Н.С. Лагутина
Ю.А. Ларина

Основы объектно-ориентированного программирования на языке JAVA

Оглавление

Введение

Учебно-методическое пособие содержит описание основных элементов и конструкций языка Java. Классы, интерфейсы, наследование, исключения, классы стандартной библиотеки, концепции объектно-ориентированного программирования описаны теоретически, а также рассматриваются на простых примерах.

Первая глава пособия является вводной и рассматривает простейшее приложение на Java с целью познакомить начинающего программиста со структурой программы. Вторая и третья главы посвящены описанию основных конструкций языка и массивов, как самых простых средств обработки и хранения данных. В четвертая и пятой главах обсуждаются возможности некоторых существующих классов стандартных пакетов Java. Шестая глава посвящена классам для работы со строками. Эти главы показывают широкий спектр задач, которые решаются уже существующими классами.

Пять следующих глав (7 – 11) описывают основные понятия и принципы объектно-ориентированного подхода к программированию. Это наиболее современная и эффективная технология создания программных продуктов. Материал этих глав полезен при разработке программ не только на языке Java, но и на многих других (C++, Python, C#, и т.д.). Все основные конструкции языка и приёмы объектно-ориентированного программирования подробно проиллюстрированы примерами кода. Последняя глава обсуждает особенности ввода и вывода в Java.

Пособие предназначено для студентов направления 02.03.02 Фундаментальная информатика и информационные технологии для использования в ходе изучения материала курса «Практикум на ЭВМ по объектно-ориентированному программированию». Кроме этого оно полезно при освоении курса «Объектно-ориентированное программирование», а также может быть использовано как учебное пособие для курсов «Языки и методы программирования» и «Практикум на ЭВМ по объектно-ориентированному программированию» для направления 01.03.02 Прикладная математика и информатика.

Пособие может быть полезно для всех преподавателей, ведущих занятия по программированию, и для программистов, начинающих изучать язык Java.

1. «Привет, JAVA!»

В основе конструкции языка Java лежит понятие класса. Класс определяет форму и сущность объекта и образует основу объектно-ориентированного программирования. По сути, класс определяет новый тип данных, являющийся идеализированным представлением сущности из предметной области. В описание класса вносятся свойства и поведение сущности, которые важны для решения поставленной задачи и игнорируются несущественные характеристики. После определения нового типа его можно использовать для создания объектов (экземпляров) данного класса.

Любой элемент предметной области, который требуется реализовать программно, используя технологию программирования Java, должен быть заключен внутри класса. Таким образом, весь программный код, написанный на языке Java, инкапсулируется в пределах поименованных классов. То есть программа, написанная на языке Java, представляет собой один или несколько классов. ***Каждый класс реализуется в отдельном файле с расширением .java. Имя класса начинается с большой буквы и должно совпадать с именем файла, в котором класс будет реализован.***

Классы, группируются по пакетам в соответствии со своим назначением. Внутри пакета у каждого класса должно быть свое уникальное имя, но в разных пакетах имена классов могут повторяться. Например, мы можем создать класс List, поместить его в пакет и не беспокоится о том, что кто-то еще может создать класс с точно таким же именем. Таким образом, пакет является специальным модулем, который содержит группу классов, объединённых в одном пространстве имён. Распределение классов по пакетам позволяет с одной стороны решить проблему конфликта имен, а с другой предоставляет возможность более эффективно осуществлять управление доступом, то есть позволяет упростить навигацию по проекту. Для обращения к элементам пакета используется путь, состоящий из набора подпакетов, разделенных точками. Внутри одного пакета используются просто имена классов без обращения к имени пакета.

В любом проекте на языке Java обязательно присутствует ***главный класс***, так как без него невозможны сборка и работа приложения. ***Главный класс*** – это класс, который обязательно содержит метод main, используемый в качестве точки входа в программу. Главный класс принято называть MainClass или именем разрабатываемого приложения.

В качестве первой программы рассмотрим традиционный пример, выводящий на экран сообщение «Привет, JAVA!». Для этого создадим пакет **myClasses** и добавим в него главный класс с именем **MainClass**. Главный класс размещается в файле с именем MainClass.java.

```
package myClasses;

public class MainClass {

   public static void main(String[] args) {
      System.out.println("Привет, JAVA!");
   }
}
```

В первой строке кода записано утверждение package, определяющее пакет myClasses, которому принадлежат классы, описанные в данном модуле.

Далее идет описание класса, начинающееся с модификатора доступа. В данном примере – public, то есть публичный, общедоступный. Начало класса отмечается служебным словом class, после которого следует имя класса.

Само описание (тело) класса содержится внутри фигурных скобок. В нашей простейшей программе тело класса содержит только один метод main(). Метод является общедоступным (модификатор public) и статическим (модификатор static). Модификатор static обеспечивает возможность вызова метода main() в начале программы без создания экземпляра (объекта) класса.

Все, что содержит метод или тело метода, записывается внутри фигурных скобок. В нашем примере метод выполняет единственное действие – вывод на консоль сообщение «Привет, JAVA!». Для этого в классе System определяется переменная с именем out, являющаяся экземпляром класса PrintStream (один из классов Java API), в котором есть метод println().

2. Основы Java

В этой главе мы рассмотрим основные конструкции языка Java.

2.1 Переменные и идентификаторы

Элементы программы, с которыми работает программист, имеют имена-идентификаторы. ***Идентификаторы*** используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и $ (доллар). Java — язык, чувствительный к регистру букв. Это означает, что, Value и VALUE — различные идентификаторы.

Переменная — это основной элемент хранения информации в программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for:

```
for(int i=0; i<25;i++){...}
```

Либо это может быть переменная экземпляра класса, доступная всем методам данного класса:

```
public class MainClass {
   private int i;
   private void method1() {... i = 0;...}
   public void method2() {... i = 1;...}
}
```

Локальные области действия объявляются с помощью фигурных скобок. Область действия и время жизни локальной переменной ограничивается блоком, в котором эта переменная описана.

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

```
final double F = 1.618;
```

2.2 Типы данных

Типы данных задают основные возможности любого языка. В Java определено 8 примитивных типов данных: семь числовых и один логический. К числовым относятся четыре целочисленных `byte, short, int, long`, два вещественных `float` и `double` и символьный `char.` Логический тип `boolean`. Главная особенность примитивных типов данных в Java состоит в том, что переменные этих типов не являются объектами. Переменные всех остальных типов, включая массивы и переменные строкового типа `String`, являются объектными ссылками. Подробнее они будут рассмотрены далее.

Тип `byte` — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла. Тип `short` — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Эти два типа используются только для специальных целей, например для манипуляций с битами. Для целочисленных счетчиков, массивов данных и в арифметических выражениях следует использовать тип `int`.

Тип `int` служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Тип `long` предназначен для представления 64-битовых чисел со знаком. Его диапазон значений от -2^{63} до 2^{63}.

Числа с плавающей точкой, часто называемые вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В переменных с обычной, или одинарной точностью, объявляемых с помощью ключевого слова `float`, для хранения вещественного значения используется 32 бита. В случае двойной точности, задаваемой с помощью ключевого слова `double`, для хранения значений используется 64 бита. Все математические функции, такие, как `sin`, `cos`, `sqrt`, возвращают результат типа `double`.

В Java для представления символов в строках используется кодировка Unicode, разрядность типа `char` в этом языке — 16 бит. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов. Хотя величины типа char не используются, как целые числа, с ними можно оперировать так же как с целыми числами. Можно сложить два символа вместе, или прибавить число к символьной переменной.

Переменная типа `boolean` имеет два значения: `false` и `true`. Они используются для вычисления логических выражений. Использование булевских переменных в арифметических выражениях и наоборот невозможно.

Преобразования типов в Java могут осуществляться при присваивании значений переменным, при выполнении цепочки операций, при передаче аргументов в методы. Они делятся на ***неявные*** и ***явные***. Неявные преобразования осуществляются компилятором автоматически по следующим правилам:

1. Типы приводимы друг к другу. Среди примитивных типов приводимыми являются числовые типы, а логический тип не приводится ни к одному другому.
2. Меньший по размеру тип приводится к большему.

Приведем пример неявного преобразования типов.

```
(1) int num;
(2) num = 200;
(3) double rez;
(4) rez = num;
(5) long longNum;
(6) longNum = 13;
(7) rez = 7 +  longNum * num;
```

В четвертой строке переменной типа `double` присвоили значение переменной типа `int`, а компилятор автоматически произвел преобразование числа 200 из целого в вещественное. В седьмой строке произошло несколько неявных преобразований в ходе выполнения цепочки операций. Сначала в ходе умножения значение переменной `num` типа `int` было преобразовано к `long`, затем целое число 7 преобразовано также к `long` в процессе выполнения операции сложения. Последняя операция в цепочке – операция присваивания, здесь результат вычисления арифметического выражения, имеющий тип `long` был преобразован к `double` и записан в переменную `rez`.

Явное преобразование типов производится, если необходимо переменную большего типа преобразовать к меньшему. Для этого перед приводимым выражением надо в скобках указать тип к которому следует осуществить приведение. Рассмотрим пример явного приведения типов при передаче аргументов в метод. Пусть существует некоторый метод с тремя целочисленными параметрами:

```
public static double method (int a, int b, int c) { … }
```

При вызове этого метода возможно явное приведение типов:

```
double a1 = 5.67;
int b1 = 20;
long c1 = 13000;
double rez = method ((int)a1, b1, (int)c1);
```

В последней строчке примера происходит преобразование двух значений переменных a1 и c1 к меньшему типу int при передаче значений аргументов в метод во время его вызова. При явном приведении типов следует помнить, что это может привести к потере значений, так как происходит усечение больших по размеру типов.

2.3 Операции

Операцией можно назвать любое действие, задаваемое некоторой синтаксической конструкцией, которое можно использовать в выражениях и которое возвращает определенный результат. Рассмотрим основные операции Java.

К арифметическим относятся следующие операции: инкремента ++ и декремента --, операции сложения +, вычитания -, умножения *, деления / и нахождения остатка от деления %, операции с присваиванием +=, -=, *=, /=.

Арифметические операции применяются к числовым типам и имеют результат также числового типа. Операция деления для целочисленных типов даёт результат целочисленного типа (неполное частное), для вещественных – вещественного. Единственной операцией в целочисленной арифметике, приводящей к ошибке, является операция деления на 0.

При выполнении операций в вещественной арифметике ошибка не возникает никогда. При переполнениях и делении на ноль результат получает одно из специальных значений NaN (не число), POSITIVE_INFINITY (плюс бесконечность), NEGATIVE_INFINITY (минус бесконечность). Эти значения определены в классах Float и Double. Результат вычисления выражения 0.0/0.0 равен Double.NaN.

К операциям сравнения относятся операции проверки на равенство ==, неравенство != , меньше <, больше >, меньше или равно <=, больше или равно >=. Их результат всегда имеет тип boolean.

Логические операции применяются только к аргументам типа boolean и дают результат того же типа. К логическим относятся:операция логического отрицания !, операции логического и &&, или ||, исключающего или ^, а также аналогичные операции с присваиванием &=, |=, ^=.

2.4 Операторы управления

Важной частью любого языка программирования являются ***операторы управления*** потоком выполнения действий. Все управляющие операторы можно разделить на три категории:

- операторы выбора, которые позволяют программе выполняться тем или иным образом в зависимости от условия.
- операторы цикла, которые позволяют повторять выполнение одного или нескольких действий.
- операторы перехода, которые позволяют пропускать выполнение одного или нескольких операторов и переходить сразу к нужному действию.

Все операторы, выполняющие проверку условий, требуют, чтобы условие имело тип boolean.

В языке Java существует всего два оператора выбора: оператор `if-else` и оператор `switch-case`. Они позволяют управлять выполнением той или иной программы в зависимости от заданного условия.

В приведенном ниже примере метод класса с помощью оператора `if-else` проверяет, могут ли три вещественных числа быть сторонами какого-нибудь треугольника.

```
public static boolean isTriangle(double a, double b, double c) {
      if ((a < b + c) && (b < c + a) && (c < a + b))
         return true;
      else
         return false;
}
```

Оператор выбора `switch-case` позволяет направить поток программы по одному из возможных путей в зависимости от значения управляющего оператора. Использование данного оператора представляется более удобным по сравнению с оператором `if` в тех случаях, когда необходимо сделать выбор из более чем двух вариантов. Управляющее выражение в операторе `switch` должно быть одного из целочисленных типов. В последних версиях Java в качестве управляющего выражения можно использовать строки (объекты класса `String`).

По ходу программы управляющее выражение сравнивается со значением в каждом блоке `case`. Если будет обнаружено совпадение, то выполниться последовательность операторов, находящаяся в данном блоке, если совпадений не будет найдено, то выполнятся действия в блоке `default`. Использование блока `default` не обязательно. Если его не будет, то не найдя совпадений, программа не выполнит ни одного оператора. В каждом блоке `case` есть оператор `break`, который прерывает выполнение всего блока switch после обнаружения совпадения. Если этого оператора не будет, то программа продолжит выполнять операторы, которые находятся после блока, в котором найдено совпадение.

Приведем два примера использования оператора `switch-case`. В первом управляющее выражение является целочисленным:

```
public class TimeYear {
   public static void main(String[] args) {
      int timeOfYear = 2;
      switch (timeOfYear) {
         case 1:
            System.out.println("Зима");
            break;
         case 2:
            System.out.println("Весна");
            break;
         case 3:
            System.out.println("Лето");
            break;
         case 4:
            System.out.println("Осень");
            break;
         default:
            System.out.println("Соответствия не найдено!");
      }
```

```
    }
}
```

Второй пример демонстрирует использование строк в управляющем выражении:

```
public class TimeYear {
    public static void main(String[] args) {
        String timeOfYear = "Зима";
        switch (timeOfYear) {
            case "Зима":
                System.out.println("Холодно");
                break;
            case "Весна":
                System.out.println("Скоро лето");
                break;
            case "Лето":
                System.out.println("Каникулы");
                break;
            case "Осень":
                System.out.println("Опять учиться");
                break;
            default:
                System.out.println("Соответствия не найдено!");
        }
    }
}
```

Циклы в Java представляют собой специальные конструкции, которые позволяют выполнять один или несколько операторов многократно до достижения условия завершения цикла. В Java существуют следующие операторы цикла: `for, while, do-while.` Оператор `while` позволяет выполнять какой-то блок операторов до тех пор, пока условие цикла является истинным. Тело цикла может не выполниться ни разу, если условие цикла изначально ложно. Однако, иногда возникают ситуации, когда нам необходимо выполнить цикл хотя бы один раз, даже если условие изначально ложно. Для этих целей в языке Java предусмотрен цикл `do-while`. Этот цикл, сначала выполняет операторы цикла, а уже затем проверяет условие, и если оно ложно, то операторы больше не будут выполняться. Таким образом, в любом случае тело цикла будет выполнено хотя бы один раз.

```
int i = 1, n = 100;
while (i < n) {
    System.out.println(i + "*" + i + "=" + (i * i));
    i++;
}
```

В этом примере будут напечатаны квадраты целых чисел от 1 до 99 включительно. Если `n` окажется меньше либо равен единицы, то программа не выведет ничего. Следующий пример решает туже самую задачу, но квадрат единицы будет выведен в любом случае.

```
int i = 1, n = 100;
do {
    System.out.println(i + "*" + i + "=" + (i * i));
    i++;
    } while (i < n);
```

Цикл `for` проводит инициализацию перед первым шагом цикла. Затем выполняется проверка условия цикла, и в конце каждой итерации происходит изменение управляющей переменной. Выглядит это следующим образом:

```
for(int i=1; i<100; i++) {
   System.out.println(i+"*"+i+"="+(i*i));
}
```

Три части заголовка цикла : инициализация, логическое выражение, шаг, разделяются точкой с запятой и могут быть пустыми. В разделе инициализации можно использовать несколько выражений, отделив их запятыми. Условие проверяется перед выполнением каждой итерации цикла. Если условие окажется ложным, то выполнение продолжится с инструкции, следующей за конструкцией `for`. Важно помнить, что выражение инициализации выполняется один раз в начале выполнения цикла, затем обязательно проверяется условие, шаг выполняется в конце каждой итерации.

