Министерство образования и науки РФ
Государственное образовательное учреждение
Высшего профессионального образования
Иркутский Государственный университет
Институт математики, экономики и информатики

Основы языка программирования Java

учебное пособие

2012

Предназначено для студентов обучающихся по направлению подготовки бакалавров 010400.62 «Прикладная математика и информатика». Может быть использовано в курсах «Основы информатики» и «Языки и методы программирования».

Составитель: Мезенцев А.В.

Оглавление.

1. Встроенные типы данных, операции над ними

В этой главе перечислены базовые типы данных, операции над ними, операторы управления. Но начнем, по традиции, с простейшей программы.

Листинг 1.1. Первая программа на языке Java;

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

На этом простом примере можно заметить целый ряд существенных особенностей языка Java.

- Всякая программа представляет собой один или несколько классов, в этом простейшем примере только один *класс*.
- Начало класса отмечается служебным словом **`class`** , за которым следует имя класса, выбираемое произвольно, в данном случае **`HelloWorld`** . Все, что содержится в классе, записывается в фигурных скобках и составляет *тело класса*.
- Все действия производятся с помощью методов обработки информации, коротко говорят просто *метод*. Это название употребляется в языке Java вместо названия "функция", применяемого в других языках.
- Методы различаются по именам. Один из методов обязательно должен называться **`main`** , с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя ему **`main`**.
- Как и положено функции, метод всегда выдает в результате (чаще говорят, *возвращает*) только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры, как в нашем случае. Тогда вместо типа возвращаемого значения записывается слово **`void`**, как это и сделано в примере.
- После имени метода в скобках, через запятую, перечисляются *аргументы* или *параметры* метода. Для каждого аргумента указывается его тип и, через пробел, имя. В примере только один аргумент, его тип - массив, состоящий из строк символов. Строка символов - это встроенный в Java API тип **`String`**, а квадратные скобки - признак массива. Имя массива может быть произвольным, в примере выбрано имя **`args`**.
- Перед типом возвращаемого методом значения могут быть записаны *модификаторы*. В примере их два: слово **`public`** означает, что этот метод доступен отовсюду; слово **`static`** обеспечивает возможность вызова метода **`main()`** в самом начале выполнения программы, не

создавая объект (экземпляр класса). Модификаторы вообще необязательны, но для метода **`main()`** они необходимы.
- Все, что содержит метод, *тело метода*, записывается в фигурных скобках.

Единственное действие, которое выполняет метод **`main()`** в примере, заключается в вызове другого метода со сложным именем **`System.out.println`** и передаче ему на обработку одного аргумента, текстовой константы **`"Hello, World!"`**. Текстовые константы записываются в кавычках, которые являются только ограничителями и не входят в состав текста.

Составное имя **`System.out.println`** означает, что в классе **`System`**, входящем в Java API, определяется переменная (поле класса) с именем **`out`** , содержащая экземпляр одного из классов Java API, класса **`PrintStream`**, в котором есть метод **`println()`**. Все это станет ясно позднее, а пока просто будем писать это длинное имя.

Действие метода **`println()`** заключается в выводе своего аргумента в выходной поток, связанный обычно с выводом на экран текстового терминала, в окно MS-DOS Prompt или Command Prompt или Xterm**,** в зависимости от операционной системы. После вывода курсор переходит на начало следующей строки экрана, на что указывает окончание **`ln`**, слово **`println`** - сокращение слов print line. В составе Java API есть и метод **`print()`**, оставляющий курсор в конце выведенной строки.

Сделаем важное замечание. Язык Java различает строчные и прописные буквы, имена **`main`**, **`Main`**, **`MAIN`** различны с "точки зрения" компилятора Java. В примере важно писать **`String`**, **`System`** с заглавной буквы, а **`main`** с маленькой. Но внутри текстовой константы неважно, писать **`World`** или **`world`** , компилятор вообще не "смотрит" на нее, разница будет видна только на экране.

Свои имена можно записывать как угодно, можно было бы дать классу имя **`helloworld`** или **`helloWorld`**, но между Java-программистами заключено соглашение, называемое "Code Conventions for the Java Programming Language", хранящееся по адресу **http://java.sun.com/docs/codeconv/index.html.** Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы;
- имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы;

- имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания.

Конечно, эти правила необязательны, хотя они и входят в JLS, п. 6.8, но сильно облегчают понимание кода и придают программе характерный для Java стиль.

Итак, программа написана в каком-либо текстовом редакторе, например, Notepad. Теперь ее надо сохранить в файле, имя которого совпадает с именем класса, содержащего метод **`main()`**, и дать имени файла расширение **`.java`**. Это правило очень желательно выполнять. При этом система исполнения Java будет быстро находить метод **`main()`** для начала работы, просто отыскивая класс, совпадающий с именем файла.

Называйте файл с программой именем класса, содержащего метод **`main()`**, соблюдая регистр букв.

В нашем примере, сохраним программу в файле с именем **`HelloWorld.java`** в текущем каталоге. Затем вызовем компилятор, передавая ему имя файла в качестве аргумента:

```
javac HelloWorld.java
```

Компилятор создаст файл с байт-кодами, даст ему имя **`Helloworld.class`** и запишет этот файл в текущий каталог.

Осталось вызвать интерпретатор, передав ему в качестве аргумента имя класса (а не файла):

```
Java HelloWorld
```

На экране появится:

```
Hello, World!
```

При работе в интегрированной среде все эти действия вызываются выбором соответствующих пунктов меню или "горячими" клавишами - единых правил здесь нет.

1.1. Комментарии

В текст программы можно вставить комментарии, которые компилятор не будет учитывать. Они очень полезны для пояснений по ходу программы. В период отладки можно выключать из действий один или несколько операторов, пометив их символами комментария. Комментарии вводятся таким образом:

- за двумя наклонными чертами подряд **//**, без пробела между ними, начинается комментарий, продолжающийся до конца строки;
- за наклонной чертой и звездочкой **/*** начинается комментарий, который может занимать несколько строк, до звездочки и наклонной черты ***/** (без пробелов между этими знаками).

Комментарии очень удобны для чтения и понимания кода, они превращают программу в документ, описывающий ее действия. Программу с хорошими комментариями называют *самодокументированной.* Поэтому в Java введены комментарии третьего типа, а в состав JDK - программа **javadoc**, извлекающая эти комментарии в отдельные файлы формата HTML и создающая гиперссылки между ними: за наклонной чертой и двумя звездочками подряд, без пробелов, **/**** начинается комментарий, который может занимать несколько строк до звездочки (одной) и наклонной черты ***/** и обрабатываться программой **javadoc**. В такой комментарий можно вставить указания программе **javadoc** , которые начинаются с символа **@**.

Именно так создается документация к JDK.

Добавим комментарии к нашему примеру.

Листинг 1.2. Первая программа с комментариями

```
/**
* Разъяснение содержания и особенностей программы...
* @author Имя Фамилия (автора)
* @version 1.0 (это версия программы)
*/
class HelloWorld{ // HelloWorld — это только имя
     // Следующий метод начинает выполнение программы
     public static void main(String[] args){
     // args не используются
     /* Следующий метод просто выводит свой аргумент
     * на экран дисплея */
     System.out.println("Hello,World!");
     // Следующий вызов закомментирован,
     // метод не будет выполняться
     // System.out.println("Goodbye, World!");
     }
}
```

Звездочки в начале строк не имеют никакого значения, они написаны просто для выделения комментария. Пример, конечно, перегружен пояснениями (это плохой стиль), здесь просто показаны разные формы комментариев.

1.2. Константы

В языке Java можно записывать константы разных типов в разных видах. Перечислим их.

1.2.1. Целые

Целые константы можно записывать в трех системах счисления:

- в десятичной форме: **`+5, -7, 12345678`**;
- в восьмеричной форме, начиная с нуля: **`027`**, **`-0326`**, **`0777`**; в записи таких констант недопустимы цифры 8 и 9;

Число, начинающееся с нуля, записано в восьмеричной форме, а не в десятичной.

- в шестнадцатеричной форме, начиная с нуля и латинской буквы **`x`** или **`X`**: **`0xff0a`**, **`0xFC2D`**, **`0x45a8`**, **`0X77FF`**; здесь строчные и прописные буквы не различаются.

Целые константы хранятся в формате типа **`int`** (см. ниже).

В конце целой константы можно записать букву прописную **`L`** или строчную **`l`** , тогда константа будет сохраняться в длинном формате типа **`long`** (см. ниже): **`+25L`**, **`-037l`**, **`0xffL`**, **`0XDFDFl`**.

1.2.2. Действительные

Действительные константы записываются только в десятичной системе счисления в двух формах:

- с фиксированной точкой: **`37.25`**, **`-128.678967`**, **`+27.035`**;
- с плавающей точкой: **`2.5e34`**, **`-0.345e-25`**, **`37.2E+4`**; можно писать строчную **`e`** или прописную латинскую букву **`E`**; пробелы и скобки недопустимы.

В конце действительной константы можно поставить букву **`F`** или **`f`**, тогда константа будет сохраняться в формате типа **`float`** (см. ниже): **`3.5f`**, **`-45.67F`**, **`4.7e-5f`**. Можно приписать и букву **`D`** (или **`d`**): **`0.045D`**, **`-456.77889d`**, означающую тип **`double`** , но это излишне, поскольку действительные константы и так хранятся в формате типа **`double`**.

1.2.3. Символы

Для записи одиночных символов используются следующие формы.

- Печатные символы можно записать в апострофах: **'a'**, **'N'**, **'?'**.
- Управляющие символы записываются в апострофах с обратной наклонной чертой:
  - **'\n'** - символ перевода строки newline с кодом ASCII 10;
  - **'\r'** - символ возврата каретки CR с кодом 13;
  - **'\f'** - символ перевода страницы FF с кодом 12;
  - **'\b'** - символ возврата на шаг BS с кодом 8;
  - **'\t'** - символ горизонтальной табуляции HT с кодом 9;
  - **'\\'** - обратная наклонная черта;
  - **'\"'** - кавычка;
  - **'\''** - апостроф.

- Код любого символа с десятичной кодировкой от 0 до 255 можно задать, записав его не более чем тремя цифрами в восьмеричной системе счисления в апострофах после обратной наклонной черты: **'\123'** - буква S, **'\346'** - буква Ж в кодировке CP1251. Не рекомендуется использовать эту форму записи для печатных и управляющих символов, перечисленных в предыдущем пункте, поскольку компилятор сразу же переведет восьмеричную запись в указанную выше форму. Наибольший код **'\377'** - десятичное число 255.
- Код любого символа в кодировке Unicode набирается в апострофах после обратной наклонной черты и латинской буквы **u** ровно четырьмя шестнадцатеричными цифрами: **'\u0053'** - буква **S**, **'\u0416'** - буква **Ж**.

Символы хранятся в формате типа **char** (см. ниже).

Прописные русские буквы в кодировке Unicode занимают диапазон от **'\u0410'** - заглавная буква **А** , до **'\u042F'** - заглавная **Я** , строчные буквы от **'\u0430'** - **а**, до **'\044F'** - **я**.

В какой бы форме ни записывались символы, компилятор переводит их в Unicode, включая и исходный текст программы.

Компилятор и исполняющая система Java работают только с кодировкой Unicode.

1.2.4. Строки

Строки символов заключаются в кавычки. Управляющие символы и коды записываются в строках точно так же, с обратной наклонной чертой, но, разумеется, без апострофов, и оказывают то же действие. Строки могут располагаться только на одной строке исходного кода, нельзя открывающую кавычку поставить на одной строке, а закрывающую - на следующей.

Вот некоторые примеры:

```
"Это строка\nc переносом"
"\"Спартак\" — Чемпион!"
```

Для строковых констант определена операция сцеплений, обозначаемая плюсом.

```
" Сцепление " + "строк" дает в результате строку "Сцепление
строк"
```

Чтобы записать длинную строку в виде одной строковой константы, надо после закрывающей кавычки на первой и следующих строках поставить плюс **+**; тогда компилятор соберет две (или более) строки в одну строковую константу, например:

```
"Одна строковая константа, записанная "+
"на двух строках исходного текста"
```

1.2.5. Имена

Имена переменных, классов, методов и других объектов могут быть простыми (общее название - *идентификаторы*) и *составными* . Идентификаторы в Java составляются из так называемых *букв Java* и арабских цифр 0 - 9, причем первым символом идентификатора не может быть цифра. (Действительно, как понять запись **2e3**: как число 2000,0 или как имя переменной?). В число букв Java обязательно входят прописные и строчные латинские буквы, знак доллара **$** и знак подчеркивания _, а так же символы национальных алфавитов.

Не указывайте в именах знак доллара. Компилятор Java использует его для записи имен вложенных классов.

Вот примеры правильных идентификаторов:

```
a1        my_var    var3_5              _var      veryLongVarName
aName     theName   a2Vh36kBnMt456dX
```

В именах лучше не использовать строчную букву **l** , которую легко спутать с единицей, и букву **o**, которую легко принять за нуль.

Не забывайте о рекомендациях "Code Conventions".

В классе **Character** , входящем в состав Java API, есть два метода, проверяющие, пригоден ли данный символ для использования в идентификаторе: **isJavaIdentifierStart()**, проверяющий, является ли

символ буквой Java, и **`isJavaIdentifierPart()`**, выясняющий, является ли символ – буквой, цифрой, знаком подчеркивания или знаком доллара.

Служебные слова Java, такие как **`class`** , **`void`** , **`static`** , зарезервированы, их нельзя использовать в качестве идентификаторов своих объектов.

Составное имя - это несколько идентификаторов, разделенных точками, без пробелов, например, уже встречавшееся нам имя **`System.out.println`**.

1.3. Базовые типы данных и операции

Все типы исходных данных, встроенные в язык Java, делятся на две группы: *базовые типы* и *ссылочные типы*

Ссылочные типы делятся на *массивы*, *классы, интерфейсы* и *перечислимый тип*.

Базовых типов всего восемь. Их можно разделить на *логический* (иногда говорят *булев)* тип **`boolean`** и *числовые*.

К числовым типам относятся *целые* и *вещественные* типы.

Целых типов пять: **`byte`**, **`short`**, **`int`**, **`long`**, **`char`**.

Символы можно использовать везде, где используется тип **`int`**, поэтому JLS причисляет их к целым типам. Например, их можно использовать в арифметических вычислениях, скажем, можно написать **`2 + 'ж'`**, к двойке будет прибавляться кодировка Unicode **`'\u0416'`** буквы **`'ж'`**. В десятичной форме это число 1046 и в результате сложения получим 1048.

Обратите внимание, в записи **`2 + "Ж"`** плюс понимается как сцепление строк, двойка будет преобразована в строку, в результате получится строка **`"2ж"`**.

Вещественных типов два: **`float`** и **`double`**.

Все переменные обязательно должны быть описаны перед их использованием. Описание заключается в том, что записывается имя типа, затем, через пробел, список имен переменных, разделенных запятой. Для всех или некоторых переменных можно указать начальные значения после знака равенства, которыми могут служить любые константные выражения того же типа. Описание каждого типа завершается точкой с запятой. В программе может быть сколько угодно описаний каждого типа.

Разберем каждый тип подробнее.

1.3.1. Логический тип

Значения логического типа **boolean** возникают в результате различных сравнений, вроде **2 > 3**, и используются, главным образом, в условных операторах и операторах циклов. Логических значений всего два: **true** (истина) и **false** (ложь). Это служебные слова Java. Описание переменных этого типа выглядит так:

```
boolean b = true, bb = false, bool2;
```

Над логическими данными можно выполнять операции присваивания, например, **bool2=true**, в том числе и составные с логическими операциями; сравнение на равенство **b == bb** и на неравенство **b != bb** , а также логические операции.

1.3.2. Логические операции

В языке Java четыре логические операции:

- отрицание (NOT) **!**;
- конъюнкция (AND) **&** ;
- дизъюнкция (OR) **|** ;
- исключающее ИЛИ (XOR) **^** .

Они выполняются над логическими данными, их результатом будет тоже логическое значение **true** или **false**. Результаты логтческих операций приведены в следующей таблице:

b1	b2	!b1	b1&b2	b1|b2	b1^b2
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Словами эти правила можно выразить так:

- отрицание меняет значение истинности;
- конъюнкция истинна, только если оба операнда истинны;
- дизъюнкция ложна, только если оба операнда ложны;
- исключающее ИЛИ истинно, только если значения операндов различны.

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного вычисления:

- сокращенная конъюнкция (conditional-AND) **`&&`** ;
- сокращенная дизъюнкция (conditional-OR) **`||`** .

Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, т.е. если левый операнд конъюнкции имеет значение **`true`**, или левый операнд дизъюнкции имеет значение **`false`**.

Это правило очень удобно, например, можно записывать выражения **`(n!=0)&&(m/n>0.001)`** или **`(n==0)||(m/n>0.001)`** не опасаясь деления на нуль.

Практически всегда в Java используются именно сокращенные логические операции.

1.3.3. Целые типы

Спецификация языка Java, JLS, определяет разрядность (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений каждого типа. Для целых типов они приведены в следующей таблице:

Тип	Разрядность (байт)	Диапазон
`byte`	1	**от –128 до 127**
`short`	2	**от –32768 до 32767**
`int`	4	**от –2147483648 до 2147483647**
`long`	8	**от –9223372036854775808 до 9223372036854775807**
`char`	2	**от '\u0000' до '\uFFFF' , в десятичной форме от 0 до 65535**

Впрочем, для Java разрядность не столь важна, на некоторых компьютерах она может отличаться от указанной в таблице, а вот диапазон значений должен выдерживаться неукоснительно.

Хотя тип **`char`** занимает два байта, в арифметических вычислениях он участвует как тип **`int`**, ему выделяется 4 байта, два старших байта заполняются нулями.

Примеры определения переменных целых типов:

```
byte b1 = 50, b2 = -99, b3;
short det = 0, ind = 1;
int i = -100, j = 100, k = 9999;
long big = 50, veryBig = 2147483648L;
char c1 = 'A', c2 = '?', newLine = '\n';
```

Целые типы хранятся в двоичном виде в дополнительном коде. Последнее означает, что для отрицательных чисел хранится не их двоичное представление, а *дополнительный код* этого двоичного представления.

Дополнительный же код получается так: в двоичном представлении все нули меняются на единицы, а единицы на нули, после чего к результату прибавляется единица, разумеется, в двоичной арифметике.

Например, значение 50 переменной **`b1`** , определенной выше, будет храниться в одном байте с содержимым 00110010 , а значение -99 переменной **`b2`** - в байте с содержимым, которое вычисляем так: число 99 переводим в двоичную форму, получая 01100011 , меняем единицы и нули, получая 10011100 , и прибавляем единицу, получив окончательно байт с содержимым 10011101 .

Смысл всех этих сложностей в том, что сложение числа с его дополнительным кодом в двоичной арифметике даст в результате нуль, старший бит просто теряется. Это означает, что в такой странной арифметике дополнительный код числа является противоположным к нему числом, числом с обратным знаком. А это, в свою очередь, означает, что вместо того, чтобы вычесть из числа A число B, можно к A прибавить дополнительный код числа B. Таким образом, операция вычитания исключается из набора машинных операций.

Над целыми типами можно производить массу операций. Их набор восходит к языку C, он оказался удобным и кочует из языка в язык почти без изменений. Особенности применения этих операций в языке Java показаны на примерах.

1.3.4. Операции над целыми типами

Все операции, которые производятся над целыми числами, можно разделить на следующие группы.

Арифметические операции

К арифметическим операциям относятся:

- сложение **+** (плюс);

- вычитание **-** (дефис);
- умножение ***** (звездочка);
- деление **/** (наклонная черта - слэш);
- взятие остатка от деления (деление по модулю) **%** (процент);
- инкремент (увеличение на единицу) **++** ;
- декремент (уменьшение на единицу) **--**

Между сдвоенными плюсами и минусами нельзя оставлять пробелы. Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате дает опять целое (так называемое *"целое деление"),* например, **5/2** даст в результате 2 , а не 2.5 , а **5/(-3)** даст -1 . Дробная часть попросту отбрасывается, происходит усечение частного.

В Java принято целочисленное деление.

Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать **5/2.0** или **5.0/2** или **5.0/2.0** и получим 2.5 как результат деления вещественных чисел.

Операция *деление по модулю* определяется так:

```
a % b = a - (a / b) * b
```

например, **5%2** даст в результате 1 , а **5%(-3)** даст, 2 , т.к. 5 = (-3) * (-1) + 2 , но **(-5)%3** даст -2 , поскольку -5 = 3 * (-1) - 2 .

Операции *инкремент* и *декремент* означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать **5++** или **(a + b)++**.

Например, после выполнения следующих операций:

```
int i=-100; j=100;
i++; j--;
```

i будет иметь значение -99 , а **j** - 99.

Эти операции можно записать и перед переменной: **++i**, **--j**. Разница проявится только в выражениях: при первой форме записи *(постфиксной)* в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи *(префиксной)* сначала изменится переменная, и ее новое значение будет участвовать в выражении.

Например, после приведенных выше описаний, **`(k++) + 5`** даст в результате 10004 , а переменная **`k`** примет значение 10000 . Но в той же исходной ситуации **`(++k) + 5`** даст 10005 , а переменная **`k`** станет равной 10000 .

Приведение типов

Результат арифметической операции имеет тип **`int`**, кроме того случая, когда один из операндов типа **`long`**. В этом случае результат будет типа **`long`**.

Перед выполнением арифметической операции всегда происходит *повышение* типов **`byte`**, **`short`**, **`char`**. Они преобразуются в тип **`int`**, а может быть, и в тип **`long`**, если другой операнд типа **`long`**. Операнд типа **`int`** повышается до типа **`long`**, если другой операнд типа **`long`**. Конечно, числовое значение операнда при этом не меняется.

Это правило приводит иногда к неожиданным результатам. Попытка откомпилировать простую программу:

```
class InvalidDef{
     public static void main(String [] args) {
          byte b1 = 50, b2 = -99;
          short k = b1 + b2; // Неверно! '
          System.out.println("k=" + k);
     }
}
```

Приведет к сообщениям компилятора:

```
InvalidDef.java:4: possible loss of precision
found   : int
required: short
     short k = b1 + b2;
                  ^
1 error
```

эти сообщения означают, что в файле InvalidDef.java, в строке 4, обнаружена возможная потеря точности (possible loss of precision). Затем приводятся обнаруженный (found) и нужный (required) типы, выводится строка, в которой обнаружена ошибка, и отмечается символ, при разборе которого найдена ошибка. Затем указано общее количество обнаруженных ошибок (1 error).

В таких случаях следует выполнить явное приведение типа. В данном случае это будет *сужение* типа **`int`** до типа **`short`**. Оно осуществляется операцией явного приведения, которая записывается перед приводимым значением в виде имени типа в скобках. Определение

```
short k = (short)(b1 + b2);
```

будет верным.

Сужение осуществляется просто отбрасыванием старших битов, что необходимо учитывать для больших значений. Например, определение

```
byte b = (byte) 300;
```

даст переменной **b** значение 44 . Действительно, в двоичном представлении числа 300 , равном 100101100 , отбрасывается старший бит и получается 00101100 .

Таким же образом можно произвести и явное *расширение* типа, если в этом есть необходимость.

Если результат целой операции выходит за диапазон своего типа **int** или **long**, то автоматически происходит приведение по модулю, равному длине этого диапазона, и вычисления продолжаются, переполнение никак не отмечается.

В языке Java нет целочисленного переполнения.

Операции сравнения

В языке Java шесть обычных операций сравнения целых чисел по величине:

- больше **>** ;
- меньше **<** ;
- больше или равно **>=** ;
- меньше или равно **<=** ;
- равно **==** ;
- не равно **!=** .

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись **=>** будет неверной.

Результат сравнения - логическое значение: **true**, в результате, например, сравнения **3!=5**; или **false**, например, в результате сравнения **3==5**.

Для записи сложных сравнений следует привлекать логические операции. Например, в вычислениях часто приходится делать проверки вида **a<x<b**. Подобная запись на языке Java приведет к сообщению об ошибке, поскольку первое сравнение, **a<x**, даст **true** или **false**, а Java не знает, больше это, чем **b**, или меньше. В данном случае следует написать выражение **(a<x)&&(x<b)**, причем здесь скобки можно опустить, написать просто **a<x && x<b**, так приоритет операций сравнения выше чем приоритет логических операций.

