Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Пермский национальный исследовательский
политехнический университет»

А.С. Григалашвили

ПРОГРАММИРОВАНИЕ НА JAVA

Учебно-методическое пособие

Издательство
Пермского национального исследовательского
политехнического университета
2017

УДК 681.3
ББК 73я7
Г83

Рецензенты:

д-р экон. наук, профессор *Р.А Файзрахманов*
(Пермский национальный исследовательский
политехнический университет);

д-р техн. наук, профессор *И.Е. Жигалов*
(Владимирский государственный университет)

Григалашвили, А.С.

Г83 Программирование на Java : учеб.-метод. пособие / А.С. Григалашвили. – Пермь : Изд-во Перм. нац. исслед. политехн. ун-та, 2017. – 56 с.

ISBN 978-5-398-01911-7

Изложены теоретические основы, указания и рекомендации для выполнения и оформления контрольной работы по дисциплине «Программирование на Java».

Предназначено для студентов заочной формы обучения по направлению 09.03.01 «Информатика и вычислительная техника».

УДК 681.3
ББК73я7

ISBN 978-5-398-01911-7

Оглавление

Введение 4
Программное обеспечение для выполнения контрольной работы 5
Содержание контрольной работы 5
Теоретические сведения 6
Основные конструкции языка Java 6
Массивы 13
Классы и объекты в JAVA 21
Конструкторы в Java 25
Массивы объектов 30
Наследование 34
Инкапсуляция 37
Полиморфизм 38
Обработка исключительных ситуаций 42
Варианты для выполнения контрольной работы 50
Варианты 1–10 50
Вариант № 11–20 53

Введение

Java представляет собой язык программирования и платформу вычислений, которая была впервые выпущена *Sun Microsystems* в 1995 г. Сначала *Java* предназначался для системного программирования небольших устройств, и язык программирования был только небольшой частью этой системы. Но популярность Интернета изменила предназначение *Java*, и *Sun* стала позиционировать его как многоплатформенную среду и язык.

Существует множество приложений и веб-сайтов, которые не работают при отсутствии установленной *Java*, и с каждым днем число таких веб-сайтов и приложений увеличивается. *Java* отличается быстротой, высоким уровнем защиты и надежностью. От портативных компьютеров до центров данных, от игровых консолей до суперкомпьютеров, используемых для научных разработок, от сотовых телефонов до сети Интернет — *Java* повсюду!

Язык *Java* является строго объектно-ориентированным языком. Поэтому, чтобы начать программировать какие-либо приложения на этом языке, необходимо познакомиться с принципами объектно-ориентированного программирования.

Для выполнения задания контрольной работы по дисциплине «Программирование на *Java*» необходимо ознакомиться со следующими темами:

1. Основные конструкции языка.
2. Массивы.
3. Массивы объектов.
4. Классы.
5. Конструкторы.
6. Наследование.
7. Обработка исключительных ситуаций.

Данное учебно-методическое пособие содержит все необходимое для выполнения задания контрольной работы: перечисляется программное обеспечение, даются теоретические сведения, примеры программ, варианты заданий и список источников литературы, где также можно найти дополнительную информацию.

Программное обеспечение для выполнения контрольной работы

Программное обеспечение, необходимое для выполнения задания контрольной работы:

1. **Минимальный инструментальный набор**

- *JDK* 8u111/8u112 (последняя версия на момент составления методических указаний) – http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

2. **Интегрированная среда программирования – любая из перечисленных:**

- *NetBeans IDE* - http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html
- *IntelliJ IDEA* – https://www.jetbrains.com/idea/download/#section=windows

Все ПО размещено на сайтах, скачивание бесплатное.

Содержание контрольной работы

Контрольная работа выполняется в электронном виде и сдается на проверку преподавателю по электронной почте. После проверки работы преподавателем можно оформлять контрольную работу на бумажном носителе.

Оформление контрольной работы осуществляется по требованиям, установленным кафедрой АТП. Ссылка но нормоконтролю https://vk.com/docs-19715918. Контрольная работа должна содержать следующее:

1. Титульный лист с указанием Ф.И.О., группа, вариант.
2. Задание, соответствующее варианту.
3. Листинг программы с комментариями.
4. Результат работы программы (скриншот). В случае возможности нескольких вариантов, например, при обработке различных исключительных ситуаций, скриншотов должно быть несколько, чтобы полностью продемонстрировать работу программы.

Теоретические сведения

Основные конструкции языка Java

Для начала рассмотрим структуру написания программы на языке *Java*.

```
public class Welcome
{
public static void main (String[] args)
   {
      System.out.println(“Welcome!”);
   }
}
```

В языке Java учитывается регистр слов.

Ключевое слово *public* называется модификатором доступа. Такие модификаторы управляют обращением к коду из других частей программы.

Ключевое слово *class* напоминает, что все элементы *Java*-программ находятся в составе классов. Пока можно считать их некоторыми контейнерами, в которых реализована логика программы, определяющая работу приложения. Таким образом, классы – это строительные блоки, из которых состоят все приложения и апплеты, написанные на языке *Java*.

За ключевым словом *class* следует имя класса. Имя класса должно начинаться с прописной буквы, а остальная его часть может состоять из букв и цифр. Длина имени не ограничена. В качестве имени класса нельзя использовать зарезервированные слова, например, *class*. Если имя состоит из нескольких слов, то каждое из них должно начинаться с прописной буквы без пробела, например, *FirstName*.

Файл, содержащий исходный текст программы, должен называться так же, как и класс, имеющий общедоступный модификатор доступа *public* и иметь расширение **.java*. Таким образом, мы должны создать файл *Welcome.java*. Компиляция вызывается следующим образом:

```
javac Welcome.java
```

В результате компиляции получим файл, содержащий байтовые коды данного класса. Компилятор языка автоматически назовет его *Welcome.class*. Осталось выполнить полученные байт-коды с помощью интерпретатора языка, набрав команду

```
java Welcome
```

Расширение **.class* не указывается! Выполняясь, программа выведет на экран сообщение: *Welcome*.

Когда вызывается последняя команда *java имя_класса*, интерпретатор начинает свою работу с выполнения метода *main()* указанного класса. Следовательно, чтобы программа могла выполняться, в классе должен присутствовать метод *main()*. Разумеется, в класс можно добавить и другие методы. В соответствии со спецификацией языка метод *main* должен быть объявлен как *public*. Аргументы этого метода – параметры командной строки. Объявление метода *main* реализуется только так, как показано ниже:

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

Фигурные скобки используются для выделения блоков программы. В языке *Java* код любого метода должен начинаться открывающейся фигурной скобкой ({) и заканчиваться закрывающейся фигурной скобкой (}).

Каждый оператор должен заканчиваться точкой с запятой (;).

В данном примере используется объект *System.out* и вызывается его метод *println()*. Заметьте, что метод отделяется от объекта точкой:

```
Объект.метод(параметры);
```

В языке *Java* методам, как и в другом любом языке, могут передаваться параметры. Их может быть несколько, а может и не быть, в таком случае после метода необходимо ставить пустые скобки, например:

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

В результате на экран выведется пустая строка.

Математические функции и константы

Стандартные математические функции в *Java* определены в классе Math. Они реализованы как статические методы и поэтому могут вызываться даже в случае, если не создан ни один экземпляр класса *Math*.

Чтобы извлечь квадратный корень из числа, применяют метод *sqrt()*.

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y);          //Выводит число 2.0
```

Возведение в степень – метод *pow()*. В результате выполнения следующей строки кода переменной *y* присваивается значение переменной *x*, возведенное в степень *a*.

```
double y = Math.pow(x,a);
```

Оба параметра метода *pow()*, а также возвращаемое им значение имеют тип *double*.

При вызове математических функций класс *Math* можно не указывать, включив вместо этого в начало файла с исходным текстом следующее выражение:

```
import static java.lang.Math.*;
```

Например, при компиляции приведенной ниже строки ошибки не возникает:

```
System.out.println("The square root is " + sqrt(PI));
```

Помимо арифметических, в *Math* определены функции *min(x,y)* и *max(x,y)*. Кроме того, в этом же классе объявлены константы *PI* и *E* (основание экспоненты).

Значения углов для тригонометрических функций указываются в радианах. Для удобства работы, в *Math* определены также функции преобразования значений из радиан в градусы (*toDegrees(x), x* - в радианах) и наоборот (*toRadians(x), x* - в градусах).

Функция	**Описание**	**Возвр. значение**
sin(x)	x - в радианах	double
cos(x)	x - в радианах	double
tan(x)	x - в радианах	double
asin(x)	арксинус от x, x из [-1.0 .. 1.0]	double
acos(x)	арккосинус от x, x из [-1.0 .. 1.0]	double
atan(x)	арктангенс от x, x из [-pi/2 .. pi/2]	double
exp(x)	e в степени x double	
log(x)	натуральный логарифм от x	double
sqrt(x)	корень квадратный из x	double
pow(x,y)	x в степени y, x - положительное	double
ceil(x)	наименьшее целое >= x	double
floor(x)	наибольшее целое <= x	double
round(x)	floor(x+0.5)	для x типа float - int, для x типа double -long
abs(x)	абсолютная величина x	соответствует типу x
toDegrees(x)	x – в радианах	
toRadians(x)	x – в градусах	

Ввод и вывод

Для того, чтобы организовать чтение информации с консоли, вам надо создать объект *Scanner* и связать его со стандартным входным потоком *System.in*.