В Java есть три оператора перехода: `break, continue, return.` Операторы перехода прерывают последовательность действий блока и передают управление другой части программы.

Оператор `break` завершает последовательность операторов в операторе `switch` или позволяет выйти из ближайшего цикла. При использовании вложенных циклов оператор `break` осуществляет выход только из самого внутреннего цикла, не оказывая влияния на внешний цикл.

Иногда требуется, чтобы повторение цикла начиналось с более раннего оператора его тела. В циклах `while` и `do-while` оператор `continue` вызывает передачу управления непосредственно управляющему условному выражению цикла. В цикле `for` управление передаётся вначале шагу цикла `for`, а потом условному выражению. При этом во всех циклах промежуточный код пропускается.

```
double sinX=0, r=1, k=1, x=0.5, e=0.0001;
for (int n = 1; n < 100; n++) {
   r=r*x*1/n;
   System.out.println(r);
   if(r<e) break;
   if (n % 2 == 0)
      continue;
   sinX+=k*r;
   k*=(-1);
}
System.out.println(sinX);
```

Пример кода рассчитывает приблизительное значение функции sin(x) в соответствии с разложением в ряд Тейлора:

$$\sin(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\ldots$$

Если значение очередного слагаемого r становится меньше заданной точности e происходит выход из цикла с помощью оператора `break`. Слагаемые с четными степенями не добавляются к сумме, так как оператор `continue` переводит выполнение действий на начало итерации цикла, заставляя компилятор игнорировать два последних оператора тела цикла. Заметим еще, что данное решение работает правильно только для неотрицательных значений x.

Оператор `return` используют для выполнения явного выхода из текущего метода. Оператор можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Важно отметить, что наличие в программе недоступного кода, например, кода, следующего за оператором `return` в Java запрещено и приводит к ошибке компиляции.

Упражнения

1. Напишите программу, которая будет печатать все трехзначные числа с заданной суммой цифр.
2. Напишите программу, которая будет печатать таблицу квадратов первых ста натуральных чисел.
3. Задан возраст человека и его пол. Напишите программу, которая определит, является ли человек пенсионером.
4. Напишите программу, которая будет печатать таблицу умножения однозначных чисел.
5. В небоскрёбе бесконечное число этажей и всего один подъезд; на каждом этаже по N квартир. Человек садится в лифт и набирает номер нужной ему квартиры M . Напишите программу, которая определит по введённым N и M на какой этаж должен доставить лифт пассажира?
6. Имеется часть катушки с автобусными билетами. Номер билета шестизначный. Задаются меньший номер билета — N и больший — M . Напишите программу, которая определит количество счастливых билетов на катушке. Билет является счастливым, если сумма первых трёх его цифр равна сумме последних трёх.
7. Имеется информация о банкомате: количество купюр достоинством 50, 100 и 1000 рублей. Задается денежная сумма S, которую должен выдать автомат. Напишите программу, которая определит, какие купюры будут выданы и сколько. Банкомат старается выдать запрашиваемую сумму максимально крупными имеющимися купюрами. Если выдать сумму невозможно, об этом выводится сообщение.

3. Массивы

Массив – это группа однотипных переменных, доступ к которым осуществляется по имени массива и индексу переменной в массиве. Массивы бывают двух видов: одномерные и многомерные. Для объявления одномерного массива используется синтаксис, аналогичный описанию переменных, но к имени переменной либо к имени типа данных добавляются пустые квадратные скобки. Например:

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

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

```
a = new int[10];
```

Таким образом, процесс создания массива происходит в два этапа:

1. создается сам массив;
2. резервируется место в памяти под этот массив.

Возможно одновременное объявление и выделение памяти:

```
int a[] = new int[10];
```

Возможна также инициализация массива при его создании:

```
int a[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Для обращения к элементам массива используется операция `[]`. Элементы массива нумеруются с нуля. Попытка обращения к несуществующему элементу массива приводит к ошибке во время выполнения программы.

Массивы хранят информацию о своих размерах, которую можно получить при помощи специальной переменной `length`. В последнем примере a.length будет равна 12. Последний элемент массива a можно записать так: a [a.length - 1].

Рассмотрим пример:

```
public class ArrayExample {
   public static void main(String[] args) {
      int n=10, k=0;
      int a = new int[n];
      for (int i = 0; i < a.length; i++) {
         a[i] = (int)(Math.random()*100) -50;
         if (a[i] != 0) {
            k++;
         }
      }
      int b[] = new int[k];
      for (int i = 0, j = 0; i < a.length; i++) {
         if (a[i] != 0) {
            b[j] = a[i];
            j++;
         }
      }
      System.out.println("Old array: ");
      for (int i = 0; i < a.length; i++) {
         System.out.print(a[i] + " ");
      }
```

```
        System.out.println("\nNew array: ");
        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i] + " ");
        }
    }
}
```

Здесь массив a из 10 элементов заполняется случайными числами из диапазона от -50 до 50. Происходит подсчет ненулевых элементов с помощью переменной k. Затем создается новый массив b, размер которого соответствует количеству ненулевых элементов первого массива. Затем эти элементы копируются во вновь созданный массив.

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

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

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

```
public class ArrayExample {
    public static void main(String[] args) {
        int[][] two = {{2, 2}, {2, 2}};
        int[][] three = {{1}, {2, 3}, {4, 5, 6}};
        System.out.println("\ntwo");
        for (int i = 0; i < two.length; ++i) {
            for (int j = 0; j < two[i].length; ++j)
                System.out.print(" " + two[i][j]);
            System.out.println();
        }
        System.out.println("\nthree");
        for (int i = 0; i < three.length; ++i) {
            for (int j = 0; j < three[i].length; ++j)
                System.out.print(" " + three[i][j]);
            System.out.println();
        }
    }
}
```

Результат работы:

```
two
 2 2
 2 2

three
 1
 2 3
 4 5 6
```

Для массивов и некоторых других хранилищ данных можно использовать цикл `for` в форме foreach. Цикл в стиле foreach предназначен для строго последовательного выполнения повторяющихся действий по отношению к набору объектов Общая форма версии foreach цикла `for` имеет следующий вид: `for` (тип *итерационная_переменная* : *хранилище*) блок-операторов. Тип это тип объектов в хранилище, *итерационная переменная* — имя итерационной переменной, которая последовательно будет принимать значения из

хранилища, от первого до последнего. Элемент *хранилище* – это имя набора данных, например массива, по которому должен выполняться цикл. На каждой итерации цикла программа извлекает следующий элемент хранилища и сохраняет его копию в итерационной переменной. Цикл выполняется до тех пор, пока не будут обработаны все элементы хранилища. Цикл можно прервать и раньше, используя оператор `break`. Следует также сказать, что в цикле foreach нельзя изменить значение самих элементов массива.

Рассмотрим пример вычисления суммы элементов массива:

```
int numb[] = {1, 2, 3, 4, 5};
int summa = 0;
for (int i = 0; i < 5; i++)
   summa += numb[i];
System.out.println("Сумма=" + summa);
```

Теперь посмотрим, как ту же задачу можно решить с помощью цикла for-each:

```
int numb[] = {1, 2, 3, 4, 5};
int summa = 0;
for (int i : numb)
   summa += i;
System.out.println("Сумма=" + summa);
```

Основное отличие здесь в том, что мы не указываем начальное и конечное значение цикла. Здесь все делается автоматически. Цикл будет продолжаться до тех пор, пока не будут перебраны все элементы массива.

Упражнения

1. Создать массив типа double размера 10 и заполнить его произвольными числами от -20 до 20. Вычислить сумму положительных чисел.
2. Создать массив типа int размера 10 и заполнить его произвольными числами. В массиве поменять местами пары соседних элементов.
3. Заполнить массив 20 числами (диапазон чисел от 0 до 1000). В массиве подсчитать количество и сумму трехзначных симметричных чисел (симметричные числа – 121, 565, 111, и т.п.).
4. Заполнить двумерный массив 4 x 5 числами (диапазон чисел от 0 до 10). В массиве найти количество чисел, равных 5.
5. Заполнить массив 20 числами (диапазон чисел от 1 до 1000). Образовать новый массив, элементами которого будут элементы исходного массива, оканчивающиеся на цифру 3.
6. Заполнить двумерный массив (матрицу) 8 x 8 числами (диапазон чисел от -10 до 10). Элемент матрицы называется локальным минимумом, если он строго меньше всех четырех соседних элементов. Посчитать количество локальных минимумов в матрице и вывести их на печать вместе с индексами.

4. Ввод и вывод данных с помощью специализированных классов

4.1 Класс Scanner

Для ввода данных используется класс Scanner из библиотеки утилит java.util.Scanner.

При работе с потоком ввода необходимо создать объект класса Scanner, при создании указав, с каким потоком ввода он будет связан.

Стандартный поток ввода (клавиатура) в Java представлен объектом — **System.in.** ***Стандартный поток вывода*** (дисплей) — уже знакомым вам объектом **System.out**.

В классе Scanner есть методы для чтения очередного символа заданного типа из стандартного потока ввода, а также для проверки существования такого символа.

```
import java.util.Scanner; // импортируем класс
public class MainClass {
   public static void main(String[] args) {
          // создаём объект класса Scanner
      Scanner sc = new Scanner(System.in);
           int i;
      System.out.print("Введите целое число: ");
          // если из потока ввода можно считать целое число
      if(sc.hasNextInt()) {
                    // считываем из потока ввода и сохраняем целое число
          i = sc.nextInt();
                    // выводим целое число на экран
          System.out.println("Вы ввели число " + i);
       }
           else
      {
       System.out.println("Вы ввели не целое число");
      }
   }
}
```

Так как класс Scanner находится в пакете java.util, то мы его импортируем в самом начале перед описанием класса. Для создания самого объекта Scanner в его конструктор передается объект System.in. После этого мы можем вводить данные, вызывая для объекта класса Scanner методы, которые позволяют получить введенные пользователем значения.

Обратите внимание, что сначала проверяется можно ли считать число из потока и только потом считывается само число. Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит).

Далее приведены некоторые методы класса Scanner, используемые для ввода данных:

`hasNextInt()` – метод возвращает истину, если из потока ввода можно считать целое число

`nextInt()` – метод, считывающий из потока целое число

`hasNextDouble()` – метод проверяет, можно ли считать из потока ввода вещественное число

`nextDouble()` – метод считывает из потока ввода вещественное число

nextLine() – метод, позволяющий считывать целую последовательность символов, т.е. строку. Полученное через этот метод значение нужно сохранять в объекте класса String.
next() – метод считывает введенную строку до первого пробела
hasNext() – метод, проверяющий остались ли в потоке ввода какие-то символы.

4.2 Вывод данных на консоль

Для вывода данных на консоль используется объект **System.out**. У этого объекта есть ряд методов, позволяющих управлять параметрами вывода текстовых данных. Методы **print()** и **println()** наиболее простые и выводят на экран строку текста, последний добавляет к ней переход на новую строку.

Для того, чтобы применить для вывода более сложное форматирование следует использовать метод **printf()**. Метод **printf** определен следующим образом: `void printf(String format, Object... args)`. Первый аргумент `format` это строка, определяющая шаблон, в соответствии с которым будет происходить форматирование, вторая часть `Object... args` – список аргументов, значения которых будут отформатированы и вставлены в исходную строку на печати.

Для правил форматирования следует вставлять в строку специальные конструкции. Опишем их в том порядке, в котором они следуют в конструкции форматирования.

- В начале конструкции ставится знак **%.**
- Далее может идти целое десятичное число со знаком доллара (**$**) в конце, указывающее позицию аргумента в списке аргументов. Ссылка на первый аргумент **"1$"**, ссылка на второй аргумент **"2$"** и т.д. Если позиция не задана, то аргументы должны находиться в том же порядке, что и соответствующие им конструкции в строке форматирования.
- Следующий необязательный знак — специальный флаг для форматирования. Например, флаг **"+"** означает, что числовое значение должно включать знак **+**, флаг **"-"** означает выравнивание результата по левому краю, флаг «**,**» устанавливает разделитель тысяч у целых чисел.
- Потом идет положительное целое десятичное число, которое определяет минимальное количество символов, которые будут выведены. После него можно указать неотрицательное целое десятичное число с точкой перед ним. Обычно оно используется для ограничения количества символов в дробной части вещественного числа.
- Следующий символ, указывает тип форматируемого аргумента. Например **d** для целых чисел, **s** для строк, **f** для чисел с плавающей точкой. Этот символ является обязательной частью инструкции.

  Приведем несколько примеров. Вывод целого числа:

  ```
  System.out.printf("%d", 1234); //  "1234"
  ```

  Вывод целого числа с разделением тысяч:

  ```
  System.out.printf("%,d", 1234); //  "1 234"
  ```

  Число менее 8 знаков будет «подвинуто» вправо на недостающее количество знаков.

  ```
  System.out.printf("%8d", 1234); //  "    1234"
  ```

  Число менее 8 знаков будет дополнено нулями слева на недостающее количество знаков.

  ```
  System.out.printf("%08d", 1234); //  "00001234"
  ```

Число будет дополнено знаком + и, если оно менее 8 знаков, то будет дополнено нулями на недостающее количество знаков.

```
System.out.printf("%+08d", 1234); //  "+0001234"
```

Число будет выровнено по левому краю и, если оно менее 8 знаков, то будет дополнено пробелами справа на недостающее количество знаков.

```
System.out.printf("%-8d", 1234); //  "7845    "
```

Вещественное число автоматически округляется до 6 знаков после запятой.

```
System.out.printf("%f", 2721.0/1001); //   "2,718282"
```

Число менее 10 знаков будет «подвинуто» вправо на недостающее количество знаков.

```
System.out.printf("%10f", 2721.0/1001); //   "  2,718282"
```

Число менее 10 знаков будет дополнено нулями слева на недостающее количество знаков.

```
System.out.printf("%010f", 2721.0/1001); //   "002,718282"
```

Число будет дополнено знаком + и, если оно менее 10 знаков, то будет дополнено нулями на недостающее количество знаков.