Побитовые операции

Иногда приходится изменять значения отдельных битов в целых данных. Это выполняется с помощью побитовых операций путем наложения маски. В языке Java есть четыре побитовые операции:

- дополнение **~**
- побитовая конъюнкция **&**
- побитовая дизъюнкция **|**
- побитовое исключающее ИЛИ **^**

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу **int** или **long**, так же как и для арифметических операций, а значит, и к одной разрядности. Операции над каждой парой битов выполняются согласно следующей таблицы:

n1	n2	~n1	n1 & n2	n1 | n2	n1 ^ n2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

В нашем примере **b1 == 50**, двоичное представление **00110010**, **b2 == -99**, двоичное представление **10011101**. Перед операцией происходит повышение до типа **int**. Получаем представления из 32-х разрядов для **b1** - **0...00110010**, для **b2** - **1...10011101**. В результате побитовых операций получаем:

- **~b2 == 98**, двоичное представление **0...01100010**;
- **b1 & b2 == 16**, двоичное представление **0...00010000**;
- **b1 | b2 == -65**, двоичное представление **1...10111111**;
- **b1 ^ b2 == -81**, двоичное представление **1...10101111**.

Двоичное представление каждого результата занимает 32 бита.

Заметьте, что дополнение **~x** всегда эквивалентно **(-x)-1**.

Сдвиги

В языке Java есть три операции сдвига двоичных разрядов:

- сдвиг влево **<<**;
- сдвиг вправо **>>**;

- беззнаковый сдвиг вправо **>>>**.

Эти операции отличаются тем, что левый и правый операнды в них имеют разный смысл. Слева стоит значение целого типа, а правая часть показывает, на сколько двоичных разрядов сдвигается значение, стоящее в левой части.

Например, операция **b1<<2** сдвинет влево на 2 разряда предварительно повышенное значение **0...00110010** переменной **b1**, что даст в результате **0...011001000**, десятичное 200. Освободившиеся справа разряды заполняются нулями, левые разряды, находящиеся за 32-м битом, теряются.

Операция **b2<<2** сдвинет повышенное значение **1...10011101** на два разряда влево. В результате получим **1...1001110100**, десятичное значение - 396.

Заметьте, что сдвиг влево на **n** разрядов эквивалентен умножению числа на **2** в степени **n**.

Операция **b1 >> 2** даст в результате 0...00001100, десятичное 12, а **b2 >> 2** - результат 1..11100111, десятичное -25, т. е. слева распространяется старший бит, правые биты теряются. Это так называемый *арифметический сдвиг.*

Операция беззнакового сдвига во всех случаях ставит слева на освободившиеся места нули, осуществляя *логический сдвиг.* Но вследствие предварительного повышения это имеет эффект только для нескольких старших разрядов отрицательных чисел. Так, **b2>>>2** имеет результатом 001...100111, десятичное число 1 073 741 799.

Если же мы хотим получить логический сдвиг исходного значения 100111101 переменной **b2**, т.е., 0...00100111, надо предварительно наложить на **b2** *маску,* обнулив старшие биты: **(b2 & 0XFF) >>> 2**.

Будьте осторожны при использовании сдвигов вправо.

1.3.5. Вещественные типы

Вещественных типов в Java два: **float** и **double**. Они характеризуются разрядностью, диапазоном значений и точностью представления, отвечающим стандарту IEEE 754-1985 с некоторыми изменениями. К обычным вещественным числам добавляются еще три значения:

- Положительная бесконечность, выражаемая константой **POSITIVE_INFINITY** и возникающая при переполнении положительного значения, например, в результате операции умножения **3.0 * 6e307**.
- Отрицательная бесконечность **NEGATIVE_INFINITY**.

- "Не число", записываемое константой **NaN** (Not a Number) и возникающее при делении вещественного числа на нуль или умножении нуля на бесконечность.

Кроме того, стандарт различает положительный и отрицательный нуль, возникающий при делении на бесконечность соответствующего знака, хотя сравнение `0.0 == -0.0` дает **true**.

Операции с бесконечностями выполняются по обычным математическим правилам.

Во всем остальном вещественные типы - это обычные, вещественные значения, к которым применимы все арифметические операции и сравнения, перечисленные для целых типов.

В языке Java взятие остатка от деления **%**, инкремент **++** и декремент **--** применяются и к вещественным типам.

Характеристики вещественных типов приведены в следующей таблице:

Тип	Разрядность	Диапазон	Точность
float	4 байта	3,4e-38 < |x| < 3,4e38	7-8 цифр
double	8 байтов	1,7e-308<|x|<1,7e308	17 цифр

Примеры определения вещественных типов:

```
float x = 0.001, y = -34.789;
double z1 = -16.2305, z2;
```

Поскольку к вещественным типам применимы все арифметические операции и сравнения, целые и вещественные значения можно смешивать в операциях. При этом правило приведения типов дополняется такими условиями:

- если в операции один операнд имеет тип **double**, то и другой приводится к типу **double**;
- если один операнд имеет тип **float**, то и другой приводится к типу **float**;
- в противном случае действует правило приведения целых значений.

1.3.6. Операции присваивания

Простая операция присваивания записывается знаком равенства **=**, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной:

x=3.5, y=2*(x-0.567)/(x+2), b=x<y, bb=x>=y&&b.

Операция присваивания действует так: выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. Результатом операции будет приведенное значение правой части.

Операция присваивания имеет еще одно, побочное, действие: переменная, стоящая слева, получает приведенное значение правой части, старое ее значение теряется.

Кроме простой операции присваивания есть еще 11 *составных* операций присваивания:

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=.

Символы записываются без пробелов, нельзя переставлять их местами.

Все составные операции присваивания действуют по одной схеме:

x *операция*= a

эквивалентно

x = (*тип* x)(x *операция* a).

Напомним, что переменная **ind** типа **short** определена у нас со значением **1**. Присваивание **ind+=7.8** даст в результате число 8, то же значение получит и переменная **ind**. Эта операция эквивалентна простой операции присваивания **ind=(short)(ind+7.8)**.

Перед присваиванием, при необходимости, автоматически производится приведение типа. Поэтому:

```
byte b = 1;
b = b + 10; // Ошибка!
b += 10; // Правильно!
b = (byte)(b + 10); // Правильно!
```

Перед сложением **b+50** происходит повышение **b** до типа **int**, результат сложения тоже будет типа **int** и, в первом случае, не может быть присвоен переменной **b** без явного приведения типа. Во втором случае перед присваиванием произойдет сужение результата сложения до типа **byte**.

1.3.7. Условная операция

Это единственная операция, которая имеет три операнда. Вначале записывается произвольное логическое выражение, т. е. имеющее в результате **`true`** или **`false`**, затем знак вопроса, потом два произвольных выражения, разделенных двоеточием, например,

```
x < 0 ? 0 : x
x > y ? x — y : x + y
```

Условная операция выполняется так. Сначала вычисляется логическое выражение. Если получилось значение **`true`**, то вычисляется первое выражение после вопросительного знака и его значение будет результатом всей операции. Последнее выражение при этом не вычисляется. Если же получилось значение **`false`**, то вычисляется только последнее выражение, его значение будет результатом операции.

Это позволяет написать **`n==0?m:m/n`** не опасаясь деления на нуль. Условная операция поначалу кажется странной, но она очень удобна для записи небольших разветвлений.

1.3.8. Выражения

Из констант и переменных, операций над ними, вызовов методов и скобок составляются *выражения*. Разумеется, все элементы выражения должны быть совместимы, нельзя написать, например, **`2 + true`**. При вычислении выражения выполняются четыре правила:

- Операции одного приоритета вычисляются слева направо: **`x+y+z`** вычисляется как **`(x+y)+z`**. Исключение: операции присваивания вычисляются справа налево: **`x=y=z`** вычисляется как **`x=(y=z)`**.
- Левый операнд вычисляется раньше правого.
- Операнды полностью вычисляются перед выполнением операции.
- Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Следующие примеры показывает особенности применения первых трех правил. Пусть

```
int a = 3, b = 5;
```

Тогда результатом выражения **`b+(b=3)`** будет число 8; но результатом выражения **`(b=3)+b`** будет число 6. Выражение **`b+=(b=3)`** даст в результате 8, потому что вычисляется как первое из приведенных выше выражений.

Четвертое правило можно продемонстрировать так. При тех же определениях **a** и **b** в результате вычисления выражения **b+=a+=b+=7** получим 20. Хотя операции присваивания выполняются справа налево и после первой, правой, операции значение **b** становится равным 12, но в последнем, левом, присваивании участвует старое значение **b**, равное 5. А в результате двух последовательных вычислений **a+=b+=7; b+=a;** получим 27, поскольку во втором выражении участвует уже новое значение переменной **b**, равное 12.

Выражения могут иметь сложный и запутанный вид. В таких случаях возникает вопрос о приоритете операций, о том, какие операции будут выполнены в первую очередь. Естественно, умножение и деление производится раньше сложения и вычитания. Остальные правила перечислены в следующем разделе.

Порядок вычисления выражения всегда можно отрегулировать скобками, их можно ставить сколько угодно. Но здесь важно соблюдать "золотую середину". При большом количестве скобок снижается наглядность выражения и легко ошибиться в расстановке скобок. Если выражение со скобками корректно, то компилятор может отследить только парность скобок, но не правильность их расстановки.

1.3.9. Приоритет операций

Операции перечислены в порядке убывания приоритета. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции **++** и **--**.
2. Префиксные операции **++** и **--**, дополнение **~** и отрицание **!**.
3. Приведение типа **(*тип*)**.
4. Умножение *****, деление **/** и взятие остатка **%**.
5. Сложение **+** и вычитание **-**.
6. Сдвиги **<<**, **>>**, **>>>**.
7. Сравнения **>**, **<**, **>=**, **<=**.
8. Сравнения **==**, **!=**.
9. Побитовая конъюнкция **&**.
10. Побитовое исключающее ИЛИ **^**.
11. Побитовая дизъюнкция **|** .
12. Конъюнкция **&&**.
13. Дизъюнкция **||** .
14. Условная операция **?:**.
15. Присваивания **=**, **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **^=**, **|=**, **<<=**, **>>=**, **>>>=**.

Здесь перечислены не все операции языка Java, список будет дополняться по мере изучения новых операций.

1.4. Операторы

Набор операторов языка Java включает:

- операторы описания переменных и других объектов (они были рассмотрены выше);
- операторы-выражения;
- операторы присваивания;
- условный оператор **`if`**;
- три оператора цикла **`while`**, **`do-while`**, **`for`**;
- оператор варианта **`switch`**;
- Операторы перехода **`break`**, **`continue`** и **`return`**;
- блок **`{}`**;
- пустой оператор **`;`** - просто точка с запятой.

Здесь приведен не весь набор операторов Java, он будет дополняться по мере изучения языка.

В языке Java нет оператора **`goto`**.

Всякий оператор завершается точкой с запятой.

Можно поставить точку с запятой в конце любого выражения, и оно станет оператором. Но смысл это имеет только для операций присваивания, инкремента и декремента и вызовов методов. В остальных случаях это бесполезно, потому что вычисленное значение выражения потеряется.

Точка с запятой в Java не разделяет операторы, а является частью оператора.

Линейное выполнение алгоритма обеспечивается последовательной записью операторов. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста.

1.4.1. Блок

Блок заключает в себе нуль или несколько операторов с целью использовать их как один оператор в тех местах, где по правилам языка можно записать только один оператор. Например, **`{x = 5; y = 7;}`**. Можно записать и пустой блок, просто пару фигурных скобок **`{}`**.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

1.4.2. Операторы присваивания

Точка с запятой в конце любой операции присваивания превращает ее в оператор присваивания. Побочное действие операции - присваивание - становится в операторе основным.

Разница между операцией и оператором присваивания носит лишь теоретический характер. Присваивание чаще используется как оператор, а не операция.

1.4.3. Условный оператор

Условный оператор в языке Java записывается так:

if(*логВыр*) *оператор1* else *оператор2*

и действует следующим образом. Сначала вычисляется логическое выражение ***логВыр***. Если результат **true**, то действует ***оператор1*** и на этом действие условного оператора завершается, ***оператор2*** не действует, далее будет выполняться следующий за **if** оператор. Если результат **false**, то действует ***оператор2***, при этом ***оператор1*** вообще не выполняется.

Условный оператор может быть сокращенным:

if(*логВыр*) *оператор1*

и в случае **false** не выполняется ничего.

Синтаксис языка не позволяет записывать несколько операторов ни в ветви **then**, ни в ветви **else**. При необходимости составляется блок операторов в фигурных скобках. Соглашения "Code Conventions" рекомендуют всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как в следующем примере:

```
if(a<x){
     x=a+b;
}else{
     x=a–b;
}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма. Мы не будем строго следовать этому правилу.

Очень часто одним из операторов является снова условный оператор, например:

```
if(n==0}{
     sign=0;
```

```
}else if(n<0){
     sign=-1;
}else{
     sign=1;
}
```

При этом может возникнуть такая ситуация:

```
int ind=5, x=100;
if(ind>=10) if(ind<=20) x=0; else x=1;
```

Сохранит переменная **x** значение 100 или станет равной 1? Здесь необходимо волевое решение, и общее для большинства языков, в. том числе и Java,. правило таково: ветвь **else** относится к ближайшему слева условию **if**, не имеющему своей ветви **else**. Поэтому в нашем примере переменная **x** останется равной 100.

Изменить этот порядок можно с помощью блока:

```
if(ind>10){if(ind<20) x=0; else x=1;}
```

Вообще не стоит увлекаться сложными вложенными условными операторами. Проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в нашем примере можно написать

```
if(ind >=10 && ind <= 20) x = 0; else x = 1;
```

В листинге 1.4 вычисляются корни квадратного уравнения $ax^2 + bx + c = 0$ для любых коэффициентов, в том числе и нулевых.

Листинг 1.4. Вычисление корней квадратного уравнения

```
class QuadraticEquation{
   public static void main(String[] args){
      double a=0.5, b=-2.7, c=3.5, d, eps=1e-8;
      if(Math.abs(a)<eps)
         if(Math.abs(b)<eps)
            if(Math.abs(c)<eps) // Все коэффициенты равны нулю
               System.out.println("Решение — любое число");
            else
               System.out.println("Решений нет");
         else
            System.out.println("xl = x2 = " +(-c / b) ) ;
      else { // Коэффициенты не равны нулю
         if((d = b**b — 4*a*c)< 0.0){ // Комплексные корни
            d = 0.5 * Math.sqrt(-d) / a;
            a = -0.5 * b/ a;
            System.out.println("xl = " +a+ " +i " + d +
               ",x2 = " + a + " -i " +d);
```

```
        } else { // Вещественные корни
          d = 0.5 * Math.sqrt(d) / a;
          a = -0.5 * b / a;
          System.out.println("x1 = " + (a + d) +
             ", x2 = " +(a - d));
        }
      }
   }
}
```

В этой программе использованы методы вычисления модуля **abs()** и квадратного корня **sqrt()** вещественного числа из встроенного в Java API класса **Math**. Поскольку все вычисления с вещественными числами производятся приближенно, мы считаем, что коэффициент уравнения равен нулю, если его модуль меньше 0,00000001. Обратите внимание на то, как в методе **println()** используется сцепление строк, и на то, как операция присваивания при вычислении дискриминанта вложена в логическое выражение.

1.4.4. Операторы цикла

Основной оператор цикла - оператор **while** - выглядит так:

while(*логВыр*)*оператор*

Вначале вычисляется логическое выражение ***логВыр;*** если его значение **true**, то выполняется ***оператор***, образующий тело цикла. Затем снова вычисляется ***логВыр*** и выполняется ***оператор***, и так до тех пор, пока не получится значение **false**. Если ***логВыр*** изначально равняется **false**, то ***оператор*** не будет выполнен ни разу. Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на нуль и других неприятностей. Поэтому оператор **while** является основным оператором цикла.

Оператор в цикле может быть и пустым, например, следующий фрагмент кода:

```
int i=0;
double s=0.0;
while((s+=1.0/++i)<10);
```

вычисляет количество **i** сложений, которые необходимо сделать, чтобы гармоническая сумма **s** достигла значения 10. Такой стиль характерен для языка C. Не стоит им увлекаться, чтобы не превратить текст программы в шифровку, на которую вы сами через пару недель будете смотреть с недоумением.

Можно организовать и бесконечный цикл:

while(true) ***оператор***

Конечно, из такого цикла следует предусмотреть какой-то выход, например, оператором **break**, как в листинге 1.5. В противном случае программа зациклится.

Если в цикл надо включить несколько операторов, то следует образовать блок операторов **{}**.

Второй оператор цикла - оператор **do-while** - имеет вид

do ***оператор*** **while(*логВыр*);**

Здесь сначала выполняется ***оператор***, а потом происходит вычисление логического выражения ***логВыр***. Цикл выполняется, пока ***логВыр*** остается равным **true**.

В цикле **do-while** проверяется условие продолжения, а не окончания цикла.

Существенное различие между этими двумя операторами цикла только в том, что в цикле **do-while** оператор обязательно выполнится хотя бы один раз.

Например, пусть задана какая-то функция *f(x)*, имеющая на отрезке,[*a*; *b*] ровно один корень. В листинге 1.5 приведена программа, вычисляющая этот корень приближенно методом деления пополам (бисекции, дихотомий).

Листинг 1.5. Нахождение корня нелинейного уравнения методом бисекции

```
class Bisection{
   static double f(double x){
      return x*x*x – 3*x*x +3; // Или что-то другое
   }
   public static void main(String[] args){
      double a=0.0, b=1.5, c, y, eps=1e-8;
      do{
         c=0.5*(a+b); y=f(c);
         if(Math.abs(y)<eps) break;
         // Корень найден. Выходим из цикла
         // Если на концах отрезка [a; c]
         //функция имеет разные знаки:
         if(f(a)*y<0.0) b=c;
         // Значит, корень здесь. Переносим точку b в точку c
         //В противном случае:
         else a=c;
         // Переносим точку a в точку c
         // Продолжаем, пока отрезок [a; b] не станет мал
      } while(Math.abs(b-a)>=eps);
      System.out.println("x = " +c+ ", f(" +c+ ") = " +y) ;
   }
}
```

Класс **Bisection** сложнее предыдущих примеров: в нем кроме метода **main()** есть еще метод вычисления функции *f(x)*. Здесь метод **f()** очень прост: он вычисляет значение многочлена и возвращает его в качестве значения функции, причем все это выполняется одним оператором:

return *выражение*

В методе **main()** появился еще один новый оператор **break**, который просто прекращает выполнение цикла, если мы по счастливой случайности наткнулись на приближенное значение корня. В объявлении метода **f()** присутсвует модификатор **static**. Он необходим потому, что метод **f()** вызывается из статического метода **main()**.

Третий оператор цикла - оператор **for** - выглядит так:

for(*списокВыр1*;*логВыр*;*списокВыр2*) *оператор;*

Перед выполнением цикла вычисляется список выражений ***списокВыр1***. Это нуль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем вычисляется логическое выражение ***логВыр***. Если оно истинно, **true**, то выполняется ***оператор***, потом вычисляются слева направо выражения из списка выражений ***списокВыр2***. Далее снова проверяется ***логВыр***. Если оно истинно, то выполняется оператор и ***списокВыр2*** и т.д. Как только ***логВыр*** станет равным **false**, выполнение цикла заканчивается.

Короче говоря, выполняется последовательность операторов

```
списокВыр1;
while(логВыр){
      оператор
      списокВыр2;
}
```

с тем исключением, что, если оператором в цикле является оператор **continue**, то ***списокВыр2*** все-таки выполняется.

Вместо ***списокВыр1*** может стоять одно определение переменных обязательно с начальным значением. Такие переменные доступны только в пределах этого цикла.

Любая часть оператора **for** может отсутствовать: тело цикла может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл:

```
for(;;) оператор
```

В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Хотя в операторе **for** заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например, фрагмент кода

```
int s=0;
for(int k=1; k<=N; k++) s+=k*k;
// Здесь переменная k уже недоступна
```

вычисляет сумму квадратов первых N натуральных чисел.

1.4.5. Оператор **continue** и метки

Оператор **continue** используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова **continue** и осуществляет немедленный переход к следующей итерации цикла. В очередном фрагменте кода оператор **continue** позволяет обойти деление на нуль:

```
for(int i=0; i<N; i++){
     if(i==j) continue;
     s+=1.0/(i-j);
}
```

Вторая форма содержит метку:

```
continue метка
```

метка записывается, как все идентификаторы, из букв Java, цифр и знака подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается *помеченный оператор* или *помеченный блок.*

Метка не требует описания и не может начинаться с цифры.

Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из объемлющих циклов, а именно, помеченного цикла.

1.4.6. Оператор **break**

Оператор **break** используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

Оператор

break ***метка***

применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию.

```
M1: { // Внешний блок
     M2: { // Вложенный блок — второй уровень
          M3: { // Третий уровень вложенности...
               if (что-то случилось) break M2;
               // Если true, то здесь ничего не выполняется
              }
              // Здесь тоже ничего не выполняется
         }
         // Сюда передается управление
    }
```

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором **break** с меткой.

1.4.7. Оператор варианта

Оператор варианта **switch** организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме **long**) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

```
switch(целВыр){
     case констВыр1: оператор1
     case констВыр2: оператор2
     ...
     case констВырN: операторN
     default: операторDef
}
```

Стоящее в скобках выражение ***целвыр*** может быть типа **byte**, **short**, **int**, **char**, но не **long**. Целые числа или целочисленные выражения, составленные из констант, ***констВыр*** тоже не должны иметь тип **long**.

Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от друга значения. Сначала вычисляется целочисленное выражение ***целВыр***. Если оно совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются все следующие операторы, включая и ***операторDef***, и работа оператора варианта заканчивается.

Если же ни одна константа не равна значению выражения, то выполняется ***операторDef*** и все следующие за ним операторы. Поэтому ветвь **default** должна записываться последней. Ветвь **default** может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает.

Таким образом, константы в вариантах **case** играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи.

После выполнения одного варианта оператор **switch** продолжает выполнять все оставшиеся варианты.

Чаще всего необходимо "пройти" только одну ветвь операторов. В таком случае используется оператор **break**, сразу же прекращающий выполнение оператора **switch**. Может понадобиться выполнить один и тот же оператор в разных ветвях **case**. В этом случае ставят несколько меток **case** подряд. Вот простой пример.

