МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования «Казанский (Приволжский) федеральный университет»

Набережночелнинский институт (филиал)

Кафедра Бизнес-информатики и математических методов в экономике

Основы программирования на языке Java

Учебно-методическое пособие

Набережные Челны
2019 г.

УДК 681.3 (076)
ББК 73я7

Печатается по решению учебно-методической комиссии экономического отделения Набережночелнинского института (филиала) федерального государственного автономного образовательного учреждения высшего образования «Казанский (Приволжский) федеральный университет», от «15» марта 2019г. (протокол №7)

Рецензенты:
Доктор физ.-мат. наук, профессор А.Г. Исавнин
Доктор экономических наук, профессор А.Н. Макаров

Бадриев А.И., Карамышев А.Н. Основы программирования на языке Java: учебно-методическое пособие / А.И. Бадриев, А.Н. Карамышев. – Набережные Челны: Изд-во Набережночелнинского института КФУ, 2019. – 39 с.

Учебно-методическое пособие содержит последовательное изложение базовых понятий теории программирования JAVA. Основы программирования на языке JAVA. Подробно изложены: использование массивов; управление выполнением программы; классы; работа с файлами.

Учебно-методическое пособие предназначено для использования в учебном процессе студентами технических направлений в экономике и экономического отделения дневной, заочной и дистанционной форм обучения.

© Бадриев А.И., Карамышев А.Н., 2019
© НЧИ КФУ, 2019
© Кафедра Бизнес-информатики и математических методов в экономике, 2019 г.

Оглавление

Введение ... 4
1. Использование массивов ... 4
2. Управление выполнением программы ... 8
2.1. Условные операторы if-else ... 8
2.2. Оператор break ... 9
2.3. Оператор switch ... 10
2.4. Оператор return ... 14
2.5. Циклы ... 14
2.5.1. Оператор цикла while ... 14
2.5.2. Оператор цикла do-while ... 15
2.5.3. Оператор цикла for ... 15
2.6. Оператор запятая ... 16
2.7. Оператор continue ... 16
3. Классы ... 17
3.1. Переменные представителей (instance variables) ... 19
3.2. Создание объекта класса ... 19
3.3. Объявление методов ... 20
3.4. Вызов метода ... 20
3.5. Скрытие переменных представителей ... 21
3.6. Конструкторы ... 21
3.7. Совмещение методов ... 22
3.8. Ссылка this в конструкторах ... 23
3.9. Наследование классов ... 24
3.10. Ссылка super ... 24
3.11. Замещение методов ... 25
3.12. Динамическое назначение методов ... 26
3.13. Директива final ... 27
3.14. Метод Finalize ... 27
3.15. Директива static ... 27
3.16. Директива abstract ... 28
4. Работа с файлами ... 29
4.1. Классы Java для работы с потоками ... 30
4.2. Стандартные потоки ... 30
4.3. Базовые классы для работы с файлами и потоками ... 30
4.3.2. Класс OutputStream ... 31
4.3.3. Класс RandomAccesFile ... 31
4.3.4. Класс File ... 31
4.3.5. Класс FileDescriptor ... 32
4.3.6. Класс StreamTokenizer ... 32
4.4. InputStream ... 32
4.5. OutputStream ... 33
4.6. Файловые потоки ... 33
4.6.1. FileInputStream ... 33
4.6.2. FileOutputStream ... 34
4.7. Работа с файлами и каталогами при помощи класса File ... 35
4.7.1. Создание объекта класса File ... 36
4.7.2. Определение атрибутов файлов и каталогов ... 36

Введение

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

Три ключевых элемента объединились в технологии языка Java и сделали ее в корне отличной от всего, существующего на сегодняшний день.

- Java предоставляет для широкого использования свои аплеты — небольшие, надежные, динамичные, не зависящие от платформы активные сетевые приложения, встраиваемые в страницы Web. Аплеты Java могут настраиваться и распространяться потребителям с такой же легкостью, как любые документы HTML.
- Java предоставляет мощные объектно-ориентированные принципы разработки приложений, сочетая простой и знакомый синтаксис с надежной и удобной в работе средой разработки. Это позволяет широкому кругу программистов быстро создавать новые программы и новые аплеты.
- Java предоставляет программисту богатый набор классов объектов для ясного абстрагирования многих системных функций, используемых при работе с окнами, сетью и для ввода-вывода. Ключевая черта этих классов заключается в том, что они обеспечивают создание независимых от используемой платформы абстракций для широкого спектра системных интерфейсов.