```
System.out.printf("%+010f", 2721.0/1001); //   "+02,718282"
```

Число будет выведено с 15 знаками после запятой.

```
System.out.printf("%.15f", 2721.0/1001); //   "2,718281718281718"
```

Число будет выведено с 3-мя знаками после запятой и, если оно менее 8 символов, то будет «подвинуто» вправо на недостающее количество знаков.

```
System.out.printf("%8.3f", 2721.0/1001); //   "   2,718"
```

Число будет выровнено по левому краю, выведено с 3-мя знаками после запятой и, если оно менее 8 знаков, то будет дополнено пробелами справа на недостающее количество знаков.

```
System.out.printf("%-8.3f", 2721.0/1001); //   "2,718   "
```

Строка выведется с переходом на новую строку.

```
System.out.printf("%s%n", "Example"); //   "Example"
```

Если строка содержит менее 10 символов, то она будет «подвинута » вправо на недостающее количество символов.

```
System.out.printf("%10s%n", "Example"); //   "   Example"
```

Строка будет выровнена по левому краю и, если она менее 10 символов, то будет дополнена справа пробелами на недостающее количество символов.

```
System.out.printf("%-10s%n", "Example"); //   "Example   "
```

Будут выведены первые 3 символа строки.

```
System.out.printf("%.3s%n", "Example"); //   "Exa"
```

Будут выведены первые 3 символа строки и «подвинуты» вправо на недостающее до 8 количество символов.

```
System.out.printf("%8.3s%n", "Example"); //   "     Exa"
```

В случае, если нет необходимости выводить отформатированную строку, а нужно просто ее сохранить для дальнейшего использования следует использовать метод `format` из класса `String`. Принципы форматирования в этом случае абсолютно такие же, как у описанного выше `printf`, но этот метод вместо вывода строки сохраняет ее как отформатированную строку.

Упражнения

1. Напишите программу, которая будет сообщать, является ли целое число, введённое пользователем, чётным или нечётным. Если пользователь введёт не целое число, то сообщать ему об ошибке.
2. Напишите программу, которая будет вычислять и выводить на экран сумму двух целых чисел, введённых пользователем. Если пользователь некорректно введёт хотя бы одно из чисел, то сообщать об ошибке.
3. Напишите программу, которая будет выводить на экран меньшее по модулю из трёх введённых пользователем вещественных чисел. Если пользователь некорректно введёт хотя бы одно из чисел, то сообщать об ошибке.
4. Напишите программу, которая вводит данные о человеке: фамилия, имя, год рождения и выводит полученную информацию на экран. Если пользователь некорректно введёт год рождения, то сообщать об ошибке.
5. Напишите программу, которая вводит массив из заданного количества целых чисел, выводит его на печать и находит в этом массиве максимальное значение. Программа должна контролировать все данные, введенные пользователем.

5. Стандартные классы Java для работы с данными

Одна из сложных задач профессионального программиста – выбрать подходящий класс из набора библиотек Java для наиболее эффективной обработки данных. В этой главе разбираются возможности и детали использования нескольких стандартных классов, предназначенных для решения часто встречающихся задач в программировании.

5.1 Обертки примитивных типов

Примитивные типы используются для числовых и логических величин из соображений производительности. Применение объектов для этих значений добавляет нежелательные накладные расходы, даже в случае простейших вычислений.

Несмотря на то что примитивные типы обеспечивают выигрыш производительности, бывают случаи, когда может понадобиться объектное представление. Например, нельзя передать в метод примитивный тип по ссылке. Кроме того, многие из стандартных структур данных, реализованных в Java, оперируют только с объектами. Чтобы справиться с такими ситуациями, Java предлагает обертки примитивных типов, которые представляют собой классы, помещающие примитивный тип в объект.

Обертки для примитивных типов — это **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character** и **Boolean**. Эти классы предоставляют широкий диапазон методов, позволяющий в полной мере интегрировать примитивные типы в иерархию объектных типов Java.

Следующий фрагмент кода демонстрирует пример создания объектов классов оберток:

```
Integer i = new Integer(100);
Character c = new Character('c');
Boolean b = new Boolean(true);
Integer i2 = new Integer("200");
Boolean b2 = new Boolean("TRUE");
```

Рассмотрим основные возможности класса **Integer.**

static int MAX_VALUE (231-1) — константа, определяющая наибольшее значение;
static int MIN_VALUE (-231) — константа, определяющая наименьшее значение;
Integer(int value) — конструктор, создающий объект из примитивного типа;
Integer(String s) — конструктор, создающий объект из строки соответствующего значения;
static int max(int a, int b) — определение максимального из двух чисел;
static int min(int a, int b) — определение минимального из двух чисел;
static int parseInt(String s) — преобразование строки в целое число;
static int parseInt(String s, int radix) — преобразование строки в целое число в заданной системе счисления;
static String toString(int i) — преобразование числа в строку;
static Integer valueOf(int i) — преобразование примитивного типа в объект класса.

Константы и методы класса **Double**:

static double MIN_VALUE (2-1074) — минимально возможное по модулю вещественное число;
static double POSITIVE_INFINITY — плюс бесконечность;
static double NEGATIVE_INFINITY — минус бесконечность;
static double NaN — не число;

`boolean isNaN()` — проверка, что объект не является числом;
`static boolean isNaN(double v)` — проверка, что выражение v не является числом, например, если в нем попытались вычислить квадратный корень из отрицательного числа;
`boolean isInfinite()` — проверка, что объект является бесконечностью;
`static boolean isInfinite(double v)` — проверка, что выражение v является бесконечностью, например, если в нем произошло деление числа на ноль;
`double doubleValue()` и `float floatValue()` — преобразование значения объекта в примитивный тип;
`static double max(double a, double b)` — определение максимального из двух чисел;
`static double min(double a, double b)` — определение минимального из двух чисел;
`public static double parseDouble(String s)` — преобразование строки в вещественное число;
`static Double valueOf(double d)` — преобразование примитивного типа в объект класса.

Приведем пример использования этого класса:

```
double x = 1.0/0.0;
System.out.println ("x = " + x);
System.out.println("x isNaN? "+Double.isNaN(x));
System.out.println("x isInfinite? " + Double.isInfinite(x));
System.out.println("x == Infinity? " +
(x == Double.POSITIVE_INFINITY) );
```

Результат работы этого примера:

```
x = Infinity
x isNaN? false
x isInfinite? true
x == Infinity? true
```

Методы класса `Character` позволяют работать со свойствами отдельных символов:
`Character(char value)` — конструктор, создающий объект из примитивного типа;
`char charValue()` — преобразование значения объекта в примитивный тип;
`static boolean isDigit(char ch)` — проверка, является ли символ цифрой;
`static boolean isLetter(char ch)` — проверка, является ли символ буквой;
`static boolean isLowerCase(char ch)` — проверка, находится ли символ в нижнем регистре;
`static boolean isUpperCase(char ch)` — проверка, находится ли символ в верхнем регистре;
`static char toLowerCase(char ch)` — преобразование символа в нижний регистр;
`static Character valueOf(char c)` — преобразование примитивного типа в объект класса.

Приведем пример проверки на то, что символы являются буквами.

```
System.out.println(Character.isLetter('ф'));
System.out.println(Character.isLetter('z'));
System.out.println(Character.isLetter('\u03C0'));
```

Результат работы фрагмента кода:

```
true
true
```

```
true
```

Попробуйте определить, какой буквой является символ из третьей строки последнего примера.

5.2 Класс `Math`

Класс **`Math`** содержит константы и методы, наиболее популярные при выполнении математических вычислений. Методы реализованы как **`static`**, поэтому можно сразу вызывать через **`Math.methodName()`** без создания экземпляра класса. В классе определены две константы типа `double`: **E** и **PI**.

Рассмотрим некоторые методы и примеры их использования.

Методы для вычисления тригонометрических функций принимают параметр типа `double`, выражающий угол в радианах и возвращают результат типа `double`.

`double sin(double d)` – вычисляет синус числа d;
`double cos(double d)` – вычисляет косинус числа d;
`double tan(double d)` – вычисляет тангенс числа d;
`double asin(double d)` – вычисляет арксинус числа d;
`double acos(double d)` – вычисляет арккосинус числа d;
`double atan(double d)` – вычисляет арктангенс числа d.

Пример использования:

```
double result = Math.acos(1); // 0.0
```

Экспоненциальные функции:

`double cbrt(double a)` – извлекает кубический корень из числа a;
`double exp(double a)` – вычисляет e в степени числа a;
`double log(double a)` – вычисляет натуральный логарифм числа a;
`double log10(double d)` – возвращает десятичный логарифм числа d;
`double pow(double a, double b)` – вычисляет a в степени b;
`double sqrt(double a)` – извлекает квадратный корень из числа a.

Пример использования:

```
double result = Math.cbrt(27); // 3
```

Функции округления:

`double ceil(double d)` – округляет d до ближайшего большего целого;
`double floor(double d)` – округляет d до ближайшего меньшего целого;
`double round(double d)` – возвращает ближайшее к числу d целое число

Пример использования:

```
double result = Math.ceil(2.34); // 3
double result = Math.floor(2.56); // 2
double result = Math.round(2.34); // 2
double result = Math.round(2.56); // 3
```

Другие функции:

`abs()` - возвращает модуль аргумента;
`int floorDiv(int a, int b)` – возвращает целочисленный результат деления a на b;
`max()` - возвращает большее из двух чисел;
`min()` - возвращает меньшее из двух чисел;
`double random()` – возвращает случайное число от 0.0 до 1.0;
`double signum(double d)` – возвращает знак d: число 1, если d положительное, и -1, если значение d отрицательное, если d равно 0, то возвращает 0;

`double toDegrees(double value)` – переводит радианы в градусы и `double toRadians(double value)` - градусы в радианы.

Пример использования:

```
int result = Math.floorDiv(9, 2); // 4
double result = Math.abs(-13.5); // 13.5
int result = Math.max(3, 5); // 5
```

5.3 Класс Arrays и его использование

Класс `Arrays` из пакета `java.util` предназначен для работы с массивами. Он содержит удобные методы для работы с целыми массивами:

`copyOf()` – предназначен для копирования массива

`copyOfRange()` – копирует часть массива

`toString()` – позволяет получить все элементы в виде одной строки

`sort()` — сортирует массив методом quick sort

`binarySearch()` – ищет элемент методом бинарного поиска в массиве упорядоченном по возрастанию

`fill()` – заполняет массив переданным значением (удобно использовать, если нам необходимо значение по умолчанию для массива)

`equals()` – проверяет на идентичность массивы

Приведем примеры их использования. Заполнение массива случайными числами:

```
int n = 10;
int a[] = new int[n];
for(int i=0; i<a.length; i++)
   a[i] = (int)(Math.random()*10);
```

Копирование массива a в массив b и вывод второго массива на консоль в виде строки:

```
int b[] = Arrays.copyOf(a, a.length);
System.out.println(Arrays.toString(b));
```

Сортировка и поиск числа в массиве:

```
Arrays.sort(a);
int k = 5;
int ik = Arrays.binarySearch(a, k);
System.out.println("Индекс числа " + k + " в массиве = "+ik);
```

Упражнения

1. Переместите в целочисленном массиве элементы таким образом, чтобы все отрицательные числа предшествовали положительным. Отрицательные и положительные числа полностью сортировать не надо, достаточно отделить их друг от друга.
2. Упорядочите по убыванию каждую строку матрицы N x M, а после этого перестановкой строк упорядочите всю матрицу по убыванию элементов главной диагонали.
3. Напишите калькулятор для вычисления значений тригонометрических функций по заданному аргументу.

6. Работа со строками

Строки — представляют собой последовательность символов. Строки в Java широко используются и являются объектами, так как платформа Java предоставляет специальный класс String для создания и работы со строками.

6.1. Класс String

В классе String существует масса полезных методов, которые можно применять к строкам. Приведем некоторые из них:

length() — возвращает длину строки (количество символов в ней);

isEmpty() — проверяет, пустая ли строка;

replace(a, b) — возвращает строку, где символ a (литерал или переменная типа char) заменён на символ b;

split() – метод позволяет разбить строку на подстроки по определенному разделителю. Разделитель – символ или набор символов передается в качестве параметра в метод;

trim() – метод позволяет удалить начальные и конечные пробелы в стоке;

toLowerCase() — возвращает строку, где все символы исходной строки преобразованы к строчным;

toUpperCase() — возвращает строку, где все символы исходной строки преобразованы к прописным;

equals(s) — возвращает истинну, если строка, к которой применён метод, совпадает со строкой s указанной в аргументе метода (с помощью оператора == строки сравнивать нельзя, как и любые другие объекты);

indexOf(ch) — возвращает индекс символа ch в строке (индекс это порядковый номер символа, но нумероваться символы начинают с нуля). Если символ совсем не будет найден, то возвратит -1. Если символ встречается в строке несколько раз, то вовзвратит индекс его первого вхождения.

lastIndexOf(ch) — аналогичен предыдущему методу, но возвращает индекс последнего вхождения, если смивол встретился в строке несколько раз.

indexOf(ch,n) — возвращает индекс символа ch в строке, но начинает проверку с индекса n (индекс это порядковый номер символа, но нумероваться символы начинают с нуля).

charAt(n) — возвращает код символа, находящегося в строке под индексом n (индекс это порядковый номер символа, но нумероваться символы начинают с нуля).

В классе String есть метод concat()для объединения строк, но чаще строки объединяют при помощи оператора « + »:

```
public static void main(String args[]) {
        String string1 = "JAVA";
        System.out.println("Я стану " + string1 + "программистом!"); }
```

В результате получится:

Я стану JAVA программистом!

Важно отметить, что ***класс строк* String *является неизменяемым***, так что как только он будет создан, строковый объект не может быть изменен. Если есть необходимость сделать много изменений в строке символов, например многократную6. Работа со строками конкатенацию в цикле, следует использовать классы строки буфера StringBuffer и построитель строки StringBuilder.

6.2 Регулярные выражения

В повседневных задачах часто возникает необходимость найти какие-то данные в тексте по какому-то шаблону, или проверить данные, которые поступили от пользователя, или каким-нибудь сложным образом модифицировать текст. Все эти задачи можно решить с помощью регулярных выражений.

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

Чтобы создать регулярное выражение Java, нужно написать его в виде строки с учётом синтаксиса регулярных выражений. Рассмотрим основные ***метасимволы***, используемые для создания регулярных выражений

- `^` – крышка, цирркумфлекс, начало проверяемой строки
- `$` – доллар, конец проверяемой строки
- `.` – точка, представляет собой сокращенную форму записи для символьного класса, совпадающего с любым символом
- `|` – означает «или». Подвыражения, объединенные этим способом, называются альтернативами
- `?` – знак вопроса, означает, что предшествующий ему символ является необязательным
- `+` – обозначает «один или несколько экземпляров непосредственно предшествующего элемента
- `*` – любое количество экземпляров элемента (в том числе и нулевое)
- `{n}` – Ровно n раз
- `{m,n}` – От m до n включительно
- `{m,}` – Не менее m
- `{,n}` – Не более n
- `[ ]` – один из символов, входящих в заданный набор
- `[ ^ ]` – один из символов, не входящих в заданный набор
- `-` – (дефис) интервал символов;
- `()` – выделение группы
- `\\d` – цифровой символ
- `\\D` – не цифровой символ
- `\\s` – пробельный символ
- `\\S` – не пробельный символ
- `\\w` – буквенный или цифровой символ или знак подчёркивания
- `\\W` – любой символ, кроме буквенного или цифрового символа или знака подчёркивания

Примеры регулярных выражений:

- `ab` – строка «ab»
- `a.b` – строка из трех символов, первый a, второй – любой , третий b
- `a*` – строка из любого количества букв a
- `[abc]` – a, b, или c (один символ)

- `[^abc]` – любой символ, кроме a, b, или c
- `[a-zA-Z]+` – непустая последовательность латинских букв в любом порядке
- `\\+` – символ +

В классе String есть метод `matches()` проверяющий соответствие строки регулярному выражению. Например можно проверить, состоит ли строка из латинских букв:

```
if (testString.matches("[A-Za-z]+")) {
    System.out.println("Yes");
}
```

Большая часть функциональности по работе с регулярными выражениями в Java сосредоточена в пакете `java.util.regex` и классах `Pattern` и `Matcher`.

6.3. Классы StringBuffer и StringBuilder

StringBuffer и **StringBuilder** - представляют расширяемые и доступные для изменений последовательности символов, позволяя вставлять символы и подстроки в существующую строку и в любом месте. Данные классы гораздо экономичнее в плане потребления памяти и поэтому рекомендуется к использованию.

В основном эти два класса идентичны. Отличается StringBuilder от StringBuffer большей производительностью, и его нельзя использовать в том случае, если к строке обращается несколько потоков.

Приведем некоторые методы этих классов:

length() – метод позволяет получить текущую длину объекта;

setLength(int length) – метод устанавливает длину строки. Значение должно быть неотрицательным.

charAt(int index) – метод, позволяющий извлечь значение отдельного символа;

setCharAt(int index, char ch) – метод, позволяющий установить новое значение символа,указав индекс символа и его значение.

getChars() – метод позволяет скопировать подстроку из объекта класса StringBuffer в массив. Необходимо позаботиться, чтобы массив был достаточного размера для приёма нужного количества символов указанной подстроки.

append() – метод соединяет объект класса и представление любого другого типа данных. Есть несколько перегруженных версий.

insert() – метод вставляет одну строку в другую. Также можно вставлять значения других типов, которые будут автоматически преобразованы в строки.

delete() – метод удаляет последовательность символов

deleteCharAt() – метод удаляет один символ из указанной позиции.

replace() – метод позволяет заменить один набор символов на другой. Нужно указать начальный и конечный индекс и строку замены.

Приведем пример использования объекта класса StringBuilder:

```
public static void main(String[] args) {
        // создаем объект класса
    StringBuilder sb = new StringBuilder("I Java!");
        // вызываем один из методов класса
    sb.insert(2, "like ");
        // выводм результат на экран
    System.out.println(sb);
}
```

```
}
```

Проверьте, что получится в результате.

Упражнения

1. Напишите программу, которая выводит строку и выводит на экран индексы всех пробелов в строке.
2. Напишите программу, которая вводит строку, заменяет в строке все пробелы на ‘ * ‘ и выводит полученную строку на экран.
3. Напишите программу, которая вводит две строки, определяет – совпадают строки или нет. Если строки не совпадают, то программа должна соединить их в одну строку и результат вывести на экран.
4. Напишите программу, которая будет проверять, является ли слово из пяти букв, введённое пользователем, палиндромом (примеры: «комок», «ротор»). Если введено слово не из 5 букв, то сообщать об ошибке. Программа должна нормально обрабатывать слово, даже если в нём использованы символы разного регистра. Например, слова «Комок» или «РОТОР» следует также считать палиндромами.
5. Напишите программу, которая вводит строку, удаляет из нее все слова, которые начинаются с буквы, заданной пользователем, и выводит на экран получившийся результат.

7. Реализация класса

Как уже говорилось ранее, программы на языке Java состоят из классов, которые являются абстракциями объектов из реального мира. Поэтому сначала разработчик решает, какие классы необходимо включить в программу и только после этого приступает к написанию кода.

Попробуем создать свой собственный класс на примере следующей задачи: *каждый абитуриент имеет фамилию и сумму баллов за ЕГЭ. Необходимо выяснить, кто из абитуриентов поступил в ВУЗ (общий балл абитуриента >= проходного балла).*

Для решения задачи необходимо реализовать класс Applicant, описывающий абитуриента. Класс будет содержать свойство «фамилия» (surname) и свойство «общий балл» (totalScore), а так же метод, определяющий, проходит ли данный абитуриент по конкурсу.

Добавим в пакет myClasses файл с именем Applicant.java, в котором реализуем описание класса Applicant.

7.1 Описание атрибутов класса

Начнем с описания свойств (еще их называют атрибуты или поля) класса. Для описания атрибутов класса чаще всего применяется модификатор доступа private (закрытый), который позволяет использовать переменные только внутри класса. Доступ к таким полям вне класса будет невозможен. Для описания атрибута «фамилия» используем имя surname и класс String, предоставляемый платформой Java и реализующий основные операции со строками. Для описания атрибута «общий балл» используем имя totalScore и уже известный по языку С тип int, так как общий балл является целым числом.

```
package myClasses;

public class Applicant {

   private String surname;
   private int totalScore;
}
```

Обратите внимание, что имена атрибутов класса пишутся с маленькой буквы. Если имя состоит из нескольких слов, то каждое новое слово в имени атрибута пишется с большой буквы без пробела.

7.2 Конструктор класса

Конструктор - это специальный метод, который вызывается при создании нового объекта и определяет действия, выполняемые для задания начальных значений атрибутов класса. Конструктор является важной частью класса, так как не позволяет создавать не проинициализированные объекты.

Конструктор обладает следующими свойствами, отличающими его от остальных методов класса:

- имя конструктора всегда совпадает с именем класса, включая регистр;
- конструктор никогда ничего не возвращает;
- конструктор есть в любом классе. Если не описано ни одного конструктора, то компилятором Java будет автоматически сгенерирован конструктор по умолчанию.

Конструктор по умолчанию – это метод, который не имеет параметров. Как уже говорилось, он генерируется автоматически, но, как правило, программисты стараются конструктор по умолчанию описать явно в теле класса.

Реализуем конструктор по умолчанию для класса Applicant.

```
public Applicant(){
  surname = "Иванов";
  totalScore = 200;
}
```

Конструктор имеет модификатор доступа public, так как объекты создаются, как правило, вне класса, например, в главном классе приложения, поэтому конструктор должен быть доступным за пределами класса.

В качестве начальных значений атрибутов класса могут быть заданы любые значения, удобные для дальнейшего применения.

Как правило, разработчики не ограничиваются реализацией только одного конструктора, потому что для удобства применения класса нужно предусмотреть возможности создания объектов различными способами. То есть, ***конструкторы можно перегружать***.

Перегруженные методы – это методы вообще и конструкторы в частности, которые имеют одинаковые имена, но различаются списками параметров.

Это означает, что класс может иметь несколько различных конструкторов с одинаковыми именами, но с различными списками параметров. Например, при создании объекта мы хотим передать конструктору следующие данные: фамилию и общий балл. Для этого нужно описать в классе ***конструктор с параметрами***.

```
public Applicant(String surname, int totalScore ){
        this.surname = surname;
      this.totalScore = totalScore;
}
```

В качестве параметров передаем конструктору переменную типа String (фамилия) и переменную типа int (общий балл). Имена переменным-параметрам можно давать любые, не обязательно совпадающие с именами атрибутов класса. Но реализованный вариант считается более удобным, так как не нужно задумываться над тем какой параметр какому атрибуту соответствует. Это особенно актуально в случае, когда атрибутов много и их типы совпадают.

Для того чтобы отличать переменную и поле класса с одинаковыми именами используется ключевое слово this. Оно требуется для того, чтобы метод мог сослаться на вызвавший его объект. То есть **this** ***является ссылкой на объект текущего класса*** (можно использовать для обращения к объекту, вызвавшему метод) ***и неявно присутствует в вызове любого метода класса*** (является неявным параметром всех методов класса). В приведенном примере обращение к атрибуту класса через this происходит при помощи оператора «точка».

Напоминаем, что ***если в теле класса реализован хотя бы один любой конструктор, то в этом случае конструктор по умолчанию автоматически не генерируется.***

Если изначально в теле класса был определен именно конструктор с параметрами, то конструктор по умолчанию можно реализовать при помощи ссылки this.Тогда конструктор по умолчанию будет выглядеть следующим образом:

```
public Applicant(){

    this("Иванов", 200);
 }
```

Использование перегруженных конструкторов через конструктор this () позволяет исключить дублирование кода, уменьшая время загрузки классов.

Следует отметить, что оба приведенные описания конструктора по умолчанию являются правильными, поэтому можно применять тот, который более понятен на данном этапе.

Мы пока использовали в качестве параметров конструктора простые типы. Но можно передать конструктору и объект самого класса Applicant. В этом случае получится ***конструктор копирования***, то есть конструктор, создающий копию объекта.

```
public Applicant ( Applicant a) {
      surname = a.surname;
      totalScore = a.totalScore;
}
```

В этом случае будет создан объект, аналогичный тому, который был передан в качестве параметра.

Попробуйте самостоятельно реализовать данный конструктор при помощи ссылки this.

7.3 Методы класса

Помимо конструкторов большинство классов в своем описании должно иметь методы, позволяющие получить доступ к закрытым атрибутам класса. Это методы, начинающиеся со слова set (установить), позволяющие изменять существующие значения атрибутов и методы, начинающиеся со слова get (получить), позволяющие получить текущие значения атрибутов класса.

В описании класса Applicant данные методы буду реализованы следующим образом:

```
//метод, изменяющий значение атрибута «фамилия»
public void setSurname(String surname){
   this.surname = surname;
}
//метод, изменяющий значение атрибута «общий балл»
public void setTotalScore(int totalScore){
   this.totalScore = totalScore;
}
// метод, возвращающий значение атрибута «фамилия»
public String getSurname(){
   return surname;
}
//метод, возвращающий значение атрибута «общий балл»
public int getTotalScore(){
   return totalScore;
}
```

Методы класса определяют с модификатором доступа public, если предполагается их использование вне класса. Далее следует тип возвращаемого методом значения и имя метода со списком параметров. Имена методов класса пишутся с маленькой буквы. Если имя состоит из нескольких слов, то каждое новое слово в имени метода пишется с большой буквы без пробела. Тело метода заключается в фигурные скобки.

Если каждый атрибут класса имеет уникальный тип, то метод set() можно перегрузить. В классе Applicant типы атрибутов различны. Попробуйте сделать метод set() перегруженным.

Если типы атрибутов повторяются, то в этом случае второе слово в имени метода set соответствует изменяемому атрибуту. Подобное правило применяется и для задания имен методов get – второе слово в имени соответствует возвращаемому атрибуту.

Помимо обязательных методов в классе можно реализовывать методы, которые описывают поведение, свойственное объектам данного класса. В нашем случае это будет метод admission (прием, зачисление), определяющий прошел абитуриент по конкурсу или нет.

```
public boolean admission (int passingScore) {

      if ( totalScore >= passingScore )
         return true;
      else
         return false;
}
```

В качестве параметра метод принимает значение проходного балла. Если абитуриент поступил (общий балл абитуриента >= проходного балла), метод возвращает значение «истина», если нет – «ложь».

7.4. Статические поля и методы класса

Статические члены класса - это атрибуты и методы класса, которые напрямую принадлежат самому классу, а не его экземпляру.

Статические атрибуты – это поля, которые будут общими для всех объектов данного класса. Если один объект изменит значение такого поля, то это изменение увидят все объекты. То есть статические атрибуты имеют одинаковые значения для всех объектов класса, в котором они объявлены.

Для того чтобы объявить такое поле, достаточно в описание атрибута добавить ключевое слово static. Для поля static означает, что ***статический атрибут создается в единственном экземпляре*** вне зависимости от количества объектов данного класса. Статическое поле существует даже в том случае, если не создано ни одного экземпляра класса. Статические поля класса размещаются отдельно от объектов класса в некоторой области памяти в момент первого обращения к данному классу.

Добавим в описание класса Applicant статический атрибут maxTotalScore (максимально возможное значение общего балла) и присвоим ему начальное значение.

```
private static int maxTotalScore = 300;
```

Теперь каждый экземпляр класса Applicant будет имеет одну и ту же копию статической переменной *maxTotalScore*, то есть ее значение будет всегда одинаковым для всех объектов данного класса.

Для работы со статическими атрибутами используются ***статические методы***, объявленные со спецификатором static. Такие методы являются методами класса, которые не привязаны ни к какому объекту и не содержат ссылки this на конкретный объект, вызвавший метод. По причине недоступности ссылки this ***статические поля и методы не могут обращаться к нестатическим полям и методам*** напрямую. ***Для обращения к статическим полям и методам достаточно имени класса, в котором они определены***, то есть для доступа к таким методам не нужно создавать объект класса. Статический метод можно вызвать, используя тип класса, в котором эти методы описаны. ***Статические методы нельзя переопределить.***

Статические методы широко используются в библиотеках Java. Например, в классе математических утитлит java.lang.Math почти все методы статические.

Опишем статический метод в классе Applicant, изменяющий значение статического поля *maxTotalScore*.

```
public static void setMaxTotalScore (int maxTotalScore)
{
  Applicant.maxTotalScore = maxTotalScore;
  }
```

Обратите внимание, что здесь, в отличие от нестатических методов, вместо слова this (так как данная ссылка недоступна статическим методам) обращение к соответствующему полю происходит с использованием типа класса.

8. Применение класса

В настоящий момент описание класса Applicant выглядит следующим образом:

```
package myClasses;
public class Applicant {
   private String surname;
   private int totalScore;
   private static int maxTotalScore = 300;
   //статический метод класса
   public static void setMaxTotalScore (int maxTotalScore)
   {
      Applicant.maxTotalScore = maxTotalScore;
   }
   //конструктор с параметрами
   public Applicant(String surname, int totalScore ){

         this.surname = surname;
      this.totalScore = totalScore;
       }
       //конструктор по умолчанию
       public Applicant(){

      this("Иванов", 200);
       }
       //конструктор копирования
       public Applicant ( Applicant a) {
      surname = a.surname;
      totalScore = a.totalScore;
       }
   //метод, изменяющий значение атрибута «фамилия»
   public void setSurname(String surname){
      this.surname = surname;
   }
    //метод, изменяющий значение атрибута «общий балл»
   public void setTotalScore(int totalScore){
      this.totalScore = totalScore;
   }
      // метод, возвращающий значение атрибута «фамилия»
   public String getSurname(){
      return surname;
   }
      //метод, возвращающий значение атрибута «общий балл»
   public int getTotalScore(){
      return totalScore;
   }
       //метод, определяющий поступил абитуриент или нет
       public boolean admission ( int passingScore ){
      if ( totalScore >= passingScore )
         return true;
      else
         return false;
       }
}
```

Можно попробовать создать и поработать с объектами реализованного класса. Для этого переходим в главный класс приложения, то есть в файл MainClass.java. (Строчка System.out.println("Привет, JAVA!"); нам больше не нужна, её можно удалить.)

8.1 Создание объекта

Создание объектов в Java представляет собой процесс, состоящий из двух этапов. Сначала объявляется переменная типа класса, которая представляет собой ссылку на объект. Например, для описанного выше класса, это выглядит следующим образом

```
Applicant ap1;
```

Но объявленная переменная не определяет сам объект, а только ссылается на него. Поэтому вторым этапом необходимо выделить память под сам объект. Для этого используется оператор new.

Оператор new динамически резервирует память для объекта и возвращает ссылку на него, которая сохраняется в переменной.