```
switch(dayOfWeek){
     case 1: case 2: case 3: case 4:case 5:
          System.out.println("Week-day");, break;
     case 6: case 7:
          System.out.println("Week-end"); break;
     default:
          System.out.printlnt"Unknown day");
}
```

Не забывайте завершать варианты оператором **break**.

1.5. Массивы

В программировании *массив* - это совокупность переменных одного типа, хранящихся в смежных ячейках оперативной памяти.

Массивы в языке Java относятся к ссылочным типам и описываются своеобразно, но характерно для ссылочных типов. Описание производится в три этапа.

Первый этап - *объявление*. На этом этапе определяется только переменная типа *ссылка на массив,* содержащая тип массива. Для этого записывается имя типа элементов массива, квадратными скобками указывается, что объявляется ссылка на массив, а не простая переменная, и перечисляются имена переменных типа ссылка, например,

```
double[] a, b;
```

Здесь определены две переменные - ссылки **a** и **b** на массивы типа **double**. Можно поставить квадратные скобки и непосредственно после имени. Это удобно делать среди определений обычных переменных:

```
int i = 0, ar[], k = -1;
```

Здесь определены две переменные целого типа **i** и **k**, и объявлена ссылка на целочисленный массив **ar**.

Второй этап — *определение*. На этом этапе указывается количество элементов массива, называемое его *длиной,* выделяется место для массива в оперативной памяти, переменная-ссылка получает адрес массива. Все эти действия производятся еще одной операцией языка Java - операцией **new *тип***, выделяющей участок в оперативной памяти для объекта указанного в операции типа и возвращающей в качестве результата адрес этого участка. Например,

```
a = new double[5];
b = new double[100];
ar = new int[50];
```

При этом все элементы массива получают нулевые значения.

Индексы массивов всегда начинаются с 0. Массив **a** состоит из пяти переменных **a[0]**, **a[1]**, …, **a[4]**. Элемента **a[5]** в массиве нет. Индексы можно задавать любыми целочисленными выражениями, кроме типа **long**, например, **a[i+j]**, **a[i%5]**, **a[++i]**. Исполняющая система Java следит за тем, чтобы значения этих выражений не выходили за границы длины массива.

Третий этап — *инициализация*. На этом этапе элементы массива получают начальные значения. Например,

```
a[0] = 0.01; a[1] = -3.4; a[2] = 2.89; a[3] = 4.5; a[4] = -6.7;
for(int i = 0; i < 100; i++) b[i] = 1.0 /i;
for(int i = 0; i < 50; i++) ar[i] = 2 * i + 1;
```

Первые два этапа можно совместить:

```
double[] a = new double[5], b = new double[100];
int i = 0, ar[] = new int[50], k = -1;
```

Можно сразу задать и начальные значения, записав их в фигурных скобках через запятую в виде констант или константных выражений. При этом даже необязательно указывать количество элементов массива, оно будет равно количеству начальных значений;

```
double[] a = {0.01, -3.4, 2.89, 4.5, -6.7};
```

Можно совместить второй и третий этап:

```
a = new double[] {0.1, 0.2, -0.3, 0.45, -0.02};
```

Можно даже создать безымянный массив, сразу же используя результат операции **new**, например, так:

```
System.out.println(new char[] {'H', 'e', 'l', 'l', 'o'});
```

Ссылка на массив не является частью описанного массива, ее можно перебросить на другой массив того же типа операцией присваивания. Например, после присваивания **a=b** обе ссылки **a** и **b** указывают на один и тот же массив из 100 вещественных переменных типа **double** и содержат один и тот же адрес.

Ссылка может присвоить "пустое" значение **null**, не указывающее ни на какой адрес оперативной памяти:

```
ar = null;
```

После этого массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

Кроме простой операции присваивания, со ссылками можно производить еще только сравнения на равенство, например, **a == b**, и неравенство, **a != b**. При этом сопоставляются адреса, содержащиеся в ссылках, мы можем узнать, не ссылаются ли они на один и тот же массив.

Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Кроме ссылки на массив, для каждого массива автоматически определяется целая константа с одним и тем же именем **length**. Она равна длине массива. Для каждого массива имя этой константы уточняется именем массива через точку. Так, после наших определений, константа **a.length** равна 5, константа **b.length** равна 100, а **ar.length** равна 50.

Последний элемент массива **a** можно записать так: **a[a.length-1]**, предпоследний - **a[a.length-2]** и т.д. Элементы массива обычно перебираются в цикле вида:

```
double aMin = a[0], aMax = aMin;
for(int i = 1; i < a.length; i++){
     if(a[i] < aMin) aMin = a[i];
     if(a[i] > aMax) aMax = a[i];
}
double range = aMax – aMin;
```

Здесь вычисляется диапазон значений массива.

Ситуация, когда надо перебрать все элементы массива в порядке возрастания их индексов встречается очень часто. Начиная с версии Java SE 5 для таких случаев в язык Java введен оператор цикла "for-each". Вот как можно записать предыдущий пример с использованием этого цикла:

```
double aMin = a[0], aMax = aMin;
for(double x: a){
     if(x < aMin) aMin = x;
     if(x > aMax) aMax = x;
}
double range = aMax – aMin;
```

Обратите внимание на то, что в цикле **`for`** определяется переменная **`x`** того же типа, что и элементы массива. Эта переменная принимает последовательно значения всех элементов массива от первого элемента до последнего.

Элементы массива - это обыкновенные переменные своего типа, с ними можно производить все операции, допустимые для этого типа: **`(a[2] + a[4]) / a[0]`** и т. д.

Массив символов в Java не является строкой, даже если он заканчивается нуль-символом **`'\u0000'`**.

1.6. Многомерные массивы

Элементами массивов в Java могут быть массивы. Можно объявить:

```
char[][] c;
```

что эквивалентно

```
char[] c[];
```

или

```
char c[][];
```

Затем определяем внешний массив:

```
c = new char[3][];
```

Становится ясно, что **`c`** - массив, состоящий из трех элементов-массивов. Теперь определяем его элементы-массивы:

```
c[0] = new char[2];
```

```
c[1] = new char[4];
c[2] = new char[3];
```

После этих определений переменная **c.length** равна 3, **c[0].length** равна 2, **c[1].length** равна 4 и **c[2].length** равна 3.

Наконец, задаем начальные значения **c[0][0] = 'a'**, **c[0][1] = 'r'**,

c[1][0] = 'r', **c[1][1] = 'a'**, **c[1][2] = 'y'** и т.д.

Двумерный массив в Java не обязан быть прямоугольным.

Описания можно сократить:

```
int[][] d = new int[3][4];
```

А начальные значения задать так:

```
int[][] inds = {{1, 2, 3}, {4, 5, 6}};
```

В листинге 1.6 приведен пример программы, вычисляющей первые 10 строк треугольника Паскаля, заносящей их в треугольный массив и выводящей его элементы на экран.

Листинг 1.6. Треугольник Паскаля

```
class PascalTriangle{
   public static final int LINES = 10; // Так определяются
   константы
   public static void main(String[] args) {
      int[][] p, = new int[LINES] [];
      p[0] = new int[1];
      System.out.println(p[0][0] = 1);
      p[1] = new int[2];
      p[1][0] = p[1][1] = 1;
      System.out.println(p[1][0] + " " + p[1][1]);
      for(int i = 2; i < LINES; i++){
         p[i] = new int[i+1];
         System.out.print((p[i][0] = 1) + " ");
         for(int j = 1; j < i; j++)
            System.out.print( (p[i][j]=p[i-1][j-1]-bp[i-1][j]) +
            " ");
         System.out.println(p[i][i]=1)
      }
   }
}
```

Эта программа выведет следующий результат:

```
1
1 1
1 2 1
```

```
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

2. Объектно-ориентированное программирование в Java

2.1. Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование развивается уже более двадцати лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий. Перечислим их.

2.1.1. Абстракция

Описывая поведение какого-либо объекта, например автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Мы должны *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Описание каждой модели производится в виде одного или нескольких *классов* (classes). Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. Класс содержит описание переменных и констант, характеризующих объект. Они называются *полями класса* (class fields). Процедуры, описывающие поведение объекта, называются *методами класса* (class methods). Внутри класса можно описать и *вложенные классы* (nested classes) и *вложенные интерфейсы*. Поля, методы и вложенные классы первого уровня являются *членами класса* (class

members). Разные школы объектно-ориентированного программирования предлагают разные термины, мы используем терминологию, принятую в технологии Java.

Вот набросок описания автомобиля:

```
class Automobile{
int maxVelocity;    // Поле, содержащее наибольшую скорость
                    //автомобиля
int speed; // Поле, содержащее текущую скорость автомобиля
int weight; // Поле, содержащее вес автомобиля
// Прочие поля...
void moveTo(int x, int y){ // Метод, моделирующий перемещение
          // автомобиля. Параметры x и y — не поля
     int a = 1; // Локальная переменная — не поле
          // Тело метода. Здесь описывается закон
          // перемещения автомобиля в точку (x, y)
}
// Прочие методы...
}
```

В Java нет вложенных процедур и функций, в теле метода нельзя описать другой метод.

После того как описание класса закончено, можно создавать конкретные объекты, *экземпляры* описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них.

```
Automobile lada2110, fordScorpio, oka;
```

Затем операцией **new** определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции **new** повторяется имя класса со скобками **Automobile()**. Это так называемый *конструктор* класса, но о нем поговорим попозже.

Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350;
oka.moveTo(35, 120);
```

Напомним, что текстовая строка в кавычках понимается в Java как объект класса **String**. Поэтому можно написать

```
int strlen = "Это объект класса String".length();
```

Объект "строка" выполняет метод **length()**, один из методов своего класса **String**, подсчитывающий число символов в строке. В результате получаем значение **strlen**, равное 24. Подобная странная запись встречается в программах на Java на каждом шагу.

Во многих ситуациях строят несколько моделей с разной степенью детализации. Скажем, для конструирования пальто и шубы нужна менее точная модель контуров человеческого тела и его движений, а для конструирования фрака или вечернего платья - уже гораздо более точная. При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Не кажется ли вам, что класс **Automobile** сильно перегружен? Действительно, в мире выпущены миллионы автомобилей разных марок и видов. Что между ними общего, кроме четырех колес? Да и колес может быть больше или меньше. Не лучше ли написать отдельные классы для легковых и грузовых автомобилей, для гоночных автомобилей и вездеходов? Как организовать все это множество классов? На этот вопрос объектно-ориентированное программирование отвечает так: надо организовать иерархию классов.

2.1.2. Иерархия

Иерархия объектов давно используете для их классификации. Особенно детально она проработана в биологии. Все знакомы с семействами, родами и видами. Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом:

```
class Pet{ // Здесь описываем общие свойства
          //всех домашних любимцев
Master person; // Хозяин животного
int weight, age, eatTime[]; // Вес, возраст, время кормления
int eat(int food, int drink, int time){ // Процесс кормления
     // Начальные действия...
     if(time == eatTime[i]) person.getFood(food, drink);
     // Метод потребления пищи
}
void voice(); // Звуки, издаваемые животным
```

```
// Прочее...
}
```

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat extends Pet{ // Описываются свойства,
     //присущие только кошкам:
int mouseCatched; // число пойманных мышей
void toMouse(); // процесс ловли мышей
// Прочие свойства
}
class Dog extends Pet{ // Свойства собак:
void preserve(); // охранять
}
```

Заметьте, что мы не повторяем общие свойства, описанные в классе **Pet**. Они наследуются автоматически. Мы можем определить объект класса **Dog** и использовать в нем все свойства класса **Pet** так, как будто они описаны в классе **Dog**:

```
Dog tuzik = new Dog(), sharik = new Dog();
```

После этого определения можно будет написать

```
tuzik.age = 3;
int p = sharik.eat(30, 10, 12);
```

А классификацию продолжить так:

```
class Pointer extends Dog{ ... } // Свойства породы Пойнтер
class Setter extends Dog{ ... } // Свойства сеттеров
```

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но, ни одно свойство не пропадает. Поэтому и употребляется слово **extends** - "расширяет" и говорят, что класс **Dog** - *расширение* (extension) класса **Pet**. С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних животных. Поэтому часто говорят, что класс **Dog** - *подкласс* (subclass) класса **Pe** , а класс **Pet** - *суперкласс* (superclass) или надкласс класса **Dog**.

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок, возникают племянники и внуки, вся беспокойная семейка вступает в отношения, достойные мексиканского сериала.

В этой терминологии говорят о *наследовании* классов, в нашем примере класс **Dog** наследует класс **Pet**.

Мы еще не определили счастливого владельца нашего домашнего зоопарка. Опишем его в классе **Master**. Делаем набросок:

```
class Master{ // Хозяин животного
String name; // Фамилия, имя
// Другие сведения
void getFood(int food, int drink); // Кормление
// Прочее
}
```

Хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и прочими. Для описания взаимодействия объектов применяется третий принцип объектно-ориентированного программирования - обязанность или ответственность

2.1.3. Ответственность

В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом **eat()**. В этом методе животное обращается к хозяину, умоляя его применить метод **getFood()**.

В англоязычной литературе подобное обращение описывается словом message. Это понятие неудачно переведено на русский язык ни к чему не обязывающим словом *"сообщение"*. Лучше было бы использовать слово "послание", "поручение" или даже "распоряжение". Но термин "сообщение" устоялся и нам придется его применять. Почему же не используется словосочетание "вызов метода", ведь говорят: "Вызов процедуры"? Потому что между этими понятиями есть, по крайней мере, три отличия.

- Сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект - текущее значение переменной **person**. У каждого объекта свое текущее состояние, свои значения полей класса, и это может повлиять на выполнение метода.
- Способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Один хозяин поставит миску с "Chappi", другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство называется *полиморфизмом* и будет обсуждаться ниже.
- Обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется *"поздним связыванием" в* противовес *"раннему связыванию",* при котором процедура присоединяется к программе на этапе компоновки.

Итак, объект **sharik**, выполняя свой метод **eat()**, посылает сообщение объекту, ссылка на который содержится в переменной **person**, с просьбой

выдать ему определенное количество еды и питья. Сообщение записано в строке **`person.getFood(food, drink)`**.

Этим сообщением заключается *контракт* между объектами, суть которого в том, что объект **`sharik`** берет на себя *ответственность* задать правильные параметры в сообщении, а объект - текущее значение **`person`** - возлагает на себя *ответственность* применить метод кормления **`getFood()`**, каким бы он ни был.

2.1.4. Модульность

Для того чтобы правильно реализовать принцип ответственности, применяется четвертый принцип объектно-ориентированного программирования - *модульность*.

Этот принцип утверждает - каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы.

В языке Java инкапсуляция достигается добавлением модификатора **`private`** к описанию члена класса. Например:

```
private int mouseCatched;
private String name;
private void preserve();
```

Эти члены классов становятся *закрытыми,* ими могут пользоваться только экземпляры того же самого класса, например, **`tuzik`** может дать поручение **`sharik.preserve()`**.

А если в классе **`Master`** мы напишем

```
private void getFood(int food, int drink);
```

то метод **`getFood()`** не будет найден, и несчастный **`sharik`** не сможет получить пищу.

В противоположность закрытости мы можем объявить некоторые члены класса *открытыми,* записав вместо слова **`private`** модификатор **`public`**, например:

```
public void getFood(int food, int drink);
```

К таким членам может обратиться любой объект любого класса.

В языке Java, в отличии от языка C++, словами **`private`**, **`public`** и **`protected`** отмечается каждый член класса в отдельности.

Принцип модульности предписывает открывать члены класса только в случае необходимости. Вспомните надпись: "Нормальное положение шлагбаума - закрытое".

Если же надо обратиться к полю класса, то рекомендуется включить в класс специальные *методы доступа*, отдельно для чтения этого поля и для записи в это поле. Имена методов доступа рекомендуется начинать со слов **`get`** и **`set`**, добавляя к этим словам имя поля.

В нашем примере класса **`Master`** методы доступа к полю **`Name`** в самом простом виде могут выглядеть так:

```
public String getName(){
     return name;
}
public void setName(String newName){
     name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в ***`set`**-методах,* меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные ***`is`**-методы,* возвращающие логическое значение **`true`** или **`false`** . Например, в класс **`Master`** можно включить метод, проверяющий, задано ли имя хозяина:

```
public boolean isEmpty(){
     return name == null;
}
```

и использовать этот метод для проверки при доступе к полю **`Name`**, например:

```
if(master01.isEmpty()) master01.setName("Иванов");
```

Итак, мы оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, было наименьшее *зацепление* между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную *связность*. Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил

достижения сильной функциональной связности, введенное Карлом Либерхером (Karl J. Lieberherr), получило название *закон Деметра.* Закон гласит: "В методе **m()** класса **A** следует использовать только методы класса **A**, методы классов, к которым принадлежат аргументы метода **m()**, и методы классов, экземпляры которых создаются внутри метода **m()**".

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовые выполнить любое поручение, на которое рассчитана их конструкция.

Будут ли закрытые члены класса доступны его наследникам? Если в классе **Pet** написано

```
private Master person;
```

то можно ли использовать **sharik.person**? Разумеется, нет. Ведь в противном случае каждый, интересующийся закрытыми полями класса **A**, может расширить его классом **B**, и просмотреть закрытые поля класса **A** через экземпляры класса **B**.

Когда надо разрешить доступ наследникам класса, но нежелательно открывать его всему миру, тогда в Java используется *защищенный* (protected) доступ, отмечаемый модификатором **protected** , например, объект **sharik** может обратиться к полю **person** родительского класса **Pet** , если в классе **Pet** это поле описано так:

```
protected Master person;
```

Следует сразу сказать, что на доступ к члену класса влияет еще и пакет, в котором находится класс, но об этом поговорим позже.

Из этого общего схематического описания принципов объектно-ориентированного программирования видно, что язык Java позволяет легко воплощать все эти принципы. Вы уже поняли, как записать класс, его поля и методы, как инкапсулировать члены класса, как сделать расширение класса и какими принципами следует при этом пользоваться. Разберем теперь подробнее правила записи классов и рассмотрим дополнительные их возможности.

2.2. Описание класса и подкласса

Описание класса начинается со слова **class**, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы.

Перед словом **class** можно записать модификаторы класса. Это одно из слов **public**, **abstract**, **final**, **strictfp** . Перед именем вложенного класса можно поставить, кроме того, модификаторы **protected**, **private**, **static** . Модификаторы мы будем вводить по мере изучения языка.

Тело класса, в котором в любом порядке перечисляются поля, методы, вложенные классы и интерфейсы, заключается в фигурные скобки.

При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением.

Описание поля может начинаться с одного или нескольких необязательных модификаторов **public**, **protected**, **private**, **static**, **final**, **transient**, **volatile**. Если надо поставить несколько модификаторов, то перечислять их JLS рекомендует в указанном порядке, поскольку некоторые компиляторы требуют определенного порядка записи модификаторов. С модификаторами мы будем знакомиться по мере необходимости.

При описании метода указывается тип возвращаемого им значения или слово **void**, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках расписывается выполняемый метод.

Описание метода может начинаться с модификаторов **public**, **protected**, **private**, **abstract**, **static**, **final**, **synchronized**, **native**, **strictfp**. Мы будем вводить их по необходимости.

В списке параметров через запятую перечисляются тип и имя каждого параметра. Перед типом какого-либо параметра может стоять модификатор **final**. Такой параметр нельзя изменять внутри метода. Список параметров может отсутствовать, но скобки сохраняются.

Перед началом работы метода для каждого параметра выделяется ячейка оперативной памяти, в которую копируется значение параметра, заданное при обращении к методу. Такой способ называется передачей параметров *по значению.*

В листинге 2.1 показано, как можно оформить метод деления пополам для нахождения корня нелинейного уравнения из листинга 1.5.

Листинг 2.1. Нахождение корня нелинейного уравнения методом бисекции

```
class Bisection2{
  private static double final EPS = 1e-8; // Константа
  private double a = 0.0, b = 1.5, root; // Закрытые поля
  public double getRoot(){return root;} // Метод доступа
  private double f(double x) {
```

```
      return x*x*x – 3*x*x + 3; // Или что-то другое
   }
   private void bisect(){// Параметров нет –
                         // метод работает с полями экземпляра
      double y = 0.0; // Локальная переменная – не поле
      do{
         root = 0.5 *(a + b); y = f(root);
         if (Math.abs(y) < EPS) break;
         // Корень найден. Выходим из цикла
         // Если на концах отрезка [a; root]
         // функция имеет разные знаки:
         if(f(a) * y < 0.0} b = root;
            // значит, корень здесь
            // Переносим точку b в точку root
            //В противном случае:
         else a = root;
            // переносим точку a в точку root
            // Продолжаем, пока [a; b] не станет мал
      }while(Math.abs(b-a) >= EPS);
   }
   public static void main(String[] args){
      Bisection2 b2 = new Bisection2();
      b2.bisect();
      System.out.println("x = " +
         b2.getRoot() + // Обращаемся к корню через метод доступа
         ", f() = " +b2.f(b2.getRoot()));
   }
}
```

В описании метода **f()** сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода **bisect()** выполнено в духе ООП: метод активен, он сам обращается к полям экземпляра **b2** и сам заносит результат в нужное поле. Метод **bisect()** - это внутренний механизм класса **Bisection2**, поэтому он закрыт (**private**).

2.2.1. Перегрузка методов

Имя метода, число и типы параметров образуют *сигнатуру* метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, различающиеся числом и/или типами параметров.

Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы.

Например, в классе **Automobile** мы записали метод **moveTo(int x, int y)**, обозначив пункт назначения его географическими координатами. Можно определить еще один метод **moveTo(String destination)** для указания географического названия пункта назначения и обратиться к нему так:

```
oka.moveTo("Москва");
```

Такое дублирование методов называется *перегрузкой*. Перегрузка методов очень удобна в использовании. Вспомните, в главе 1 мы выводили данные любого типа на экран методом **println()** не заботясь о том, данные какого именно типа мы выводим. На самом деле мы использовали разные методы с одним и тем же именем **println**, даже не задумываясь об этом. Конечно, все эти методы надо тщательно спланировать и заранее описать в классе. Это и сделано в классе **PrintStream**, где представлено около двадцати методов **print()** и **println()**.

2.2.2. Переопределение методов

Если же записать метод с тем же именем в подклассе, например:

```
class Truck extends Automobile{
     void moveTo(int x, int y){
          // Какие-то действия
     }
     // Что-то еще
}
```

то он перекроет метод суперкласса. Определив экземпляр класса **Truck**, например:

```
Truck gazel = new Truck();
```

и записав **gazel.moveTo(25, 150)**, мы обратимся к методу класса **Truck** . Произойдет *переопределение* метода.

При переопределении права доступа к методу можно только расширить. Открытый метод **public** должен остаться открытым, защищенный **protected** может стать открытым.

Можно ли внутри подкласса обратиться к методу суперкласса? Да, можно, если уточнить имя метода, словом **super** , например, **super.moveTo(30, 40)** . Можно уточнить и имя метода, записанного в этом же классе, словом **this** , например, **this.moveTo(50, 70)** , но в данном случае это уже излишне. Таким же образом можно уточнять и совпадающие имена полей, а не только методов.

Данные уточнения подобны тому, как мы говорим про себя "я", а не "Иван Петрович", и говорим "отец", а не "Петр Сидорович".

2.2.3. Реализация полиморфизма в Java

Переопределение методов приводит к интересным результатам. В классе **Pet** мы описали метод **voice()**. Переопределим его в подклассах и используем в классе chorus , как показано в листинге 2.2.

Листинг 2.2. Пример полиморфного метода

```
abstract class Pet{
abstract void voice();
}
class Dog extends Pet{
int k = 10;
void voice(){
System.out.printin("Gav-gav!");
}
}
class Cat extends Pet{
void voice () {
System.out.printin("Miaou!");
}
}
class Cow extends Pet{
void voice(){
System.out.printin("Mu-u-u!");
}
}
public class Chorus(
public static void main(String[] args){
Pet[] singer = new Pet[3];
singer[0] = new Dog();
singer[1] = new Cat();
singer[2] = new Cow();
for (int i = 0; i < singer.length; i++)
singer[i].voice();
}
}
```

Все дело здесь в определении поля **singer[]**. Хотя массив ссылок **singer[]** имеет тип **Pet**, каждый его элемент ссылается на объект своего типа **Dog**, **Cat**, **Cow**. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.

В языке Java все методы являются виртуальными функциями.

Вы наверное заметили в описании класса **Pet** новое слово **abstract**. Класс **Pet** и метод **voice()** являются абстрактными.

2.2.4. Абстрактные методы и классы

При описании класса **`Pet`** мы не можем задать в методе **`voice()`** никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса.

В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет *абстрактным*, что необходимо указать компилятору модификатором **`abstract`**.

Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится *абстрактным,* что обязательно надо указать модификатором **`abstract`**.

Как же использовать абстрактные классы? Только порождая от них подклассы, в которых переопределены абстрактные методы.

Зачем же нужны абстрактные классы? Не лучше ли сразу написать нужные классы с полностью определенными методами, а не наследовать их от абстрактного класса? Для ответа снова обратимся к листингу 2.2.

Хотя элементы массива **`singer[]`** ссылаются на подклассы **`Dog`**, **`Cat`**, **`Cow`** , но все-таки это переменные типа **`Pet`** и ссылаться они могут только на поля и методы, описанные в суперклассе **`Pet`**. Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к полю **`k`** класса **`Dog`**, написав **`singer[0].k`**. Компилятор "скажет", что он не может реализовать такую ссылку. Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявить абстрактным. Таким образом, абстрактные классы группируются на вершине иерархии классов.

Кстати, можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например:

```
void voice(){}
```

Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Замкнуть же иерархию можно окончательными классами.

2.2.5. Окончательные члены и классы

Пометив метод модификатором **`final`**, можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так

определены математические функции **sin()**, **cos()** и прочие в классе **Math**. Мы уверены, что метод **Math.cos(x)** вычисляет именно косинус числа **x**. Разумеется, такой метод не может быть абстрактным.

Для полной безопасности, поля, обрабатываемые окончательными методами, следует сделать закрытыми (**private**).

Если же пометить модификатором **final** весь класс, то его вообще нельзя будет расширить. Так определен, например, класс **Math**:

```
public final class Math{ ... }
```

Для переменных модификатор **final** имеет совершенно другой смысл. Если пометить модификатором **final** описание переменной, то ее значение (а оно должно быть обязательно задано или здесь же, или в блоке инициализации или в конструкторе) нельзя изменить ни в подклассах, ни в самом классе. Переменная превращается в константу. Именно так в языке Java определяются константы:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

По соглашению "Code Conventions" константы записываются прописными буквами, слова в них разделяются знаком подчеркивания.

На самой вершине иерархии классов Java стоит класс **Object**.

2.2.6. Класс Object

Если при описании класса мы не указываем никакое расширение, т. е. не пишем слово **extends** и имя класса за ним, как при описании класса **Pet**, то Java считает этот класс расширением класса **Object**, и компилятор дописывает это за нас:

```
class Pet extends Object{ ... }
```

Можно записать это расширение и явно.

Сам же класс **Object** не является ничьим наследником, от него начинается иерархия любых классов Java. В частности, все массивы - прямые наследники класса **Object**.

Поскольку такой класс может содержать только общие свойства всех классов, в него включено лишь несколько самых общих методов, например, метод **equals()**, сравнивающий данный объект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Его можно использовать так:

```
Object obj1 = new Dog(), obj 2 = new Cat();
if(obj1.equals(obj2)) ...
```

Оцените объектно-ориентированный дух этой записи: объект **obj1** активен, он сам сравнивает себя с другим объектом. Можно, конечно, записать и **obj2.equals(obj1)**, сделав активным объект **obj2**, с тем же результатом.

Ссылки можно сравнивать на равенство и неравенство:

```
obj1 == obj2; obj1 != obj 2;
```

В этом случае сопоставляются адреса объектов, мы можем узнать, не указывают ли обе ссылки на один и тот же объект.

Метод **equals()** же сравнивает содержимое объектов в их текущем состоянии, фактически он реализован в классе **Object** как тождество: объект равен только самому себе. Поэтому его часто переопределяют в подклассах, более того, правильно спроектированные, "хорошо воспитанные", классы должны переопределить методы класса **Object**, если их не устраивает стандартная реализация.

Второй метод класса **Object**, который следует переопределять в подклассах, - метод **toString()**. Это метод без параметров, который пытается содержимое объекта преобразовать в строку символов и возвращает объект класса **String**.

К этому методу исполняющая система Java обращается каждый раз, когда требуется представить объект в виде строки, например, в методе **println()**.

2.2.7. Конструкторы класса

Вы уже обратили внимание на то, что в операции **new**, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса?

Такой "метод" называется *конструктором класса*. Его своеобразие заключается не только в имени. Перечислим особенности конструктора.

- Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст *конструктор по умолчанию*, который, впрочем, пуст, он не делает ничего, кроме вызова конструктора суперкласса.
- Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и обнуления полей, но до начала использования создаваемого объекта.

- Конструктор не возвращает никакого значения. Поэтому в его описании не пишется даже слово **void**, но можно задать один из трех модификаторов **public**, **protected** или **private**.
- Конструктор не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе.
- Тело конструктора может начинаться:

    - с вызова одного из конструкторов суперкласса, для этого записывается слово **super()** с параметрами в скобках, если они нужны;
    - с вызова другого конструктора того же класса, для этого записывается слово **this()** с параметрами в скобках, если они нужны.

    Если же **super()** в начале конструктора не указан, то вначале выполняется конструктор суперкласса без аргументов, затем происходит инициализация полей значениями, указанными при их объявлении, а уж потом то, что записано в конструкторе.

Во всем остальном конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор **return**, но только пустой, без всякого возвращаемого значения.

В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

В наших примерах мы ни разу не рассматривали конструкторы классов, поэтому при создании экземпляров наших классов вызывался конструктор класса **Object**.

2.2.8. Операция **new**

Пора подробнее описать операцию с одним операндом, обозначаемую словом **new**. Она применяется для выделения памяти массивам и объектам.

В первом случае в качестве операнда указывается тип элементов массива и количество его элементов в квадратных скобках, например:

```
double a[] = new double[100];
```

Во втором случае операндом служит конструктор класса. Если конструктора в классе нет, то вызывается конструктор по умолчанию.

Числовые поля класса получают нулевые значения, логические поля - значение **false**, ссылки - значение **null**.

Результатом операции **new** будет ссылка на созданный объект. Эта ссылка может быть присвоена переменной типа ссылка на данный тип:

```
Dog k9 = new Dog();
```

но может использоваться и непосредственно

```
new Dog().voice();
```

Здесь после создания безымянного объекта сразу выполняется его метод **voice()**. Такая странная запись встречается в программах, написанных на Java, на каждом шагу.

2.2.9. Статические члены класса

Разные экземпляры одного класса имеют совершенно независимые друг от друга поля, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти. Поэтому такие поля называются переменными *экземпляра класса* или переменными *объекта.*

Иногда надо определить поле, общее для всего класса, изменение которого в одном экземпляре повлечет изменение того же поля во всех экземплярах. Например, мы хотим в классе **Automobile** отмечать порядковый заводской номер автомобиля. Такие поля называются *переменными класса*. Для переменных класса выделяется только одна ячейка памяти, общая для всех экземпляров. Переменные класса образуются в Java модификатором **static**. В листинге 2.3 мы записываем этот модификатор при определении переменной **number**.

Листинг 2.3. Статическая переменная

```
class Automobile {
private static int number;
Automobile(){
number++;
System.out.println("From Automobile constructor:"+
" number = "+number);
}
}
public class AutomobiieTest{
public static void main(String[] args){
Automobile lada2105 = new Automobile(),
fordScorpio = new Automobile(),
oka = new Automobile!);
}
}
```

Интересно, что к статическим переменным можно обращаться с именем класса, **`Automobile.number`**, а не только с именем экземпляра, **`lada2105.number`**, причем это можно делать, даже если не создан ни один экземпляр класса.

Для работы с такими *статическими переменными* обычно создаются *статические методы,* помеченные модификатором **`static`**. Для методов слово **`static`** имеет совсем другой смысл. Исполняющая система Java всегда создает в памяти только одну копию машинного кода метода, разделяемую всеми экземплярами, независимо от того, статический это метод или нет.

Основная особенность статических методов - они выполняются сразу во всех экземплярах класса. Более того, они могут выполняться, даже если не создан ни один экземпляр класса. Достаточно уточнить имя метода именем класса (а не именем объекта), чтобы метод мог работать. Именно так мы пользовались методами класса **`Math`**, не создавая его экземпляры, а просто записывая **`Math.abs(x)`**, **`Math.sqrt(x)`**. Точно так же мы использовали метод **`System.out.println()`**. Да и методом **`main()`** мы пользуемся, вообще не создавая никаких объектов.

Поэтому статические методы называются *методами класса*, в отличие от нестатических методов, называемых *методами экземпляра*.

Отсюда вытекают другие особенности статических методов:

- в статическом методе нельзя использовать ссылки **`this`** и **`super`**;
- в статическом методе нельзя прямо, не создавая экземпляров, ссылаться на нестатические поля и методы;
- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические.

Именно поэтому в листинге 1.5 мы пометили метод **`f()`** модификатором **`static`**. Но в листинге 2.1 мы работали с экземпляром **`b2`** класса **`Bisection2`**, и нам не потребовалось объявлять метод **`f()`** статическим.

Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом **`static`**, который тоже будет выполнен до запуска конструктора:

```
static int[] a = new a[10];
static {
for(int k = 0; k < a.length; k++)
```

```
a[k] = k * k;
}
```

Операторы, заключенные в такой блок, выполняются только один раз, при первой загрузке класса, а не при создании каждого экземпляра.

2.2.10. Класс `Complex`

Комплексные числа широко используются не только в математике. Они часто применяются в графических преобразованиях, в построении фракталов, не говоря уже о физике и технических дисциплинах. Но класс, описывающий комплексные числа, почему-то не включен в стандартную библиотеку Java. Восполним этот пробел.

Листинг 2.4 длинный, но просмотрите его внимательно, при обучении языку программирования очень полезно чтение программ на этом языке.

Листинг 2.4. Класс Complex

```
class Complex {
private static final double EPS = le-12; // Точность вычислений
private double re, im; // Действительная и мнимая часть
// Четыре конструктора
Complex(double re, double im) {
this, re = re; this.im = im;
}
Complex(double re){this(re, 0.0); }
Complex(){this(0.0, 0.0); }
Complex(Complex z){this(z.re, z.im) ; }
// Методы доступа
public double getRe(){return re;}
public double getlmf){return im;}
public Complex getZ(){return new Complex(re, im);}
public void setRe(double re){this.re = re;}
public void setlm(double im){this.im = im;}
public void setZ(Complex z){re = z.re; im = z.im;}
// Модуль и аргумент комплексного числа
public double mod(){return Math.sqrt(re * re + im * im);}
public double arg()(return Math.atan2(re, im);}
// Проверка: действительное число?
public boolean isReal(){return Math.abs(im) < EPS;}
public void pr(){ // Вывод на экран
System.out.println(re + (im < 0.0 ? "" : '"+") + im + "i");
}
// Переопределение методов класса Object
public boolean equals(Complex z){
return Math.abs(re -'z.re) < EPS &&
Math.abs(im - z.im) < EPS;
}
public String toString(){
return "Complex: " + re + " " + im;
```

```
}
// Методы, реализующие операции +=, -=, *=, /=
public void add(Complex z){re += z.re; im += z.im;}
public void sub(Complex z){re -= z.re; im —= z.im;}
public void mul(Complex z){
double t = re * z.re — im * z. im;
im = re * z.im + im * z.re;
re = t;
}
public void div(Complex z){
double m = z.mod();
double t = re * z.re — im * z.im;
im = (im * z.re — re * z.im) / m;
re = t / m;
}
// Методы, реализующие операции +, -, *, /
public Complex plus(Complex z){
return new Complex(re + z.re, im + z im);
}
public Complex minus(Complex z){
return new Complex(re - z.re, im - z.im);
}
public Complex asterisk(Complex z){
return new Complex(
re * z.re - im * z.im, re * z.im + im * z re);
}
public Complex slash(Complex z){
double m = z.mod();
return new Complex(
(re * z.re - im * z.im) / m, (im * z.re - re * z.im) / m);
}
}
// Проверим работу класса Complex
public class ComplexTest{
public static void main(Stringf] args){
Complex zl = new Complex(),
z2 = new Complex(1.5),
z3 = new Complex(3.6, -2.2),
z4 = new Complex(z3);
System.out.printlnf); // Оставляем пустую строку
System.out.print("zl = "); zl.pr();
System.out.print("z2 = "); z2.pr();
System.out.print("z3 = "); z3.pr();
System.out.print ("z4 = "}; z4.pr();
System.out.println(z4); // Работает метод toString()
z2.add(z3);
System.out.print("z2 + z3 = "}; z2.pr();
z2.div(z3);
System.out.print("z2 / z3 = "); z2.pr();
z2 = z2.plus(z2);
System.out.print("z2 + z2 = "); z2.pr();
z3 = z2.slash(zl);
System.out.print("z2 / zl = "); z3.pr();
```

```
}
}
```

2.2.11. Метод `main()`

Всякая программа, оформленная как *приложение*, должна содержать метод с именем **`main`**. Он может быть один на все приложение или содержаться в некоторых классах этого приложения, а может находиться и в каждом классе.

Метод **`main()`** записывается как обычный метод, может содержать любые описания и действия, но он обязательно должен быть открытым (**`public`**), статическим (**`static`**), не иметь возвращаемого значения (**`void`**). Его аргументом обязательно должен быть массив строк (**`String[]`**). По традиции этот массив называют **`args`**, хотя имя может быть любым.

Эти особенности возникают из-за того, что метод **`main()`** вызывается автоматически исполняющей системой Java в самом начале выполнения приложения. При вызове интерпретатора **`java`** указывается класс, где записан метод **`main()`**, с которого надо начать выполнение. Поскольку классов с методом **`main()`** может быть несколько, можно построить приложение с дополнительными точками входа, начиная выполнение приложения в разных ситуациях из различных классов.

Часто метод **`main()`** заносят в каждый класс с целью отладки. В этом случае в метод **`main()`** включают тесты для проверки работы всех методов класса.

При вызове интерпретатора **`java`** можно передать в метод **`main()`** несколько параметров, которые интерпретатор заносит в массив строк. Эти параметры перечисляются в строке вызова **`java`** через пробел сразу после имени класса. Если же параметр содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в параметр, это только ограничители.

Все это легко понять на примере листинга 2.5, в котором записана программа, просто выводящая параметры, передаваемые в метод **`main()`** при запуске.

Листинг 2.5. Передача параметров в метод main()

```
class Echo {
     public static void main(String[] args){
          for (int i = 0; i < args.length; i++)
               System.out.println("args[" + i +"]="+ args[i]);
     }
}
```

Как видно, имя класса не входит в число параметров. Оно и так известно в методе **`main()`**.

Поскольку в Java, в отличии от C++, имя файла всегда совпадает с именем класса, содержащего метод **`main()`**, оно не заносится в **`args[0]`**. Вместо **`argc`** используется **`args.length`**. Доступ к переменным среды разрешен не всегда и осуществляется другим способом. Некоторые значения можно просмотреть так:

```
System.getProperties().list(System.out);
```

2.2.12. Область видимости переменных

В языке Java нестатические переменные можно объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не могут объявляться внутри методов и блоков. Какова же *область видимости* переменных? Из каких методов мы можем обратиться к той или иной переменной? В каких операторах использовать? Рассмотрим на примере листинга 2.6 разные случаи объявления переменных.

Листинг 2.6. Видимость и инициализация переменных

```
class ManyVariables{
static int x = 9, y; // Статические переменные — поля класса
// Они известны во всех методах и блоках класса
// Переменная y получает значение 0
static{ // Блок инициализации статических переменных
// Выполняется один раз при первой загрузке класса после
// инициализаций в объявлениях переменных
x = 99; // Оператор выполняется вне всякого метода!
}
int a = 1, p; // Нестатические переменные — поля экземпляра
// Известны во всех методах и блоках класса, в которых они
//не перекрыты другими переменными с тем же именем
// Переменная p получает значение 0
{ // Блок инициализации экземпляра
// Выполняется при создании, каждого экземпляра после
// инициализаций при объявлениях переменных
p = 999; // Оператор выполняется вне всякого метода!
}
static void f(int b){ // Параметр метода b — локальная
// переменная, известна только внутри метода
int a = 2; // Это вторая переменная с тем же именем "a"
// Она известна только внутри метода f() и
// здесь перекрывает первую "a"
int c; // Локальная переменная, известна только в методе f()
//Не получает никакого начального значения
//и должна быть определена перед применением
{ int c = 555; // Ошибка! Попытка повторного объявления
int x = 333; // Локальная переменная, известна только в этом
блоке
}
// Здесь переменная x уже неизвестна
```

```
for(int d = 0; d < 10; d++){
// Переменная цикла d известна только в цикле
int a = 4; // Ошибка!
int e = 5; // Локальная переменная, известна только в цикле for
e++; // Инициализируется при каждом выполнении цикла
System.out.println("e = " + e) ; // Выводится всегда "e = 6"
}
// Здесь переменные d и e неизвестны
}
public static void main(String!] args){
int a = 9999; // Локальная переменная, известна
// только внутри метода main()
f(a);
}
}
```

Обратите внимание на то, что переменным класса и экземпляра неявно присваиваются нулевые значения. Символы неявно получают значение `'\u0000'`, логические переменные - значение **`false`** , ссылки получают неявно значение **`null`**.

Локальные же переменные неявно не инициализируются. Им должны либо явно присваиваться значения, либо они обязаны определяться до первого использования. К счастью, компилятор замечает неопределенные локальные переменные и сообщает о них.

Поля класса при объявлении обнуляются, локальные переменные автоматически не инициализируются.

В листинге 2.6 появилась еще одна новая конструкция: *блок инициализации экземпляра*. Это просто блок операторов в фигурных скобках, но записывается он вне всякого метода, прямо в теле класса. Этот блок выполняется при создании каждого экземпляра, после инициализации при объявлении переменных, но до выполнения конструктора. Он играет такую же роль, как и **`static`**-блок для статических переменных. Зачем же он нужен, ведь все его содержимое можно написать в начале конструктора? В тех случаях, когда конструктор написать нельзя, а именно, в безымянных внутренних классах.

2.2.13. Вложенные классы

В этой главе уже несколько раз упоминалось, что в теле класса можно сделать описание другого, *вложенного* класса. А во вложенном классе можно снова описать вложенный, *внутренний* класс и т. д. Эта матрешка кажется вполне естественной, однако возникает масса вопросов.

- Можем ли мы из вложенного класса обратиться к членам внешнего класса? Можем, для того это все и задумывалось.

- А можем ли мы в таком случае определить экземпляр вложенного класса, не определяя экземпляры внешнего класса? Нет, не можем, сначала надо определить хоть один экземпляр внешнего класса, матрешка ведь!
- А если экземпляров внешнего класса несколько, как узнать, с каким экземпляром внешнего класса работает данный экземпляр вложенного класса? Имя экземпляра вложенного класса уточняется именем связанного с ним экземпляра внешнего класса. Более того, при создании вложенного экземпляра операция new тоже уточняется именем внешнего экземпляра.

Все вложенные классы можно разделить на вложенные *классы-члены* класса, описанные вне методов, и вложенные *локальные классы*, описанные внутри методов и/или блоков. Локальные классы, как и все локальные переменные, не являются членами класса.

Классы-члены могут быть объявлены статическим модификатором **`static`**. Поведение статических классов-членов ничем не отличается от поведения обычных классов, отличается только обращение к таким классам. Поэтому они называются *вложенными классами верхнего уровня*, хотя статические классы-члены можно вкладывать друг в друга. В них можно объявлять статические члены. Используются они обычно для того, чтобы сгруппировать вспомогательные классы вместе с основным классом.

Все нестатические вложенные классы называются *внутренними*. В них нельзя объявлять статические члены.

Локальные классы, как и все локальные переменные, известны только в блоке, в котором они определены. Они могут быть *безымянными*.

В листинге 2.7 рассмотрены все эти случаи.

Листинг 2.7. Вложенные классы

```
class Nested{
static private int pr; // Переменная pr объявлена статической
// чтобы к ней был доступ из статических классов A и AB
String s = "Member of Nested";
// Вкладываем статический класс.
static class .A{ // Полное имя этого класса — Nested.A
private int a=pr;
String s = "Member of A";
// Во вложенньм класс A вкладываем еще один статический класс
static class AB{ // Полное имя класса — Nested.A.AB
private int ab=pr;
String s = "Member of AB";
}
}
```

```
//В класс Nested вкладываем нестатический класс
class B{ // Полное имя этого класса — Nested.B
private int b=pr;
String s = "Member of B";
// В класс B вкладываем еще один класс
class BC{ // Полное имя класса — Nested.B.BC
private int bc=pr;
String s = "Member of BC";
}
void f(final int i){ // Без слова final переменные i и j
final int j = 99; // нельзя использовать в локальном классе D
class D{ // Локальный класс D известен только внутри f()
private int d=pr;
String s = "Member of D";
void pr(){
// Обратите внимание на то, как различаются
// переменные с одним и тем же именем "s"
System.out.println(s + (i+j)); // "s" эквивалентно "this.s"
System.out.println(B.this.s);
System.out.println(Nested.this.s);
// System.out.println(AB.this.s); // Нет доступа
// System.out.println(A.this.s); // Нет доступа
}
}
D d = new D(); // Объект определяется тут же, в методе f()
d.pr(); // Объект известен только в методе f()
}
}
void m(){
new Object(){ // Создается объект безымянного класса,
// указывается конструктор его суперкласса
private int e = pr;
void g(){
System.out.println("From g()) ;
}
}.g(); // Тут же выполняется метод только что созданного объекта
}
}
public class NestedClasses{
public static void main(String[] args){
Nested nest = new Nested(); // Последовательно раскрываются
// три матрешки
Nested.A theA = nest.new A(); // Полное имя класса и уточненная
// операция new. Но конструктор только вложенного класса
Nested.A.AB theAB = theA.new AB(); // Те же правила. Операция
// new уточняется только одним именем
Nested.B theB = nest.new B(); // Еще одна матрешка
Nested.B.BC theBC = theB.new BC();
theB.f(999); // Методы вызываются обычным образом
nest.m();
}
}
```

Дадим пояснения.

- Как видно, доступ к полям внешнего класса **`Nested`** возможен отовсюду, даже к закрытому полю **`pr`**. Именно для этого в Java и введены вложенные классы. Остальные конструкции введены вынужденно, для того чтобы увязать концы с концами.
- Язык Java позволяет использовать одни и те же имена в разных областях видимости - пришлось уточнять константу **`this`** именем класса: **`Nested.this`**, **`B.this`**.
- В безымянном классе не может быть конструктора, ведь имя конструктора должно совпадать с именем класса, - пришлось использовать имя суперкласса, в примере это класс **`Object`**. Вместо конструктора в безымянном классе используется блок инициализации экземпляра.
- Нельзя создать экземпляр вложенного класса, не создав предварительно экземпляр внешнего класса, - пришлось подстраховать это правило уточнением операции **`new`** именем экземпляра внешнего класса - **`nest.new`**, **`theA.new`**, **`theB.new`**.
- При определении экземпляра указывается полное имя вложенного класса, но в операции **`new`** записывается просто конструктор класса.

Введение вложенных классов сильно усложнило синтаксис и поставило много задач разработчикам языка.

Механизм вложенных классов станет понятнее, если посмотреть, какие файлы с байт-кодами создал компилятор:

- Nested$l$D.class — локальный класс **`D`**, вложенный в класс **`Nested`**;
- NestedSl.class — безымянный класс;
- Nested$A$AB.class — класс **`Nested.A.AB`**;
- Nested$A.class — класс **`Nested.A`**;
- Nested$B$BC.class — класс **`Nested.B.BC`**;
- NestedSB.class — класс **`Nested.B`**;
- Nested.class — внешний класс **`Nested`**;
- NestedClasses.class - класс с методом **`main()`**.

Компилятор разложил матрешки и, как всегда, создал отдельные файлы для каждого класса. При этом, поскольку в идентификаторах недопустимы точки, компилятор заменил их знаками доллара. Для безымянного класса компилятор придумал имя. Локальный класс компилятор пометил номером.

Оказывается, вложенные классы существуют только на уровне исходного кода. Виртуальная машина Java ничего не знает о вложенных классах. Она работает с обычными внешними классами. Для взаимодействия объектов вложенных классов компилятор вставляет в них специальные закрытые поля.

Поэтому в локальных классах можно использовать только константы объемлющего метода, т.е. переменные, помеченные словом **`final`**. Виртуальная машина просто не догадается передавать изменяющиеся значения переменных в локальный класс. Таким образом, не имеет смысла помечать вложенные классы **`private`**, все равно они выходят на самый внешний уровень.

Все эти вопросы можно не брать в голову. Вложенные классы - это прямое нарушение принципа KISS, и в Java используются только в самом простом виде, главным образом, при обработке событий, возникающих при действиях с мышью и клавиатурой.

В каких же случаях создавать вложенные классы? В теории ООП вопрос о создании вложенных классов решается при рассмотрении отношений "быть частью" и "являться".

3. Пакеты и интерфейсы

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих. Множество классов становится необозримым. Уже давно принять классы объединять в библиотеки. Но библиотеки классов, кроме стандартной, не являются частью языка.

Разработчики Java включили в язык дополнительную конструкцию - *пакеты*. Все классы Java распределяются по пакетам. Кроме классов пакеты могут включать в себя интерфейсы и вложенные *подпакеты*. Образуется древовидная структура пакетов и подпакетов.

Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением **`.class`** (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы. Подпакеты собраны в подкаталоги этого каталога.

Каждый пакет образует одно *пространство имен*. Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: ***`пакет.класс`***. Такое уточненное имя называется *полным именем класса*.

Все эти правила, опять-таки, совпадают с правилами хранения файлов и подкаталогов в каталогах.

Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса **private**, **protected** и **public** еще один, "пакетный" уровень доступа.

Если член класса не отмечен ни одним из модификаторов **private**, **protected**, **public**, то, по умолчанию, к нему осуществляется *пакетный доступ*, а именно, к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком - если класс не помечен модификатором **public**, то все его члены, даже открытые, **public**, не будут видны из других пакетов.

Как же создать пакет и разместить в нем классы и подпакеты?

3.1. Пакет и подпакет

Чтобы создать пакет надо просто в первой строке **java**-файла с исходным кодом записать строку **package *имя*;**, например:

```
package mypack;
```

Тем самым создается пакет с указанным именем **mypack** и все классы, записанные в этом файле, попадут в пакет **mypack**. Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы.

Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, **subpack**, следует в первой строке исходного файла написать;