1. Использование массивов

Для создания массива необходимо использовать квадратные скобки, расположив, их справа от имени массива или от типа элементов, из которых будет состоять массив, например:

```
int Numbers[]; int[] AnotherNumbers;
```

Допустимы оба варианта. При объявлении массивов в языке Java их размер не указывается. Приведенные выше две строки не вызывают

резервирования памяти для массива. Здесь просто создаются ссылки на массивы, которые нельзя использовать без инициализации.

Для того чтобы зарезервировать память для массива, необходимо создать соответствующие объекты с помощью оператора **new**, например:

```
int[] AnotherNumbers;
AnotherNumbers = new int[15];
```

В приведенной выше строке кода с помощью оператора **new** массиву AnotherNumbers выделяется память для хранения пятнадцати целых чисел. Нумерация элементов в массивах начинается с нуля.

Инициализацию элементов массива можно выполнить либо статически, либо динамически. В первом случае, инициализатор массива представляет собой список выражений разделенных запятыми и заключенный в фигурные скобки. Запятые отделяют друг от друга значения элементов массива. При таком способе создания массив будет содержать ровно столько элементов, сколько требуется для хранения значений, указанных в списке инициализации.

```
int[] ColorRed = {255, 255, 100, 0, 10};
```

В приведенной выше строке кода создается массив ColorRed из пяти элементов.

Динамическая инициализация выполняется с использованием индекса массива, например, в цикле:

```
int nInitialValue = 7; int[] AnotherNumbers;
AnotherNumbers = new int[15]; for(int i = 0; i < 15; i++)
{
AnotherNumbers[i] = nInitialValue;
}
```

Можно создавать массивы не только из переменных базовых типов, но и из произвольных объектов. Каждый элемент такого массива должен инициализироваться оператором **new**.

Массивы могут быть многомерными и несимметричными. В следующем примере создается массив массивов. В нулевом и первом элементе

создается массив из четырех чисел, а во втором - из восьми:

```
int[][] nDim = new int[5][10]; nDim[0] = new int [4]; nDim[1] = new int [4]; nDim[2]  = new int  [8];
```

Приведенный ниже код создает традиционную матрицу из шестнадцати элементов типа **double**, каждый из которых инициализируется нулем. Внутренняя реализация этой матрицы - массив массивов типа **double**.

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

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

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

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

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

```
m[2][3]);
System.out.println(m[3][0] +М М+ m[3][1] +М М+ m[3][2] +М ”+
m[3][3]);
```

Виртуальная машина Java строго следит за тем, чтобы вы случайно не записали или не попытались получить значения, выйдя за границы массива. Если же вы попытаетесь использовать в качестве индексов значения, выходящие за границы массива - отрицательные числа, либо числа, которые больше или равны количеству элементов в массиве, то получите сообщение об ошибке.

Массивы в языке Java являются объектами встроенного класса. Для этого класса существует возможность определить размер массива, обратившись к элементу данных класса с именем **length**, например:

```
int[] AnotherNumbers;
AnotherNumbers = new int[15];
for(int i = 0; i < AnotherNumbers.length; i++)
{
AnotherNumbers[i] = nInitialValue;
}
```

Для задания начальных значений многомерных массивов существует специальная форма инициализатора. В этом случае, во вложенных фигурных скобках указывается блок инициализации для элементов строки многомерного массива. Внутри инициализатора массива можно использовать не только литералы, но и выражения. Ниже показан пример класса, в котором находится максимальный элемент многомерного массива.

```
public class lr1 {
//Объявление и инициализация массива int ar[][] = {{1,2,3,4,5},
{6,7,60,9,10},
{11,12,13,14,15}};
//Конструктор по умолчанию public lr1() {
}
//Метод, возвращающий максимальное значение массива public int
```

```
getMax()
{
int max =0; for(int i=0 ; i<3; i++)
{
for(int j=0; j<5; j++)
{
if (ar[i][j] > max)
{ max = ar[i][j]; }
}
}
return max;
}
public static void main(String[]  args) { lr1 lr11 = new lr1();
System.out.println("Максимальное число =" + lr11.getMax());
}
}
```

2. Управление выполнением программы

2.1. Условные операторы if-else

В обобщенной форме этот оператор записывается следующим образом:

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

Раздел **else** необязателен. На месте «оператор1» или «оператор2» может стоять составной оператор, заключенный в фигурные скобки. «Логическое выражение» — это любое выражение, возвращающее значение типа **boolean**. Например,

```
int x;
if (x > 0) {
ProcessDataQ;
x+= n;
} else
```

```
waitForMoreData();
```

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

```
class A {
public static void main(String args[])
{
int month = 4;
String season;
if (month == 12 || month == 1 || month == 2) { season = "зима";
} else if (month ==3 || month == 4 || month == 5) { season = "весна";
} else if (month == 6 || month == 7 || month == 8) { season = "лето";
} else if (month == 9 || month == 10 || month == 11) { season = "осень";
}
System.out.println( "Сейчас " + season + ".");
}
}
```

После выполнения программы вы должны получить следующий результат:

Сейчас весна.