```
ap1 = new Applicant();
```

Оператор new при создании объекта класса всегда используется вместе с конструктором, который инициализирует объект. В зависимости от вызванного конструктора, можно создавать объекты различными способами, реализованными в описании класса.

```
package myClasses;
public class MainClass {
   public static void main(String[] args) {

        // создание объекта класса при помощи конструктора по умолчанию
      Applicant ap1 = new Applicant();

        // создание объекта класса при помощи конструктора с параметрами
      Applicant ap2 = new Applicant("Сидоров",195);

        // создание объекта класса при помощи конструктора копирования
         Applicant ap3 = new Applicant(ap2);
   }
}
```

Итак, в нашем примере созданы три объекта при помощи трех различных конструкторов, попробуем поработать с ними.

8.2. Работа с объектами

В классе Applicant объявлено несколько методов, которые можно использовать при работе с объектами. Например, применим методы void setSurname(String surname) и void setTotalScore(int totalScore) к объекту ap3. Этот объект создан как копия объекта ap2, поэтому имеет те же самые значения атрибутов. Для того чтобы их изменить, вызовем для ap3 соответствующие методы, так как непосредственно к атрибутам класса из main() мы обратиться не можем.

```
ap3.setSurname("Петров");
ap3.setTotalScore(210);
```

Теперь объект ap3 отличается от объекта ap2, так как значения атрибутов в нем изменены.

Далее проверим, прошли ли по конкурсу все три абитуриента. Для этого вызовем метод boolean admission (int passingScore) для всех имеющихся объектов и выведем на экран фамилии поступивших.

Добавим новую переменную «проходной балл» и присвоим ей значение 200.

```
int passingScore = 200;
```

Далее для каждого объекта вызовем метод `boolean admission (int passingScore)`. Так как метод возвращает «истина», если абитуриент прошел по конкурсу и «ложь», если нет, удобно поместить вызов метода в условный оператор `if`.

```
if(ap1.admission(passingScore))
  System.out.println(ap1.getSurname());

if(ap2.admission(passingScore))
  System.out.println(ap2.getSurname());

if(ap3.admission(passingScore))
  System.out.println(ap3.getSurname());
```

Для того чтобы вывести на печать фамилию абитуриента, нужно ее получить из объекта класса. Так как напрямую к закрытым атрибутам мы обратиться не можем, то используем метод `String getSurname()`, возвращающий значение атрибута «фамилия».

Таким образом, в результате будет выведено на экран:

```
Иванов
Петров
```

И, наконец, приведем пример вызова статического метода, описанного в классе:

```
Applicant.setMaxTotalScore(350);
```

Обратите внимание, что метод вызывается не для конкретного объекта, а для «класса вообще». После вызова этого метода значение соответствующего статического атрибута изменится у всех созданных объектов класса `Applicant`, а будущие объекты будут создаваться уже с новым заданным значением атрибута *maxTotalScore*.

Упражнения.

1. Создайте класс «Одномерный массив».

Атрибуты класса:

- размер массива;
- последовательность элементов массива типа `int`.

Методы класса:

- конструкторы: по умолчанию, с параметром «размер массива», копирования;
- метод, возвращающий размер массива;
- метод, возвращающий элемент массива по его номеру и метод, изменяющий значение элемента массива под заданным номером на новое значение;
- метод, находящий максимальный элемент массива;
- метод, определяющий упорядочен ли массив по возрастанию или нет.

В главном классе создайте объект класса и проверьте работу объявленных методов.

2. Создайте класс «Студент», в котором задаются фамилия студента и набор оценок. Максимальное количество оценок у каждого студента 20. В описании класса должны быть представлены следующие методы:

- конструкторы, необходимые для создания объектов;
- метод, добавляющий оценку к списку оценок,
- метод, вычисляющий средний балл,
- метод, определяющий является ли студент успевающим (среди оценок нет двоек).

В главном классе создайте несколько объектов класса и проверьте работу объявленных методов.

Добавьте возможность увеличения максимального количества оценок у каждого студента.

3. Создайте приложение для хранения и обработки информации о грузовых перевозках. Для каждого рейса определяются фамилия водителя, наименование груза, последовательность городов в маршруте.

Разработайте соответствующий класс, который должен содержать следующие обязательные методы:

- конструктор по умолчанию, копирования, с параметром;
- методы, изменяющие значения полей класса;
- методы, возвращающие значения полей класса.

Разработайте следующие методы:

- метод, возвращающий количество городов в маршруте;
- метод, "разворачивающий" маршрут в противоположном направлении;
- метод, добавляющий город в маршрут, если там его еще нет.

В главном классе

- организуйте ввод данных пользователем
- создайте объекты с использованием различных конструкторов
- выведите на печать все перевозки, прошедшие меньше чем через 3 города.

9. Наследование

Наследование является одним из основных понятий объектно-ориентированного программирования. С помощью наследования можно расширять функционал имеющихся классов (***классов предков***) за счет создания на их основе новых классов (***классов наследников***). При этом модификация происходит без внесения изменений в реализацию уже существующих классов. Создаваемые на их основе новые классы автоматически учитывают уже имеющиеся свойства и поведение. Сама модификация происходит за счет добавления в создаваемые классы новых свойств и поведения, а так же переопределения в новых классах уже имеющегося функционала.

9.1 Наследование от базового класса

В языке Java классы-наследники или ***подклассы создаются ни основе одного (и только одного!) класса-предка***, то есть множественное наследование в Java запрещено.

Чтобы сделать один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово extends, после которого пишется имя класса-предка. Класс потомок унаследует все поля и методы класса, но доступ будет иметь только к тем, которые не являются закрытыми. Будем исходить из того, что все члены класса предка доступны классам потомкам, то есть являются открытыми (public) или видимыми в рамках пакета (protected). То есть объект класса-потомка обязан уметь все, о чем объявил его родитель, поэтому можно безопасно использовать класс-потомок везде, где ожидается использование класса-родителя.

Рассмотрим простой пример реализации наследования. Не забывайте, что каждый класс должен размещаться в отдельном одноименном файле.

```
public class Animal {
   String name;
   int age;

   public Animal(String name, int age){
      this.name=name;
      this.age=age;
   }

   public String getName(){
      return name;
      }

       public int getAge(){
      return age;
      }

       public void setName(String name){
              this.name = name;
       }

       public void setAge(int age){
              this.age = age;
       }
}
```

Выше приведена реализация базового класса Animal, в котором описаны характеристики «имя» и «возраст», а так же конструктор и методы, возвращающие и

изменяющие значении атрибутов. Обратите внимание, что перед описанием атрибутов нет модификатора доступа. Как уже говорилось выше, атрибуты базового класса должны быть открытыми или, как в данном случае (то есть по умолчанию), видимыми в рамках пакета.

Теперь создадим класс Cat, который будет унаследован от Animal.

```
public class Cat extends Animal {

  public Cat(String name,int age){
  super(name,age);
  }
}
```

В описании класса Cat присутствует только конструктор, так как ***конструкторы не наследуются***, но подкласс может вызывать конструктор, определенный его суперклассом, с помощью следующей формы ключевого слова super:

```
super (список_аргументов);
```

С помощью ключевого слова super можно обращаться к любому члену класса-предка, если они не определены с модификатором private.

В классе Cat не объявлено собственных полей и методов, но он может успешно использоваться, так как все что определено в базовом классе будет унаследовано классом потомком. То есть могут создаваться объекты данного класса и вызываться реализованные методы, например:

```
Cat cat = new Cat("Барсик",3);
cat.setName("Бонифаций");
```

В классе наследнике могут быть определены свои собственные методы и поля. Добавим в класс Cat поле «цвет» и метод «говорить». Теперь описание класса выглядит следующим образом:

```
public class Cat extends Animal {
  String color;
  public Cat(String name, int age, String color){
            super(name,age);
            this.color = color;
  }
  public String speak(){
            String phrase = "...мяу...\n";
            return phrase;
  }
}
```

Обратите внимание на то, как изменился конструктор класса. Теперь создание объекта выглядит следующим образом:

```
Cat cat = new Cat("Барсик",3,"белый");
```

И для объекта cat, помимо всех описанных в базовом классе методов, может быть вызван метод speak():

```
System.out.println(cat. getName() + " сказал " + cat. speak());
```

На экран будет выведено:

```
Барсик сказал "...мяу..."
```

Расширим нашу иерархию наследования, добавив в нее класс Dog.

```
public class Dog extends Animal {
  String breed;
  public Dog(String name, int age, String breed){
            super(name,age);
```

```
                this.breed = breed;
    }

    public String speak(){
                String phrase = "...гав-гав...\n";
                return phrase;
    }
}
```

В классе Dog мы добавили поле breed (порода) и изменили метод speak(). Остальные методы и атрибуты, объявленные в базовом классе Animal, являются общими для всех классов наследников.

Аналогичным образом можно и далее расширять иерархию наследования, добавляя в нее новые классы потомки. При этом важно отметить, что реализация уже существующих классов меняться не будет.

9.2 Абстрактные методы и классы

Класс-предок может содержать объявление метода, но при этом не иметь его реализации. Такие методы называются абстрактными. Абстрактный метод объявляется с ключевым слово abstract в начале описания. Сразу после сигнатуры такого метода ставится точка с запятой.

Например, в нашем случае в обоих классах наследниках есть метод speak(), но при этом у каждого класса своя собственная реализация этого метода. Поэтому можно объявить этот метод абстрактным в базовом классе, то есть определить, что все объекты потомки класса Animal (а не только Cat и Dog, в случае расширения иерархии наследования) будут иметь свою собственную реализацию метода speak().

Классы, имеющие в своем описании хотя бы один абстрактный метод, так же называются ***абстрактными классами***. Для абстрактного класса не может быть создан его экземпляр. Классы-потомки такого класса должны реализовать все абстрактные методы, заявленные в классе предке. Только в этом случае они смогут стать конкретными классами, то есть смогут создавать свои собственные экземпляры.

Для того чтобы сделать класс абстрактным, нужно поместить ключевое слово abstract перед объявлением класса.

После внесенных изменений описание класса Animal выглядит следующим образом:

```
public abstract class Animal {
    String name;
    int age;

    public Animal(String name, int age){
        this.name=name;
        this.age=age;
    }

    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
    public void setName(String name){
                this.name = name;
    }
```

```
    public void setAge(int age){
            this.age = age;
    }

    // абстрактный метод
    public abstract String speak();
}
```

Класс-наследник может изменять функциональность, унаследованную от класса-предка. Эта способность подклассов называется ***полиморфизмом*** и является одним из ключевых аспектов объектно-ориентированного программирования наряду с наследованием и инкапсуляцией.

9.3. Переопределение методов

При полиморфизме в классе потомке происходит ***переопределение*** (не путать с перегрузкой!) методов класса предка. Реализация класса-предка определяет, как остальной код сможет использовать тот или иной метод, поэтому метод, переопределенный в подклассе должен иметь туже сигнатуру, что и метод класса-предка.

Совершенно необязательно переопределять в классах потомках все методы базового класса. Если метод реализован в классе предке и такая реализация подходит для классов-потомков, то они могут использовать реализацию поведения, описанную в базовом классе. Например, в описанной выше иерархии наследования, мы не переопределяем методы getName(), getAge(), setName(), setAge(), поэтому классы-наследники будут использовать реализацию этих методов из базового класса.

В Java каждый класс неявно, но расширяет **java.lang.Object** класс, так Object-класс находится на верхнем уровне иерархии наследования в Java. То есть, описанные выше классы, так же являются наследниками класса Object. При этом писать в объявлении класса extends Object не нужно.

Большинство, реализуемых на языке Java классов, переопределяет следующие методы, объявленные в классе Object – это toString(), equals(), hashCode(), так как эти методы, широко используются в специализированных классах стандартных библиотек Java. Обратите внимание, что если переопределение перечисленных методов в классе отсутствует, будут вызываться соответствующие методы из класса Object.

Метод **toString()** ***используется для строкового представления объекта***, поэтому, как правило, при описании класса не реализуется метод, выводящий данные об объекте на печать, а переопределяется именно метод toString().

Добавим переопределение этого метода в описание класса Applicant.

```
    @Override
  public String toString()
  {
    return surname + " " + totalScore;
  }
```

Аннотация **@Override** указывает, что далее будет переопределен метод базового класса. Данная аннотация никак не влияет на сам факт переопределения метода. При совпадении сигнатур с методом базового класса он и так будет переопределен, независимо от наличия, либо отсутствия этой аннотации. Аннотация служит лишь для контроля успешности действия при сборке проекта. То есть наличие @Override позволяет избежать выполнения перегрузки метода вместо его переопределения. В присутствии этой аннотации

метод можно будет только переопределить, а при попытке перегрузить метод будет выдаваться сообщение об ошибке.

После переопределения toString() мы можем его использовать, например, для вывода информации об объектах класса Applicant. Для этого в метод main() главного класса нужно добавить следующий код:

```
System.out.println(ap1);
System.out.println(ap2);
System.out.println(ap3);
```

В результате на экран будет выведено:

```
Иванов 200
Сидоров 195
Петров 210
```

Обратите внимание, что метод toString() вызывается для объекта класса неявно при вызове метода println(). То есть в качестве аргумента println() можно указать только соответствующий объект без вызова метода toString(), поэтому вместо

```
System.out.println(ap1.toString());
```

можно написать

```
System.out.println(ap1);
```

Теперь рассмотрим, как будет переопределен метод toString() в описанной выше иерархии наследования. В нашем примере метод будет переопределяться как в базовом классе, так и в классах наследниках, так как в их описание добавлены новые атрибуты. Но если класс-наследник не добавляет новых свойств к описанию базового класса, то достаточно будет только реализации метода toString(), описанной в базовом классе.

Переопределение в базовом классе Animal буде выглядеть следующим образом:

```
@Override
public String toString()
{
  return name + " " + age;
   }
```

В классе Cat будем использовать уже описанную в базовом классе реализацию, используя ключевое слово super, для вызова toString() из базового класса:

```
@Override
public String toString()
{
  return super.toString() + " "+ color;
   }
```

Для класса Dog попробуйте переопределить метод toString()сами.

Метод **equals()** – ***предназначен для сравнения объектов на равенство***, если объекты равны – возвращает true, если не равны – false.

Обратите внимание, что оператор = = не подходит для сравнения объектов. Этот оператор сравнивает значения только при работе с примитивными типами. В случае применения его к объектам, сравниваться будут только ссылки на эти объекты, а не само содержимое объектов, и на основании равенства или неравенства ссылок будет возвращаться результат (true или false). Поэтому для объектов нельзя применять оператор = = вместо метода equals().

Несмотря на то, что этот метод уже определен в классе Object (мы уже говорили, что все классы наследуют от класса Object), его необходимо переопределять для всех

пользовательских классов. В случае отсутствия такого переопределения, будет вызываться реализация метода equals() из класса Object, что приведет к сравнению только ссылок на объекты, а не самих объектов.

Ниже приведено переопределение метода equals() для класса Applicant.