```
package mypack.subpack;
```

и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет **subpack** пакета **mypack**.

Можно создать и подпакет подпакета, написав что-нибудь вроде

```
package mypack.subpack.sub;
```

и т.д. сколько угодно раз.

Поскольку строка **package *имя*;** только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет.

Компилятор Java может сам создать каталог с тем же именем **mypack**, а в нем подкаталог **subpack**, и разместить в них **class**-файлы с байт-кодами.

Полные имена классов **A**, в будут выглядеть так: **mypack.A**, **mypack.subpack.B**.

Фирма SUN рекомендует записывать имена пакетов строчными буквами, тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной. Кроме того, фирма SUN советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например:

```
com.sun.developer
```

До сих пор мы ни разу не создавали пакет. Куда же попадали наши файлы с откомпилированными классами?

Компилятор всегда создает для таких классов *безымянный пакет*, которому соответствует текущий каталог файловой системы. Вот поэтому у нас **class**-файл всегда оказывался в том же каталоге, что и соответствующий **java**-файл.

Безымянный пакет служит обычно хранилищем небольших пробных или промежуточных классов. Большие проекты лучше хранить в пакетах. Например, библиотека классов Java API хранится в пакетах **java**, **javax**, **org**. Пакет **java** содержит только подпакеты **applet**, **awt**, **beans**, **io**, **lang**, **math**, **net**, **nio**, **rmi**, **security**, **sql**, **text**, **util** и ни одного класса. Эти пакеты имеют свои подпакеты, например, пакет создания GUI и графики **java.awt** содержит подпакеты **color**, **datatransfer**, **dnd**, **event**, **font**, **geom**, **im**, **image**, **print**.

Конечно, состав пакетов меняется от версии к версии.

3.2. Права доступа к членам класса

Пришло время подробно разобрать различные ограничения доступа к полям и методам класса.

Рассмотрим большой пример. Пусть имеется пять классов, размещенных в двух пакетах. В файле **Base.java** описаны три класса: **Inp1**, **Base** и класс **Derivedp1**, расширяющий класс **Base**. Эти классы размещены в пакете **p1**. В классе **Base** определены переменные всех четырех типов доступа, а в методах **f()** классов **Inp1** и **Derivedp1** сделана попытка доступа ко всем полям класса **Base**. Неудачные попытки отмечены комментариями. В комментариях помещены сообщения компилятора. Листинг 3.1 показывает содержимое этого файла.

Листинг 3.1. Файл **Base.java** с описанием пакета **p1**

```
package p1;
class Inp1{
public void f() {
```

```
Base b = new Base();
// b.priv = 1; // "priv has private access in p1.Base"
b.pack = 1;
b.prot = 1;
b.publ = 1;
}
}
public class Base{
private int priv = 0;
int pack = 0;
protected int prot = 0;
public int pub1 = 0;
}
class Derivedp1 extends Base{
public void f(Base a) {
// a.priv = 1; // "priv hds private access in pi.Base"
a.pack = 1;
a.prot = 1;
a.publ = 1;
// priv = 1; // "priv has private access in pi.Base"
pack = 1;
prot = 1;
publ = 1;
}
}
```

Как видно из листинга 3.1, в пакете недоступны только закрытые, **private**, поля другого класса.

В файле **Inp2.java** описаны два класса: **Inp2** и класс **Derivedp2**, расширяющий класс **Base**. Эти классы находятся в другом пакете **p2**. В этих классах тоже сделана попытка обращения к полям класса **Base**. Неудачные попытки прокомментированы сообщениями компилятора. Листинг 3.2 показывает содержимое этого файла.

Напомним, что класс **Base** должен быть помечен при своем описании в пакете **p1** модификатором **public**, иначе из пакета **p2** не будет видно ни одного его члена.

Листинг 3.2. Файл **Inp2.java** с описанием пакета **p2**

```
package p2;
import pl.Base;
class Inp2{
public static void main(String[] args){
Base b = new Base();
// b.priv = 1; // "priv has private access in pl.Base"
// b.pack = 1; // "pack is not public in pl.Base;
// cannot be accessed from outside package"
// b.prot = 1; //„"prot has protected access in pi.Base"
```

```
b.publ = 1;
}
}
class Derivedp2 extends Base{
public void f(Base a){
// a.priv = 1; // "priv has private access in. p1.Base"
// a.pack = 1; // "pack, is not public in pi.Base; cannot
//be accessed from outside package"
// a.prot = 1; // "prot has protected access in p1.Base"
a.publ = 1;
// priv = 1; // "priv has private access in pi.Base"
// pack = 1; // "pack is not public in pi.Base; cannot
// be accessed from outside package"
prot = 1;
publ = 1;
super.prot = 1;
}
}
```

Здесь, в другом пакете, доступ ограничен в большей степени.

Из независимого класса можно обратиться только к открытым, **public**, полям класса другого пакета. Из подкласса можно обратиться еще и к защищенным, **protected**, полям, но только унаследованным непосредственно, а не через экземпляр суперкласса.

Все указанное относится не только к полям, но и к методам. Подытожим все сказанное в следующей таблице

	Класс	Пакет	Пакет и подклассы	Все классы
private	+			
"package"	+	+		
protected	+	+	*	
public	+	+	+	+

Особенность доступа к **protected** -полям и методам из чужого пакета отмечена звездочкой.

3.3. Импорт классов и пакетов

Наверно вы заметили во второй строке листинга 3.2 новый оператор **import**. Для чего он нужен?

Дело в том, что компилятор будет искать классы только в двух пакетах: в том, что указан в первой строке файла, и в пакете стандартных классов **java.lang**. Для классов из другого пакета надо указывать полные имена. В нашем примере они короткие, и мы могли бы писать в листинге 3.2 вместо **Base** полное имя **p1.Base**.

Но если полные имена длинные, а используются классы часто, то стучать по клавишам, набирая полные имена, становится утомительно. Вот тут-то мы и пишем операторы **import**, указывая компилятору полные имена классов.

Правила использования оператора **import** очень просты: пишется слово **import** и, через пробел, полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько операторов **import** и пишется.

Это тоже может стать утомительным и тогда используется вторая форма оператора **import** - указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка *****. Этой записью компилятору предписывается просмотреть весь пакет. В нашем примере можно было написать

```
import p1.*;
```

Напомним, что импортировать можно только открытые классы, помеченные модификатором **public**.

Мы пользовались методами классов стандартной библиотеки, не указывая ее пакетов. Пакет **java.lang** просматривается всегда, его необязательно импортировать. Остальные пакеты стандартной библиотеки надо указывать в операторах **import**, либо записывать полные имена классов.

Подчеркнем, что оператор **import** вводится только для удобства программистов и слово "импортировать" не означает никаких перемещений классов.

Оператор **import** не эквивалентен директиве препроцессора **include** в языке C - он не подключает никакие файлы.

3.4. Java-файлы

Теперь можно описать структуру исходного файла с текстом программы на языке Java.

- В первой строке файла может быть необязательный оператор **package**.
- В следующих строках могут быть необязательные операторы **import**.
- Далее идут описания классов и интерфейсов.

Еще два правила.

- Среди классов файла может быть только один открытый **`public`** -класс.
- Имя файла должно совпадать с именем открытого класса, если последний существует.

Отсюда следует, что, если в проекте есть несколько открытых классов, то они должны находиться в разных файлах.

Соглашение "Code Conventions" рекомендует открытый класс, который, если он имеется в файле, нужно описывать первым.

3.5. Интерфейсы

Вы уже заметили, что получить расширение можно только от одного класса. Но иногда возникает необходимость породить класс от двух классов. Это называется *множественным наследованием*. Во множественном наследовании нет ничего плохого. Трудности возникают, если классы сами порождены от одного класса.

Создатели языка Java после долгих споров и размышлений поступили радикально - запретили множественное наследование вообще. При расширении класса после слова **`extends`** можно написать только одно имя суперкласса. С помощью уточнения **`super`** можно обратиться только к членам непосредственного суперкласса.

Но что делать, если все-таки при порождении надо использовать несколько предков? Например, у нас есть общий класс автомобилей **`Automobile`**, от которого можно породить класс грузовиков **`Truck`** и класс легковых автомобилей **`Car`**. Но вот надо описать пикап **`Pickup`**. Этот класс должен наследовать свойства и грузовых, и легковых автомобилей.

В таких случаях используется еще одна конструкция языка Java - интерфейс. Внимательно проанализировав ромбовидное наследование, теоретики ООП выяснили, что проблему создает только реализация методов, а не их описание.

Интерфейс, в отличие от класса, содержит только константы и заголовки методов, без их реализации.

Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются тоже в **`class`**-файлы.

Описание интерфейса начинается со слова **`interface`** , перед которым может стоять модификатор **`public`** , означающий, как и для класса, что интерфейс

доступен всюду. Если же модификатора **public** нет, интерфейс будет виден только в своем пакете.

После слова **interface** записывается имя интерфейса, потом может стоять слово **extends** и список интерфейсов-предков через запятую. Таким образом, интерфейсы могут порождаться от интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет корня, общего предка.

Затем, в фигурных скобках, записываются в любом порядке константы и заголовки методов. Можно сказать, что в интерфейсе все методы абстрактные, но слово **abstract** писать не надо. Константы всегда статические, но слова **static** и **final** указывать не нужно.

Все константы и методы в интерфейсах всегда открыты, не надо даже указывать модификатор **public**.

Вот какую схему можно предложить для иерархии автомобилей:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
interface Truck extends Automobile{ . . . }
interface Pickup extends Car, Truck{ . . . }
```

Таким образом, интерфейс - это только набросок, эскиз. В нем указано, что делать, но не указано, как это делать.

Как же использовать интерфейс, если он полностью абстрактен, в нем нет ни одного полного метода?

Использовать нужно не интерфейс, а его *реализацию* (implementation). Реализация интерфейса - это класс, в котором расписываются методы одного или нескольких интерфейсов. В заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово **implements** и, через запятую, перечисляются имена интерфейсов.

Вот как можно реализовать иерархию автомобилей:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
class Truck implements Automobile{ . . . }
class Pickup extends Truck implements Car{ . . . }
```

или так:

```
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
interface Truck extends Automobile{ . . . }
```

```
class Pickup implements Car, Truck{ . . . }
```

Реализация интерфейса может быть неполной, некоторые методы интерфейса расписаны, а другие - нет. Такая реализация - абстрактный класс, его обязательно надо пометить модификатором **abstract**.

Как реализовать в классе **Pickup** метод **f()**, описанный и в интерфейсе **Car**, и в интерфейсе **Truck** с одинаковой сигнатурой? Ответ простой - никак. Такую ситуацию нельзя реализовать в классе **Pickup**. Программу надо спроектировать по-другому.

Итак, интерфейсы позволяют реализовать средствами Java чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта.

Мы можем, приступая к разработке проекта, записать его в виде иерархии интерфейсов, не думая о реализации, а затем построить по этому проекту иерархию классов, учитывая ограничения одиночного наследования и видимости членов классов.

Интересно то, что мы можем создавать ссылки на интерфейсы. Конечно, указывать такая ссылка может только на какую-нибудь реализацию интерфейса. Тем самым мы получаем еще один способ организации полиморфизма.

Листинг 3.3 показывает, как можно собрать с помощью интерфейса хор домашних животных из листинга 2.2.

Листинг 3.3. Использование интерфейса для организации полиморфизма

```
interface Voice{
void voice();
}
class Dog implements Voice{
public void voice(){
System.out.println("Gav-gav!");
}
}
class Cat implements Voice{
public void voice(){
System.out.println("Miaou!");
}
}
class Cow implements Voice{
public void voice(){
System.out.println("Mu-u-u!");
}
}
public class Chorus{
```

```
public static void main(String[] args){
Voiced singer = new Voice[3];
singer[0] = new Dog();
singer[1] = new Cat();
singer[2] = new Cow();
for(int i = 0; i < singer.length; i++)
singer[i].voice();
}
}
```

Здесь используется интерфейс **Voice** вместо абстрактного класса **Pet**, описанного в листинге 2.2.

Что же лучше использовать: абстрактный класс или интерфейс? На этот вопрос нет однозначного ответа.

Создавая абстрактный класс, вы волей-неволей погружаете его в иерархию классов, связанную условиями одиночного наследования и единым предком - классом **Object**. Пользуясь интерфейсами, вы можете свободно проектировать систему, не задумываясь об этих ограничениях.

С другой стороны, в абстрактных классах можно сразу реализовать часть методов. Реализуя же интерфейсы, вы обречены на скучное переопределение всех методов.

Вы, наверное, заметили и еще одно ограничение: все реализации методов интерфейсов должны быть открытыми, **public**, поскольку при переопределении можно лишь расширять доступ, а методы интерфейсов всегда открыты.

Вообще же наличие и классов, и интерфейсов дает разработчику богатые возможности проектирования. В нашем примере, вы можете включить в хор любой класс, просто реализовав в нем интерфейс **Voice**.

Наконец, можно использовать интерфейсы просто для определения констант, как показано в листинге 3.4.

Листинг 3.4. Система управления светофором

```
interface Lights{
int RED = 0;
int YELLOW = 1;
int GREEN = 2;
int ERROR = -1;
}
class Timer implements Lights{
private int delay;
private static int light = RED;
Timer(int sec)(delay = 1000 * sec;}
```

```
public int shift(){
int count = (light++) % 3;
try{
switch(count){
case RED: Thread.sleep(delay); break;
case YELLOW: Thread.sleep(delay/3); break;
case GREEN: Thread.sleep(delay/2); break;
}
}catch(Exception e){return ERROR;}
return count;
}
}
class TrafficRegulator{
private static Timer t = new Timer(1);
public static void main(String[] args){
for(int k = -0; k < 10; k++)
switch(t.shift()){
case Lights.RED: System.out.println("Stop!"); break;
case Lights.YELLOW: System.out.println("Wait!"); break;
case Lights.GREEN: System.out.println("Go!"); break;
case Lights.ERROR: System.err.println("Time Error"); break;
default: System.err.println("Unknown light."); return;
}
}
}
```

Здесь, в интерфейсе **Lights**, определены константы, общие для всего проекта.

Класс **Timer** реализует этот интерфейс и использует константы напрямую как свои собственные. Метод **shift()** этого класса подает сигналы переключения светофору с разной задержкой в зависимости от цвета. Задержку осуществляет метод **sleep()** класса **Thread** из стандартной библиотеки, которому передается время задержки в миллисекундах. Этот метод нуждается в обработке исключений **try{}catch(){}**, о которой мы будем говорить в *главе 6*.

Класс **TrafficRegulator** не реализует интерфейс **Lights** и пользуется полными именами **Lights.RED** и т.д. Это возможно потому, что константы **RED**, **YELLOW** и **GREEN** по умолчанию являются статическими.

4. Работа со строками

Очень большое место в обработке информации занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они представляются экземплярами класса **String** или класса **StringBuffer**.

Конечно, можно занести текст в массив символов типа **char** или даже в массив байтов типа **byte**, но тогда нельзя будет использовать готовые методы работы с текстовыми строками.

Зачем в язык введены два класса для хранения строк? В объектах класса **String** хранятся строки-константы неизменной длины и содержания. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее. Длину строк, хранящихся в объектах класса **StringBuffer**, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку. Во многих случаях, когда надо изменить длину строки типа **String**, компилятор Java неявно преобразует ее к типу **StringBuffer**, меняет длину, потом преобразует обратно в тип **String**. Например, следующее действие

```
String s = "Это" + " одна " + "строка";
```

компилятор выполнит так:

```
String s = new StringBuffer().append("Это").append(" одна ")
     .append("строка").toString();
```

Будет создан объект класса **StringBuffer**, в него последовательно добавлены строки **"Это"**, **" одна "**, **"строка"**, и получившийся объект класса **StringBuffer** будет приведен к типу **String** методом **toString()**.

Напомним, что символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа **char**.

4.1. Класс **String**

Перед работой со строкой ее следует создать. Это можно сделать разными способами.

Самый простой способ создать строку - это организовать ссылку типа **String** на строку-константу:

```
String s1 = "Это строка.";
```

Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления:

```
String s2 = "Это длинная строка, " +
     "записанная в двух строках исходного текста";
```

Не забывайте разницу между пустой строкой **`String s = ""`**, не содержащей ни одного символа, и пустой ссылкой **`String s = null`**, не указывающей ни на какую строку и не являющейся объектом.

Самый правильный способ создать объект с точки зрения ООП - это вызвать его конструктор в операции **`new`**. Класс **`String`** предоставляет вам девять конструкторов:

- **`String()`** - создается объект с пустой строкой;
- **`String(String str)`** - из одного объекта создается другой, поэтому этот конструктор используется редко;
- **`String(StringBuffer str)`** - преобразованная копия объекта класса **`StringBuffer`**;
- **`String(byte[] byteArray)`** - объект создается из массива байтов **`byteArray`**;
- **`String(char[] charArray)`** - объект создается из массива charArray символов Unicode;
- **`String(byte[] byteArray, int offset, int count)`** - объект создается из части массива байтов **`byteArray`**, начинающейся с индекса **`offset`** и содержащей **`count`** байтов;
- **`String(char[] charArray, int offset, int count)`** - то же, но массив состоит из символов Unicode;
- **`String(byte[] byteArray, String encoding)`** - символы, записанные в массиве байтов, задаются в Unicode-строке, с учетом кодировки **`encoding`**;
- **`String(byte[] byteArray, int offset, int count, String encoding)`** - то же самое, но только для части массива.

При неправильном заданий индексов **`offset`**, **`count`** или кодировки **`encoding`** возникает исключительная ситуация.

Конструкторы, использующие массив байтов **`byteArray`**, предназначены для создания Unicode-строки из массива байтовых ASCII-кодировок символов. Такая ситуация возникает при чтении ASCII-файлов, извлечении информации из базы данных или при передаче информации по сети.

В самом простом случае компилятор для получения двухбайтовых символов Unicode добавит к каждому байту старший нулевой байт. Получится диапазон **`'\u0000'`** - **`'\u00ff'`** кодировки Unicode, соответствующий кодам Latin 1. Тексты на кириллице будут выведены неправильно.

Если же на компьютере сделаны местные установки, как говорят на жаргоне "установлена локаль" (locale) (в MS Windows это выполняется утилитой Regional Options в окне **Control Panel**), то компилятор, прочитав эти установки, создаст символы Unicode, соответствующие местной кодовой

странице. В русифицированном варианте MS Windows это обычно кодовая страница CP1251.

Если исходный массив с кириллическим ASCII-текстом был в кодировке CP1251, то строка Java будет создана правильно. Кириллица попадет в свой диапазон `'\u0400' - '\u04FF'` кодировки Unicode.

Но у кириллицы есть еще, по меньшей мере, четыре кодировки.

- В MS-DOS применяется кодировка CP866.
- В UNIX обычно применяется кодировка KOI8-R.
- На компьютерах Apple Macintosh используется кодировка MacCyrillic.
- Есть еще и международная кодировка кириллицы ISO8859-5;

Например, байт `11100011` (`0xE3` в шестнадцатеричной форме) в кодировке CP1251 представляет кириллическую букву `г`, в кодировке CP866 - букву `у`, в кодировке KOI8-R - букву `ц`, в ISO8859-5 - букву `у` , в MacCyrillic - букву `г`.

Если исходный кириллический ASCII-текст был в одной из этих кодировок, а местная кодировка CP1251, то Unicode-символы строки Java не будут соответствовать кириллице.

В этих случаях используются последние два конструктора, в которых параметром `encoding` указывается, какую кодовую таблицу использовать конструктору при создании строки.

Листинг 5.1 показывает различные случаи записи кириллического текста. В нем создаются три массива байтов, содержащих слово "Россия" в трех кодировках.

- Массив `byteCP1251` содержит слово "Россия" в кодировке CP1251.
- Массив `byteCP866` содержит слово "Россия" в кодировке CP866.
- Массив `byteKOI8R` содержит слово "Россия" в кодировке KOI8-R.

Из каждого массива создаются по три строки с использованием трех кодовых таблиц.

Кроме того, из массива символов `c[]` создается строка `s1`, из массива байтов, записанного в кодировке CP866, создается строка `s2`. Наконец, создается ссылка `s3` на строку-константу.

Листинг 5.1. Создание кириллических строк

```
class StringTest{
public static void main(String[] args){
String winLikeWin = null, winLikeDOS = null, winLikeUNIX = null;
String dosLikeWin = null, dosLikeDOS = null, dosLikeUNIX = null;
```

```
String unixLikeWin = null, unixLikeDOS = null, unixLikeUNIX =
null;
String msg = null;
byte[] byteCp!251 = {
(byte)0xD0, (byte)0xEE, (byte)0xFl,
(byte)0xFl, (byte)0xES, (byte)0xFF
};
byte[] byteCp866 = {
(byte)0x90, (byte)0xAE, (byte)0xE1,
(byte)0xEl, (byte)0xA8, (byte)0xEF
};
byte[] byteKOISR = (
(byte)0xF2, (byte)0xCF, (byte)0xD3,
(byte)0xD3, (byte)0xC9, (byte)0xDl
};
char[] c = {'P', 'o', 'c', 'c', 'и', 'я'};
String s1 = new String(c);
String s2 = new String(byteCp866); // Для консоли MS Windows
String s3 = "Россия";
System.out.println();
try{
// Сообщение в Cp866 для вывода на консоль MS Windows.
msg = new String("\"Россия\" в ".getBytes("Cp866"), "Cpl251");
winLikeWin = new String(byteCp1251, "Cpl251"); //Правильно
winLikeDOS = new String(byteCpl251,: "Cp866");
winLikeUNIX – new String(byteCp1251, "KOI8-R");
dosLikeWin = new String(byteCp866, "Cpl251"); // Для консоли
dosLikeDOS = new String(byteCp866, "Cp866"); // Правильно
dosLikeUNIX = new String(byteCp866, "KOI8-R");
unixLikeWin = new String(byteKOISR, "Cpl251");
unixLikeDOS = new String(byteKOISR, "Cp866");
unixLikeUNIX = new String(byteKOISR, "KOI8-R"); // Правильно
System.out.print(msg + "Cpl251: ");
System.out.write(byteCp1251);
System.out.println();
System.out.print(msg + "Cp866 : ");
System, out.write (byteCp866} ;
System.out.println();
System.out.print(msg + "KOI8-R: ");
System.out.write(byteKOI8R);
{catch(Exception e)(
e.printStackTrace();
}
System.out.println();
System.out.println();
System.out.println(msg + "char array : " + s1);
System.out.println(msg + "default encoding : " + s2);
System.out.println(msg + "string constant : " + s3);
System.out.println();
System.out.println(msg + "Cp1251 -> Cp1251: " + winLikeWin);
System.out.println(msg + "Cp1251 -> Cp866 : " + winLikeDOS);
System.out.println(msg + "Cp1251 -> KOI8-R: " + winLikeUNIX);
System.out.println(msg + "Cp866 -> Cp1251: " + dosLikeWin);
```

```
System.out.println(msg + "Cp866 -> Cp866 : " + dosLikeDOS);
System.out.println(msg + "Cp866 -> KOI8-R: " + dosLikeUNIX);
System.out.println(msg + "KOI8-R -> Cpl251: " + unixLikeWin);
System.out.println(msg + "KOI8-R -> Cp866 : " + unixLikeDOS);
System.out.println(msg + "KOI8-R -> KOI8-R: " + unixLikeUNIX);
}
}
```

Еще один способ создать строку - это использовать два статических метода

```
copyValueOf(chart] charArray);
copyValueOf(char[] charArray, int offset, int length);
```

Они создают строку по заданному массиву символов и возвращают ее в качестве результата своей работы. Например, после выполнения следующего фрагмента программы

```
chart] c = ('С', 'и', 'м', 'в', 'о 1 , 'л', 'ь', 'н', 'ы', 'й'};
String s1 = String.copyValueOf(c);
String s2 = String.copyValueOf(c, 3, 7);
```

получим в объекте **s1** строку **"Символьный"**, а в объекте **s2** - строку **"вольный"**.

4.1.1. Сцепление строк

Со строками можно производить операцию *сцепления строк*, обозначаемую знаком плюс **+**. Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк, как показано в начале данной главы. Ее можно применять и к константам, и к переменным. Например:

```
String attention = "Внимание: ";
String s = attention + "неизвестный символ";
```

Вторая операция — присваивание **+=** - применяется к переменным в левой части:

```
attention += s;
```

Поскольку операция **+** перегружена со сложения чисел на сцепление строк, встает вопрос о приоритете этих операций. У сцепления строк приоритет выше, чем у сложения, поэтому, записав **"2" + 2 + 2**, получим строку **"222"**. Но, записав **2 + 2 + "2"**, получим строку **"42"**, поскольку действия выполняются слева направо. Если же запишем **"2" + (2 + 2)**, то получим **"24"**.

4.1.2. Манипуляции строками

В классе **String** есть множество методов для работы со строками. Посмотрим, что они позволяют делать.

Длина строки

Для того чтобы узнать длину строки, т.е. количество символов в ней, надо обратиться к методу **length()**:

```
String s = "Write once, run anywhere.";
int len = s.length();
```

или еще проще

```
int len = "Write once, run anywhere.".length();
```

поскольку строка-константа - полноценный объект класса **String**. Заметьте, что строка - это не массив, у нее нет поля **length**.

Выбор символа из строки

Выбрать символ с индексом **ind** (индекс первого символа равен нулю) можно методом **charAt(int ind)**. Если индекс **ind** отрицателен или не меньше чем длина строки, возникает исключительная ситуация. Например, после определения

```
char ch = s.charAt(3);
```

переменная **ch** будет иметь значение **'t'**

Все символы строки в виде массива символов можно получить методом **toCharArray()**.

Если же надо включить в массив символов **dst**, начиная с индекса **ind** массива подстроку от индекса **begin** включительно до индекса **end** исключительно, то используйте метод
getChars(int begin, int end, char[] dst, int ind) типа **void**.

В массив будет записано **end-begin** символов, которые займут элементы массива, начиная с индекса **ind** до индекса **ind+(end-begin)-1**.

Этот метод создает исключительную ситуацию в следующих случаях:

- ссылка **dst == null**;
- индекс **begin** отрицателен;
- индекс **begin** больше индекса **end**;

- индекс **end** больше длины строки;
- индекс **ind** отрицателен;
- **ind+(end–begin)** больше **dst.length**.

Например, после выполнения

```
char[] ch = ('К', 'о', 'н', 'е', 'ц', ' ', 'л', 'е', 'т', 'а'};
"Пароль легко найти".getChars(2, 8, ch, 2);
```

результат будет таков:

```
ch = ('К', 'о', 'р', 'о', 'л', 'ь ', ' ', 'л', 'е', 'т', 'а'};
```

Если надо получить массив байтов, содержащий все символы строки в байтовой кодировке ASCII, то используйте метод **getBytes()**.

Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, используйте метод **getBytes(String encoding)**.

Так сделано в листинге 5.1 при создании объекта **msg**. Строка "\"Россия в\"" перекодировалась в массив CP866-байтов для правильного вывода кириллицы в консольное окно **Command Prompt** операционной системы Windows 2000.

Выбор подстроки

Метод **substring(int begin, int end)** возвращает подстроку от символа с индексом **begin** включительно до символа с индексом **end** исключительно. Длина подстроки будет равна **end–begin**.

Метод **substring(int begin)** возвращает подстроку от индекса **begin** включительно до конца строки.

Если индексы отрицательны, индекс **end** больше длины строки или **begin** больше чем **end**, то возникает исключительная ситуация.

Например, после выполнения

```
String s = "Write once, run anywhere.";
String sub1 = s.substring(6, 10);
String sub2 = s.substring(16);
```

получим в строке **sub1** значение **"once"**, а в **sub2** - значение **"anywhere"**.

Разбиение строки на подстроки

Метод **split(String regExp)** разбивает строку на подстроки, используя в качестве разделителей символы, входящие в параметр **regExp**, записывает подстроки в массив строк и возвращает ссылку на этот массив. Сами разделители не входят в подстроки.

Например, после выполнения следующего фрагмента

```
String s = "Write:once,:run:anywhere.";
String[] sub = s.split(":");
```

получим в строке **sub[0]** значение **"Write"**, в строке **sub[1]** значение **"once,"**, в строке **sub[2]** значение **"run"**, в строке **sub[3]** - значение **"anywhere."**.

Метод **split(String regExp, int n)** разбивает строку на **n** подстрок. Если параметр **n** меньше числа подстрок, то весь остаток строки заносится в последний элемент создаваемого массива строк. Применение метода

```
String[] sub = s.split(":", 2);
```

в предыдущем примере даст массив **sub** из двух элементов со значением **sub[0]**, равным **"Write"**, и значением **sub[1]**, равным **"once,:run:anywhere."**.

Разбить строку можно практически на любые подстроки, поскольку значением параметра **regExp** может быть любое регулярное выражение.

Сравнение строк

Операция сравнения **==** сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк

```
String s1 = "Какая-то строка";
String s2 = "Другая-строка";
```

сравнение **s1 == s2** дает в результате **false**.

Значение **true** получится, только если обе ссылки указывают на одну и ту же строку, например, после присваивания **s1 = s2**.

Интересно, что если мы определим **s3** так:

```
String s3 = "Какая-то строка";
```

то сравнение **s1 == s3** даст в результате **true**, потому что компилятор создаст только один экземпляр константы **"Какая-то строка"** и направит на него все ссылки - и ссылку **s1**, и ссылку **s3**.

Мы, разумеется, хотим сравнивать не ссылки, а содержимое строк. Для этого есть несколько методов.

Логический метод **equals(Object obj)**, переопределенный из класса **Object**, возвращает **true**, если аргумент **obj** не равен **null**, является объектом класса **String**, и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение **false**.

Логический метод **equalsIgnoreCase(Object obj)** работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими.

Например, **s2.equals("другая строка")** даст в результате **false**, а **s2.equalsIgnoreCase("другая строка")** возвратит **true**.

Метод **compareTo(String str)** возвращает целое число типа **int**, вычисленное по следующим правилам:

1. Сравниваются символы данной строки **this** и строки **str** с одинаковым индексом, пока не встретятся различные символы с индексом, допустим **k**, или пока одна из строк не закончится.
2. В первом случае возвращается значение **this.charAt(k)-str.charAt(k)**, т.е. разность кодировок Unicode первых несовпадающих символов.
3. Во втором случае возвращается значение **this.length()-str.length()**, т.е. разность длин строк.
4. Если строки совпадают, возвращается 0.

Если значение **str** равно **null**, возникает исключительная ситуация.

Нуль возвращается в той же ситуации, в которой метод **equals()** возвращает **true**.

Метод **compareToIgnoreCase(String str)** производит сравнение без учета регистра букв, точнее говоря, выполняется метод

```
this.toUpperCase().toLowerCase().compareTo(
     str.toUpperCase().toLowerCase());
```

Эти методы не учитывают алфавитное расположение символов в локальной кодировке.

Русские буквы расположены в Unicode по алфавиту, за исключением одной буквы. Заглавная буква Ё расположена перед всеми кириллическими буквами, ее код **'\u0401'**, а строчная буква ё - после всех русских букв, ее код **'\u0451'**.

Если вас такое расположение не устраивает, задайте свое размещение букв с помощью класса **RuleBasedCollator** из пакета **java.text**.

Сравнить подстроку данной строки **this** с подстрокой той же длины **len** другой строки **str** можно логическим методом

```
regionMatches(int ind1, String str, int ind2, int len);
```

Здесь **ind1** - индекс начала подстроки данной строки **this**, **ind2** - индекс начала подстроки другой строки **str**. Результат **false** получается в следующих случаях:

- хотя бы один из индексов **ind1** или **ind2** отрицателен;
- хотя бы одно из **ind1+len** или **ind2+len** больше длины соответствующей строки;
- хотя бы одна пара символов не совпадает.

Этот метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод:

```
regionMatches(boolean flag, int ind1,
     String str, int ind2, int len)
```

Если первый параметр **flag** равен **true**, то регистр букв при сравнении подстрок не учитывается, если **false** - учитывается.

Поиск символа в строке

Поиск всегда ведется с учетом регистра букв.

Первое появление символа **ch** в данной строке **this** можно отследить методом **indexOf(int ch)**, возвращающим индекс этого символа в строке или -1 , если символа **ch** в строке **this** нет.

Например, **"Молоко".indexOf('o')** выдаст в результате 1.

Конечно, этот метод реализован так, что он выполняет в цикле последовательные сравнения **this.charAt(k++)==ch**, пока не получит значение **true**.

Второе и следующие появления символа **ch** в данной строке **this** можно отследить методом **indexOf(int ch, int ind)**. Этот метод начинает поиск символа **ch** с индекса **ind**. Если **ind<0**, то поиск идет с начала строки, если **ind** больше длины строки, то символ не ищется, т.е. возвращается -1.

Например, **"Молоко".indexOf('o', indexOf('o')+1)** даст в результате 3.

Последнее появление символа **ch** в данной строке **this** отслеживает метод **lastIndexOf(int ch)**. Он просматривает строку в обратном порядке. Если символ **ch** не найден, возвращается.-1.

Например, **"Молоко".lastIndexOf('o')** даст в результате 5.

Предпоследнее и предыдущие появления символа **ch** в данной строке **this** можно отследить методом **lastIndexOf(int ch, int ind)**, который просматривает строку в обратном порядке, начиная с индекса **ind**.

Если **ind** больше длины строки, то поиск идёт от конца строки, если **ind<0**, то возвращается -1.

Поиск подстроки в строке

Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки **sub** в данную строку **this** отыскивает метод **indexOf(String sub)**. Он возвращает индекс первого символа первого вхождения подстроки **sub** в строку или -1, если подстрока **sub** не входит в строку **this** . Например, **"Раскраска".indexOf("рас")** даст в результате 4.

Если вы хотите начать поиск не с начала строки, а с какого-то индекса **ind**, используйте метод **indexOf(String sub, int ind)**. Если **ind<0**, то поиск идет с начала строки, если **ind** больше длины строки, то символ не ищется, т.е. возвращается -1.

Последнее вхождение подстроки **sub** в данную строку **this** можно отыскать методом **lastIndexOf(String sub)**, возвращающим индекс первого символа последнего вхождения подстроки **sub** в строку **this** или -1, если подстрока **sub** не входит в строку **this**.

Последнее вхождение подстроки **sub** не во всю строку **this**, а только в ее начало до индекса **ind** можно отыскать методом **lastIndexOf(String stf, int ind)**. Если **ind** больше длины строки, то поиск идет от конца строки, если **ind<0**, то возвращается -1.

Для того чтобы проверить, не начинается ли данная строка **this** с подстроки **sub**, используйте логический метод **startsWith(string sub)**, возвращающий **true**, если данная строка **this** начинается с подстроки **sub**, или совпадает с ней, или подстрока **sub** пуста.

Можно проверить и появление подстроки **sub** в данной строке **this**, начиная с некоторого индекса **ind** логическим методом **startsWith(String sub, int ind)**. Если индекс **ind** отрицателен или больше длины строки, возвращается **false**.

Для того чтобы проверить, не заканчивается ли данная строка **this** подстрокой **sub**, используйте логический метод **endsWith(String sub)**. Он возвращает **true**, если подстрока **sub** совпадает со всей строкой или подстрока **sub** пуста.

Например, **if(fileName.endsWith(".java"))** отследит имена файлов с исходными текстами Java.

Перечисленные выше методы создают исключительную ситуацию, если **sub==null**.

Если вы хотите осуществить поиск, не учитывающий регистр букв, измените предварительно регистр всех символов строки.

Изменение регистра

Метод **toLowerCase()** возвращает новую строку, в которой все буквы переведены в нижний регистр, т. е. сделаны строчными.

Метод **toUpperCase()** возвращает новую строку, в которой все буквы переведены в верхний регистр, т.е. сделаны прописными.

При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы **toLowerCase(Locale loc)** и **toUpperCase(Locale loc)**.

Замена символа в строке

Метод **replace(int old, int new)** возвращает новую строку, в которой все вхождения символа **old** заменены символом **new** . Если символа **old** в строке нет, то возвращается ссылка на исходную строку.

Например, после выполнения
"Рука в руку сует хлеб".replace('y', 'e') получим строку
"Река в реке сеет хлеб".

Регистр букв при замене учитывается.

Удаление пробелов в строке

Метод **`trim()`** возвращает новую строку, в которой удалены начальные и конечные символы с кодами, не превышающими **`'\u0020'`**.

Преобразование данных другого типа в строку

В языке Java принято соглашение - каждый класс отвечает за преобразование других типов в тип этого класса и должен содержать нужные для этого методы.

Класс **`String`** содержит восемь статических методов **`valueOf(`*`тип`*` elem)`** преобразования в строку базовых типов **`boolean`**, **`char`**, **`int`**, **`long`**, **`float`**, **`double`**, массива **`char[]`**, и просто объекта типа **`Object`**.

Девятый метод **`valueOf(char[] ch, int offset, int len)`** преобразует в строку подмассив массива **`ch`** , начинающийся с индекса **`offset`** и имеющий **`len`** элементов.

Кроме того, в каждом классе есть метод **`toString()`**, переопределенный или просто унаследованный от класса **`Object`** . Он преобразует объекты класса в строку. Фактически, метод **`valueOf()`** вызывает метод **`toString()`** соответствующего класса. Поэтому результат преобразования зависит от того, как реализован метод **`toString()`**.

Еще один простой способ - сцепить значение **`elem`** какого-либо типа с пустой строкой: **`""+elem`**. При этом неявно вызывается метод **`elem.toString()`**.

4.2. Класс `StringBuffer`

Объекты класса **`StringBuffer`** - это строки переменной длины. Только что созданный объект имеет буфер определенной *емкости*, по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.

Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

В любое время емкость буфера можно увеличить, обратившись к методу

`ensureCapacity(int minCapacity)`

Этот метод изменит емкость, только если **`minCapacity`** будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему

правилу. Пусть емкость буфера равна **N**. Тогда новая емкость будет равна **Max(2 * N + 2, minCapacity)**.

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

Методом **setLength(int newLength)** можно установить любую длину строки.

Если она окажется больше текущей длины, то дополнительные символы будут равны **'\u0000'**. Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются, точнее, будут заменены символом **'\u0000'**. Емкость при этом не изменится.

Если число **newLength** окажется отрицательным, возникнет исключительная ситуация.

Будьте осторожны, устанавливая новую длину объекта.

Количество символов в строке можно узнать, как и для объекта класса **String**, методом **length()**, а емкость - методом **capacity()**.

Создать объект класса **StringBuffer** можно только конструкторами.

4.2.1. Конструкторы

В классе **StringBuffer** три конструктора:

StringBuffer() - создает пустой объект с емкостью 16 символов;

StringBuffer(int capacity) - создает пустой объект заданной емкости **capacity**;

StringBuffer(String str) - создает объект емкостью **str.length()+16**, содержащий строку **str.**

4.2.2. Добавление подстроки

В классе **StringBuffer** есть десять методов **append()**, добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод **append(String str)** присоединяет строку **str** в конец данной строки. Если ссылка **str==null**, то добавляется строка **"null"**.

Шесть методов **append(*тип* elem)** добавляют базовые типы **boolean**, **char**, **int**, **long**, **float**, **double**, преобразованные в строку.

Два метода присоединяют к строке массив **str** и подмассив **sub** символов, преобразованные в строку:

append(char[] str)

и

append(char[] sub, int offset, int len).

Десятый метод добавляет просто объект **append(Object obj)**. Перед этим объект **obj** преобразуется в строку своим методом **toString()**.

4.2.3. Вставка подстроки

Десять методов **insert()** предназначены для вставки строки, указанной параметром метода, в данную строку. Место вставки задается первым параметром метода **ind**. Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же преобразованную строку.

Основной метод **insert(int ind, String str)** вставляет строку **str** в данную строку перед ее символом с индексом **ind**. Если ссылка **str==null** вставляется строка **"null"**.

Например, после выполнения

```
String s = new StringBuffer("Это большая строка").insert(
     4, "не").toString();
```

Получим **s == "Это небольшая строка"**.

Метод **sb.insert(sb.length(), "xxx")** будет работать так же, как метод **sb.append("xxx")**.

Шесть методов **insert(int ind, type elem)** вставляют базовые типы **boolean**, **char**, **int**, **long**, **float**, **double**, преобразованные в строку.

Два метода вставляют массив **str** и подмассив **sub** символов, преобразованные в строку:

```
insert(int ind, char[] str)
insert(int ind, char[] sub, int offset, int len)
```

Десятый метод вставляет просто объект :

insert(int ind, Object obj)

Объект **obj** перед добавлением преобразуется в строку своим методом **toString()**.

4.2.4. Удаление подстроки

Метод **delete(int begin, int end)** удаляет из строки символы, начиная с индекса **begin** включительно до индекса **end** исключительно, если **end** больше длины строки, то до конца строки.

Например, после выполнения

```
String s = new StringBuffer(
     "Это небольшая строка").delete(4,6).toString();
```

получим **s == "Это большая строка"**.

Если **begin** отрицательно, больше длины строки или больше **end** , возникает исключительная ситуация.

Если **begin == end**, удаление не происходит.

4.2.5. Удаление символа

Метод **deleteCharAt(int ind)** удаляет символ с указанным индексом **ind**. Длина строки уменьшается на единицу.

Если индекс **ind** отрицателен или больше длины строки, возникает исключительная ситуация.

4.2.6. Замена подстроки в строке

Метод **replace(int begin, int end, String str)** удаляет символы из строки, начиная с индекса **begin** включительно до индекса **end** исключительно, если **end** больше длины строки, то до конца строки, и вставляет вместо них строку **str**.

Если **begin** отрицательно, больше длины строки или больше **end**, возникает исключительная ситуация.

Разумеется, метод **replace()** - это последовательное выполнение методов

delete() и **insert()**.

4.2.7. Обращение строки

Метод **reverse()** меняет порядок расположения символов в строке на обратный порядок.

Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка").
     reverse().toString();
```

получим **s == "акортс яашьлобен отЭ"**.

4.3. Синтаксический разбор строки

Задача разбора введенного текста - *парсинг* (parsing) - вечная задача программирования, наряду с сортировкой и поиском.

В пакет **java.util** входит простой класс **StringTokenizer**, облегчающий разбор строк.

4.3.1. Класс **StringTokenizer**

Класс **StringTokenizer** из пакета **java.util** небольшой, в нем три конструктора и шесть методов.

Первый конструктор **StringTokenizer(String str)** создает объект, готовый разбить строку **str** на слова, разделенные пробелами, символами табуляций **'\t'**, перевода строки **'\n'** и возврата каретки **'\r'**. Разделители не включаются в число слов.

Второй конструктор **StringTokenizer(String str, String delimeters)** задает разделители вторым параметром **delimeters**, например:

```
StringTokenizer("Казнить,нельзя:пробелов-нет", " \t\n\r,:-");
```

Здесь первый разделитель - пробел. Потом идут символ табуляции, символ перевода строки, символ возврата каретки, запятая, двоеточие, дефис. Порядок расположения разделителей в строке **delimeters** не имеет значения. Разделители не включаются в число слов.

Третий конструктор позволяет включить разделители в число слов:

```
StringTokenizer(String str, String deiimeters, boolean flag);
```

Если параметр **flag** равен **true**, то разделители включаются в число слов, если **false** - нет. Например:

```
StringTokenizer("a - (b + c) / b * c", " \t\n\r+*-/()", true);
```

В разборе строки на слова активно участвуют два метода:

- метод **nextToken()** возвращает в виде строки следующее слово;
- логический метод **hasMoreTokens()** возвращает **true**, если в строке еще есть слова, и **false**, если слов больше нет.

Третий метод **countTokens()** возвращает число оставшихся слов.

Четвертый метод **nextToken(string newDelimeters)** позволяет "на ходу" менять разделители. Следующее слово будет выделено по новым разделителям **newDelimeters**; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе **nextToken()**.

Оставшиеся два метода **nextElement()** и **hasMoreElements()** реализуют интерфейс **Enumeration**. Они просто обращаются к методам **nextToken()** и **hasMoreTokens()**.

Схема очень проста (листинг 4.2).

Листинг 4.2. Разбиение строки на слова :

```
String s = "Строка, которую мы хотим разобрать на слова";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while(st.hasMoreTokens()){
// Получаем слово и что-нибудь делаем с ним, например,
// просто выводим на экран
System.out.println(st.nextToken()) ;
}
```

5. Потоки ввода/вывода

Программы, написанные нами в предыдущих главах, воспринимали информацию только из параметров командной строки и графических компонентов, а результаты выводили на консоль или в графические компоненты. Однако во многих случаях требуется выводить результаты на принтер, в файл, базу данных или передавать по сети. Исходные данные тоже часто приходится загружать из файла, базы данных или из сети.

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие *потока* (stream). Считается, что в программу идет *входной поток* (input stream) символов Unicode или просто байтов, воспринимаемый в программе методами **read()**. Из программы методами **write()** или **print()**, **println()** выводится *выходной поток* (output stream) символов или байтов. При этом неважно, куда направлен

поток: на консоль, на принтер, в файл или в сеть, методы **`write()`** и **`print()`** ничего об этом не знают.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

Три потока определены в классе **`System`** статическими полями **`in`**, **`out`** и **`err`**. Их можно использовать без всяких дополнительных определений. Они называются соответственно *стандартным вводом*, *стандартным выводом* и *стандартным выводом сообщений*. Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки **`out`** и **`err`** - это экземпляры класса **`PrintStream`**, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами **`print()`**, **`println()`** и **`write()`**, которых в классе **`PrintStream`** имеется около двадцати для разных типов аргументов.

Поток **`err`** предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток **`in`** - это экземпляр класса **`InputStream`**. Он назначен на клавиатурный ввод с консоли методами **`read()`**. Класс **`InputStream`** абстрактный, поэтому реально используется какой-то из его подклассов.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода/вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т.е. связывающих программу с областью оперативной памяти. Более того, можно создать поток, связанный со строкой типа **`String`**, находящейся, опять-таки, в оперативной памяти. Кроме того, можно создать *канал* (pipe) обмена информацией между подпроцессами.

Еще один вид потока - поток байтов, составляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется *сериализацией* (serialization) объектов.

Методы организации потоков собраны в классы пакета **`java.io`**.

Кроме классов, организующих поток, в пакет **`java.io`** входят классы с методами преобразования потока, например, можно преобразовать поток байтов, образующих целые числа, в поток этих чисел.

Еще одна возможность, предоставляемая классами пакета **java.io**, - слить несколько потоков в один поток.

Итак, в Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии четыре класса, непосредственно расширяющих класс **Object**:

- **Reader** - абстрактный класс, в котором собраны самые общие методы символьного ввода;
- **Writer** - абстрактный класс, в котором собраны самые общие методы символьного вывода;
- **InputStream** - абстрактный класс с общими методами байтового ввода;
- **OutputStream** - абстрактный класс с общими методами байтового вывода.

Классы входных потоков **Reader** и **InputStream** определяют по три метода ввода:

- **read()** - возвращает один символ или байт, взятый из входного потока, в виде целого значения типа **int**; если поток уже закончился, возвращает -1;
- **read(char[] buf)** - заполняет заранее определенный массив **buf** символами из входного потока; в классе inputstream массив типа **byte[]** и заполняется он байтами; метод возвращает фактическое число взятых из потока элементов или -1, если поток уже закончился;
- **read(char[] buf, int offset, int len)** - заполняет часть символьного или байтового массива **buf**, начиная с индекса **offset**, число взятых из потока элементов равно **len**; метод возвращает фактическое число взятых из потока элементов или -1.

Эти методы выбрасывают **IOException**, если произошла ошибка ввода/вывода.

Четвертый метод **skip(long n)** "проматывает" поток с текущей позиции на **n** символов или байтов вперед. Эти элементы потока не вводятся методами **read()**. Метод возвращает реальное число пропущенных элементов, которое может отличаться от **n**, например поток может закончиться.

Текущий элемент потока можно пометить методом **mark(int n)**, а затем вернуться к помеченному элементу методом **reset()**, но не более чем через **n** элементов. Не все подклассы реализуют эти методы, поэтому перед расстановкой пометок следует обратиться к логическому методу **markSupported()**, который возвращает **true**, если реализованы методы расстановки и возврата к пометкам.

Классы выходных потоков **Writer** и **OutputStream** определяют по три почти одинаковых метода вывода:

- **write(char[] buf)** - выводит массив в выходной поток, в классе **OutputStream** массив имеет тип **byte[]**;
- **write(char[] buf, int offset, int len)** - выводит **len** элементов массива **buf**, начиная с элемента с индексом **offset**;
- **write(int elem)** в классе **Writer** - выводит 16, а в классе **OutputStream** 8 младших битов аргумента **elem** в выходной поток,

В классе **Writer** есть еще два метода:

- **write(string s)** - выводит строку **s** в выходной поток;
- **write(String s, int offset, int len)** - выводит **len** символов строки **s**, начиная с символа с номером **offset**.