2.2. Оператор break

Этот оператор прекращает выполнение текущего блока и передает управление оператору, следующему за данным блоком. Для именования блоков в языке Java могут использоваться метки. Оператор **break** при работе с циклами и в операторах **switch** может использоваться без метки. В таком случае подразумевается выход из текущего блока. Можно использовать оператор **break** только для перехода за один из текущих вложенных блоков.

Например, в следующей программе имеется три вложенных блока, и у каждого своя уникальная метка. Оператор **break**, стоящий во внутреннем

блоке, вызывает переход на оператор, следующий за блоком b. При

этом пропускаются два оператора **println**.

```
class Break {
public static void main(String args[]) { boolean t = true; a:
{ b:
{ c:
{
System.out.println("Перед break ");
if (t) break b;
System.out.println("Не будет выполнено ");
}
System.out.println("Не будет выполнено ");
}
System.out.println("После b ");
```

В результате исполнения программы вы получите следующий результат: ***Перед break После b***

2.3. Оператор switch

Оператор **switch** обеспечивает способ переключения между различными частями программного кода в зависимости от значения одной переменной или выражения. Общая форма этого оператора следующая:

```
switch ( выражение ) { case значение 1:
break;
case значение 2:
break;
case значение n:
break;
default:
}
```

Результатом вычисления «выражения» может быть значение любого простого типа, при этом каждое из значений, указанных в операторах **case**,

должно быть совместимо по типу с выражением в операторе **switch**. Все эти значения должны быть уникальными литералами. Если же вы укажете в двух операторах **case** одинаковые значения, тогда будет ошибка.

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

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

```
class SwitchSeason {
public static void main(String args[]) {
int month = 4;
String season; switch (month) { case 12: case 1: case 2:
season = "Зима";
break;
case 3:
case 4:
case 5:
season = "Весна";
break;
case 6:
case 7:
case 8:
season = "Лето";
break;
case 9:

case 10:

case 11:

season = "Осень";

break;
```

default:season = "Неправильный
номер месяца";

} ".");

System.out.println("Апрель - это " +
season +

}

}

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

class WordCount {

// Задаем текстовую строку

static String text = "Основы программирования^" +

"на языке Java\n" +

"для начинающих^" +

"разработчиков^";

//Получаем длину строки static int len = text.length(); public static void main(String args[]) { boolean inWord = false;

int numChars = 0; //переменная для хранения количества символов в тексте int numWords = 0; //переменная для хранения количества слов в тексте int numLines = 0; //переменная для хранения количества строк в тексте //Организуем цикл по длине текстовой строки for (int i=0; i < len; i++) {

char c = text.charAt(i); //Преобразуем элемент текстовой строки в символ numChars++; //Увеличиваем на 1 счетчик символов

switch (c) { //Анализируем символы в текстовой строке

case '\n': numLines++; // Если символ перевода строки, то увеличиваем //счетчик строк на 1

case '\t': // Тоже самое

case ' ' : if (inWord) { //Если пробел увеличиваем счетчик слов на 1

```
numWords++; inWord = false;
          }
          break;
          default:  inWord = true;
          }
          }
          System.out.println("\t"  + numLines  +"\t" + numWords + "\t" + numChars);
//Выводим на консоль количество строк, слов и символов в текстовой строке
          }
          }
```

2.4. Оператор return

В любом месте программного кода метода можно поставить оператор **return**, который приведет к немедленному завершению работы и передаче управления коду, вызвавшему этот метод. Ниже приведен пример, иллюстрирующий использование оператора **return** для немедленного возврата управления.

```
class RetumDemo {
public static void main(String args[]) {
boolean t = true;
System^^^r'intlnC^ оператора return"); //Перед оператором return if (t)
return;
System.out.println("”Это не выполнится"); //Это не будет выполнено
}
}
```

2.5. Циклы

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

2.5.1. Оператор цикла while

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

[инициализация;] while (завершение) {

тело;

[итерация;] }

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

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

2.5.2. Оператор цикла do-while

Иногда возникает потребность выполнить тело цикла, по крайней мере, один раз - даже в том случае, когда логическое выражение с самого начала принимает значение **false**. Для таких случаев в Java используется циклическая конструкция do-while. Ее общая форма записи такова:

[инициализация;]

do {

тело;

[итерация;] } while (завершение);

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

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

2.5.3. Оператор цикла for

Общая форма записи оператора **for** следующая:

for (инициализация; завершение; итерация)

{

тело;

}

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

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

Следующий пример - вариант программы, ведущей обратный отсчет.