```
@Override
public boolean equals(Object obj)
{
if(obj==null)
      return false;
if(this==obj)
          return true;

if (!(obj instanceof Applicant))
  return false;
Applicant a = (Applicant) obj;
return surname.equals(a.surname)&& totalScore ==a.totalScore;
}
```

Сначала проверяется ссылка на объект, переданная в качестве параметра. Если такой ссылки не существует, то естественно объекты не равны.

Далее проверяются на равенство ссылки на текущий объект (this) и объект – параметр метода (obj), если они одинаковые, то значит они ссылаются на один и тот же объект, то есть объекты равны.

Оператор* instanceof *служит для проверки какому классу принадлежит объект.

A instanceof B - возвращает истину, если в переменной A содержится ссылка на экземпляр класса B. В данном случае, проверяется содержится ли в obj ссылка на объект класса Applicant, если нет, то объекты не равны.

Далее преобразуем ссылку obj к соответствующему типу (Applicant) и возвращаем результат сравнения соответствующих атрибутов двух объектов. Так атрибут surname имеет тип String (то есть является объектом класса String), то для того, что бы сравнить этот атрибут с таким же атрибутом другого объекта, необходимо применить метод equals(). Он уже переопределен в классе String стандартной библиотеки Java. Через = = сравнивать строки нельзя, потому что это объекты. А вот атрибуты totalScore, имеющие тип int, можно сравнивать, используя оператор = =, так это обычные переменные, а не ссылки.

Вместе с методом equals() обязательно переопределяется и метод hashCode(). ***Метод* hashCode () *– используется для генерации целочисленного кода объекта*** и вызывается автоматически при вызове метода equals().

Метод hashCode () реализован таким образом, что для одного и того же входного объекта, хеш-код всегда будет одинаковым. Хеш-код вычисляется на основании содержимого объекта (значения полей), поэтому если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые. Таким образом, если объекты равны, то и хеш-коды этих объектов тоже обязательно должны быть равны. Но в обратную сторону это правило не действует, то есть, если хеш-коды равны, то входные объекты равны не всегда.

Переопределение метода hashCode() в классе Applicant:

```
@Override
public int hashCode()
{
    return surname.hashCode() + totalScore;
```

```
}
```

Обратите внимание, что при вычислении хеш-кода следует использовать те же поля, которые сравниваются в equals().

Для атрибута surname мы вызвали метод hashCode(), так как он уже переопределен для объектов класса String. Далее к полученному значению просто прибавили количество баллов, так как это целое число.

Упражнения

1. Для классов Animal, Cat, Dog из описанной в примере иерархии наследования переопределите методы equals() и hashCode().
2. Расширьте описанную в примере иерархию наследования, добавив разделение на домашних и диких животных. Добавьте классы:
   - «Cow» (как зовут, что ест, дает молоко);
   - «Alligator» (что ест, где обитает, охотится);
   - «Bear» (что ест, где обитает, впадает в спячку).
3. Опишите иерархию наследования для геометрических фигур: эллипс, окружность, четырехугольник, прямоугольник, квадрат, ромб. Добавьте возможность рассчитать площади и периметры перечисленных фигур.
4. В описание классов из упражнений 1, 2, 3 раздела 5 добавьте переопределение методов toString(), equals(), hashCode().

10. Интерфейсы

Продолжением идеи абстрактных классов является концепция интерфейсов, которая широко используется в языке Java. Смысл использования интерфейсов состоит в унификации работы с разнотипными объектами. То есть интерфейсы позволяют сосредоточиться не на типах объектов, а на той роли, которую могут играть классы, независимо от их расположения в иерархии наследования.

10.1 Создание пользовательского интерфейса

Интерфейс содержит только абстрактные методы, то есть в нем методы только описываются, но не определяются, при этом ключевое ***слово*** **abstract** ***в описании интерфейса не указывается***. В интерфейсе могут присутствовать поля, но они должны являться статическими и неизменными, поэтом у поля интерфейса принято писать большими буквами, как это делается при описании констант. Все методы и поля интерфейса неявно объявляются как public . ***В интерфейсе нет конструкторов***.

Давайте попробуем разработать собственный интерфейс, который смогут реализовать описанные в примере классы Applicant, Dog, Cat. Очевидно, что эти классы описывают совершенно разные абстракции, то есть объекты этих классов имеют разные типы. При этом абитуриенты любят играть в компьютерные игры, собаки любят играть с мячиком, а коты бегать за лучом от лазерной указки. То есть разнотипные объекты выполняют одно и то же действие, но выполняют каждый по-своему. Чтобы описать такую ситуацию в Java как раз и переменяется интерфейс. Интерфейс описывает поведение (совокупность методов), которым должны обладать классы, реализующие этот интерфейс.

В нашем случае описание интерфейса будет выглядеть следующим образом:

```
public interface Game {
        void play();
}
```

Объявление интерфейса похоже на объявление обычного класса и размещается в отдельном файле. Вместо слова class в заголовке указывается слово interface. Далее идет имя интерфейса Game и объявлен метод play() без реализации. Методы в интерфейсе всегда «пустые», потому что интерфейс только описывает поведение, а не реализует его.

Чтобы использовать объявленный интерфейс, он должен быть реализован в классах Applicant, Dog, Cat. То есть все классы, реализующие данный интерфейс, должны уметь «играть», а как конкретно они будут это делать – вопрос уже к самим классам Applicant, Dog, Cat.

Рассмотрим реализацию интерфейса Game в классе Applicant.

```
public class Applicant implements Game{
   ...
   @Override
   public void play()
   {
      System.out.println("... World of Warcraft ...");
   }
}
```

Класс Applicant «связывается» с интерфейсом Game при помощи ключевого слова implements. Это значит, что класс Applicant реализует интерфейс Game, то есть Applicant должен реализовать метод play() , объявленный в интерфейсе Game, поэтому реализация метода play() добавляется в описание класса Applicant.

Аналогично тот же самый интерфейс реализуется в классе Cat.

```
public class Cat implements Game
{
   ...
   @Override
   public void play()
   {
      System.out.println("... Лазерная указка ...");
   }
}
```

Обратите внимание, что метод интерфейса, переопределяемый в классе, должен иметь модификатор доступа public – это обязательное условие при реализации интерфейса.

Для класса Dog допишите реализацию интерфейса Game самостоятельно.

Важно отметить, что ***интерфейсы — это не классы***, поэтому ***создать экземпляр интерфейса нельзя***, но можно объявлять интерфейсные переменные. Например, можно создать массив типа Game, состоящий из объектов классов Applicant, Dog, Cat. Для этого, в главном классе создадим объекты соответствующих классов и поместим их в массив типа Game:

```
Applicant ap = new Applicant("Захаров",250);
Cat cat = new Cat("Барсик",3,"белый");
Dog dog = new Dog("Шарик",5,"дворянин");
// объявление массива из элементов типа Game
Game [] g = new Game[3];
// размещение объектов в массиве
g[0] = ap;
g[1] = cat;
g[2] = dog;
// применение элементов массива
for (int i=0; i<3 ; i++)
     g[i].play();
```

В цикле для каждого элемента массива вызовем метод play(). Проверьте, что будет выведено в результате. (Не забыли, что в классе Dog нужно было дописать реализацию интерфейса Game?)

Обратите внимание, что в массив мы помещаем объекты разных классов, но получить из массива мы можем только объекты, на которые ссылается переменная типа Game. Поэтому для элементов массива g доступен единственный определенный для них метод play(),а методы соответствующих классов не доступны. Они будут доступны только для самих объектов ap, cat и dog.

10.2 Интерфейс Comparable

Интерфейсы являются одной из основных частей языка программирования Java. Для того, что бы пользовательский класс мог использовать имеющиеся встроенные возможности Java, он должен реализовать соответствующие интерфейсы из библиотеки Java. Например, нам требуется в программе отсортировать объекты класса в определенном порядке. Java предоставляет встроенные методы для сортировки массива примитивных типов, объектов, массивов и списков, но эти методы для объектов собственного класса можно использовать только, если этот класс реализует интерфейс Comparable или интерфейс Comparator.

Интрефейс Comparable применяется, как правило, для «естественного» (по возрастанию) упорядочивания объектов.

```
public interface Comparable<T> {
        int compareTo(T o);
}
```

Интерфейс Comparable содержит один единственный метод compareTo (T o), который сравнивает текущий объект с объектом того же класса, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

В классе Applicant реализация интерфейса Comparable, который задает порядок сравнения по атрибуту surname (по фамилии), будет выглядеть следующим образом:

```
public class Applicant implements Game, Comparable <Applicant>
{
...
        @Override
         public int compareTo(Applicant ap)
         {
         return this.surname.compareTo(ap.surname);
         }
 }
```

Так как Applicant уже реализует интерфейс Game, то в заголовке остальные реализуемые классом интерфейсы перечисляются через запятую. Если бы реализовывался только интерфейс Comparable, то объявление класса выглядело бы так:

```
public class Applicant implements Comparable <Applicant>
```

В угловых скобках указывается класс, объекты которого будут сравниваться. Далее в описание класса добавляется переопределение метода compareTo(Applicant ap). Метод возвращает результат сравнения атрибута surname текущего объекта (this.surname) и атрибута surname объекта, переданного в качестве параметра (ap.surname). Так как атрибут surname имеет тип String, а этот класс реализует интерфейс Comparable, то для сравнения указанных полей используем реализацию метода compareTo из класса String.

Теперь в главном классе создадим массив из объектов класса Applicant и отсортируем его, используя метод sort() из библиотеки Java.

```
// создаем объекты класса
Applicant ap1 = new Applicant("Григорьев",210);
Applicant ap2 = new Applicant("Сидоров",200);
Applicant ap3 = new Applicant("Акопян",195);
// создаем массив из объектов класса
Applicant [] a = new Applicant[3] ;
a[0]=ap1;
a[1]=ap2;
a[2]=ap3;
//сортируем массив объектов, вызывая метод sort класса Arrays     //библиотеки Java
 Arrays.sort(a);
//распечатываем массив объектов после сортировки
 for(int i=0; i<3; i++)
        System.out.println(a[i]);
```

В результате на экран буде выведено:

Акопян 195
Григорьев 210
Сидоров 200

Обратите внимание, что Comparable задает единственный способ упорядочивания объектов, в данном случае по фамилии. Если нужно объекты класса Applicant упорядочивать по общему баллу, придется изменить реализацию метода compareTo() или использовать интерфейс Comparator.

Попробуйте сами изменить метод compareTo() так, чтобы объекты сортировались по возрастанию общего балла.

10.3 Интерфейс Comparator

Если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность и нужен другой порядок сортировки объектов, то на этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator, который определяет свои собственные способы сравнения двух экземпляров класса.

Основные отличия интерфейса Comparator от интерфейса Comparable состоят в следующем:

– интерфейс Comparator реализуется внешним образом, то есть не затрагивает непосредственно реализацию самого класса;

– при помощи интерфейса Comparator можно определить несколько правил сортировки объектов класса, при этом уже существующие реализации интерфейса не меняются.

Интерфейс Comparator содержит ряд методов, ключевым из которых является метод compare():

```
public interface Comparator<T> {

   int compare(T a, T b);
   // остальные методы
   ...
}
```

Метод compare() возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. Если метод возвращает ноль, то объекты равны.

Для применения интерфейса надо создать класс компаратора, который реализует этот интерфейс. То есть в нашем примере для реализации сравнения объектов по фамилии при помощи интерфейса Comparator, нужно в отдельном файле описать новый класс, который будет реализовывать интерфейс Comparator для сравнения объектов класса Applicant. Не забывайте, что каждый класс реализуется в отдельном файле.

```
import java.util.Comparator;
public class SurnameComparator implements Comparator <Applicant>{
   @Override
   public int compare(Applicant ap1, Applicant ap2)
   {
            return ap1.getSurname().compareTo(ap2.getSurname());
   }
}
```

Обратите внимание, что в файл с реализацией класса, нужно импортировать пакет `import java.util.Comparator`.

Имя класса, реализующего интерфейс, принято начинать с названия соответствующего атрибута и заканчивать словом Comparator, чтобы было сразу из имени класса понятно, что этот класс реализует интерфейс, определяющий сравнение объектов по заданному полю. Поэтому в нашем случае класс называется SurnameComparator. Если в приложении интерфейс реализуется для нескольких классов, то можно к имени класса, реализующего интерфейс, добавить название соответствующего класса, например ApplicantSurnameComparator.

Далее переопределяется метод compare, который в качестве параметров принимает два сравниваемых объекта класса Applicant.Так как класс SurnameComparator реализуется внешним образом, то для того чтобы получить значение атрибута surname у соответствующего объекта, применяется метод getSurname() класса Applicant. В остальном реализация метода compare() аналогична описанной выше реализации метода compareTo().

Теперь мы можем сортировать объекты класса Applicant по фамилии при помощи компаратора. Для этого в главном классе создадим объект класса SurnameComparator и применим его для сортировки имеющегося массива объектов.

```
SurnameComparator sc = new SurnameComparator();
Arrays.sort(a,sc);
```

Если необходима сортировка объектов еще и по общему баллу, то есть по полю totalScore, нужно описать новый клаcс TotalScoreComparator:

```
import java.util.Comparator;
public class TotalScoreComparator implements Comparator <Applicant>{
   @Override
   public int compare(Applicant ap1, Applicant ap2)
   {
            return ap1.getTotalScore() - ap2.getTotalScore();
   }
}
```

Так как атрибут totalScore имеет тип int, то самый простой вариант реализации метода compare() - это просто вернуть результат разности значений соответствующих атрибутов. Значения атрибутов получаются в результате вызова метода getTotalScore() класса Applicant.

Для сортировки объектов по общему баллу создадим объект класса TotalScoreComparator и применим его для сортировки массива.