Многие подклассы классов **Writer** и **OutputStream** осуществляют буферизованный вывод. При этом элементы сначала накапливаются в буфере, в оперативной памяти, и выводятся в выходной поток только после того, как буфер заполнится. Это удобно для выравнивания скоростей вывода из программы и вывода потока, но часто надо вывести информацию в поток еще до заполнения буфера. Для этого предусмотрен метод **flush()**. Данный метод сразу же выводит все содержимое буфера в поток.

По окончании работы с потоком его необходимо закрыть методом **close()**.

Все классы пакета **java.io** можно разделить на две группы: классы, *создающие поток*, и классы, *управляющие потоком*.

Классы, создающие потоки, в свою очередь, можно разделить на пять групп:

- классы, создающие потоки, связанные с файлами:

  FileReader **FilelnputStream**
  FileWriter **FileOutputstream**

- классы, создающие потоки, связанные с массивами:

  CharArrayReader **ByteArraylnputStream**
  CharArrayWriter **ByteArrayOutputStream**

- классы, создающие каналы обмена информацией между подпроцессами:

  PipedReader **PipedlnputStream**
  PipedWriter **PipedOutputStream**

- классы, создающие символьные потоки, связанные со строкой:

  StringReader
  StringWriter

- классы, создающие байтовые потоки из объектов Java:

  ObjectInputStream
  ObjectOutputStream

Слева перечислены классы символьных потоков, справа - классы байтовых потоков.

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток. Можно представлять их себе как "переходное кольцо", после которого идет труба другого диаметра.

Четыре класса созданы специально для преобразования потоков:

FilterReader	**FilterInputStream**
FilterWriter	**FilterOutputStream**

Сами по себе эти классы бесполезны - они выполняют тождественное преобразование. Их следует расширять, переопределяя методы ввода/вывода. Но для байтовых фильтров есть полезные расширения, которым соответствуют некоторые символьные классы. Перечислим их.

Четыре класса выполняют буферизованный ввод/вывод:

BufferedReader	**BufferedInputStream**
BufferedWriter	**BufferedOutputStream**

Два класса преобразуют поток байтов, образующих восемь простых типов Java, в эти самые типы:

DataInputStream
DataOutputStream

Два класса содержат методы, позволяющие вернуть несколько символов или байтов во входной поток:

PushbackReader	**PushbackInputStream**

Два класса связаны с выводом на строчные устройства - экран дисплея, принтер:

PrintWriter	**PrintStream**

Два класса связывают байтовый и символьный потоки:

- **`InputStreamReader`** - преобразует входной байтовый поток в символьный поток;
- **`OutputStreamWriter`** - преобразует выходной символьный поток в байтовый поток.

Класс **`StreamTokenizer`** позволяет разобрать входной символьный поток на отдельные элементы (tokens) подобно тому, как класс **`StringTokenizer`**, рассмотренный нами в *главе 4,* разбирал строку.

Из управляющих классов выделяется класс **`SequenceInputStream`**, сливающий несколько потоков, заданных в конструкторе, в один поток, и класс **`LineNumberReader`**, "умеющий" читать выходной символьный поток построчно. Строки в потоке разделяются символами **`'\n'`** и/или **`'\r'`**.

особняком стоит класс **`RandomAccessFile`**, реализующий прямой доступ к файлу. Он не создает поток байтов или символов, а позволяет непосредственно обратиться к любому байту файла.

Еще один особенный класс **`Console`**, не создающий поток, выполняет ввод-вывод, связанный с консолью.

Этот обзор классов ввода/вывода немного проясняет положение, но не объясняет, как их использовать. Перейдем к рассмотрению реальных ситуаций.

5.1. Консольный ввод/вывод

Для вывода на консоль мы всегда использовали метод **`println()`** класса **`PrintStream`**, никогда не определяя экземпляры этого класса. Мы просто использовали статическое поле **`out`** класса **`System`**, которое является объектом класса **`PrintStream`**. Исполняющая система Java связывает это поле с консолью.

Кстати говоря, если вам надоело писать **`System.out.println()`**, то вы можете определить новую ссылку на **`System.out`**, например:

```
PrintStream pr = System.out;
```

и писать просто **`pr.println()`**.

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin 1 с кодами **`'\u0000'`** - **`'\u00FF'`** при этом просто откидывается нулевой старший байт и выводятся байты **`'0x00'`** - **`'0xFF'`**. Для кодов кириллицы, которые

лежат в диапазоне **'\u0400' - '\u04FF'** кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали. Мы обсуждали это в *главе 4*.

Трудности с отображением кириллицы возникают, если вывод на консоль производится в кодировке, отличной от локали. Именно так происходит в русифицированных версиях MS Windows NT/2000. Обычно в них устанавливается локаль с кодовой страницей CP1251, а вывод на консоль происходит в кодировке CP866.

В этом случае надо заменить **PrintStream**, который не может работать с символьным потоком, на **PrintWriter** и "вставить переходное кольцо" между потоком символов Unicode и потоком байтов **System.out**, выводимых на консоль, в виде объекта класса **OutputStreamWriter**. В конструкторе этого объекта следует указать нужную кодировку, в данном случае, CP866. Все это можно сделать одним оператором:

```
PrintWriter pw = new PrintWriter(
     new OutputStreamWriter(System.out, "Cp866"), true);
```

Класс **PrintStream** буферизует выходной поток. Второй аргумент **true** его конструктора вызывает принудительный сброс содержимого буфера в выходной поток после каждого выполнения метода **println()**. Но после **print()** буфер не сбрасывается! Для сброса буфера после каждого **print()** надо писать **flush()**, как это сделано в листинге 5.2.

Методы класса **PrintWriter** по умолчанию не очищают буфер, а метод **print()** не очищает его в любом случае. Для очистки буфера используйте метод **flush()**.

После этого можно выводить любой текст методами класса **PrintWriter**, которые просто дублируют методы класса **PrintStream**, и писать, например,

```
pw.println("Это русский текст");
```

как показано в листинге 5.1.

Следует заметить, что конструктор класса **PrintWriter**, в котором задан байтовый поток, всегда неявно создает объект класса **OutputStreamWriter** с локальной кодировкой для преобразования байтового потока в символьный поток.

Ввод с консоли производится методами **read()** класса **InputStream** с помощью статического поля **in** класса **System**. С консоли идет поток байтов, полученных из scan-кодов клавиатуры. Эти байты должны быть

преобразованы в символы Unicode такими же кодовыми таблицами, как и при выводе на консоль. Преобразование идет по той же схеме - для правильного ввода кириллицы удобнее всего определить экземпляр класса **BufferedReader**, используя в качестве "переходного кольца" объект класса **InputStreamReader**:

```
BufferedReader br = new BufferedReader(
     new InputStreamReader(System.in, "Cp866"));
```

Класс **BufferedReader** переопределяет три метода **read()** своего суперкласса **Reader**. Кроме того, он содержит метод **readLine()**.

Метод **readLine()** возвращает строку типа **String**, содержащую символы входного потока, начиная с текущего, и заканчивая символом **'\n'** и/или **'\r'**. Эти символы-разделители не входят в возвращаемую строку. Если во входном потоке нет символов, то возвращается **null**.

В листинге 5.1 приведена программа, иллюстрирующая перечисленные методы консольного ввода/вывода.

Листинг 5.1. Консольный ввод/вывод

```
import java.io.*;
class PrWr{
public static void main(String[] args){
try{
BufferedReader br =
     new BufferedReader(new InputstreamReader(System.in,
"Cp866"));
PrintWriter pw = new PrintWriter(
     new OutputstreamWriter(System.out, "Cp866"), true);
String s = "Это строка с русским текстом";
System.out.println("System.out puts: " + s);
pw.println("PrintWriter puts: " + s) ;
int c = 0;
pw.println("Посимвольный ввод:");
while((c = br.read()) != -1)
pw.println((char)c);
pw.println("Построчный ввод:");
do{
s = br.readLine();
pw.println(s);
}while(!s.equals("q"));
}catch(Exception e){
System.out.println(e);
}
}
}
```

5.3. Форматированный вывод.

В классе **PrintStream** есть метод форматированного вывода:

```
PrintStream printf(String format, Object … args);
```

Строка символов **format** описывает шаблон для вывода данных, перечисленных в следующих аргументах метода, а также содержит надписи, которые должны появиться на консоли.

Например, тот же самый вывод на консоль, который мы можем сделать методом

```
System.out.println("x = " + x + ", y = " + y);
```

можно сделать методом

```
System.out.printf("x = %d, y = %d\n", x, y);
```

В строке формата мы пишем поясняющий текст **"x = , y = \n"**, который будет просто выводиться на консоль. В текст вставлены спецификации формата **"%d"**. На месте этих спецификаций во время вывода будут подставлены значения данных, перечисленных в следующих аргументах метода. Вместо первой спецификации появится значение первой переменной в списке, т.е. **x**, вместо второй – значение второй переменной - **y**. Если знак процента надо вывести в надписи, то его следует удвоить, например:

```
System.out.printf("Увеличение на %d%% процентов\n", x);
```

Если спецификаций окажется больше, чем данных в списке, то будет выброшено исключение класса **MissingFormatArgumentException**, если меньше, то последние данные, для которых не хватило спецификаций, просто не станут выводиться.

Если нужно изменить порядок вывода, то в спецификации можно явно задать порядковый номер выводимого аргумента, завершив его знаком доллара: **%1$d**, **%2$d**, **%3$d**. Например, если написать

```
System.out.printf("y = %2$d, x = %1$d\n", x, y);
```

то на консоль сначала будет выведено значение переменной **y**, а потом значение переменной **x**.

Можно несколько раз вывести одно и то же значение, например два раза вывести значение второй переменной:

```
System.out.printf("x = %2$d, y = %2$d\n", x, y);
```

Каждая спецификация начинается со знака процента, а заканчивается одной или двумя буквами **a**, **A**, **b**, **B**, **c**, **C**, **d**, **e**, **E**, **f**, **g**, **G**, **h**, **H**, **n**, **o**, **s**, **S**, **t**, **T**, **x**, **X**, показывающими тип выводимого данного. Буква **d**, использованная нами в примере, показывает, что соответствующее этой спецификации данное следует выводить на консоль как целое число в десятичной системе счисления.

Спецификации вывода целых чисел

Для спецификации **d** можно задать количество позиций, выделяемых для выводимого данного, например **%10d**. Если данное содержит меньше цифр, то оно будет выравнено по правому краю, а слева будут выведены пробелы. Если же данное содержит больше цифр, то поле будет расширено так, чтобы все число было выведено. В спецификации сразу же после знака процента можно поставить дефис: **%-12d**, и тогда число будет выравнено по левому краю поля, пробелы останутся справаю.

Если вместо пробелов надо вывести слева незначащие нули, то надо перед числом задающим ширину поля поставить ноль: **%010d**.

В больших числах группы по три цифры: тысячи, сотни тысяч, часто разделяют запятыми. Такую разбивку можно указать запятой: **%,10d**, но этот элемент форматирования сильно зависит от локальных установок, и надо проверить, как он действует на конкретной машине.

Положительные числа обычно выводятся без начального плюса. Если поставить знак плюс в спецификации: **%+10d**, то положительные числа будут выведены с плюсом, а отрицательные с минусом. Если вместо плюса оставить один пробел, то знак плюс у положительных чисел выводиться не будет, вместо него выведется пробел.

Целое число по спецификации **d** выводится в десятичной системе счисления. Спецификация **o** задает вывод числа в восьмеричной системе счисления, спецификации **x** и **X** - в шестнадцатеричной системе счисления. Если использовать **x**, то шестнадцатеричные цифры будут записаны малыми буквами, а если **X**, то заглавными буквами. В этих спецификациях можно использовать символ **#**. Тогда восьмеричное число будет выведено с начальным нулем, а шестнадцатеричное – с начальными символами **0x** или **0X**, как это принято при записи констант Java.

Спецификации вывода вещественных чисел

Для вывода вещественных чисел используются спецификации **a**, **A**, **e**, **E**, **f**, **g**, **G**. Спецификация **f** выводит вещественное число в десятичной системе счисления с фиксированной точкой, спецификации **e** и **E** - с плавающей точкой. Отличаются друг от друга только регистром выводимой буквы **E**. Спецификации **g**, **G** универсальны - они выводят короткие числа с фиксированной точкой, а длинные - с плавающей.

Во всех этих спецификациях можно использовать точность вывода чисел - количество цифр в его дробной части. Точность задается после точки в конце спецификации, например **%.8f**, **%+10.5E**, **%-12.4G**. По умолчанию точность равна 6 цифрам.

Спецификация **a**, **A** выводит числа в шестнадцатеричной системе счисления с плавающей точкой. В этих спецификациях точность задавать нельзя.

Спецификация вывода символов

Спецификация **c** выводит один символ в кодировке Unicode. В этой спецификации можно задавать только ширину поля и дефис для вывода символа с левым выравниванием в поле. Например: **%c**, **%6c**, **%-3c**.

Спецификации вывода строк

Спецификации **s**, **S** выводят строку символов. Соответствующий аргумент должен быть ссылкой на объект, который преобразуется в строку своим методом **toString()**. Если строка пуста, то выводится слово **null**. В этих спецификациях можно задавать только ширину поля и дефис для выравнивания строки по левой границе поля. Спецификация **S** преобразует все символы строки в верхний регистр. Например: **%6s**, **%-10s**, **%-50S**.

Спецификации вывода логических значений

Спецификации **b**, **B** выводят логическое значение словом **true** или **false**. В данной спецификации можно задавать только ширину поля и дефис для выравнивания по левой границе поля. Спецификация **B** выводит слово заглавными буквами: **TRUE**, **FALSE**. Например: **%b**, **%6B**, **%-10b**.

Спецификации вывода хеш-кода объекта

Спецификации **h**, **H** выводят хеш-код объекта в шестнадцатеричной системе счисления. В этой спецификации можно задавать только ширину поля и дефис.

Спецификации для вывода даты и времени

Спецификации **t**, **T** выводят дату и время по правилам заданным локальными установками. Дата и время для вывода по этим спецификациям должны быть заданы в секундах и миллисекундах в виде целого числа типа **long** или объекта типа **Long**, а также в виде объектов классов **Calendar** или **Date**.

После буквы **t** или **T** обязательно должна быть записана еще одна буква, указывающая объем выводимой информации: дата и время, только дата, только время, только часы и т.д. Например, метод

```
System.out.printf("Местное время: %tT\n",
Calendar.getInstance());
```

выведет только текущее время, без даты, используя локальные установки. В русской версии получим запись вида 12:36:14.

Метод

```
System.out.printf("Сейчас %tH часов\n", Calendar.getInstance());
```

выведет только часы, например 12. Спецификация **%tM** выведет только минуты, а спецификация **%tS** - только секунды.

Мы можем отформатировать время по-своему, написав, например, метод

```
System.out.printf("Местное время: %1$tH часов, %1$tM минут\n",
     Calendar.getInstance());
```

Кроме этого, по спецификации **%ts** можно получить время в секундах, начиная с 1 января 1970 года, а по спецификации **%tQ** - в миллисекундах, осчитанных от той же даты.

Дату в формате 10/24/08 можно получить по спецификации **%tD**, а в виде 2008-10-24 - по спецификации **%tF**. Ни та, ни другая формы не соответствуют российским стандартам. Привычный для нас вид представления даты - 24.10.08 - можно получить методом

```
System.out.printf("Сегодня %1$td.%1$tm.%1$ty\n",
     Calendar.getInstance());
```

Спецификация **%td** дает день месяца всегда двумя цифрами. Если вы хотите записать первые дни месяца одной цифрой, то используйте спецификацию **%te**.

Спецификация **%ty** записывает только две последние цифры года. Полная запись года четырьмя цифрами получится по спецификации **%tY**.

Название месяца, записанное полным словом, можно получить по спецификации **%tB**, а записанное сокращением из первых трех букв - по спецификации **%tb** или **%th**.

День недели полным словом можно получить по спецификации **%tA**, а сокращением из трех букв - по спецификации **%ta**.

Наконец, день недели, дату и время полностью можно получить по спецификации **%tc**.

5.3. Файловый ввод/вывод

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помощью классов **FileInputStream** и **FileOutputStream**. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Но много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байтов в символы и обратно.

Чтобы облегчить это преобразование, в пакет **java.io** введены классы **FineReader** и **FileWriter**. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла - байтовые. Это происходит потому, что данные классы расширяют классы **InputStreamReader** и **OutputStreamWriter**, соответственно, значит, содержат "переходное кольцо" внутри себя.

Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех четырех файловых потоков задается имя файла в виде строки типа **String** или ссылка на объект класса **File**. Конструкторы не только создают объект, но и отыскивают файл и открывают его. Например:

```
FileInputStream fis = new FileInputStream("PrWr.java");
FileReader fr = new FileReader("D:\\jdkl.3\\src\\PrWr.java");
```

При неудаче выбрасывается исключение класса **FileNotFoundException**, но конструктор класса **FileWriter** выбрасывает более общее исключение **IOException**.

После открытия выходного потока типа **FileWriter** или **FileOutputStream** содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла, и в том и в другом классе предусмотрен конструктор с двумя аргументами. Если второй аргумент равен **true**, то происходит дозапись в конец файла, если **false**, то файл заполняется новой информацией. Например:

```
FileWriter fw = new FileWriter("ch5.txt", true);
FileOutputStream fos = new FileOutputStream(
     "D:\\samples\\newfile.txt");
```

Содержимое файла, открытого на запись конструктором с одним аргументом, стирается.

Сразу после выполнения конструктора можно читать файл:

```
fis.read(); fr.read();
```

или записывать в него:

```
fos.write((char)c); fw.write((char)c);
```

По окончании работы с файлом поток следует закрыть методом **close()**.

Преобразование потоков в классах **FileReader** и **FileWriter** выполняется по кодовым таблицам установленной на компьютере локали. Для правильного ввода кирилицы надо применять **FileReader**, а не **FileInputStream**. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять "переходное кольцо" вручную, как это делалось для консоли, например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R"));
```

Байтовый поток **fis** определен выше.

5.3.1. Получение свойств файла

В конструкторах классов файлового ввода/вывода, описанных в предыдущем разделе, указывалось имя файла в виде строки. При этом оставалось неизвестным, существует ли файл, разрешен ли к, нему доступ, какова длина файла.

Получить такие сведения можно от предварительно созданного экземпляра класса **File**, содержащего сведения о файле. В конструкторе этого класса

```
File(String filename)
```

указывается путь к файлу или каталогу, записанный по правилам операционной системы. В UNIX имена каталогов разделяются наклонной чертой /, в MS Windows - обратной наклонной чертой \, в Apple Macintosh - двоеточием :. Этот символ содержится в системном свойстве **`file.separator`**. Путь к файлу предваряется префиксом. В UNIX это наклонная черта, в MS Windows - буква раздела диска, двоеточие и обратная наклонная черта. Если префикса нет, то путь считается относительным и к нему прибавляется путь к текущему каталогу, который хранится в системном свойстве **`user.dir`**.

Конструктор не проверяет, существует ли файл с таким именем, поэтому после создания объекта следует это проверить логическим методом **`exists()`**.

Класс **`File`** содержит около сорока методов, позволяющих узнать различные свойства файла или каталога.

Прежде всего, логическими методами **`isFile()`**, **`isDirectory()`** можно выяснить, является ли путь, указанный в конструкторе, путем к файлу или каталогу.

Для каталога можно получить его содержимое - список имен файлов и подкаталогов - методом **`list()`**, возвращающим массив строк **`String[]`**. Можно получить такой же список в виде массива объектов класса **`File[]`** методом **`listFiles()`**. Можно выбрать из списка только некоторые файлы, реализовав интерфейс **`FileNameFilter`** и обратившись к методу

`list(FileNameFilter filter)` или **`listFiles(FileNameFilter filter)`**.

Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом **`mkdir()`**. Этот метод возвращает **`true`**, если каталог удалось создать. Логический метод **`mkdirs()`** создает еще и все несуществующие каталоги, указанные в пути.

Пустой каталог удаляется методом **`delete()`**.

Для файла можно получить его длину в байтах методом **`length()`**, время последней модификации в секундах с 1 января 1970 г. методом **`lastModified()`**. Если файл не существует, эти методы возвращают нуль.

Логические методы **`canRead()`**, **`canWrite()`**, **`canExecute()`** показывают права доступа к файлу или каталогу, а методы - **`setReadable()`**, **`setWritable()`**, **`setExecutable()`** - позволяют их установить.

Файл можно переименовать логическим методом **renameTo(File *newName*)** или удалить логическим методом **delete()**. Эти методы возвращают **true**, если операция прошла удачно.

Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом **createNewFile()**, возвращающим **true**, если файл не существовал, и его удалось создать, и **false**, если файл уже существовал.

Статическими методами

```
createTempFile(String prefix, String suffix, File tmpDir);
createTempFile(String prefix, String suffix);
```

можно создать временный файл с именем **prefix** и расширением **suffix** в каталоге **tmpDir** или каталоге, указанном в системном свойстве **java.io.tmpdir**. Имя **prefix** должно содержать не менее трех символов. Если **suffix == null**, то файл получит суффикс **.tmp**.

Перечисленные методы возвращают ссылку типа **File** на созданный файл. Если обратиться к методу **deleteOnExit()**, то по завершении работы JVM временный файл будет уничтожен.

Несколько методов **get*Xxx*()** возвращают имя файла, имя каталога и другие сведения о пути к файлу. Эти методы полезны в тех случаях, когда ссылка на объект класса **File** возвращается другими методами и нужны сведения о файле.

В листинге 5.2 показан пример использования класса **File**.

Листинг 5.2. Определение свойств файла и каталога

```
import java.io.*;
class FileTest{
public static void main(String[] args) throws IOException{
PrintWriter pw = new PrintWriter(
     new OutputStreamWriter(System.out, "Cp866"), true);
File f = new File("FileTest.Java");
pw.println();
pw.println("Файл \"" + f.getName() + "\" " +
(f.exists()?"":"не ") + "существует");
pw.println("Вы " + (f.canRead()?"":"не ") + "можете читать
файл");
pw.println("Вы " + (f.canWrite()?"":"не ") +
     "можете записывать в файл");
pw.println("Длина файла " + f.length() + " б");
pw.println() ;
File d = new File(" D:\\jdkl.3\\MyProgs ");
```

```
pw.println("Содержимое каталога:");
if(d.exists() && d.isDirectory()) {
String[] s = d.list();
for (int i = 0; i < s.length; i++)
pw.println(s[i]);
}
}
}
```

5.3.2. Буферизованный ввод/вывод

Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область - буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Классы файлового ввода/вывода не занимаются буферизацией. Для этой цели есть четыре специальных класса **`BufferedXxx`**, перечисленных выше. Они присоединяются к потокам ввода/вывода как "переходное кольцо", например:

```
BufferedReader br = new BufferedReader(isr);
BufferedWriter bw = new BufferedWriter(fw);
```

Потоки **`isr`** и **`fw`** определены выше.

Программа листинга 5.3 читает текстовый файл, написанный в кодировке CP866, и записывает его содержимое в файл в кодировке KOI8_R. При чтении и записи применяется буферизация. Имя исходного файла задается в командной строке параметром **`args[0]`**, имя копии - параметром **`args[1]`**.

Листинг 5.3. Буферизованный файловый ввод/вывод

```
import java.io.*;
class DOStoUNIX{
public static void main(String[] args) throws IOException{
if (args.length != 2){
System.err.println("Usage: DOStoUNIX Cp866file KOI8_Rfile");
System.exit(0);
}
BufferedReader br = new BufferedReader(
new InputStreamReader(
new FileInputStream(args[0]), "Cp866"));
BufferedWriter bw = new BufferedWriter(
new OutputStreamWriter(
new FileOutputStreamtargs[1]), "KOI8_R"));
int c = 0;
while ((c = br.readO) != -1)
bw.write((char)c);
```

```
br.closeO; bw.close();
System.out.println("The job's finished.");
}
}
```

5.3.3. Поток простых типов Java

Класс **DataOutputStream** позволяет записать данные простых типов Java в выходной поток байтов методами