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

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

2.6. Оператор запятая

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

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

Вывод этой программы показывает, что цикл выполняется всего два раза. ***a = 1 b = 4 a = 2 b = 3***

2.7. Оператор continue

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

```
class ContinueDemo {
public static void main(String args[]) {
for (int i=0; i < 10; i++) {
System.out.print(i + " "); if (i % 2 == 0) continue;
```

```
System.out.println("");
}
}
}
```

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

01 2 3

4 5

5 7 8 9

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

```
class ContinueLabel { public static void main(String args[]) { outer: for (int
i=0; i < 10; i++) { for (int j = 0; j < 10; j++) {
if G > i) {
System.out.println(MM); continue outer;
}
System.out.print(" " + (i * j)); }
}
}
}
```

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

0

01 0 2 4 0 3 6 9 0 4 81216 0 51015 20 25 0 6 12 18 24 30 36 0 714 21 28 35 42 49 0 8 16 24 32 40 48 56 64 0 9 18 27 36 45 54 63 72 81

3. Классы

Базовым элементом объектно-ориентированного программирования в языке Java является класс. В этой главе Вы научитесь создавать и расширять свои собственные классы, работать с экземплярами этих классов и начнете

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

```
class Point { }
```

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

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

```
class имя_класса extends имя_суперкласса {
type переменная 1_объекта:
type переменная2_объекта:
type переменная^объекта:
type имяметода1(список_параметров) {
тело метода;
}
type имяметода2(список_параметров) { тело метода;
}
type имя методаM(список_параметров) { тело метода;
}
}
```

Ключевое слово extends указывает на то, что «имя_класса» - это подкласс класса «имя_суперкласса». Во главе классовой иерархии Java стоит

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

3.1. Переменные представителей (instance variables)

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

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

В качестве типа для переменных объектов можно использовать как любой из простых типов, так и классовые типы.

3.2. Создание объекта класса

С помощью оператора **new** создается экземпляр указанного класса и возвращается ссылка на созданный объект. Ниже приведен пример создания и присваивание переменной p экземпляра класса Point.

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

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

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

```
p2.y = 99;
System.out.println("x = " + p1.x + " y = " + p1.y);
System.out.println("x = " + p2.x + " y = " + p2.y);
} }
```

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

x = 10 y = 20 x = 42 y = 99

3.3. Объявление методов

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

тип имя_метода (список формальных параметров) { тело метода;

}

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

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

3.4. Вызов метода

В Java отсутствует возможность передачи параметров «по ссылке» на

простой тип. В Java все параметры простых типов передаются «по значению», а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке. Можно изменять содержимое того объекта, на который ссылается данная переменная.

3.5. Скрытие переменных представителей

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

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

3.6. Конструкторы

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

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

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

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

3.7. Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещением методов (**method overloading**). В качестве примера приведена версия класса Point, в которой совмещение методов использовано для определения альтернативного конструктора, который инициализирует координаты x и y значениями по умолчанию (-1).

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

Point() {
x = -1;
y = -1;
}

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

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

X = *-1 y = -1*

Решение о том, какой конструктор нужно вызвать в том или ином случае, принимается в соответствии с количеством и типом параметров, указанных в операторе **new**. Недопустимо объявлять в классе методы с

одинаковыми именами и сигнатурами. В сигнатуре метода не учитываются имена формальных параметров, учитываются лишь их типы и количество.

3.8. Ссылка this в конструкторах

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

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

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

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

```
class Point { int x, y;
Point(int x, int y) { this.x = x; this. y = y;
}
double distance(int x, int y) { int dx = this.x - x; int dy = this^ - y; return
Math.sqrt(dx*dx + dy*dy);
}
double distance(Point p) { return distance(p.x, p.y);
}
} class PointDist {
public static void main(String args[]) {
Point pi = new Point(0, 0);
Point p2 = new Point(30, 40);
System.out.println(Mp1 = " + pi.x + "," + pi.y); System.out.println("p2 = " +
p2.x + ", " + p2.y); System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(60, 80) = " + p1.distance(60, 80));
}
}
```

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

работы этой программы:***p1 = 0, 0***

p2 = 30, 40 p1.distance(p2) = 50.0 p1.distance(60, 80) = 100.0

3.9. Наследование классов

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование. Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и методы классов-предков. Классы-потомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс Point таким образом, чтобы включить в него третью координату z. class Point3D extends Point { int z;

```
Point3D(int x, int y, int z) { this.x = x; this.y = y; this.z = z;
}

Point3D() { this(-1,-1,-1);
}

}
```

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

3.10. Ссылка super

В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе. Вспомните, как во втором конструкторе мы использовали **this** для вызова первого конструктора того же класса. Аналогичным образом ключевое слово **super** позволяет обратиться непосредственно к конструктору суперкласса.