```
TotalScoreComparator tsc = new TotalScoreComparator();
Arrays.sort(a,tsc);
```

Проверьте, что получиться в результате.

Обратите внимание, что предложенная в примере реализация интерфейса Comparator никак не затрагивает имеющееся описание класса Applicant, при этом заданы сразу две возможности для сортировки объектов.

Кроме того, интерфейс Comparator определяет специальный метод по умолчанию thenComparing(), который позволяет использовать цепочки компараторов для сортировки набора объектов, то есть можно сортировать объекты сразу по нескольким полям:

```
Comparator <Applicant> comp = sc.thenComparing(tsc);
Arrays.sort(a,comp);
```

В этом случае сортировка происходит сразу по фамилии и по общему баллу. Для того чтобы проверить результат попробуйте, например, у двух из трех имеющихся объектов задать одинаковые фамилии и разные общие баллы.

Упражнения.

1. Добавьте в пример с описанием иерархии наследования возможность сортировки животных по имени и возрасту. Создайте массив из объектов и отсортируйте его по различным параметрам.
2. Добавьте в приложение из упражнения 2 раздела 5 возможность сортировки объектов по фамилии и успеваемости (величине среднего балла). Создайте массив из объектов и отсортируйте его по различным параметрам.
3. Добавьте в упражнение 5 из раздела 6:

- возможность сортировки телеграмм по типу, стоимости и количеству слов;
- возможность рассчитать стоимость различных почтовых отправлений.

Создайте массив из объектов и отсортируйте его по различным параметрам.

4. Разработайте приложение для решения следующей задачи. Работник характеризуется фамилией и окладом. Заработная плата вычисляется дней следующим образом: полный оклад начисляется за 22 отработанных дня, если дней меньше или больше, то зарплата рассчитывается пропорционально количеству отработанных дней.

Разработайте соответствующий класс, содержащий необходимые поля и методы:

- создайте в классе конструкторы: по умолчанию, копирования, с параметрами;
- реализуйте методы, изменяющие и возвращающие значения полей класса;
- переопределите метод для представления информации об объекте в виде строки;
- метод для сравнения объектов на равенство;
- реализуйте интерфейс для сравнения объектов по окладу;
- реализуйте метод, рассчитывающий заработную плату.

В главном классе приложения создайте три объекта с использованием различных конструкторов и проверьте работу всех описанных в классе методов.

11. Обработка исключений

При разработке программы могут возникать логические ошибки, которые связаны с выполнением алгоритма программы, а не с синтаксисом программного кода. При возникновении таких ошибок работа программы будет автоматически завершена. Это, конечно, плохо, а иногда просто катастрофично, поэтому задача любого приличного разработчика состоит в том, чтобы не допускать возникновения таких ситуаций в работе программы, то есть программа должна завершать свою работу, когда это нужно программисту, а не когда этого хочет сама программа.

11.1 Генерация и обработка исключений

При возникновении ошибки выполнение программы приостанавливается и в соответствии с типом ошибки создается ***объект, содержащий сведения о том что, где и когда произошло***. Этот объект называют ***исключением***. Далее исключение передается для обработки программе (методу), в котором это исключение возникло. Если программа не обрабатывает исключение, то оно возвращается по умолчанию к обработчику исполняющей системы. Обработчик, в свою очередь, поступает очень просто: выводит на консоль сообщение о произошедшем исключении и прекращает выполнение программы. ***Задача программиста - перехватить и обработать исключения таким образом, чтобы не происходило аварийного завершения работы программы.***

В Java механизм обработки исключений использует следующие ключевые слова.

try {} – блок, в который помещаются потенциально опасные в отношении исключений участки кода.

catch(){} – блок, в котором исключения перехватываются и обрабатываются. Каждый блок catch(){} перехватывает и обрабатывает исключение одного типа, который указывается в его аргументе (в круглых скобках). Для перехвата нескольких типов исключений можно написать несколько блоков catch(){}.

finally{} – блок, в который помещается код, который выполняется в программе при любом исходе

throw – выбрасывает объект-исключение, указанного типа, а дальше обработка этого исключения идет как обычно. То есть, в случае необходимости, в любом месте кода мы можем сами сгенерировать исключение. Для этого и используется оператор throw, после которого указывается тип выбрасываемого исключения.

throws – используется в сигнатуре методов для предупреждения, о том, что метод может выбросить исключение.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch(), try{}catch(){}finally{}, try{}finally{}.

В Java существует иерархия классов, предназначенных для обработки исключительных ситуаций. Иерархия исключений в Java представлена следующим образом: родительский класс для всех Throwable. От него унаследовано два класса: Exception и Error. От класса Exception унаследован еще RuntimeException.

Error – критические ошибки, которые могут возникнуть в системе (например, StackOverflowError). Как правило, обрабатывает их система. Если они возникают, то приложение закрывается, так как при данной ситуации работа не может быть продолжена.

Exception – это проверенные исключения. Это значит, что если метод бросает исключение, которое унаследовано от Exception, то этот метод должен быть обязательно заключен в блок try-catch. Сам метод, который бросает исключение, должен в сигнатуре содержать конструкцию throws. Проверенные (checked) исключения означают, что исключение можно было предвидеть и, соответственно, оно должно быть обработано, работа приложения должна быть продолжена.

RuntimeException – это непроверенные исключения. Они возникают во время выполнения приложения и не требуют обязательного заключения в блок try-catch. Когда RuntimeException возникает, это свидетельствует об ошибке, допущенной программистом (неинициализированный объект, выход за пределы массива и т.д.). Поэтому данное исключение чаще всего не нужно обрабатывать, а нужно исправлять ошибку в коде, чтобы такое исключение не возникало вновь.

Далее приведен список системных исключений, которые требуются в работе чаще всего.

Непроверяемые системные исключения:

- ArithmeticException – арифметическая ошибка, например, деление на ноль
- ArrayIndexOutOfBoundsException – выход индекса за границу массива
- ArrayStoreException – присваивание элементу массива объекта несовместимого типа
- ClassCastException – неверное приведение
- IllegalArgumentException – неверный аргумент при вызове метода
- IndexOutofBoundsException – тип индекса вышел за допустимые пределы
- NegativeArraySizeException – создан массив отрицательного размера
- NullPointerException – неверное использование пустой ссылки
- NumberFormatException – неверное преобразование строки в числовой формат
- StringIndexOutOfBounds – попытка использования индекса за пределами строки
- TypeNotPresentException – тип не найден
- UnsupportedOperationException – обнаружена неподдерживаемая операция

Проверяемые системные исключения:

- ClassNotFoundException – класс не найден
- IllegalAccessException – запрещен доступ к классу
- InstantiationException – попытка создать объект абстрактного класса или интерфейса
- NoSuchFieldException – запрашиваемое поле не существует
- NoSuchMethodException – запрашиваемый метод не существует

Рассмотрим простой пример обработки исключений, когда в программе происходит деление на ноль. Если такая ошибка возможна при выполнении кода, то она будет вызывать аварийное завершение программы. Используем механизм обработки исключений для предотвращения этой ситуации.

```
public class MainClass {
  public static void main(String[] args) {
    int[] m = {-1,0,1};
    int a;
    a=0;
    try {
         //если a=0, то возникает ошибка  - деление на 0
      m[a] = 4/a;
```

```
        System.out.println(m[a]);
    }
    catch (ArithmeticException e) {
        System.out.println("Недопустимая арифметическая операция");
    }
  }
}
```

В блок try поместили код, в котором может возникнуть исключение – «деление на ноль», тип которого в иерархии исключений Java определяется как ArithmeticException, поэтому аргумент именно этого типа помещаем в блок catch. Теперь в случае возникновения ошибки будет выведено на экран соответствующее сообщение и программа просто завершится.

Изменим значение переменной a и рассмотрим еще одну исключительную ситуацию – выход за границы массива.

```
public class MainClass {
  public static void main(String[] args) {
    int[] m = {1,0,1};
    int a;
    a=-1;
    try {
        //если a = -1, происходит обращение к несуществующему индексу
        m[a] = 4/a;
        System.out.println(m[a]);
    }
    catch (ArithmeticException e) {
        System.out.println("Недопустимая арифметическая операция");
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Недопустимый индекс массива");
    }
  }
}
```

Обратите внимание, что в блоке try не добавилось никаких дополнительных проверок, а добавился еще один блок catch, то есть механизм обработки исключений позволяет отделить основной код от кода обработки ошибок. Блок catch обрабатывает соответствующее системное исключение типа ArrayIndexOutOfBoundsException, при этом на экран выводится соответствующее сообщение и программа завершает работу.

Теперь попробуем сами сгенерировать исключение при помощи оператора throw. Для этого введем следующее ограничение на элементы массива – элементами массива int[] m = {1,0,1}; могут быть только числа 0 и 1.

```
public class MainClass {
  public static void main(String[] args) {
    int[] m = {1,0,1};
    int a;
    a=2;
    try {
        //если a = 2, то элемент массива с номером 2 не равен 1 или 0
        m[a] = 4/a;
            if(m[a]!= 0 && m[a]!=1)
                throw new Exception("Недопустимый элемент массива");
        System.out.println(m[a]);
    }
```

```
    catch (ArithmeticException e) {
        System.out.println("Недопустимая арифметическая операция");
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Недопустимый индекс массива");
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
  }
}
```

Если элемент не равен 0 и не равен 1, генерируем исключение типа Exception, которое создается прямо в операторе throw с соответствующим сообщением и добавляем еще один блок catch, который обрабатывает это исключение. Оператор throw можно использовать в любом месте приложения. Этот оператор генерирует («выбрасывает») описанный в нем объект-исключение и далее обработка этого исключения идет как обычно в соответствующем блоке catch.

Обратите внимание, что при использовании стандартных типов исключений важно учитывать расположение соответствующих классов в иерархии наследования. В нашем примере есть исключение типа Exception и если исключение этого типа разместить в первом блоке catch, то этот блок будет перехватывать любые исключения, так как этот класс находится вверху иерархии классов исключений, то есть происходит обобщение исключений. Поэтому всегда нужно стараться сделать свои исключения максимально конкретными, чтобы не потерять информацию, описывающую исключительную ситуацию.

11.2. Создание собственных классов исключений

Помимо использования встроенных классов для обработки исключительных ситуаций можно описывать свои собственные классы исключений. Такая необходимость появляется в том случае, когда разработчику нужно исключение типа, который не предоставляется платформой Java. Например, при создании объекта класса Applicant при помощи конструктора с параметрами, в качестве общего балла может быть передано отрицательное значение или значение, превышающее максимальный итоговый балл. То есть возникает опасность создания объекта с неверно заданным значением атрибута. Чтобы обработать такую ситуацию опишем собственный класс исключение.

Для того чтобы создать собственное Java исключение, необходимо унаследовать собственный класс от класса Exception и переопределить требуемые методы.

Простейший вариант реализации собственного класса исключения выглядит следующим образом:

```
public class ApplicantException extends Exception{
   public ApplicantException(String s)
   {
      super(s);
   }
}
```

В классе ApplicantException определен только конструктор, который использует конструктор класса предка Exception, передавая ему в качестве параметра строку, которая буде содержать сообщение с информацией об ошибке.

Теперь мы можем использовать объект-исключение класса ApplicantException в конструкторе класса Applicant:

```
public Applicant(String surname, int totalScore ) throws ApplicantException{

      if(totalScore<=0||totalScore>maxTotalScore)
         throw new ApplicantException("Ошибка при создании объекта");

      this.surname = surname;
      this.totalScore = totalScore;
   }
```

Обратите внимание, что если метод не обрабатывает возникающее в нем исключение, а выбрасывает (throw) его, это следует отмечать в заголовке метода служебным словом throws и указанием класса исключения, поэтому к заголовку конструктора добавлено объявление того, что этот метод может выбрасывать исключение типа ApplicantException. В само тело конструктора добавлено условие, при котором генерируется исключение. Если условие истинно, то создается и выбрасывается исключение соответствующего типа. Остальные операторы конструктора при этом игнорируются. То есть работа конструктора прерывается, и соответствующий объект не создается.

Если в реализации конструктора по умолчанию использовался вызов конструктора с параметром, в котором может быть сгенерировано исключение, то и в заголовок конструктора по умолчанию добавляется соответствующее объявление о возможности возникновения исключения.

```
   public Applicant() throws ApplicantException{
      this("Ivanov", 200);
   }
```

Такое же объявление добавляется и в заголовок функции main( ) в главном классе:

```
public class MainClass {
   public static void main(String[] args) throws ApplicantException {

         try {
         Applicant ap1 = new Applicant("Григорьев",210);
                  Applicant ap2 = new Applicant("Сидоров",-200);
                  Applicant ap3 = new Applicant("Акопян",195);
      . . .

      }
           catch(ApplicantException e){
         System.out.println(e.getMessage());
      }
   }
}
```

Теперь создание объектов класса Applicant происходит в блоке try (в данном случае видимы они будут только внутри этого блока), и если допущена ошибка в количестве баллов, как в данном случае, то на экран буде выведено соответствующее сообщение:

Ошибка при создании объекта

Для вывода сообщения используется метод getMessage() класса Exception. Приложение в данном случае просто завершит свою работу. Если же при создании объектов ошибок допущено не было, то приложение продолжит свою дальнейшую работу и выполнит все предписанные действия.

Упражнения

1. Для класса из упражнения 1 раздела 5 предусмотрите обработку исключительных ситуаций для следующих случаев:

- в конструкторе с параметрами генерируется исключение, если передан отрицательный размер массива;
- в методе, возвращающем значение элемента, генерируется исключение, если номер элемента, передаваемый в качестве аргумента, выходит за границы массива;
- в методе, изменяющем значение элемента, генерируется исключение, если номер элемента, передаваемый в качестве аргумента, выходит за границы массива.

В главном классе обработайте перечисленные исключения и протестируйте работу приложения для всех возможных случаев.

2. Для класса из упражнения 2 раздела 5 предусмотрите обработку исключительных ситуаций в методе, добавляющем оценку к списку оценок:

- список оценок уже заполнен;
- передаваемая оценка не из пятибалльной системы оценивания, то есть не является целым числом от 1 до 5. Для этого случая разработайте свой собственный класс исключение.

В главном классе обработайте перечисленные исключения и протестируйте работу приложения для всех возможных случаев.

12. Потоковые классы для ввода и вывода данных

Ввод/вывод в Java осуществляется потоковыми классами. Понятие потока позволяет абстрагироваться от конкретного физического устройства и писать одинаковый код для вывода данных на консоль, в файл или внешнее устройство, такое как принтер. Эта особенность обусловлена тем, что все потоковые классы унаследованы от одних и тех же абстрактных классов, определяющих базовые методы для взаимодействия пользовательских приложений с потоками. Средства потокового ввода/вывода размещаются в пакете java.io. В случае ошибок ввода-вывода методы потоковых классов выбрасывают исключение класса `IOException` или его подклассов, например, `FileNotFoundException`. Классы этих исключений не являются подклассами класса `RuntimeException` и потому нуждаются в обязательной обработке.

12.1 Байтовые и символьные потоки ввода и вывода

Потоки ввода-вывода и соответствующие классы можно разделить на байтовые и символьные. Байтовые потоки предназначены для передачи бинарных данных, а символьные - текстовых. Для представления информации символьные потоки используют Unicode.

Все классы ввода вывода образуют единую иерархию классов. Базовыми являются байтовые потоки. Большинство остальных классов представляют собой «обертки базовых классов», добавляя к ним специализацию, такую как работа с символами, буферизация, работа с файлами.

Поскольку вся символьная информация представляется в Java в Unicode, в то время как внешние данные зачастую представлены в других кодировках, то для эффективного взаимодействии Java с внешним миром требуется преобразование кодировок. Эту операцию выполняют символьные потоки. Они могут «обертывать» любой байтовый поток и обращаться с полученным потоком как с символьным, выполняя все необходимые преобразования.

Рассматриваемым потокам соответствуют классы **`InputStreamReader`**, используемый для чтения, и **`OutputStreamWriter`**, используемый для записи. Для создания объектов данных классов используются следующие конструкторы:

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

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

Для оптимизации операций ввода-вывода используются буферизуемые потоки. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи данных. Функциональность буферизованных потоков обеспечивается классами **`BufferedInputStream`**, **`BufferedOutputStream`** (входной и выходной байтовые потоки с буферизацией), **`BufferedReader`** и **`BufferedWriter`** (аналогичные символьные потоки). Для создания объектов этих классов используются конструкторы, принимающие в качестве аргумента ссылку на обёртываемый поток:

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

BufferedWriter(Writer out)

Во входной символьный поток с буферизацией добавляет метод `String readLine() throws IOException`, осуществляющий чтение одной строки из входного символьного потока. В случае достижения конца файла этот метод возвращает значение `null`.

12.2 Работа с файлами

Работа с файлами часто начинается с создания объектов класса **File**. Класс **File** работает не с потоками, а непосредственно с файлами. Он может хранить информацию об определённом файле, а также о группе файлов, находящихся в каталоге. Если объект класса представляет каталог, то его метод `list()` возвращает массив строк с именами всех файлов.

Класс **File** позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. Для создания объектов класса **File** можно использовать один из следующих конструкторов.

File(File dir, String name) - указывается объекта класса **File** (каталог) и имя файла,

File(String path) - указывается путь к файлу без указания имени файла,

File(String dirPath, Sring name) - указывается путь к файлу и имя файла,

File(URI uri) - указывается объект URI, описывающий файл.

Перечислим некоторые методы класса **File** и результат их работы.

boolean canRead() - доступен ли файл для чтения

boolean canWrite() - доступен ли файл для записи

boolean delete() - удаляет файл, также можно удалить пустой каталог

boolean exists() - файл существует или нет

String getAbsolutePath() - абсолютный путь файла, начиная с корня системы

String getName() - возвращает имя файла

String getParent() - возвращает имя родительского каталога

String getPath() - путь к файлу

boolean isFile() - проверяет, что объект является файлом, а не каталогом

boolean isDirectory - проверяет, что объект является каталогом

long lastModified() - дата последнего изменения, возвращает количество миллисекунд, прошедшее с 1 января 1970 года,

boolean renameTo(File newPath) - переименовывает файл. В параметре указывается имя нового файла. Если переименование прошло неудачно, то возвращается `false`.

Приведем пример вывода на консоль даты последней модификации файла.