```
Scanner in = new Scanner (System.in);
```

Сделав это, вы получите многочисленные методы класса *Scanner*, предназначенные для чтения входных данных. Например, метод *nextLine()* обеспечивает прием строки текста.

```
System.out.print("Как вас зовут?");
```

```
String name = in.nextLine();
```

В данном случае использовался метод *nextLine()*, потому что входная строка может содержать пробелы. Чтобы прочитать одно слово, можно использовать следующий вызов:

```
String firstName = in.next();
```

Для чтения целочисленного значения предназначен метод *nextInt()*.

```
System.out.print("Сколько вам лет?");
int age = in.nextInt();
```

NextDouble() читает очередное число с плавающей точкой.
boolean hasNext() – проверяет, существует ли во входном потоке еще одно слово.
boolean hasNextInt(), boolean hasNextDouble() – проверяет, существует ли во входном потоке последовательность символов, представляющих целое число или число с плавающей точкой.

Форматирование выходных данных

Число *x* можно вывести с помощью выражения *System.out.println(x)*. В результате на экране отобразится число с максимальным количеством значащих цифр, допустимых для данного типа. Например,

```
double x = 10000.0\3.0;
System.out.println(x);          // ответ-3333.3333333333335
```

Для форматирования существует метод *printf()*. Размер поля составляет 8 цифр, дробная часть равна двум цифрам (число цифр дробной части – точность).

```
System.out.printf("%8.2f", x);  //3333.33
```

Метод *printf()* позволяет задавать произвольное число параметров.

```
System.out.printf("%s, в следующем году вам будет %d", name, age);
```

Каждый спецификатор формата, начинающийся с символа %, заменяется соответствующим параметром. Символ преобразования, которым завершается спецификатор формата, задает тип форматируемого значения: *f* – число с плавающей точкой, *s* – строка, *d* – десятичное число, *c* – символ, *b* – логическое значение, % – символ процента и др.

Условные выражения

Условный оператор в языке *Java* имеет следующий вид:

```
If (условие) оператор
```

Условие должно быть выражением, возвращающим логическое значение (тип *boolean*). Условие должно указываться в скобках. Например,

```
if (x < y)
{
f = 10;v = 5 + x;
}
```

Также оператор условия может иметь следующую форму:

```
If (условие) оператор_1 else оператор_2
```

Например,

```
if (x < y)
{
f = 10; v = 5 + x;
}
else
{
a = 10; v = 5 + y;
}
```

Также встречаются операторы такого типа: ***if…else if…***

```
if (yourSales >= 2 * target)
{
performance = ”Отлично”;
bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
performance = ”Хорошо”;
bonus = 500;
}
else if (yourSales = target)
{
performance = ”Удовлетворительно”;
bonus = 100;
}
else
{
System.out.println(“Вы уволены”);
}
```

Неопределенные циклы

Цикл ***while*** обеспечивает выполнение выражения (или группы операторов, составляющих блок) до тех пор, пока условие равно *true*. Данный цикл записывается в следующем виде:

```
while (условие) оператор
```

От инструкции *for() while()* принципиально отличается тем, что инициализация индексной переменной, если такая имеется, выполняется до вызова инструкции, а команда изменения этой переменной размещается в теле цикла.

Тело цикла не будет выполнено ни разу, если его условие изначально равно *false*. Программа, подсчитывающая, сколько лет надо вносить деньги на счет, чтобы накопить заданную сумму. Считается, что каждый год вносится одна и та же сумма и процентная ставка не меняется. Увеличиваем счетчик и обновляем в теле цикла сумму, накопленную к данному моменту, пока общий итог не превысит заданную величину.

```
while (balance < goal)
{
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
years++;
}
System.out.println(years + ” years.”);
```

Условие цикла *while* проверяется в самом начале. Следовательно, возможно ситуация, что код, содержащийся в блоке, не будет выполнен никогда.

Чтобы блок выполнялся хотя бы один раз, проверку условия нужно перенести в конец цикла. Это можно сделать с помощью цикла *do/while*. Условие выполняется лишь после выполнения тела цикла. Затем тело цикла повторяется вновь, если условие равно *true*.

```
do
{
//Добавление ежегодного взноса и процентов
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
years++;
/Вывести текущий баланс
…
System.out.println(“Уходите на пенсию? (Y/N)”);
input = in.next();
}
while(input.equals(“N”));
```

Если пользователь отвечает *N*, цикл повторяется.

Определенные циклы

В цикле *for* число повторений контролируется переменной, выполняющей роль счетчика и обновляемой на каждой итерации.

```
for (int i = 1; i <= 10; i++)
System.out.println(i);
```

Первый элемент оператора *for* обычно выполняет инициализацию счетчика, второй формулирует условие выполнения тела цикла, третий определяет способ обновления счетчика.

Цикл с убывающим счетчиком:

```
for (int i = 10; i > 0; --i)
System.out.println("Обратный счет …" + i);
```

При объявлении переменной в первой части оператора *for* область видимости простирается до конца тела цикла. Если переменная определена в теле цикла, то ее нельзя использовать вне этого цикла. Если использовать конечное значение счетчика вне цикла *for*, то переменную нужно объявлять до начала цикла.

```
int i;
for (i = 0; i < 10; i++)
{…}
```

С другой стороны, можно объявить переменные, имеющие одинаковое имя в разных циклах *for*.

```
for (i = 0; i < 10; i++)
{…}
for (i = 11; i < 20; i++)        //Переопределение переменной i
{…}
```

Действия, выполняемые посредством цикла *for*, можно реализовывать с помощью цикла *while*.

```
for (i = 10; i > 10; i--)
System.out.println("Обратный отсчет …" + i);
```

Эквивалентно

```
int i = 10;
while (i > 0)
{
System.out.println("Обратный отсчет …" + i); i--;
}
```

Массивы

Массив – это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется с помощью целочисленного индекса. Например, если *a* – массив целых чисел, то значение выражения *a[i]* равно *i*-му целому числу в массиве.

Массивы бывают статическими и динамическими. Под статические массивы память выделяется при компиляции программы, а для динамических массивов память выделяется в процессе выполнения программы. **В *Java* массивы динамические!**

Массив объявляется следующим образом: сначала указывается тип массива, т.е. тип элементов, затем следует пара пустых квадратных скобок, а после них – имя переменной.

```
int[] a;
```

Допускается указывать квадратные скобки либо после имени типа массива, либо после имени массива. Например, *int a[];*

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

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

Ниже приведен пример, в котором создается массив, элементы которого содержат число дней в месяцах года (невисокосного).

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

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

```
April has 30 days.
```

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

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

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

Поскольку для экземпляра массива длина является постоянной характеристикой, для всех массивов существует специальная переменная ***length***, позволяющая узнать это значение. Эта переменная создается при объявлении массива, и ее значением является количество элементов массива. Поскольку для каждого массива создается своя переменная *length*, обращение к таким переменным осуществляется с одновременным указанием имени массива. В частности, сначала указывается имя массива, а затем, через точку, имя переменной *length*. Например, *a.length*.

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

```
int mas[] = new int[12];
for (int i = 0; i < mas.length; i++)
{
mas[i] = 1 + i;
}
```

В *Java SE* 5.0 был реализован новый цикл *«for each»*, позволяющий перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик. Новый вариант цикла записывается следующим образом:

```
for (переменная : набор_данных) оператор
```

При обработке цикла переменной последовательно присваивается каждый элемент набора данных, после чего выполняется оператор (или блок операторов). Например,

```
for (int element : a)
System.out.println(element);
```

В результате выполнения данного фрагмента кода каждый элемент массива *a* будет напечатан в отдельной строке.

Эти же действия можно и реализовать с помощью традиционного цикла *for*:

```
for (int i = 0; i < a.length; i++)
System.out.println (a[i]);
```

Однако при использовании нового цикла запись получается более краткой (необходимость в начальном выражении и условии завершения цикла отпадает), и вероятность возникновения ошибок уменьшается.

Традиционный цикл *for* вовсе не устарел. Без него нельзя обойтись, если надо обработать не весь набор данных, а лишь его часть; либо тогда, когда счетчик явно используется в теле цикла.

Чтобы напечатать все значения массива, можно использовать метод ***toString()*** класса ***Arrays***. Вызов *Arrays.toString(a)* вернет строку, содержащую все элементы массива, заключенную в квадратные скобки и разделенную запятыми:

```
System.out.println(Arrays.toString(a));
```

Результат:

```
[2, 3, 5, 7, 11, 13]
```

Копирование массивов

Переменная, обозначающая массив (переменная массива), объявляется независимо от фактического выделения памяти под массив. Другими словами, непосредственно массив и переменная массива – это далеко не одно и то же. В этом смысле показательным является двухэтапный (двумя командами) процесс создания массива. Например,

```
int [] nums;
nums = new int[] {1,2,3,4};
```

В данном случае команда *int [] nums* есть не что иное, как объявление переменной *nums*. Тип этой переменной – «массив целых чисел». Значением переменной может быть ссылка (адрес) на какой-нибудь массив, состоящий из целых чисел.

Оператор `new` в общем случае служит для динамического выделения памяти под различные объекты, в том числе массивы. Командой *new int[] {1,2,3,4}* в памяти выделяется место для целочисленного массива из четырех элементов, соответствующие значения присваиваются элементам массива. У этого вновь созданного массива есть адрес (ссылка на массив). В качестве значения оператор *new* возвращает ссылку на созданный объект. В данном случае возвращается ссылка на массив. Эта ссылка в качестве значения присваивается переменной *nums*. Теперь несложно догадаться, каким будет результат выполнения следующих команд:

```
int [] nums, data;
nums = new int[] {1,2,3,4};
data = nums;
```

После присваивания переменная *data* ссылается на тот же массив, что и переменная *nums*.

При сравнении массивов с помощью операторов равно **==** и не равно **!=** (*nums == data*) сравниваются значения переменных массива, а не элементы этих массивов. Поэтому результатом этого выражения является *true*, если обе переменные ссылаются на один и тот же массив.

Особенности сравнения массивов на предмет равенства (неравенства) иллюстрируется следующим программным кодом:

```
class ArrayDemo2
{
public static void main(String[] args)
     {
//объявление массивов
int[] nums = new int[] {1,2,3,4};
int[] data = new int[] {1,2,3,4};
//проверка совпадения ссылок
if (data == nums)
  {
System.out.println("Massivy sovpadayut!");
return;
  }
//проверка размеров массивов
if (data.length != nums.length)
  {
System.out.println("Raznye massivy!");
```

```
return;
  }
//поэлементная проверка массивов
for (int i = 0; i < data.length; i++)
   {
if (data[i] != nums[i])
  {
System.out.println("Nesovpadayuw'ie massivy");
return;
  }
   }
System.out.println("Odinakovye massivy!");
     }
}
```

Если необходимо скопировать все или несколько элементов массива используется метод *arraycopy()* из класса *System*. Его вызов выглядит следующим образом:

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

Массив *to* должен иметь достаточный размер, чтобы в нем поместились все копируемые элементы.

Сначала создаются два массива, а затем последние четыре элемента первого массива копируются во второй. Копирование исходного массива начинается с элемента с номером 2, а в целевой массив копируемые данные помещаются, начиная с элемента с номером 3.

```
int[] one = {2,3,5,7,11,13};
int[] two = {1001,1002,1003,1004,1005,1006,1007};
System.arraycopy(one, 2, two, 3, 4);
for (int i = 0; i < two.length; i++)
     System.out.println(i + ": " + two[i]);
```

Выполнение данного фрагмента приводит к следующему результату:

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```

Сортировка массива

Если нужно упорядочить массив чисел, можно применить метод *sort()* из класса *Arrays.*

```
int [] mas = {3,10,8,12,65};
Arrays.sort(mas);
System.out.println(Arrays.toString(mas));
```

Другие методы для работы с массивами

- `Arrays.copyOf(type[] a, int length)`
- `Arrays.copyOf(type[] a, int start, int end)`

Возвращает массив того же типа, что и *a*, длиной либо *length*, либо (*end-start*), заполненный значениями из *a*.

Параметры:

a – массив любого из простых типов

start – начальный индекс (включительно)

end – конечный индекс (исключительно). Может быть больше, чем *a.length*, при этом результирующий массив дополняется нулями или значениями *false*.

length – длина копии. Если *length* больше, чем *a.length*, результат дополняется нулями или значениями *false*. В противном случае копируются только начальные *length* значений.

- *Arrays.fill(mun[] a, mun v)* – присваивает каждому элементу массива значение *v*. *V* – значение того же типа, что и элементы массива.
- *Arrays.equals(mun [] a, mun [] b)* – возвращает величину *true*, если массивы имеют одинаковую длину, и элементы с соответствующими индексами совпадают.

Многомерные массивы

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

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

После инициализации массива к его отдельным элементам можно обращаться с помощью двух пар квадратных скобок, например, *matrix[i][j]*.

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

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

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

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

Запустив эту программу, вы получите следующий результат:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

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

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

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

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

Запустив эту программу, вы получите следующий результат:

```
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
```

Классы и объекты в JAVA

Объекты реального мира, например, регистрационная форма для регистрации на курсе, состоят из атрибутов и вариантов поведения.

Атрибут – это данные, связанные с объектом. Название курса, номер курса, имя и номер студента – вот примеры данных, связанные с регистрационной формой.

Поведение – это что-то, что может делать объект, например обработка, модификация или отмена регистрации на курсе.

Класс – это шаблон, который определяет атрибуты и методы объекта реального мира. Если вам необходим объект, представляемый классом, вы создаете экземпляр класса.

Класс описывается с помощью определения класса. В определении класса перечисляются атрибуты и методы, которые являются членами класса.

```
class имя_класса
{тело_класса};
```

Ключевое слово *class* сообщает компилятору, что вы определяете класс.

Название класса – последовательность символов, которая идентифицирует класс среди других классов.

Тело класса – это часть определения класса, которая находится между фигурными скобками. Атрибуты и методы определяются между скобками.

Метод – группа операторов, которые осуществляют некоторое поведение и определяются следующим образом:

- название метода,
- список аргументов метода,
- тело метода,
- возвращаемое значение.

Название метода должно отражать тип поведения, который осуществляет метод.

Список аргументов – это данные, находящиеся вне определения метода, которые необходимы методу для осуществления поведения.

Данные в этом списке называются *аргументами.* В списке аргументов может быть один или несколько аргументов. Аргумент объявляется с помощью указания типа данных и имени аргумента. Объявления аргументов должны быть разделены запятой. После того как аргумент объявлен, его имя используется в операторах в определении метода для ссылки на данные, которые были присвоены аргументам при вызове метода.

```
тип_возвращаемого_значения название_метода (тип аргумент_1, тип аргумент_2)
      {тело метода;
      возвращаемое значение;
      }
```

Например,

```
boolean dropCourse (int courseNumber, int studentNumber)
{ return true;}
```

Тело метода – это часть метода, содержащая операторы, которые выполняются, когда вызывается метод. Тело метода определяется внутри блока кода, который выделяется фигурными скобками. Операторы выполняются в теле метода последовательно, начиная с первого оператора и пока не будет выполнен оператор возврата или достигнут конец тела метода.

Определение метода помещается в определение класса. Если метод не возвращает никакого значения, то тип возвращаемого значения – *void* (пустой). Чтобы вернуть значение, необходимо указать тип возвращаемого значения слева от определения метода.

```
class RegistrationForm {
int StudentNumber;
int CourseNumber;
void dropCourse (intcourseNumber, intstudentNumber){
//удаление курса
}
}
```

Определение класса помещается вне основной части программы. Основная часть программы на *Java* – это метод *main* класса для приложения. И уже в основной части программы происходит обращение к атрибутам и методам класса, но после того, как создан экземпляр класса. Каждый экземпляр содержит те же самые атрибуты и методы, которые определены в классе, хотя каждый экземпляр имеет свою копию этих атрибутов. Экземпляры используют одинаковые методы. Создание экземпляра класса происходит следующим образом:

```
RegistrationForm regForm=new RegistrationForm();
```

где *regForm* – ссылка на экземпляр класса. Именно эту ссылку необходимо использовать при вызове методов данного класса и использовании переменных класса.

Значение переменной экземпляра может быть изменено с помощью оператора присваивания. Для обращения к переменной следует использовать имя экземпляра класса:

```
regForm.status=”no status”;
```

Вызов метода класса происходит подобным образом:

```
regForm.dropCourse(102,1234);
```

Объединение программы и определение класса

```
class MyJavaApplication {
public static void main (String [] args){
      RegistrationForm regForm=new RegistrationForm();
      regForm.dropCourse(102,1234);
       }
}
class RegistrationForm{
void dropCourse(int courseNumber, int studentNumber){
System.out.println(“course”+studentNumber+”has been dropped from
      student”+studentNuber);
}
}
```

Пример выполнения задания

Задание 1. Класс: пенсионер, атрибуты: имя – *String*, пенсия – *int*. Вычислить размер пенсии в следующем году при увеличении ее на 5 %. Описать методы-члены класса вне тела класса.

Листинг программы:

```
import java.util.*;
class Pensioner
{
public static void main(String [] args)
      {
String name;
int pens;
Scanner in=new Scanner(System.in);
System.out.println("Kak vas zovut? ");
name=in.nextLine();
System.out.println("Kakaya u vas pensia? ");
pens=in.nextInt();
pens+=pens/20;
System.out.println(name+", v sledushem godu vasha pensia budet ravna
vsego "+pens+" rubley");
      }
}
```

Задание 2. Класс: пенсионер, атрибуты: имя – *String*, пенсия – *int*. Вычислить размер пенсии в следующем году при увеличении ее на 5%. Описать методы-члены класса в теле класса.

Описанная выше декларация класса является удобной только в случае очень небольшого класса. Если же методов достаточно много, а сами они

сложны и объемны, имеет смысл отделить описание класса от реализации. Это тем более важно, так как язык *Java* допускает раздельную компиляцию модулей. Поэтому, например, описание класса может быть выделено в отдельный, заголовочный модуль, а реализация методов будет производиться в другом, основном модуле. Поэтому наш класс может быть записан следующим образом:

```
import java.util.*;
class Pensioner
{
private String name;
private int pensia;

public void Write(String n, int pens)
      {
name=n;
pensia=pens;
      }
public void Vy4iclenie()
      {
pensia+=(pensia/20);
System.out.println(name+", v sledushem godu vasha pensia budet
ravna vsego "+pensia+" rubley");
      }
}

class Odin
{
public static void main(String [] args)
      {
String n;
int s;
Pensioner p=new Pensioner();
Scanner in=new Scanner(System.in);
System.out.println("Kak vas zovut? ");
n=in.nextLine();
System.out.println("Kakaya u vas pensia? ");
s=in.nextInt();
p.Write(n, s);
p.Vy4iclenie();
      }
}
```

Конструкторы в Java

Создание объектов – очень важная задача, поэтому на языке в *Java* предусмотрено много разнообразных механизмов для инициализации переменных экземпляра. Инициализация – это процесс присваивания значения переменной при ее объявлении. Переменные экземпляра должны быть инициализированы с использованием специального метода – *конструктора,* который автоматически вызывается, когда объявляется экземпляр класса. Формат определения конструктора следующий:

```
public имя_класса()
{…}
```

При создании конструктора нужно помнить о следующих правилах:

- имя конструктора всегда совпадает с именем класса,
- класс может иметь несколько конструкторов,
- конструктор может иметь один или несколько параметров или не иметь их вообще,
- конструктор не возвращает никакого значения, **тип *void* недопустим**,
- конструктор всегда вызывается совместно с оператором *new*.

Конструктор автоматически вызывается при определении или размещении в памяти с помощью оператора *new* каждого объекта класса, например:

```
public Employee(double aName, double aSalary)
      {
name=aName;
salary=aSalary;
      }
```

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

Если значение атрибута в конструкторе явно не задано, ему автоматически присваивается значение по умолчанию: числам – нули, логическим переменным – *false*, а ссылкам на объект – *null*. Однако полагаться на значения по умолчанию считается плохим стилем. Программа в таком случае становится плохо читаемой.

Конструктор по умолчанию

Конструктор по умолчанию не имеет параметров. Например, конструктор класса *Employee* (работник) имеет следующий вид:

```
public Employee()
```

```
{
name="";
salary=0;
hireDay=new Date();
}
```

Если в классе вовсе не определены конструкторы, вызывается конструктор по умолчанию. Конструктор по умолчанию вызывается, только если в классе не определены другие конструкторы. Если есть хотя бы один конструктор с параметрами и необходимо создавать экземпляр класса, то при создании экземпляра необходимо задавать параметры конструктора.

Конструктор с параметрами

Конструктор может иметь один или несколько параметров. Создавая такой конструктор, трудно выбрать подходящие имена для его параметров. Обычно в качестве имен параметров используются отдельные буквы.

```
public Employee(String n, double s)
{
name=n;
salary=s;
}
```

Недостаток такого подхода заключается в том, что читая программу, невозможно понять, что означает параметры *n* и *s*.

Некоторые программисты добавляют к осмысленным именам параметров какую-нибудь букву. Такая программа вполне понятна. Любой читатель поймет, в чем смысл параметра.

```
public Employee(String aName, double aSalary)
{
name= aName;
salary= aSalary;
}
```

Есть еще один широко распространенный прием.

```
public Employee(String  Name, double Salary)
{
this.name= Name;
this.salary= Salary;
}
```

Доступ к атрибуту экземпляра осуществляется с помощью ключевого слова **this,** которое означает неявный параметр, т.е. создаваемый объект. Если вызвать метод с параметром *salary*, то эта переменная будет ссылаться на параметр, а не на атрибут экземпляра.

Конструктор с параметрами вызывается автоматически при создании экземпляра класса в том случае, если при создании объекта передаются соответствующие значения параметров.

```
//вызов конструктора с параметрами
Employee emp = new Employee (“Vasya”, 25000);
```

Если конструкторов в классе определено несколько, то компилятор подбирает нужный в соответствии с количеством и типами передаваемых параметров. Этот процесс поиска компилятором необходимого метода называется *разрешением перегрузки*. Если компилятор не находит подходящего конструктора, то он выдает соответствующую ошибку.

Конструктор копирования

Конструктор копирования – конструктор, который создает объект на основе уже существующего. В качестве параметра такой конструктор принимает уже созданный объект этого же класса.

```
public Employee(Employee obj)
{
this.name= obj.name;
this.salary= obj.Salary;
}
```

Вызов такого конструктора выглядит следующим образом:

```
Employee emp1 = new Employee (“Vasya”, 25000);
Employee emp2 = new Employee (emp1);
```

В результате будет создано два объекта *emp* и *emp2* с одинаковым значениями переменных name и salary.

Пример определения нескольких конструкторов

```
class MyObjs{
// Полякласса:
double Re,Im;

// Присваивание значений полям:
void set(double Re,double Im){
this.Re=Re;
this.Im=Im;
show();}

// Отображение значений полей:
void show(){
System.out.println("Re="+Re+" и "+"Im="+Im);}
```

```
// Конструктор без аргументов:
MyObjs(){
set(0,0);}

// Конструктор с одним аргументом:
MyObjs(double x){
set(x,x);}

// Конструктор с двумя аргументами:
MyObjs(double x,double y){
set(x,y);}

// Конструктор копирования:
MyObjs(MyObjs obj){
set(obj.Re,obj.Im);}

// Аргумент и результат - объекты:
MyObjs getSum(MyObjs obj){

// Создание локального объекта:
MyObjs tmp=new MyObjs();

// Определение параметров локального объекта:
tmp.Re=Re+obj.Re;
tmp.Im=Im+obj.Im;

// Возвращение результата методом:
return tmp;}

// "Прибавление" объекта к объекту:
void add(MyObjs obj){
Re+=obj.Re;
Im+=obj.Im;}
}

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

// Создание объектов:
MyObjs a=new MyObjs(1);
MyObjs b=new MyObjs(-3,5);
MyObjs c=new MyObjs(b);

// Вычисление "суммы" объектов:
c=a.getSum(b);
// Проверка результата:
c.show();
// Изменение объекта:
a.add(c);
// Проверка результата:
```

```
a.show();
}
}
```

Для понимания принципов работы метода *getSum()* следует учесть особенности возвращения объекта в качестве результата. Так, если методом возвращается объект, при выполнении метода выделяется место в памяти для записи результата. Этот процесс (выделение места в памяти) не следует путать с созданием объекта. После того как в методе выполнена инструкция возврата значения, локальный объект, используемый как результат метода, копируется в место, выделенное в памяти для результата. Ссылка на эту область фактически и является тем значением, которое присваивается переменной, указанной слева от оператора присваивания в команде вызова метода.

Массивы объектов

Во многих языках программирования размер всех массивов должен создаваться еще на этапе компиляции программы.

В языке *Java* ситуация намного лучше - размер массива можно задавать уже во время выполнения программы.

```
int actualSize=…;
Employee[] staff=new Employee[actualSize];
```

Разумеется, этот код не решает проблему динамической модификации массивов во время выполнения программы. Задав размер массива, его потом нелегко изменить. Проще всего решить эту проблему можно, используя списочный массив. Для их создания применяются экземпляры класса *ArrayList*. Данный класс ведет себя как массив, но может динамически изменять размеры по мере добавления новых элементов или удаления существующих. При этом программисту не приходится писать дополнительный код.

Ниже представлена строка кода, посредством которого создается списочный массив для хранения объектов *Employee*.

```
ArrayList <Employee> staff=new ArrayList<Employee>();
```

Для добавления новых элементов в списочный массив используется метод *add()*.

```
staff.add(new Employee("Petya", 25000));
staff.add(new Employee("Vasya", 45000));
```

Если вы заранее знаете, сколько элементов нужно хранить, то перед заполнением списка массивов вызовете метод *ensureCapacity()*:

```
staff.ensureCapacity(100);
```

Этот метод выделит память для внутреннего массива, состоящего из ста объектов. Затем можно вызвать метод *add()*, который не будет иметь проблем с перераспределением памяти.

Количество элементов, которые будут храниться в списке массивов, можно передать конструктору класса *ArrayList* в качестве параметра:

```
ArrayList<Employee> staff=new ArrayList<Employee>(100);
```

Метод *size()* возвращает фактическое количество элементов в списочном массиве на текущий момент.