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

Вот результат работы этой программы: ***x = 10 y = 20 z = 30***

3.11. Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (пример PointDist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(int x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны заместить (**override**) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и совмещение (overloading), и замещение (overriding) метода distance.

```
class Point { int x, y;
Point(int x, int y) { this.x = x; this.y = y;
}
double distance(int x, int y) { int dx = this.x - x; int dy = this.y - y: return
Math,sqrt(dx*dx + dy*dy);
}
double distance(Point p) { return distance(p.x, p.y);
}
}
class Point3D extends Point { int z;
Point3D(int x, int y, int z) { super(x, y); this.z = z;
}
double distance(int x, int y, int z) { int dx = this.x - x; int dy = this.y - y; int dz
= this.z - z;
return Math.sqrt(dx*dx + dy*dy + dz*dz); }
double distance(Point3D other) { return distance(other.x, other.y, other.z);
}
double distance(int x, int y) { double dx = (this.x / z) - x; double dy = (this.y
/ z) - y; return Math.sqrt(dx*dx + dy*dy);
}
} class Point3DDist {
public static void main(String args[]) {
Point3D pi = new Point3D(30, 40, 10);
Point3D p2 = new Point3D(0, 0, 0);
Point p = new Point(4, 6);
System.out.println("p1 = " + pl.x + "," + pl.y + "," + pl.z);
System.out.println("p2 = " + p2.x + ", " + p2.y + ", " + p2.z); System.out.println("p = "
+ p.x + ", " + p.y); System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
System.out.println("p1.distance(p) = " + p1.distance(p));
}
}
```

Ниже приводится результат работы этой программы:

pi = 30, 40,10

p2 = 0, 0, 0

p = 4, 6

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

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

3.12. Динамическое назначение методов

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

```
class A { void callme() {
System.out.println("Вызван callme метод класса A");
}
class B extends A { void callme() {
System.out.println("Вызван callme метод класса B");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new B(); a.callme();
}
}
```

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

«Вызван callme метод класса B»

Рассмотренная форма динамического полиморфизма времени выполнения представляет собой один из наиболее мощных механизмов

объектно-ориентированного программирования, позволяющих писать надежный, многократно используемый код.

3.13. Директива final

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

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

По общепринятому соглашению при выборе имен переменных типа **final** - используются только символы верхнего регистра. Использование **final**-методов порой приводит к выигрышу в скорости выполнения кода - поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы встроенным (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

3.14. Метод Finalize

В Java существует возможность объявлять методы с именем **finalize.** Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

3.15. Директива static

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

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

Ниже приведен результат запуска этой программы.

Инициализирован статический блок X = ***42 A = 3 B = 12***

В следующем примере мы создали класс со статическим методом и несколькими статическими переменными. Второй класс может вызывать статический метод по имени и ссылаться на статические переменные непосредственно через имя класса. class StaticClass { static int a = 42; static int b = 99; static void callme() {

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

А вот и результат запуска этой программы: ***a = 42 b = 99***

3.16. Директива abstract

Бывают ситуации, когда нужно определить класс, в котором задана структура какой-либо абстракции, но полная реализация всех методов отсутствует. В таких случаях вы можете с помощью модификатора типа **abstract** объявить, что некоторые из методов обязательно должны быть замещены в подклассах. Любой класс, содержащий методы **abstract**, также должен быть объявлен, как **abstract**. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора **new**.

Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A { abstract void callme(); void metoo() {
System.out.println("Метод metoo класса A");
}
}
class B extends A { void callme() {
System.out.println("Метод callme класса B");
}
}

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

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

Метод callme класса B Метод metoo класса A

4. Работа с файлами

Библиотека классов языка программирования Java содержит многочисленные средства, предназначенные для работы с файлами. Приложения Java могут работать как с локальными, так и с удаленными файлами (через сеть Internet или Intranet).

Обобщенное понятие источника ввода относится к различным способам получения информации: к чтению дискового файла, символов с клавиатуры, либо получению данных из сети. Аналогично, под обобщенным понятием вывода также могут пониматься дисковые файлы, сетевое соединение. Эти абстракции дают удобную возможность для работы с вводом-выводом (I/O), не требуя при этом, чтобы каждая часть вашего кода понимала разницу между, скажем, клавиатурой и сетью. В Java эта абстракция называется потоком (stream)

и реализована в нескольких классах пакета java.io. Ввод инкапсулирован в классе InputStream, вывод - в OutputStream. В Java есть несколько специализаций этих абстрактных классов, учитывающих различия при работе с дисковыми файлами, сетевыми соединениями и даже с буферами в памяти.