```
writeBoolean(boolean b)
writeByte(int b)
writeShort(int h)
writeChar(int c)
writeInt(int n)
writeLong(long l)
writeFloat(float f)
writeDouble(double d).
```

Кроме того, метод **writeBytes(String s)** записывает каждый символ строки **s** в один байт, отбрасывая старший байт кодировки каждого символа Unicode, а метод **writeChars(String s)** записывает каждый символ строки **s** в два байта, первый байт - старший байт кодировки Unicode, так же, как это делает метод **writeChar()**.

Еще один метод **writeUTF(String s)** записывает строку **s** в выходной поток в кодировке UTF-8. Надо пояснить эту кодировку.

Кодировка UTF-8

Запись потока в байтовой кодировке вызывает трудности с использованием национальных символов, запись потока в Unicode увеличивает длину потока в два раза. Кодировка UTF-8 (Universal Transfer Format) является компромиссом. Символ в этой кодировке записывается одним, двумя или тремя байтами.

Символы Unicode из диапазона **'\u0000' - '\u007F'**, в котором лежит английский алфавит, записываются одним байтом, старший байт просто отбрасывается.

Символы Unicode из диапазона **'\u0080' - '\u07FF'**, в котором лежат наиболее распространенные символы национальных алфавитов, записываются двумя байтами следующим образом: символ Unicode с кодировкой **00000xxxxxyyyyyy** записывается как **110xxxxx10yyyyyy**.

Остальные символы Unicode из диапазона **'\u0800' - '\uFFFF'** записываются тремя байтами по следующему правилу: символ Unicode с

кодировкой **xxxxyyyyyyzzzzzz** записывается как **1110xxxx10yyyyyy10zzzzzz**.

Такой странный способ распределения битов позволяет по первым битам кода узнать, сколько байтов составляет код символа, и правильно отсчитывать символы в потоке.

Так вот, метод **writeUTF(String s)** сначала записывает в поток в первые два байта потока длину строки **s** в кодировке UTF-8, а затем символы строки в этой кодировке. Читать эту запись потом следует парным методом **readUTF()** класса **DataInputStream**.

Класс **DataInputStream** преобразует входной поток байтов типа **InputStream**, составляющих данные простых типов Java, в данные этого типа. Такой поток, как правило, создается методами класса **DataOutputStream**. Данные из этого потока можно прочитать методами **readBoolean()**, **readByte()**, **readShort()**, **readChar()**, **readInt()**, **readLong()**, **readFloat()**, **readDouble()**, возвращающими данные соответствующего типа.

Кроме того, методы **readUnsignedByte()** и **readUnsignedShort()** возвращают целое типа **int**, в котором старшие три или два байта нулевые, а младшие один или два байта заполнены байтами из входного потока.

Метод **readUTF()**, двойственный методу **writeUTF()**, возвращает строку типа **String**, полученную из потока, записанного методом **writeUTF()**.

Еще один, статический, метод **readUTF(DataInput in)** делает то же самое со входным потоком **in**, записанным в кодировке UTF-8. Этот метод можно применять, не создавая объект класса **DataInputStream**.

Программа в листинге 5.4 записывает в файл **fib.txt** числа Фибоначчи, а затем читает этот файл и выводит его содержимое на консоль. Для контроля записываемые в файл числа тоже выводятся на консоль.

Листинг 5.4. Ввод/вывод данных

```
import java.io.*;
class DataPrWr{
public static void main(String[] args) throws IOException{
DataOutputStream dos = new DataOutputStream (
new FileOutputStream("fib.txt"));
int a = 1, b = 1, c = 1;
for(int k = 0; k < 40; k++){
System.out.print(b + " ");
dos.writeInt(b);
a = b; b = c; c = a + b;
```

```
}
dos.close();
System.out.println("\n");
DataInputStream dis = new DataInputStream (
new FileInputStream("fib.txt")) ;
while(true)
try{
a = dis.readInt();
System.out.print(a + " ">;
}catch(lOException e){
dis.close();
System.out.println("End of file");
System.exit(0);
}
}
}
```

Обратите внимание на то, что попытка чтения за концом файла выбрасывает исключение класса **IOException**, его обработка заключается в закрытии файла и окончании программы.

5.3.4. Прямой доступ к файлу

Если необходимо интенсивно работать с файлом, записывая в него данные разных типов Java, изменяя их, отыскивая и читая нужную информацию, то лучше всего воспользоваться методами класса **RandomAccessFile**.

В конструкторах этого класса

```
RandomAccessFile(File file, String mode);
RandomAccessFile(String fileName, String mode);
```

вторым аргументом **mode** задается режим открытия файла. Это может быть строка **"r"** - открытие файла только для чтения, или **"rw"** - открытие файла для чтения и записи, **"rwd"** - чтение и запись с немедленным обновлением источника данных и **"rws"** - чтение и запись с немедленным обновлением не только данных, но и метаданных.

Этот класс собрал все полезные методы работы с файлом. Он содержит все методы классов **DataInputStream** и **DataOutputStream**, кроме того, позволяет прочитать сразу целую строку методом **readLine()** и отыскать нужные данные в файле.

Байты файла нумеруются, начиная с 0, подобно элементам массива. Файл снабжен неявным *указателем* (file pointer) текущей позиции. Чтение и запись производится, начиная с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. Текущую позицию можно узнать методом **getFilePointer()**. Каждое чтение или

запись перемещает указатель на длину прочитанного или записанного данного. Всегда можно переместить указатель в новую позицию, **pos** методом **seek(long pos)**. Метод **seek(0)** перемещает указатель на начало файла.

В классе нет методов преобразования символов в байты и обратно по кодовым таблицам, поэтому он не приспособлен для работы с кириллицей.

6. Обработка исключительных ситуаций

Исключительные ситуации могут возникнуть во время выполнения (runtime) программы, прервав ее обычный ход. К ним относится деление на нуль, отсутствие загружаемого файла, отрицательный или вышедший за верхний предел индекс массива, переполнение выделенной памяти и масса других неприятностей, которые могут случиться в самый неподходящий момент.

Конечно, можно предусмотреть такие ситуации и застраховаться от них как-нибудь так:

```
if(something == wrong){
// Предпринимаем аварийные действия
}else{
// Обычный ход действий
}
```

Но при этом много времени уходит на проверки, и программа превращается в набор этих проверок. Посмотрите любую штатную производственную программу, написанную на языке C или Pascal, и увидите, что она на 2/3 состоит из таких проверок.

В объектно-ориентированных языках программирования принят другой подход. При возникновении исключительной ситуации исполняющая система создает объект определенного класса, соответствующего возникшей ситуации, содержащий сведения о том, что, где и когда произошло. Этот объект передается на обработку программе, в которой возникло исключение. Если программа не обрабатывает исключение, то объект возвращается обработчику по умолчанию исполняющей системы. Обработчик поступает очень просто: выводит на консоль сообщение о произошедшем исключении и прекращает выполнение программы.

Приведем пример. В программе листинга 6.1 может возникнуть деление на нуль, если запустить ее с аргументом 0. В программе нет никаких средств обработки такой исключительной ситуации.

Листинг 6.1. Программа без обработки исключений

```
class SimpleExt {
```

```
public static void main(String[] args){
int n = Integer.parseInt(args[0]);
System.out.println("10 / n = " + (10 / n));
System.out.println("After all actions");
}
}
```

Программа **SimpleExt** запускается три раза. Первый раз аргумент **args[0]** равен 5 и программа выводит результат: **"10/ n = 2"**. После этого появляется второе сообщение: **"After all actions"**.

Второй раз аргумент равен 0, и вместо результата мы получаем сообщение о том, что в подпроцессе **"main"** произошло исключение класса **ArithmeticException** вследствие деления на нуль: **"/ by zero"**. Далее уточняется, что исключение возникло при выполнении метода main класса **SimpleExt**, а в скобках указано, что действие, в результате которого возникла исключительная ситуация, записано в четвертой строке файла **SimpleExt.java**. Выполнение программы прекращается, заключительное сообщение не появляется.

Третий раз программа запущена вообще без аргумента. В массиве **args[]** нет элементов, его длина равна нулю, а мы пытаемся обратиться к элементу **args[0]**. Возникает исключительная ситуация класса **ArrayIndexOutOfBoundsException** вследствие действия, записанного в третьей строке файла **SimpleExt.java**. Выполнение программы прекращается, обращение к методу printino не происходит.

6.1. Блоки перехвата исключения

Мы можем перехватить и обработать исключение в программе. При описании обработки применяется бейсбольная терминология. Говорят, что исполняющая система или программа "выбрасывает" (throws) объект-исключение. Этот объект "пролетает" через всю программу, появившись сначала в том методе, где произошло исключение, а программа в одном или нескольких местах пытается (try) его "перехватить" (catch) и обработать. Обработку можно сделать полностью в одном месте, а можно обработать исключение в одном месте, выбросить снова, перехватить в другом месте и обрабатывать дальше.

Мы уже много раз сталкивались с необходимостью обрабатывать различные исключительные ситуации, но не делали этого, потому что не хотели отвлекаться от основных конструкций языка. Хорошо написанные объектно-ориентированные программы обязательно должны обрабатывать все возникающие в них исключительные ситуации.

Для того чтобы попытаться (try) перехватить (catch) объект-исключение, надо весь код программы, в котором может возникнуть исключительная ситуация, охватить оператором **try{}catch(){}**. Каждый блок **catch(){}** перехватывает исключение только одного типа, того, который указан в его аргументе. Но можно написать несколько блоков **catch(){}** для перехвата нескольких типов исключений.

Например, мы знаем, что в программе листинга 6.1 могут возникнуть исключения двух типов. Напишем блоки их обработки, как это сделано в листинге 6.2.

Листинг 6.2. Программа с блоками обработки исключений

```
class SimpleExt1{
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
System.out.println(" 10 / n = " + (10 / n));
Systtem.out.println("After results output");
}catch(ArithmeticException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
System.out.println("From Array.Exc.catch: "+arre);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}
```

В программу листинга 6.1 вставлен блок **try{}** и два блока перехвата **catch(){}** для каждого типа исключений. Обработка исключения здесь заключается просто в выводе сообщения и содержимого объекта-исключения, как оно представлено методом **toString()** соответствующего класса-исключения.

После блоков перехвата вставлен еще один, необязательный блок **finally{}**. Он предназначен для выполнения действий, которые надо выполнить обязательно, чтобы ни случилось. Все, что написано в этом блоке, будет выполнено и при возникновении исключения, и при обычном ходе программы, и даже если выход из блока **try{}** осуществляется оператором **return**.

Если в операторе обработки исключений есть блок **finally{}**, то блок **catch(){}** может отсутствовать, т.е. можно не перехватывать исключение, но при его возникновении все-таки проделать какие-то обязательные действия.

Кроме блоков перехвата в листинге 6.2 после каждого действия делается трассировочная печать, чтобы можно было проследить за порядком выполнения программы. Программа запущена три раза: с аргументом 5, с аргументом 0 и вообще без аргумента.

После первого запуска, при обычном ходе программы, выводятся все сообщения.

```
After parseInt()
10 / n = 2
After result output
From finally
After all actions
```

После второго запуска, приводящего к делению на нуль, управление сразу же передается в соответствующий блок **`catch(ArithmeticException ae){}`**, потом выполняется то, что написано в блоке **`finally{}`**.

```
After parseInt()
From Arithm.Exc. catch: java.lang.AritmeticException: / by zero
From finally
After all actions
```

После третьего запуска управление после выполнения метода **`parseInt()`** передается в другой блок **`catch(ArraylndexOutOfBoundsException arre){}`**, затем в блок **`finally{}`**.

```
From Array.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions
```

Обратите внимание, что во всех случаях - и при обычном ходе программы, и после этих обработок - выводится сообщение **`After all actions`**. Это свидетельствует о том, что выполнение программы не прекращается при возникновении исключительной ситуации, как это было в программе листинга 6.1, а продолжается после обработки и выполнения блока **`finally{}`**.

При записи блоков обработки исключений надо совершенно четко представлять себе, как будет передаваться управление во всех случаях.

Интересно, что пустой блок **`catch(){}`**, в котором между фигурными скобками нет ничего, даже пробела, тоже считается обработкой исключения и приводит к тому, что выполнение программы не прекратится. Именно так мы "обрабатывали" исключения в предыдущих главах.

Немного выше было сказано, что выброшенное исключение "пролетает" через всю программу. Что это означает? Изменим программу листинга 6.2, вынеся деление в отдельный метод **f()**. Получим листинг 6.3.

Листинг 6.3. Выбрасывание исключения из метода

```
class SimpleExt2{
private static void f(int n){
System.out.println(" 10 / n = " + (10 / n));
}
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt());
f(n);
System.out.println("After results output");
}catch(ArithmeticException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
System.out.println("From Array.Exc. catch: "+arre);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}
```

Откомпилировав и запустив программу листинга 6.3, убедимся, что вывод программы не изменился, он такой же, как на рис. 6.2. Исключение, возникшее при делении на нуль в методе **f()**, "пролетело" через этот метод, "вылетело" в метод **main()**, там перехвачено и обработано.

6.2. Часть заголовка метода - **throws**

То обстоятельство, что метод не обрабатывает возникающее в нем исключение, а выбрасывает (throws) его, следует отмечать в заголовке метода служебным словом **throws** и указанием класса исключения:

```
private static void f(int n) throws ArithmeticException{
System.out.println(" 10 / n = " + (10 / n)) ;
}
```

Почему же мы не сделали это в листинге 6.3? Дело в том, что спецификация JLS делит все исключения на *проверяемые*, те, которые проверяет компилятор, и *непроверяемые*. При проверке компилятор замечает необработанные в методах и конструкторах исключения и считает ошибкой отсутствие в заголовке таких методов и конструкторов пометки **throws**. Именно для предотвращения подобных ошибок мы в предыдущих главах вставляли в листинги блоки обработки исключений.

Так вот, исключения класса **RuntimeException** и его подклассов, одним из которых является **ArithmeticException**, непроверяемые, для них пометка **throws** необязательна. Еще одно большое семейство непроверяемых исключений составляет класс **Error** и его расширения.

Почему компилятор не проверяет эти типы исключений? Причина в том, что исключения класса **RuntimeException** свидетельствуют об ошибках в программе, и единственно разумный метод их обработки - исправить исходный текст программы и перекомпилировать ее. Что касается класса **Error**, то эти исключения очень трудно локализовать и на стадии компиляции невозможно определить место их появления.

Напротив, возникновение проверяемого исключения показывает, что программа недостаточно продумана, не все возможные ситуации описаны. Такая программа должна быть доработана, о чем и напоминает компилятор.

Если метод или конструктор выбрасывает несколько исключений, то их надо перечислить через запятую после слова **throws**. Заголовок метода **main()** листинга 6.1, если бы исключения, которые он выбрасывает, не были бы объектами подклассов класса **RuntimeException**, следовало бы написать так:

```
public static void main(String[] args)
     throws ArithmeticException, ArrayIndexOutOfBoundsException{
     // Содержимое метода
}
```

Перенесем теперь обработку деления на нуль в метод **f()** и добавим трассировочную печать, как это сделано в листинге 6.4.

Листинг 6.4. Обработка исключения в методе

```
class SimpleExt3{
private static void f(int n){ // throws ArithmeticException{
try{
System.out.println(" 10 / n = " + (10 / n));
System.out.println("From f() after results output");
}catch(ArithmeticException ae)(
System.out.println("From f() catch: " + ae) ;
// throw ae;
}finally{
System.out.println("From f() finally");
}
}
public static void main(String[] args){
try{
inf n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
f(n);
System.out.println("After results output");
```

```
}catch(ArithmeticException ae){
System.out.println("From Arithm.Exc. catch: "+ae);
}catch(ArrayIndexOutOfBoundsException arre){
System.out.println("From Array.Exc. catch: "+arre);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}
```

Внимательно проследите за передачей управления и заметьте, что исключение класса **ArithmeticException** уже не выбрасывается в метод **main()**.

Оператор **try{}catch(){}** в методе **f()** можно рассматривать как вложенный в оператор обработки исключений в методе **main()**.

При необходимости исключение можно выбросить оператором **throw ae**. В листинге 6.4 этот оператор показан как комментарий. Уберите символы комментария **//**, перекомпилируйте программу и посмотрите, как изменится ее вывод.

6.3. Оператор **throw**

Этот оператор очень прост: после слова **throw** через пробел записывается объект класса-исключения. Достаточно часто он создается прямо в операторе **throw**, например:

```
throw new ArithmeticException();
```

Оператор можно записать в любом месте программы. Он немедленно выбрасывает записанный в нем объект-исключение и дальше обработка этого исключения идет как обычно, будто бы здесь произошло деление на нуль или другое действие, вызвавшее исключение класса **ArithmeticException**.

Итак, каждый блок **catch(){}** и перехватывает один определенный тип исключений. Если требуется одинаково обработать несколько типов исключений, то можно воспользоваться тем, что классы-исключения образуют иерархию. Изменим еще раз листинг 6.2, получив листинг 6.5.

Листинг 6.5. Обработка нескольких типов исключений

```
class SimpleExt4{
public static void main(String[] args){
try{
int n = Integer.parseInt(args[0]);
System.out.println("After parseInt()");
```

```
System.out.println(" 10 / n = " + (10 / n) ) ;
System.out.println("After results output");
}catch(RuntimeException ae){
System.out.println("From Run.Exc. catch: "+ae);
}finally{
System.out.println("From finally");
}
System.out.println("After all actions");
}
}
```

В листинге 6.5 два блока **`catch(){}`** заменены одним блоком, перехватывающим исключение класса **`RuntimeException`**.

```
After parseInt()
10 / n = 2
After result output
From finally
After all actions

After parseInt()
From Run.Exc. catch: java.lang.ArithmeticException: / by zero
From finally
After all actions

From Run.Exc. catch: java.lang.ArrayIndexOutOfBoundsException
From finally
After all actions
```

Как видно, этот блок перехватывает оба исключения. Почему? Потому что это исключения подклассов класса **`RuntimeException`**.

Таким образом, перемещаясь по иерархии классов-исключений, мы можем обрабатывать сразу более или менее крупные совокупности исключений. Рассмотрим подробнее иерархию классов-исключений.

6.4. Иерархия классов-исключений

Все классы-исключения расширяют класс **`Throwable`** - непосредственное расширение класса **`Object`**.

У класса **`Throwable`** и у всех его расширений по традиции два конструктора:

- **`Throwable()`** - конструктор по умолчанию;
- **`Throwable(String message)`** - создаваемый объект будет содержать произвольное сообщение **`message`**.

Записанное в конструкторе сообщение можно получить затем методом **`getMessage()`**. Если объект создавался конструктором по умолчанию, то данный метод возвратит **`null`**.

Метод **`toString()`** возвращает краткое описание события, именно он работал в предыдущих листингах.

Три метода выводят сообщения обо всех методах, встретившихся по пути "полета" исключения:

- **`printStackTrace()`** - выводит сообщения в стандартный вывод, как правило, это консоль;
- **`printStackTrace(PrintStream stream)`** - выводит сообщения в байтовый поток **`stream`**;
- **`printStackTrace(PrintWriter stream)`** - выводит сообщения в символьный поток **`stream`**.

У класса **`Throwable`** два непосредственных наследника - классы **`Error`** и **`Exception`**. Они не добавляют новых методов, а служат для разделения классов-исключений на два больших семейства - семейство классов-ошибок (error) и семейство собственно классов-исключений (exception).

Классы-ошибки, расширяющие класс **`Error`**, свидетельствуют о возникновении сложных ситуаций в виртуальной машине Java. Их обработка требует глубокого понимания всех тонкостей работы JVM. Ее не рекомендуется выполнять в обычной программе. Не советуют даже выбрасывать ошибки оператором **`throw`**. Не следует делать свои классы-исключения расширениями класса **`Error`** или какого-то его подкласса.

Имена классов-ошибок, по соглашению, заканчиваются словом **`Error`**.

Классы-исключения, расширяющие класс **`Exception`**, отмечают возникновение обычной нештатной ситуации, которую можно и даже нужно обработать. Такие исключения следует выбросить оператором **`throw`**. Классов-исключений очень много, более двухсот. Они разбросаны буквально по всем пакетам J2SDK. В большинстве случаев вы способны подобрать готовый класс-исключение для обработки исключительных ситуаций в своей программе. При желании можно создать и свой класс-исключение, расширив класс **`Exception`** или любой его подкласс.

Среди классов-исключений выделяется класс **`RuntimeException`** - прямое расширение класса **`Exception`**. В нем и его подклассах отмечаются исключения, возникшие при работе JVM, но не столь серьезные, как ошибки. Их можно обрабатывать и выбрасывать, расширять своими классами, но лучше доверить это JVM, поскольку чаще всего это просто ошибка в

программе, которую надо исправить. Особенность исключений данного класса в том, что их не надо отмечать в заголовке метода пометкой **throws**.

Имена классов-исключений, по соглашению, заканчиваются словом **Exception**.

6.5. Порядок обработки исключений

Блоки **catch(){}** перехватывают исключения в порядке написания этих блоков. Это правило приводит к интересным результатам.

В листинге 6.2 мы записали два блока перехвата **catch(){}** и оба блока выполнялись при возникновении соответствующего исключения. Это происходило потому, что классы-исключения **ArithmeticException** и **ArrayIndexOutOfBoundsException** находятся на разных ветвях иерархии исключений. Иначе обстоит дело, если блоки **catch(){}** перехватывают исключения, расположенные на одной ветви. Если в листинге 6.4 после блока, перехватывающего **RuntimeException**, поместить блок, обрабатывающий выход индекса за пределы:

```
try{
     // Операторы, вызывающие исключения
}catch(RuntimeException re){
     // Какая-то обработка
}catch(ArrayindexOutofBoundsException ae){
     // Никогда не будет выполнен!
}
```

то он не будет выполняться, поскольку исключение этого типа является, к тому же, исключением общего типа **RuntimeException** и будет перехватываться Предыдущим блоком **catch(){}**.

6.6. Создание собственных исключений

Прежде всего, нужно четко определить ситуации, в которых будет возникать ваше собственное исключение, и подумать, не станет ли его перехват невольно перехватывать также и другие, не учтенные вами исключения.

Потом надо выбрать суперкласс создаваемого класса-исключения. Им может быть класс **Exception** или один из его многочисленных подклассов.

После этого можно написать класс-исключение. Его имя, по соглашению, должно завершаться словом **Exception**. Как правило, этот класс состоит только из двух конструкторов и переопределения методов **toString()** и **getMessage()**.

Рассмотрим простой пример. Пусть метод **handle(int cipher)** обрабатывает арабские цифры 0 - 9, которые передаются ему в аргументе **cipher** типа **int**. Мы хотим выбросить исключение, если аргумент **cipher** выходит за диапазон 0 - 9.

Прежде всего, убедимся, что такого исключения нет в иерархии классов **Exception**. Ко всему прочему, не отслеживается и более общая ситуация попадания целого числа в какой-то диапазон. Поэтому будем расширять наш класс. Назовем его **cipherException**, прямо от класса **Exception**. Определим класс **cipherException**, как показано в листинге 6.6, и используем его в классе **ExceptDemo**.

Листинг 6.6. Создание класса-исключения

```
class CipherException extends Exception{
private String msg;
CipherException(){ msg = null;}
CipherException(String s){ msg = s;}
public String toString(){
return "CipherException (" + msg + ");
}
}
class ExceptDemo(
static public void handle(int cipher) throws CipherException{
System.out.println("handle()'s beginning");
if(cipher < 0 || cipher > 9)
     throw new CipherException("" + cipher);
System.out.println("handle()'s ending");
}
public static void main(String[] args){
try{
handle(1) ;
handle(10);
}catch(CipherException ce){
System.out.println("caught " + ce) ;
ce.printStackTrace();
}
}
}
```

Вывод этой программы:

```
handle()’s beginning
handle()’s ending
handle()’s beginning
caught CipherException (10)
CipherException (10)
     at ExceptDemo.handle(ExceptDemo.java:13)
     at ExceptDemo.main(ExceptDemo.java:20)
```

6.7. Заключение

Обработка исключительных ситуаций стала сейчас обязательной частью объектно-ориентированных программ. Применяя методы классов J2SDK и других пакетов, обращайте внимание на то, какие исключения они выбрасывают, и обрабатывайте их. Исключения резко меняют ход выполнения программы, делают его запутанным. Не увлекайтесь сложной обработкой, помните о принципе KISS.

Например, из блока **`finally{}`** можно выбросить исключение и обработать его в другом месте. Подумайте, что произойдет в этом случае с исключением, возникшем в блоке **`try{}`**? Оно нигде не будет перехвачено и обработано.