```
staff.size();
```

Для обычного массива это выглядит так: *a.length*.

Если вы уверены, что список массивов будет иметь постоянное количество элементов, можете вызвать метод *trimToSize()* – сокращает емкость списочного массива до его текущего размера**.** Этот метод устанавливает размер блока памяти так, чтобы он точно соответствовал количеству элементов, подлежащих хранению. В результате блок памяти будет перемещен, что потребует дополнительного времени. Поэтому вызывать данный метод нужно в том случае, когда вы уверены, что дополнительные элементы в списочный массив не будут добавлены.

Доступ к элементам списочных массивов

Вместо квадратных скобок для доступа к элементам программистам приходится использовать методы *get()* и *set()*. Например, чтобы задать *i*-тый элемент, используется выражение

```
staff.set(i, harry);  //a[i]=harry;
```

Как в обычных, так и в списочных массивах элементы отсчитываются от нуля. Получить *i*-тый элемент списочного массива можно, используя метод *get()*:

```
Employee e=staff.get(i);   //Employee e=a[i];
```

Возможность осуществлять перебор элементов списочных массивов с помощью цикла *“for each”*:

```
for (Employee e: staff)    //выполнение действий с переменной e
```

В более ранних версиях:

```
for (int i=0; i<staff.size(); i++)
{
Employee e=(Employee) new Staff.get(i);
//Выполнение действий с переменной e
}
```

Элементы можно добавлять не только в конец массива, но и в его середину:

```
int n =staff.size()/2;
staff.add(n, e);
```

Элемент, имеющий индекс *n*, и расположенные за ним элементы сдвигаются, чтобы освободить место для нового элемента. Если новый размер списка массивов оказывается больше, чем его емкость, происходит копирование массива.

Аналогично можно удалить элемент из середины списочного массива:

```
Employee e=(Employee)staff.remove(n);
```

Элементы, расположенные после удаленного элемента, сдвигаются влево, а размер списка массивов уменьшается на единицу.

Задание. Реализовать класс СТУДЕНТ, поля: имя – *String*, возраст – *int*, определить среднее количество рабочих в цехе. Создать массив объектов класса СТУДЕНТ, вывести информацию о каждом, вывести на экран средний возраст студентов.

Пример выполнения задания

```
import java.util.*;
class Student
{
private String Name;
private int age;

public Student(String sName,int nAge)
      {
Name=sName;
age=nAge;
      }

public void Show()
      {
System.out.println("Student "+Name+" is "+age+" years old");
      }
public int GetAge()
      {
return age;
      }
}

class MassiveObj
{
public static void main(String [] args)
      {
double s=0;
int n=0;
//создаем списочный массив
ArrayList <Student> mas=new ArrayList <Student>();

//заполняем списочный массив
mas.add(new Student("Petya", 23));
mas.add(new Student("Vasya", 21));
mas.add(new Student("Andrey", 20));

//складываем возраст
for(Student i:mas)
   {
```

```
s+=i.GetAge();
//выводим информацию о студентах
i.Show();
n++;
   }
//вычисляем средний возраст студентов
double sr=s/n;
System.out.printf("Sredniy vozrast studentov %5.2f",sr);
     }
}
```

Наследование

Наследование – это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются **суперклассами** (или родителями), а новые – **подклассами** (или потомками). Подклассы «получают в наследство» атрибуты и методы своих суперклассов и, кроме того, могут пополняться собственными компонентами (атрибутами и собственными методами). Наследуемые компоненты не перемещаются в подкласс, а остаются в суперклассах.

Существует три способа реализации наследования в программе.

1. *Простое наследование* используется, когда существует одно отношение «родитель-потомок».
2. *Многоуровневое наследование* появляется, когда потомок наследуется от родителя, а затем сам становится родителем. У класса может быть только один родитель!
3. *Множественное наследование* используется, когда отношение включает несколько родителей и одного потомка. Такой тип наследованиЯ в Java не поддерживается.

В иерархии производный объект наследует разрешенные для наследования атрибуты всех базовых объектов (*public, protected*).

Синтаксис определения производного класса:

```
режим_доступа class имя_класса extends суперкласс
{список_компонентов_класса};
```

Вызов метода суперкласса в определении метода подкласса выглядит следующим образом:

```
super.метод_наследуемого_класса;
```

Пример выполнения простого наследования

```
class Student
{
protected int ID, Graduation; // доступны только методам класса
protected String First, Last;

public void Write(int nID, int Grad, String sFirst, String sLast)
      //присваивает значения, открытый доступ
      {
ID=nID;
First=sFirst;
Last=sLast;
```

```
Graduation=Grad;
      }
public void Display()         //выводит значения атрибутов
      {
System.out.println("Student    "+ID+"    "+First+"    "+Last+"    Graduated:
"+Graduation);
      }
}

class GradStudent extends Student   //выпускник наследует студента
{
private int Year;       //доступны наследуемому классу
private String School;

//переопределение метода суперкласса
public void Write(int nID, int Grad, String sFirst, String sLast, int yr,
String sch)
      {
super.Write(nID, Grad, sFirst, sLast);          //вызываем метод класса
                                                //Студент
Year=yr;
School=sch;
      }

//переопределение метода суперкласса
public void Display()
      {
super.Display();                    //вызываем метод класса Студент
System.out.println(" "+School+" in "+Year);
      }
}

class StudentInfo2
{
public static void main(String [] args)
      {
GradStudent g=new GradStudent();          //экземпляр класса Выпускник
g.Write(101,1,"Masha", "Titova", 2009, "PNIPU");//передаем параметры
g.Display();                                    //выводим значения
      }
}
```

Приведение типов при наследовании

В случае если у нескольких классов общий родитель, их можно объединить в массив потомков. При этом тип элементов такого массива будет имя класса родителя. Это позволит обработать объекты потомков в одном цикле, например, вызвать какой-либо метод, принадлежащий всем потомкам.

```
class Student extends Human {…}
class Artist extends Human {…}
```

```
class Sportsmen extends Human {…}
…
//массив потомков класса Human
Human [] mas = new Human [3];
mas[0] = new Student();
mas[1] = new Artist();
mas[2] = new Sportsmen();

//вывод информации обо всех объектах в массиве, метод Display
//реализован во всех классах-потомках
for (int i = 0; I < mas.length; i++)
{
mas[i].Display();
}
```

Проверить, является ли объект потомком какого-либо класса, можно проверить с помощью метода *instanceof*, возвращающего значение логического типа (*boolean*).

```
//проверяем, является ли элемент объектом класса Student
if (mas[i] instanseof Student)
{…}
```

Инкапсуляция

Инкапсуляция – это процесс соединения сходных вещей для формирования нового объекта. Объектно-ориентированные языки позволяют программистам инкапсулировать атрибуты и методы и связывать их с объектами. Инкапсуляция – это способ связывания атрибутов и методов для формирования объектов.

Основная цель инкапсуляции – защита. Инкапсуляция позволяет программисту определить правила доступа к атрибутам и методам.

Программисты контролируют доступ к атрибутам и методам класса с помощью спецификаторов доступа в определении класса. Спецификатор доступа – это ключевое слово зыка программирования, которое говорит компьютеру, какая часть программы может получить доступ к атрибутам и методам, являющимся членами класса.

Java имеет три спецификатора доступа – *public*, *private*, *protected*:

1. Спецификатор доступа **public** определяет атрибуты и методы, которые доступны при использовании экземпляра класса. Другими словами обращаться к таким данным можно напрямую через объект класса.
2. Спецификатор доступа **private** определяет атрибуты и методы, которые доступны только методам, определенным в классе. Переменные и методы, отмеченные как *private*, не наследуются.
3. Спецификатор доступа **protected** определяет атрибуты и методы, которые могут быть наследованы другими классами.

Полиморфизм

Полиморфизм означает, что одна вещь может принимать много форм. В программировании полиморфизм – это когда метод с одним названием может осуществлять множество вариантов поведения. Существует несколько возможных вариантов полиморфизма:

1. В одном классе – перегрузка метода.
2. В нескольких классах:
    а) методы со схожим поведением,
    б) переопределение метода при наследовании.

Перегрузка метода – два или более метода имеют одно название, но разные списки элементов.

Похожее поведение в нескольких классах

Для того, чтобы не придумывать названия методам со схожей функциональностью, но принадлежащим разным классам, разработчики могут дать им одно имя. Компилятор, как и в случае с перегрузкой метода, будет искать подходящий метод, но уже в зависимости от того, к объекту какого класса был обращен вызов. Например,