4.1. Классы Java для работы с потоками

Приложение Java, может работать с потоками нескольких типов:

- стандартные потоки ввода и вывода;
- потоки, связанные с локальными файлами;
- потоки, связанные с файлами в оперативной памяти;
- потоки, связанные с удаленными файлами

Рассмотрим кратко классы, связанные с потоками.

4.2. Стандартные потоки

Для работы со стандартными потоками в классе **System** имеется три статических объекта: **System.in**, **System.out** и **System.err**. По своему

назначению эти потоки больше всего напоминают стандартные потоки ввода, вывода и вывода сообщений об ошибках операционной системы MS-DOS.

Поток **System.in** связан с клавиатурой, поток **System.out** и **System.err** - с консолью приложения Java.

4.3. Базовые классы для работы с файлами и потоками

Рассмотрим иерархию классов, предназначенных для организации ввода и вывода (рис. 1).

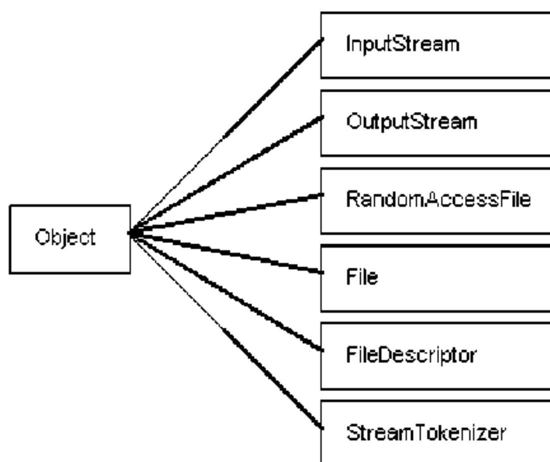

Рис. 1. Основные классы для работы с файлами и потоками

4.3.1. Класс InputStream

Класс **InputStream** является базовым для большого количества классов, на основе которых создаются потоки ввода. Именно производные классы применяются при создании программ, так как в них имеются намного более мощные методы, чем в классе **InputStream**. Эти методы позволяют работать с потоком ввода не на уровне отдельных байт, а на уровне объектов различных классов, например, класса **String** и других.

4.3.2. Класс OutputStream

Аналогично, класс **OutputStream** служит в качестве базового для различных классов, имеющих отношение к потокам вывода.

4.3.3. Класс RandomAccesFile

С помощью класса **RandomAccesFile** можно организовать работу с файлами в режиме прямого доступа, когда программа указывает смещение и размер блока данных, над которым выполняется операция ввода или вывода. Классы **InputStream** и **OutputStream** также можно использовать для обращения к файлам в режиме прямого доступа.

4.3.4. Класс File

Класс **File** предназначен для работы с оглавлениями каталогов. С помощью этого класса можно получить список файлов и каталогов, расположенных в заданном каталоге, создать или удалить каталог,

переименовать файл или каталог, а также выполнить некоторые другие операции.

4.3.5. Класс FileDescriptor

С помощью класса FileDescriptor вы можете проверить идентификатор открытого файла.

4.3.6. Класс StreamTokenizer

Очень удобен класс StreamTokenizer. Он позволяет организовать выделение из входного потока данных элементов, отделенных друг от друга заданными разделителями, такими, например, как запятая, пробел, символы возврата каретки и перевода строки.

4.4. InputStream

InputStream - абстрактный класс, задающий используемую в Java модель входных потоков. Все методы этого класса при возникновении ошибки возбуждают исключение IOException. Ниже приведен краткий обзор методов класса InputStream.

- read() возвращает представление очередного доступного символа во входном потоке в виде целого.
- read(byte b[]) пытается прочесть максимум b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока.
- read(byte b[], int off, int len) пытается прочесть максимум len байтов, расположив их в массиве b, начиная с элемента off. Возвращает количество реально прочитанных байтов.
- skip(long n) пытается пропустить во входном потоке n байтов. Возвращает количество пропущенных байтов.
- available() возвращает количество байтов, доступных для чтения в настоящий момент.
- close() закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.
- mark(int readlimit) ставит метку в текущей позиции входного потока,

которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байтов.

- reset() возвращает указатель потока на установленную ранее метку.
- markSupported() возвращает true, если данный поток поддерживает операции mark/reset.