```
import java.io.File;
import java.util.Date;
import java.text.SimpleDateFormat;
...
 File f = new File("data.txt");
 Date myDate = new Date(f.lastModified());
 SimpleDateFormat formatDate = new SimpleDateFormat("dd-MM-yyyy");
 System.out.println(formatDate.format(myDate));
```

Результат работы фрагмента кода:

```
 19-11-2014
```

Можно использовать более короткую запись:

```
 System.out.println(new SimpleDateFormat("dd-MM-yyyy HH-mm-ss").
 format( new Date(new File("data.txt").lastModified())));
```

В этом случае на консоли отобразится строка:

```
19-11-2014 19-45-19
```

Подумайте, что общего в последних двух примерах и чем они отличаются.

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

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

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

Символьные потоки, связанные с файлами, предназначены для работы с текстовыми файлами. Фактически они представляют собой комбинацию байтового потока и символьного потока-обёртки, осуществляющего преобразование. При этом всегда используется кодировка системной локали. Данным потокам соответствуют классы **`FileReader`** и **`FileWriter`** со следующими конструкторами:

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

Используются они полностью аналогично конструкторам байтовых потоков, связанных с файлами.

Рассмотрим примеры ввода строк текста с консоли. В первом примере с консоли считываются несколько строк, которые вводит пользователь. Каждая строка может состоять из произвольного количества символов и завершается нажатием клавиши Enter. Окончанием работы является ввод пустой строки – нажатие клавиши Enter без ввода других символов.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class InOutExamlpe {
   public static void main(String[] args) {
      try {
       System.out.println("Enter line:");
       BufferedReader reader = new BufferedReader
                                              (new InputStreamReader(System.in));
         String w = reader.readLine();
         while (!w.isEmpty()) {
         // обработка строки w
            w = reader.readLine();
         }
      }
      catch (IOException e) {
```

```
            System.out.println("Error consol data");
        }
}
```

Во втором примере вводится одна строка символов, содержащая информацию об имени ученика и его оценках. Эти данные разделены пробелами. Первоначальная строка w делится на массив строк a, первый элемент которого `a[0]` – имя, а следующие элементы массива a в цикле преобразуются в числа и для них рассчитывается среднее арифметическое. На консоль выводится имя и средний балл ученика.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class InOutExamlpe {
   public static void main(String[] args) {
      try {
        BufferedReader reader = new BufferedReader
                                        (newInputStreamReader(System.in));
        System.out.println("Enter data:");
        w = reader.readLine();
        String[] a = w.trim().split(" +");
        double sb = 0;
        for (int i = 1; i < a.length; i++) {
           sb += Double.parseDouble(a[i]);
        }
        System.out.println("Student " + a[0] + ". Average = " +
                                                        (sb / (a.length — 1)));
          reader.close();
      }
      catch (IOException e) {
         System.out.println("Error consol data");
      }
}
```

Следующие два примера решают те же самые задачи, но ввод данных осуществляется из файлов. В первом примере все строки из файла `data.txt` дублируются на консоль и в файл `rezult.txt`.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;

public class InOutExamlpe {
  public static void main(String[] args) {
    try {
         BufferedReader in = new BufferedReader
                                    (new FileReader("data.txt"));
         PrintWriter out = new PrintWriter("rezult.txt");
         while ((str = in.readLine()) != null) {
            out.println(str);
```

```
            System.out.println(str);
          }
          out.close();
          in.close();
      } catch (IOException e) {
           System.out.println("Error data");
      }
    }
}
```

Во втором примере имя и средний балл ученика выводятся только на консоль.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class InOutExamlpe {
   public static void main(String[] args) {
      try {
         BufferedReader reader = new BufferedReader
                                              (new FileReader("student.txt"));
         String w = reader.readLine();
         String[] a = w.trim().split(" +");
         double sb = 0;
         for (int i = 1; i < a.length; i++) {
            sb += Double.parseDouble(a[i]);
         }
         System.out.println("Student " + a[0] + ". Average = " +
                                                 (sb / (a.length — 1)));
         reader.close();
      } catch (IOException e) {
         System.out.println("Error data");
      }
   }
}
```

Важно обратить внимание, что имя файла в примерах указывается в виде строки и подразумевает относительную адресацию места нахождения этого файла в файловой системе компьютера. В случае работы с программным проектом в среде разработки файл должен находиться в основном каталоге проекта.

Рассмотрим дополнительно примеры, демонстрирующие другие особенности использования объектов классов для ввода и вывода данных. В следующем примере в файл добавляется строка текста, старое содержание сохраняется.

```
PrintWriter add = new PrintWriter(new FileOutputStream("test.txt", true));
add.write("AAA");
add.close();
```

Фрагмент кода создает потоки для чтения и записи символов в кодировке MS-DOS и Windows:

```
BufferedReader br = new BufferedReader(new InputStreamReader(new
                                              FileInputStream("test.txt")), "Cp866");
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter
```

```
                                    (new FileOutputStream("rezult.txt")), "Cp1251");
```

Последний пример содержит метод, копирующий содержимое одного файла в другой.

```
public static void copy(File src, File dst) {
   int tmp;
   try {
      if (src.exists()) {
         BufferedInputStream in = new BufferedInputStream
                                                  (new FileInputStream(src));
         BufferedOutputStream out = new BufferedOutputStream
                                                    (new FileOutputStream(dst));
         while ((tmp = in.read()) != -1) {
            out.write(tmp);
            tmp = in.read();
         }
         out.close();
         in.close();
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Подумайте: символьные или байтовые потоки используются в этом примере?

12.3 Сериализация

Процесс записи объекта в выходной поток получил название ***сериализации***, а чтения объекта из входного потока и восстановления его в оперативной памяти — ***десериализации***. Эти операции выполняются, если необходимо сохранить состояние объекта между моментом завершения программы и новым ее запуском. Таким образом можно сохранять, например, настройки программы, которые меняет пользователь в ходе очередного сеанса работы с программой.

Сериализация объекта нарушает его безопасность, поскольку какой-нибудь процесс может сериализовать объект, переписать некоторые элементы, представляющие private-поля объекта, обеспечив себе, например, доступ к секретному файлу, а затем десериализовать объект с измененными полями и совершить с ним недопустимые действия. Поэтому сериализации можно подвергнуть не каждый объект, а только тот, который реализует интерфейс **`Serializable`**. Этот интерфейс не содержит ни полей, ни методов. В сущности запись `class A implements Serializable{...}` – это только пометка, разрешающая сериализацию класса `A`.

Если в объекте присутствуют ссылки на другие объекты, то они тоже сериализуются. Метод `writeObject()` распознает две ссылки на один объект и выводит его в выходной поток только один раз. К тому же, он распознает ссылки, замкнутые в кольцо, и избегает зацикливания.

Все классы объектов, входящих в такое сериализуемое множество, а также все их внутренние классы, должны реализовать интерфейс **`Serializable`**, в противном случае будет выброшено исключение класса `NotSerializableException` и процесс сериализации прервется.

Сериализация объектов Java позволяет вам взять любой объект, который реализует интерфейс **`Serializable`** и включит его в последовательность байт, которые могут быть

полностью восстановлены для регенерации оригинального объекта. Это также выполняется при передаче по сети, что означает, что механизм сериализации автоматически поддерживается на различных операционных системах. То есть, вы можете создать объект на машине с Windows, сериализовать его и послать по сети на Unix машину, где он будет корректно реконструирован. Вам не нужно будет беспокоиться о представлении данных на различных машинах, порядке следования байт и любых других деталях.

Для сериализации выбранного объекта программы необходимио создать объект класса **`OutputStream`** или его наследника, а затем вложить его в объект **`ObjectOutputStream`**. После этого достаточно вызвать метод **`writeObject(),`** аргументом которого будет выбранный объект**,** и этот объект будет сериализован и послан в **`OutputStream`**. Чтобы провести обратный процесс, создаются объекты потоковых классов **`InputStream`** и **`ObjectInputStream`** и вызывается метод **`readObject( )`**. Результатом работы этого метода является ссылка на общий базовый класс **`Object`**, так что необходимо выполнить обратное приведение типов, чтобы получить нужный объект.

Рассмотрим пример сериализации объекта пользовательского класса. Сначала приведем код самого класса. Он хранит информацию об имени и возрасте человека.

```
import java.io.Serializable;
public class Person implements Serializable {
   private String name;
   public String getName() {
      return name;
   }
   private int age;
   public int getAge() {
      return age;
   }
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   @Override
   public String toString() {
      return name + " (" + age + ")";
   }
}
```

Теперь рассмотрим код основной программы, где осуществляется сериализация и десериализация объекта класса `Person`. Данные об объекте в результате сериализации сохраняются во внешнем файле `data.ser` в бинарном виде. Если файл существует, объект десериализуется (восстанавливается) из него. Если этого файла нет, значит объект еще не был создан в программе и это необходимо сделать. Для этого пользователя просят ввести данные в консоли. После создания или восстановления объекта, информация о нем выводится на консоль и происходит сериализация этого объекта класса `Person`.

```
public class ExampleSerialize {
   public static void main(String[] args) {
    try {
      File data = new File("person.ser");
      Person person;
```

```
        if (data.exists()) {
            ObjectInputStream ois = new ObjectInputStream
                                                    (new FileInputStream(data));
            person = (Person) ois.readObject();
            ois.close();
        } else {
            System.out.println("Enter person data");
            String str = new BufferedReader
                                (new InputStreamReader(System.in)).readLine();
                if (str.split(" +").length != 2) {
                    throw new IOException("incorrectly: " + str);
                }
                person = new Person(str.split(" +")[0],
                                                Integer.parseInt(str.split(" +")[1]));
        }
        System.out.println(person.getName()+" "+person.getAge());
        ObjectOutputStream oos = new ObjectOutputStream
                                                    (new FileOutputStream(data));
        oos.writeObject(person);
        oos.close();
    } catch (IOException e) {
        System.out.println(e.getMessage());
       }
     catch (Exception e) {
        System.out.println(e.getMessage());
     }
    }
}
```

Упражнения

1. Дан файл с произвольным текстом. Разделители слов – пробелы. Отформатировать текст по ширине по следующим правилам: длина строки M символов, отступ абзаца – A символов. Параметры M и A задает пользователь. Сохраняйте введенные значения параметров между запусками программы. Результат форматирования вывести в отдельный файл.
2. Дан файл с текстом программы на Java. Записать в выходной файл все переменные, указать, в какой строке расположено определение переменной и номера строк, где значение переменной изменяется. Информация о новой переменной должна располагаться с новой строки.
3. Программа должна читать последовательность целых чисел из входного файла и выводить в выходной файл максимальное, минимальное, среднее значение, количество появлений каждого числа в последовательности. Также должна выводиться гистограмма, показывающая распределение частот появления чисел в последовательности: по одной оси – числа, входящие в последовательность, по другой – частота встречаемости).

Заключение

Учебно-методическое пособие включает в себя материалы для начинающих изучение языка Java. Подробная документация о классах стандартных пактов Java SE содержится на официальном сайте https://docs.oracle.com/javase/8/docs/api/. Учебные материалы от разработчиков можно найти также на сайте https://docs.oracle.com/javase/tutorial/.

Дополнительные учебные материалы можно найти в списке литературы. Для начинающих рекомендуются книги [1–4]. Для более углубленного изучения Java и программирования [5–7]

Литература

1. Шилдт, Г. Java 8. Полное руководство, 9-е изд. / Герберт Шилдт – М.: ООО «ИД Вильямс», 2015. – 1376 с.

2. Монахов, В. Язык программирования Java и среда NetBeans /В. Монахов. - М.: БХВ-Петербург, 2012. - 720 с.

3. Васильев, А. Н. Самоучитель Java с примерами и программами / А.Н. Васильев. - М.: Наука и техника, 2016. - 368 с.

4. Хорстманн, Кей С. Java SE 8. Вводный курс / Хорстманн Кей С.. - М.: Диалектика / Вильямс, 2014. - 898 с.

5. Эккель, Б. Философия Java / Брюс Эккель. - М.: Питер, 2016. - 809 с.

6. Гарнаев, А. WEB-программирование на Java и JavaScript / Андрей Гарнаев , Сергей Гарнаев. - Москва: СПб. [и др.] : Питер, 2017. - 718 с.

7. Давыдов, С. IntelliJ IDEA. Профессиональное программирование на Java / Станислав Давыдов , Алексей Ефимов. - М.: БХВ-Петербург, 2015. - 800 с.