```
class StudentInfo{
public static void main (String [] args)
      {
      Student student = new Student();
      GradStudent grad = new GradStudent();
      student.Write(100, "Bob", "Smith", 2000);
      grad.Write(10, "Bob", "Smith", 2000, "Columbia University",
"CS");

      student.Display();
      grad.display();
      }
}

class Student
{
public void Write(int ID, int Grad, String Fname, String Lname)
      {
      m_ID=ID;
      m_Graduation=Grad;
      m_First=Fname;
      m_Last=Lname;
      }

public void Display()
```

```
      {
      System.out.println("Student:          "+m_ID+"          "+m_First+"
"+m_Last+"Graduated:
      "+m_Graduation);
      }

private int m_ID, m_Graduation;
private String m_First;
private String m_Last;
}

class GradStudent extends Student
{
public  void  Write(int  ID,  int  Grad,  String  Fname,  String  Lname,
intyrGrad, String unSch, String major)
      {
      super.Write(ID, Fname, Lname, Grad);
      m_UndergradSchool=unSch;
      m_Major=major;
      YearGraduation=yrGrad;
      }

public void Display()
      {
super.Display();
System.out.println("          "+m_UndergradSchool+"          "+m_Major+"
"+YearGraduation);
      }

private int YearGraduation;
private String m_UndergradSchool;
privateStringm_Major;
}
```

Каждый класс имеет методы-члены *Write()* и *Display()*. Метод *Write()* присваивает значения атрибутам, метод *Display()* отображает значения методов. У каждого класса свои методы, они выполняют схожие задачи, но делают это по-разному. Это и есть полиморфизм в действии.

Программисты стараются не забивать себе голову методами. Благодаря полиморфизму они определяют методы с похожим поведением, используя одинаковые названия. Благодаря этому достаточно запомнить только одно название метода, соответствующего некоторому поведению.

Переопределение метода при наследовании

Между переопределением и перегрузкой методов существует принципиальное различие. При перегрузке методы имеют одинаковые названия, но разные сигнатуры. При переопределении совпадают не только

названия методов, но и полностью сигнатуры (тип результата, имя и список аргументов). Переопределение реализуется при наследовании. Для перегрузки в наследовании необходимости нет.

Может сложиться такая ситуация. Допустим, в суперклассе определен некий метод, а в подклассе определяется метод с таким же названием, но другой сигнатурой. В этом случае в подклассе будут доступны обе версии метода: версия, описанная в суперклассе, и версия, описанная в подклассе.

```
class ClassA{
static int count=0;
private int code;
int number;

ClassA(int n){
set(n);
count++;
code=count;
System.out.println("Объект №"+code+" создан!");}

void set(int n){
number=n;}

void show(){
System.out.println("Для объекта №"+code+":");
System.out.println("Поле number: "+number);}
}

class ClassB extends ClassA{
char symbol;

ClassB(int n,char s){
super(n);
symbol=s;}

void set(int n,char s){
number=n;
symbol=s;}

void show(){
super.show();
System.out.println("Поле symbol: "+symbol);}
}

class MyMethDemo{
public static void main(String[] args){
ClassA objA=new ClassA(10);
ClassB objB=new ClassB(-20,'a');
objA.show();
```

```
objB.show();
objB.set(100);
objB.show();
objB.set(0,'z');
objB.show();}
}
```

Метод *show()* в классе *ClassB* переопределяется. Сигнатура описанного в классе *ClassB* метода *show()* совпадает с сигнатурой метода *show()* класса *ClassA*. Если в классе *ClassA* отображается информация о номере объекта и значении его поля *number*, то в классе *ClassB* метод *show()* выводит еще и значение поля *symbol*. При этом в переопределенном методе *show()* вызывается также прежняя (исходная) версия метода из класса *ClassA*. Для этого используется инструкция вида *super.show()*. Этот исходный вариант метода, кроме прочего, считывает из ненаследуемого (но реально существующего) поля *code* порядковый номер объекта и отображает его в выводимом на экран сообщении.

Метод *set()* в классе *ClassB* перегружается. Хотя в классе *ClassA* есть метод с таким же названием, сигнатуры методов в суперклассе и подклассе разные. В суперклассе у метода *set()* один числовой аргумент, а в подклассе у этого метода два аргумента: числовой и символьный. Поэтому в классе *ClassB* имеется два варианта метода *set()* – с одним и двумя аргументами. Первый наследуется из суперкласса *ClassA*, а второй определен непосредственно в подклассе *ClassB*.

Обработка исключительных ситуаций

Исключительная ситуация – это ошибка, которая возникает в результате выполнения программы. Исключение в *Java* – это объект, который описывает исключительную ситуацию (ошибку).

При возникновении исключительной ситуации в процессе выполнения программы автоматически создается объект, описывающий эту ситуацию. Этот объект передается для обработки методу, в котором возникла ошибка. Говорят, что исключение выбрасывается в метод. По получении объекта исключения метод может обработать его или передать для обработки дальше.

В *Java* для обработки ошибок используется блок *try-catch-finally*. В блок *try* помещается программный код, который отслеживается на случай, если возникнет исключительная ситуация. Если она возникает, то управление передается блоку *catch*. Программный код в этом блоке выполняется, только если возникает ошибка, причем не любая, а определенного типа. Аргумент, определяющий какого типа исключительные ситуации обрабатываются в блоке *catch*, указывается после ключевого слова *catch* в круглых скобках.

Поскольку в блоке *try* могут возникать исключения разных типов, для каждого из них можно предусмотреть свой блок *catch*. Если таких блоков несколько, при возникновении ошибки они перебираются последовательно до совпадения типа исключительной ситуации с аргументом блока *catch*.

После блоков *try* и *catch* можно указать блок *finally* с кодом, который выполняется в любом случае, независимо была ли исключительная ситуация.

Общая схема использования блока *try-catch-finally* выглядит следующим образом:

```
try{
//код, который генерирует исключение
}
catch (тип_исключения_1 объект)
{//код для обработки исключения}
catch (тип_исключения_2 объект)
{//код для обработки исключения}
…
finally{
//код, который выполняется обязательно
}
```

Если при исполнении программного кода в блоке *try* ошибок не возникает, после выполнения этого блока выполняется блок *finally* (если он имеется), затем управление передается следующей команде после конструкции *try-catch-finally*.

При возникновении ошибки код в блоке *try* останавливается, начинается поиск подходящего блока *catch*. Если же нужного блока нет, исключение выбрасывается из метода и должно быть обработано внешним к методу обратным кодом.

Если возникает ошибка, обработка которой не предусмотрена, то вызывается обработчик исключительной ситуации по умолчанию. После вызова этого обработчика, программа завершает работу.

Классы исключений

В *Java* существует целая иерархия классов, предназначенных для обработки исключений. Во главе этих классов находится класс *Throwable*, который в свою очередь является подклассом класса *Object*. У класса *Throwable* есть два подкласса *Exception* и *Error*. К классу *Error* относятся «катастрофические» ошибки, которые невозможно обработать в программе, например переполнение стека памяти. У класса *Exception* есть подкласс *RuntimeException*, к нему относятся ошибки времени выполнения программы, которые перехватываются программами пользователя. Исключения для этого класса определяются автоматически. К ним относятся такие ошибки, как деление на ноль и выход за пределы массива, ошибка операций с указателем, ошибка доступа (рис. 1).

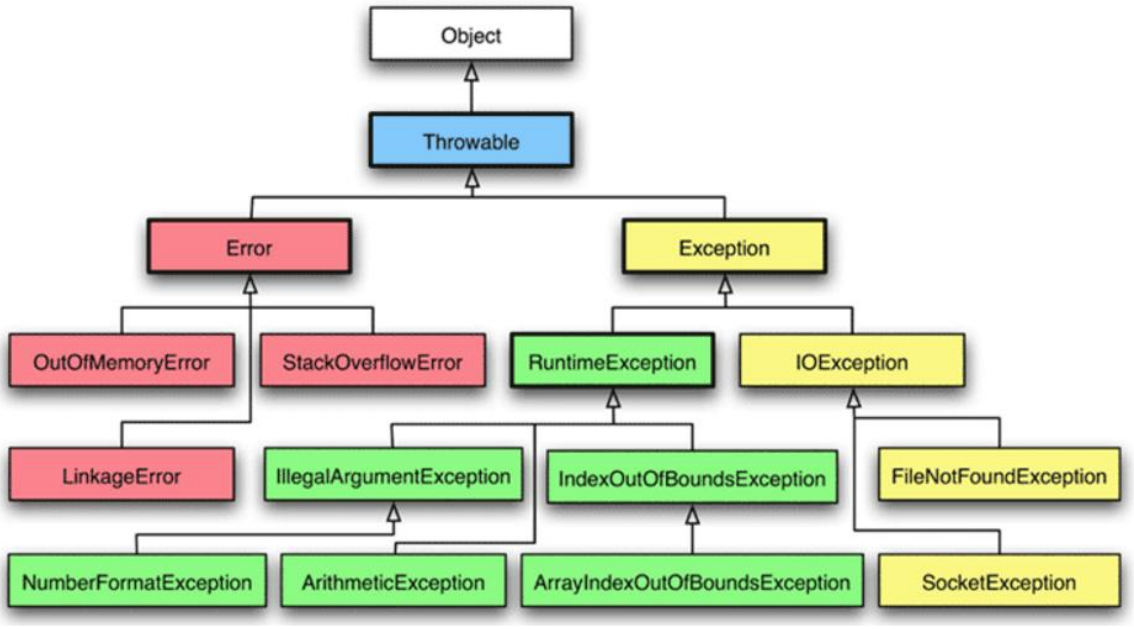

Рис. 1 Иерархия классов исключений

Пример обработки исключительной ситуации деления на ноль:

```
// Импорт класса Random:
import java.util.Random;
```

```
class ArithExcepionDemo{
public static void main(String args[]){
int a=0,b=0,c=0;
// Объект для генерирования случайных чисел:
Random r=new Random();
for(int i=0;i<32000;i++){
try{
b=r.nextInt(200);
c=r.nextInt(100);
// Возможно деление на ноль:
a=10000/b/c;
}catch(ArithmeticException e){
// Обработка ошибки:
System.out.println("Деление на ноль!");
a=0;}
System.out.println("a="+a);}
}
}
```

В *Java* все исключения делятся на два типа – **контролируемые** или **проверяемые** (*checked*) и **неконтролируемые** или **непроверяемые** (*unchecked*): те, которые перехватывать обязательно, и те, которые перехватывать не обязательно. По умолчанию – все исключения обязательно нужно перехватывать. Например, если в методе выбрасываются (возникают) исключения *ClassNotFoundException* и *FileNotFoundException*, программист обязан указать их в сигнатуре метода (в заголовке метода). Для этого после имени класса следует ключевое слово *throws*, после которого через запятую перечисляются исключения. Это *checked*-исключения. Вот как это будет выглядеть:

```
public static void method1() throws ClassNotFoundException,
FileNotFoundException

public static void main() throws IOException

public static void main() //не выбрасывает никаких исключений
```

Метод, который вызывает *method1*, должен сделать одно из двух:

1. Перехватить и обработать эти исключения.
2. Или пробросить эти исключения дальше.

Способ 1. Пробрасывание исключения выше (вызывающему методу):

```
public static void main(String[] args)  throws FileNotFoundException,
ClassNotFoundException
{
    method1();
}

public static void method1() throws FileNotFoundException,
ClassNotFoundException
```

```
{
 //тут возникнет исключение  FileNotFoundException - такого файла нет
 FileInputStream fis = new FileInputStream("C2:\\badFileName.txt");
}
```

Способ 2. Перехват исключения с помощью *try-catch* (табл. 1):

```
public static void main(String[] args)
{
    try
    {
        method1();
    }
    catch(Exception e)
    {
    }
}

public static void method1() throws FileNotFoundException,
ClassNotFoundException
{
 //тут возникнет исключение  FileNotFoundException - такого файла нет
 FileInputStream fis = new FileInputStream("C2:\\badFileName.txt");
}
```

Таблица 1. Контролируемые исключения

Исключение	**Описание**
ArithmeticException	Арифметическая ошибка (например, деление на ноль)
ArrayIndexOutOfBoundsException	Индекс массива за пределами допустимых границ
ArrayStoreException	Присваивание элементу массива недопустимого значения
ClassCastException	Недопустимое приведение типов
IllegalArgumentException	Методу передан недопустимый аргумент
IndexOutOfBoundsException	Некоторый индекс находится за пределами допустимых для него границ
NegativeArraySizeException	Создание массива отрицательного размера
NullPointerException	Недопустимое использование нулевого указателя
NumberFormatException	Недопустимое преобразование текстовой строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексирования вне пределов строки
UnsupportedOperationException	Недопустимая операция

Но есть вид исключений – это исключения класса *RuntimeException* и классы, унаследованные от него. Их перехватывать не обязательно. Это

unchecked-исключения. Считается, что это трудно прогнозируемые исключения и предсказать их появление практически невозможно. С ними можно делать все то же самое, но указывать в *throws* их не нужно (табл. 2).

Таблица 2. Неконтролируемые исключения

Исключение	Описание
ClassNotFoundException	Класс не найден
IllegalAccessException	В доступе к классу отказано
InstantiationException	Попытка создать объект абстрактного класса или интерфейса
InterruptedException	Один поток прерван другим потоком
NoSuchFieldException	Поле не существует
NoSuchMethodException	Метод не существует

Описание исключительной ситуации

В классе *Trowable* переопределен метод *toString*, в качестве результата он возвращает строку, описывающую соответствующую ошибку. Напомним, что этот метод вызывается автоматически, например, при передаче объекта исключения методу *println()* в качестве аргумента.

```
class MoreExceptionDemo{
public static void main(String args[]){
int a,b;
try{
b=0;
// Деление на ноль:
a=100/b;
}catch(ArithmeticException e){
// При обработке ошибки использован объект исключения:
System.out.println("Ошибка: "+e);}
System.out.println("Выполнение программы продолжено!");}
}
```

При запуске этой программы в первой строке, выведенной на экран, текст после слова «Ошибка» появился в результате преобразования объекта *e* в текстовый формат.

```
Ошибка: java.lang.ArithmeticException: / by zero
Выполнение программы продолжено!
```

Множественный блок catch{}

Для каждого типа исключения можно создавать свой блок *catch*. Блоки размещаются один за другим и им в соответствии с типом передаются разные аргументы в соответствии с типом обрабатываемой ошибки.

```
import java.util.Random;
class MultiCatchDemo{
public static void main(String args[]){
Random r=new Random();
int MyArray[]={0,2};
int a,b;
for(int i=1;i<10;i++){
try{
a=r.nextInt(3);
b=10/MyArray[a];
System.out.println(b);
}catch(ArithmeticException e){
System.out.println("Деление на ноль!");}
catch(ArrayIndexOutOfBoundsException e){
System.out.println("Выход за границы массива!");}
finally{System.out.println("**************");}}
System.out.println("Цикл for завершен!");}
}
```

В этом примере в цикле командой *a=r.nextInt(3)* переменной a присваивается случайное число от 0 до 2 включительно. После команды *b=10/MyArray[a]* может случиться две исключительные ситуации. Первая, это деление на ноль, когда переменная a примет значение 0. Вторая – выход за пределы массива, когда переменная примет значение 2, т.к. элемента с индексом 2 в массиве *MyArray* нет.

Для обработки этих ошибок используются соответствующие блоки *catch*. Блок *finally* содержит команду разделительной «звездной» полосы, которая выводится независимо от того, произошла ли ошибка или нет.

Создание собственных исключений

Классы встроенных исключений в *Java* описывают только наиболее часто встречаемые ошибки, поэтому иногда возникает необходимость в описании собственного исключения. В *Java* такая возможность имеется.

Технически создание собственного класса исключения сводится к созданию подкласса класса *Exception*, который, в свою очередь, является подклассом класса *Trowable*.

Программа с пользовательским классом исключения:

```
// Класс исключения:
class MyException extends Exception{
private double min;
private double max;
private String error;

// Конструктор:
MyException(double x,double y,String str){
min=x;
```

```
max=y;
error=str;}

// Переопределение метода toString():
public String toString(){
return "Произошла ошибка ("+error+"): попадание в диапазон
["+min+","+max+"]";}
}

class MyExceptionDemo{
// Метод выбрасывает исключение пользовательского типа:
static double MyLog(double x) throws MyException{
if(x<0||x>1) return Math.log(x*(x-1));
else throw new MyException(0,1,"неверный аргумент");
}