4.5. OutputStream

Как и InputStream, OutputStream - абстрактный класс. Он задает модель выходных потоков Java. Все методы этого класса имеют тип void и возбуждают исключение IOException в случае ошибки. Ниже приведен список методов этого класса: [1]

4.6. Файловые потоки

4.6.1. FileInputStream

Класс FileInputStream используется для ввода данных из файлов. В приведенном ниже примере создается два объекта этого класса, использующие один и тот же дисковый файл.

```
InputStream f0 = new FileInputStream(7autoexec.bat");
File f = new File("/autoexec.bat"):
InputStream f1 = new FileInputStream(f);
```

Когда создается объект класса FileInputStream, он одновременно с этим

1 write(int b) записывает один байт в выходной поток. Обратите внимание, что аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.

- write(byte b[]) записывает в выходной поток весь указанный массив байтов.
- write(byte b[], int off, int len) записывает в поток часть массива - len байтов, начиная с элемента b[off].
- flush() очищает любые выходные буферы, завершая операцию вывода.
- close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

открывается для чтения. FileInputStream замещает шесть методов абстрактного класса InputStream. Попытки применить к объекту этого класса методы mark и reset приводят к возбуждению исключения IOException. В приведенном ниже примере показано, как можно читать одиночные байты, массив байтов и поддиапазон массива байтов. В этом примере также показано, как методом available можно узнать, сколько еще осталось непрочитанных байтов, и как с помощью метода skip можно пропустить те байты, которые вы не хотите читать.

```
import java.io.*; import java.util.*; class FileInputStreamS {
public static void main(String args[]) throws Exception { int size;
InputStream f1 = new FileInputStream("www.root.com/default.htm"); size
= f1.available();
System.out.println("Всего доступно байтов: " + size);
System.out.println("Читаем первую 1/4 часть файла"); for (int i=0; i <
size/4; i++) {
System.out.print((char) f1.read());
}
System.out.println("Теперь доступно : " + f1.available());
System.out.println("Читаем следующую 1/8 часть "); byte b[] = new byte[size/8]; if
(fl.read(b) != b.length) {
System.err.println(" Ошибка");
}
String tmpstr = new String(b, 0, 0, b.length);
System.out.println(tmpstr);
System.out.println("Осталось доступным: " + f1.available());
System.out.println("Пропустить следующую 1/4 часть"); f1.skip(size/4);
System.out.println( "Осталось доступными: " + f1.available());
8ув1еш.ои1.рг1п11п("Читаем 1/16 часть в конец массива"); if (f1.read(b,
b.length-size/16, size/16) != size/16) { System.err.print1n(" Ошибка");
}
System.out.print1n("Осталось доступно: " + f1.available()); f1.close();
}}
```

4.6.2. FileOutputStream

У класса FileOutputStream - два таких же конструктора, что и у FileInputStream. Однако, создавать объекты этого класса можно независимо от того, существует файл или нет. При создании нового объекта класс FileOutputStream перед тем, как открыть файл для вывода, сначала создает его.

В очередном нашем примере символы, введенные с клавиатуры, считываются из потока System.in - по одному символу за вызов, до тех пор, пока

не заполнится 12-байтовый буфер. После этого создаются три файла. В первый из них, file1.txt, записываются символы из буфера, но не все, а через один - нулевой, второй и так далее. Во второй, file2.txt, записывается весь ввод, попавший в буфер. И наконец в третий файл записывается половина буфера, расположенная в середине, а первая и последняя четверти буфера не выводятся.

```
import java.io.*;
class FileOutputStreamS {
public static byte getlnput()[] throws Exception {
byte buffer[] = new byte[12];
for (int i=0; i<12; i++) {
buffer[i] = (byte) System.in.read();
}
return buffer;
}
public static void main(String args[]) throws Exception { byte buf[] =
getlnput();
OutputStream f0 = new FileOutputStream("file1.txt");
OutputStream f1 = new FileOutputStream("file2.txt");
OutputStream f2 = new FileOutputStream("file3.txt");
for (int i=0; i < 12; i += 2) {
f0.write(buf[i]);
}
f0.close();
f1.write(buf);
f1.close();
f2.write(buf, 12/4, 12/2); f2.close();
} }
```

4.7. Работа с файлами и каталогами при помощи класса File

В предыдущих разделах мы рассмотрели классы, предназначенные для чтения и записи потоков. Однако часто возникает необходимость выполнения и таких операций, как определение атрибутов файла, создание или удаление каталогов, удаление файлов, получение списка всех файлов в каталоге и так далее. Для выполнения всех этих операций в приложениях Java используется класс с именем File.