public static void main(String args[]){
double x=-1.2,y=1.2,z=0.5;
try{
System.out.println("ln("+x+")="+MyLog(x));
System.out.println("ln("+y+")="+MyLog(y));
System.out.println("ln("+z+")="+MyLog(z));
}catch(MyException e){// Обработка исключения
System.out.println(e);}
}
}
```

В программе описывается класс пользовательского исключения *MyException*, который наследует класс *Exception*. У класса есть три закрытых поля *min, max, erro*r. Также в классе переопределяется метод *toString()*. Этим методом возвращается строка, в которой содержится информация всех трех полей класса.

В главном методе программы определяется статический метод *MyLog()*, описанный как способный выбрасывать исключение пользовательского класса *MyException*.

В теле метода для значения аргумента *x* вычисляется натуральный логарифм по формуле:

$$\ln(x \cdot (x-1)).$$

Если аргумент, переданный методу *MyLog()*, лежит вне диапазона [0,1], методом возвращается значение *Math.log(x*(x-1))*. Если же аргумент попадает в этот диапазон, приведенное выражение вычислено быть не может, поскольку у натурального логарифма аргумент отрицательный. В этом случае методом *MyLog()* генерируется и выбрасывается исключение пользовательского типа *MyException*. Аргументами конструктору передаются границы диапазона [0,1] и описание ошибки (неверный аргумент).

В главном методе программы выполняется попытка вычислить значение методом *MyLog()* и вывести его на экран. При этом отслеживается соответствующая ошибка класса *MyException*. Если она возникает, то обрабатывается.

Варианты для выполнения контрольной работы

Варианты 1–10

Задание. Создайте программу из трех классов: суперкласса, подкласса и класса приложения, демонстрирующего работу всех методов объекта подкласса. Продемонстрируйте работу вех принципов ООП: наследование, полиморфизм, инкапсуляция. Предусмотрите обработку исключительных ситуаций.

Вариант № 1.
Суперкласс: ПАРА_ЧИСЕЛ (*PAIR*).
Первое_число (*first*) - *int*.
Второе_число (*second*) – *int* .
Определить методы изменения полей и вычисления произведения чисел.
Создать подкласс ПРЯМОУГОЛЬНИК (*RECTANGLE*), с полями-сторонами. Определить методы для вычисления площади и периметра прямоугольника.

Вариант № 2.
Суперкласс: ПАРА_ЧИСЕЛ (*PAIR*).
Первое_число (*first*) – *int.*
Второе_число (*second*) – *int.*
Определить методы изменения полей и операцию сложения пар *(a,b)+(c,d)=(a+b,c+d).*
Создать подкласс ДЕНЕЖНАЯ_СУММА (*MONEY*), с полями Рубли и Копейки. Переопределить операцию сложения и определить операции вычитания и деления денежных сумм.

Вариант № 3.
Суперкласс: ПАРА_ЧИСЕЛ (*PAIR*).
Первое_число (*first*) – *int.*
Второе_число (*second*) – *int*.
Определить методы проверки на равенство и операцию перемножения полей.
Реализовать операцию вычитания пар по формуле *(a,b)-(c,d)=(a-b,c-d).*
Создать подкласс ПРОСТАЯ_ДРОБЬ (*RATIONAL*), с полями Числитель и Знаменатель. Переопределить операцию вычитания и определить операции сложения и умножения простых дробей.

Вариант № 4.
Суперкласс: ТРОЙКА_ЧИСЕЛ (*TRIAD*).
Первое_число (*first*) – *int.*
Второе_число (*second*) – *int.*
Третье_число (*third*) – *int.*
Определить методы изменения полей и сравнения триады. Создать производный класс *TIME* с полями часы, минуты и секунды. Определить полный набор операций сравнения временных промежутков.

Вариант № 5.
Суперкласс: ТРОЙКА_ЧИСЕЛ (*TRIAD*).
Первое_число (*first*) – *int.*
Второе_число (*second*) – *int.*
Третье_число (*third*) – *int.*
Определить методы изменения полей и увеличения полей на 1.
Создать подкласс *TIME* с полями часы, минуты и секунды. Переопределить методы увеличения полей на 1 и определить методы увеличения на *n* секунд и минут.

Вариант № 6.
Суперкласс: ЧЕЛОВЕК (*PERSON*).
Имя (*name*) – *String.*
Возраст (*age*) – *int.*
Определить методы изменения полей.
Создать подкласс *EMPLOYEE*, имеющий поля Должность – *String* и Оклад – *double*. Определить методы изменения полей и вычисления зарплаты сотрудника по формуле Оклад + Премия (% от оклада).

Вариант № 7.
Суперкласс: ЧЕЛОВЕК (*PERSON*).
Имя (*name*) – *String*.
Возраст (a*ge*) – *int*.
Определить методы изменения полей.
Создать подкласс *STUDENT*, имеющий поля Предмет – *String* и Оценка – *int*. Определить методы изменения полей и метод, выдающий сообщение о неудовлетворительной оценке.

Вариант № 8.
Суперкласс: ЧЕЛОВЕК (*PERSON*).
Имя (*name*) – *String*.
Возраст *(age*) – *int.*
Определить методы изменения полей.
Создать подкласс *TEACHER*, имеющий поля Предмет – *String* и Количество часов – *int*. Определить методы изменения полей, а также увеличения и уменьшения часов.

Вариант № 9.
Суперкласс: ЧЕЛОВЕК (*PERSON*).
Имя (*name*) – *String*.
Возраст (*age*) – *int.*
Определить методы изменения полей.
Создать подкласс *STUDENT*, имеющий поле год обучения. Определить методы изменения и увеличения года обучения.

Вариант № 10.
Суперкласс: ПАРА_ЧИСЕЛ (*PAIR*).
Первое_число (*first*) – *int.*
Второе_число (*second*) – *int.*
Определить методы изменения полей и вычисления произведения чисел.
Создать подкласс ПРЯМОУГОЛЬНЫЙ_ТРЕУГОЛЬНИК (*RIGHTANGLED*), с полями-катетами. Определить метод вычисления гипотенузы.

Вариант № 11–20

Задание:
1) реализовать класс с методами согласно заданию;
2) предусмотреть все возможные исключительные ситуации и обработать их;
3) в классе приложения создать массив объектов этого класса и продемонстрировать их работу в цикле.

Вариант № 11.

Класс-контейнер ВЕКТОР с элементами типа *int.*
Реализовать методы: доступ к элементу по индексу; добавить элемент в вектор (постфиксная операция добавляет элемент в конец, префиксная в начало); вывод вектора на экран.

Вариант № 12.

Класс-контейнер ВЕКТОР с элементами типа *int.*
Реализовать методы: доступ к элементу по индексу; определение размера вектора; умножение элементов векторов *a[i]*b[i];* переход вправо к элементу с номером n; вывод вектора на экран.

Вариант № 13.

Класс-контейнер МНОЖЕСТВО с элементами типа *int.* (Используйте в качестве множества любой известный Вам массив).
Реализовать методы: доступ к элементу по индексу; определение размера множества; поиск элемента по значению (вернуть номер элемента); удаление элемента из множества; вывод множества на экран.

Вариант № 14.

Класс-контейнер МНОЖЕСТВО с элементами типа *int*. (Используйте в качестве множества любой известный Вам массив).
Реализовать методы: доступа по индексу; проверка на неравенство; принадлежность числа множеству; переход вправо на один элемент к элементу с номером *n*; вывод вектора на экран.

Вариант № 15.

Класс-контейнер СПИСОК с ключевыми значениями типа *int*.
Реализовать методы: доступ по индексу; определение размера списка; сложение элементов списков *a[i]+b[i]*; - *n* - переход влево к элементу с номером *n*; вывод вектора на экран.

Вариант № 16.

Класс-контейнер СПИСОК с ключевыми значениями типа *int*.
Реализовать методы: доступа по индексу; добавление списка *b* к списку *a (a+b);* добавление элемента в начало списка; вывод вектора на экран.

Вариант № 17.

Класс-контейнер СПИСОК с ключевыми значениями типа *int*.
Реализовать методы: доступ по индексу; определение размера списка; умножение значений элементов списков *a[i]*b[i];* переход вправо к элементу с номером *n*; вывод вектора на экран.

Вариант № 18.

Класс-контейнер ВЕКТОР с элементами типа *int.*
Реализовать методы: доступ по индексу; определение размера вектора; удаление *n* элементов из конца вектора; добавление *n* элементов в конец вектора; вывод вектора на экран.

Вариант № 19.

Класс-контейнер ВЕКТОР с элементами типа *int.*
Реализовать методы: доступ к элементу по индексу; определение размера вектора; добавление константы ко всем элементам вектора; удаление *n* элементов из конца вектора; вывод вектора на экран.

Вариант № 20.

Класс-контейнер СПИСОК с ключевыми значениями типа *int*.
Реализовать методы: доступ по индексу; определение размера списка; умножение всех элементов списка на число; переход влево к элементу с номером *n*; вывод вектора на экран.

Список литературы

1. А.Н. Васильев. Java. Объектно-ориентированное программирование: Учебное пособие. – СПб: Питер, 2012
2. И.Ю. Баженова. Языки программирования: Учебное пособие. – М.: Академия, 2012
3. Кей С. Хорстманн, Гари Корнелл. Java 2. Библиотека профессионала. Т. 1 Основы. – М.: Вильямс, 2010
4. Хортсман К.С. Java 2.Библиотека профессионала.В 2-х т. Т.2.Тонкости программирования [Текст] / Хортсман К.С. - 8-е изд.:пер. с англ. - Москва : ООО «И.Д.Вильямс», 2012. - 992 с. : ил.
5. Эккель, Б. Философия Java [Текст], 4-е изд. - СПб. : Питер, 2011. - 640 с. : ил. - (Б-ка программиста). - ISBN 978-5-459-00859-3.
6. Java для начинающих [Электронный ресурс] – http://study-java.ru/
7. Интернет-проект JAVARUSH [Электронный ресурс] – http://javarush.ru/
8. Викиучебник Java [Электронный ресурс] - https://ru.wikibooks.org/wiki/Java

Учебное издание

Григалашвили Алена Сергеевна

ПРОГРАММИРОВАНИЕ НА JAVA

Учебно-методическое пособие

Корректор *Н.В. Шиляева*

Подписано в печать 01.10.2017. Формат 60×90/16.
Усл. печ. л. 3,5. Тираж 70 экз. Заказ 373/2017

Отпечатано с готового оригинал-макета в типографии Центра
«Издательство Пермского национального исследовательского
политехнического университета».
Адрес: 614990, г. Пермь, Комсомольский проспект, 29, к. 113
Тел. (342) 219-80-33