4.7.1. Создание объекта класса File

У вас есть три возможности создать объект класса File, вызвав для этого один из трех конструкторов:

```
public File(String path);
public File(File dir, String name);
public File(String path, String name);
```

Первый из этих конструкторов имеет единственный параметр - ссылку на строку пути к файлу или каталогу. С помощью второго конструктора вы можете указать отдельно каталог dir и имя файла, для которого создается объект в текущем каталоге. И, наконец, третий конструктор позволяет указать полный путь к каталогу и имя файла.

Если первому из перечисленных конструкторов передать ссылку со значением null, возникнет исключение NullPointerException.

Пользоваться конструкторам очень просто. Вот, например, как создать объект класса File для файла c:\autoexec.bat и каталога d:\winnt:

```
f1 = new File("c:\\autoexec.bat");
f2 = new File("d:\\winnt");
```

4.7.2. Определение атрибутов файлов и каталогов

После того как вы создали объект класса File, нетрудно определить атрибуты этого объекта, воспользовавшись соответствующими методами класса File.

С помощью метода exists вы можете проверить существование файла или каталога, для которого был создан объект класса File. Этот метод можно применять перед созданием потока на базе класса FileOutputStream, если вам нужно избежать случайной перезаписи существующего файла. В этом случае перед созданием выходного потока класса FileOutputStream следует создать объект класса File, указав конструктору путь к файлу, а затем проверить существование файла методом exists().

Методы canRead и canWrite позволяют проверить возможность чтения из файла и записи в файл, соответственно. Их полезно применять перед созданием соответствующих потоков, если нужно избежать возникновение исключений, связанных с попыткой выполнения доступа неразрешенного типа.

Если доступ разрешен, эти методы возвращают значение true, а если запрещен - false.

С помощью методов isDirectory и isFile вы можете проверить, чему соответствует созданный объект класса File - каталогу или файлу.

Метод getName возвращает имя файла или каталога для заданного объекта класса File (имя выделяется из пути).

Метод getAbsolutePath возвращает абсолютный путь к файлу или каталогу, который может быть машинно-зависимым.

С помощью метода isAbsolute вы можете определить, соответствует ли данный объект класса File файлу или каталогу, заданному абсолютным (полным) путем, либо относительным путем.

Метод getPath позволяет определить машинно-независимый путь файла или каталога.

Если вам нужно определить родительский каталог для объекта класса File, то это можно сделать методом getParent.

Длину файла в байтах можно определить с помощью метода length.

Для определения времени последней модификации файла или каталога вы можете вызвать метод lastModified. Этот метод возвращает время в относительных единицах с момента запуска системы, поэтому его удобно использовать только для относительных сравнений.

Метод toString возвращает текстовую строку, представляющую объект класса File.

Метод hashCode возвращает значение хэш-кода, соответствующего объекту File.

Для удаления ненужного файла или каталога вы должны создать соответствующий объект File и затем вызвать метод delete.

С помощью методов mkdir и mkdirs можно создавать новые каталоги. Первый из этих методов создает один каталог, второй - все подкаталоги, ведущие к создаваемому каталогу (то есть полный путь).

Для переименования файла или каталога вы должны создать два

объекта класса File, один из которых соответствует старому имени, а второй - новому. Затем для первого из этих объектов нужно вызвать метод renameTo, указав ему в качестве параметра ссылку на второй объект.

В случае успеха метод возвращает значение true, при возникновении ошибки - false. Может также возникать исключение SecurityException.

Для сравнения объектов класса File вы должны использовать метод equals. Этот метод сравнивает пути к файлам и каталогам, но не сами файлы или каталоги.

С помощью метода list вы можете получить список содержимого каталога, соответствующего данному объекту класса File. В классе File предусмотрено два варианта этого метода - без параметра и с параметром. Первый из этих методов возвращает массив строк с именами содержимого каталога, не включая текущий каталог и родительский каталог. Второй позволяет получить список не всех объектов, хранящихся в каталоге, а только тех, что удовлетворяют условиям, определенным в фильтре filter класса FilenameFilter.

Бадриев А.И.,Карамышев А.Н.

Основы программирования на языке Java

Учебно-методическое пособие

Подписано в печать 22.04.2019.

Формат 60x84/16. Печать ризографическая.

Бумага офсетная. Гарнитура «Times New Roman».

Усл.п.л. 2.5   Уч.-изд. л. 2.4

Тираж 100 экз. Заказ № 1243

Отпечатано в Издательско-полиграфическом центре

Набережночелнинского института

Казанского (Приволжского) федерального университета

423810, г. Набережные Челны, Новый город, пр.Мира, 68/19

тел./факс (8552) 39-65-99 e-mail: ic-nchi-kpfu@mail.ru