1 Как выглядят программы на Java

На начальном этапе будем считать, что наши программы находятся в одном файле и все действия выполняются также в одном месте. Общая структура такого файла выглядит так:

```
public class Main { // начало тела класса
    public static void main(String [] args) { // начало тела метода
        // здесь будем писать код
    }
}
```

Имя класса ОБЯЗАТЕЛЬНО совпадает с именем файла, в котором он находится.
Важно понимать, что тело класса всегда ограничивается фигурными скобками. То же самое относится к телу метода.
На данном этапе мы считаем, что есть один "главный"метод, в котором выполняются все основные операции программы. Для Java "главный"метод является указателем входа в программу. Другими словами, когда мы запускаем программу, ее выполнение начинается с этого main.

2 Hello Java

Теперь мы вполне можем написать первую программу:

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

```
>> Hello, Java!
```

Еще несколько моментов оформления кода:

- Код пишется "лесенкой"
- В конце каждой строки кода ставится точка с запятой
- Весь написанный код находится ТОЛЬКО в методах

3 Переменные и константы

Все данные в программе как правило хранятся в переменных. У каждой переменной есть тип, имя и значение. Например:

```java
public class Main {
    public static void main(String [] args) {
        int a = 1; // создаем целочисленную переменную
        String s = "hello"; // создаем строчную переменную
        System.out.println(a);
        System.out.println(s);
    }
}
```

```
>> 1
>> hello
```

В данном примере мы создаем две переменные целочисленного и строчного типа и печатаем их на экран. Как мы видим, создание переменных соответствует схеме тип-имя-значение.

4 Примитивные типы данных

К примитивным типам данных относятся целочисленные и дробные типы данных.

- int - 32 битное целое число
- long - 62 битное целое число
- short - 16 битное целое число
- char - 16 битное беззнаковое число
- byte - 8 битное число
- float - дробное число с одинарной точностью
- double - дробное число с двойной точностью

С указанными типами можно совершат основные математические операции: сложение, вычитание, умножение, деление, взятие остатка от деления.
В коде это выглядит так:

```java
public class PrimitiveOperations {
    public static void main(String[] args) {
        int a = 125;
        int b = 25;

        int sum = a + b;
        int sub  = a - b;
```

```
        int mul = a * b;
        int div = a / b;
        int mod = a % b;
        System.out.println(sum);
        System.out.println(sub);
        System.out.println(mul);
        System.out.println(div);
        System.out.println(mod);

        float c = 123;
        float d = 12.2f;
        System.out.println(c - d);
        System.out.println(c * d);
        System.out.println(c % d);
    }
}
```

5 Правило именования переменных и классов

Есть договоренность использовать camelCase стиль при именовании переменных и классов. Это означает, что если название переменной содержит больше одного слова, то каждое новое слово начинается с большой буквы.
Например, вместо sumofallusers нужно писать sumOfAllUsers.
Вместо класса Playerinfo нужно писать PlayerInfo.
ВАЖНО: имена классов начинаются с большой буквы, переменных всегда с маленькой, методов также с маленькой буквы.

6 Строчный тип данных

Отдельно хочется сказать про строки, так как они уже не относятся к примитивным типам. Класс String является встроенным в Java и уже имеет внутри себя функционал для работы со строчными выражениями. Например, мы можем узнать длину строки, значение символа на конкретной позиции в строке и, например, взять подстроку.

```
public class SimpleStrings {
    public static void main(String[] args) {
        String hello = "Hello";
        String world = " world!";


        // есть два способа объединить две строки в одну
        String helloWorld = hello + world;
```

```
        String anotherHelloWorld = hello.concat(world);

        System.out.println(helloWorld);
        System.out.println(anotherHelloWorld);

        // у каждой строки мы можем узнать ее длину
        System.out.println(hello.length());
        System.out.println(world.length());
        System.out.println(helloWorld.length());

        // печатаем подстроку с 1 по 3 символ невключительно [1, 3)
        System.out.println(helloWorld.substring(1, 3));

        // печатаем символ на первой позиции
        // ВАЖНО: позиции нумеруются с нуля
        System.out.println(world.charAt(1));
    }
}
>> Hello world!
>> Hello world!
>> 5
>> 7
>> 12
>> el
>> w
```

7 Преобразование типов

Строки и числа часто приходится перекидывать между собой, для этого в языке есть встроенные функции: Integer.toString(123) преобразует число в строку, Integer.valueOf("123") преобразует строку в число.

```
import java.util.*;

public class Casts {
        public static void main(String [] args) {
                Scanner sc = new Scanner(System.in);
                int a = sc.nextInt();
                String s = Integer.toString(a);
                s = s + "!";
                System.out.println(s);
```

```
                String number = sc.nextLine();
                int b = Integer.valueOf(number);
                b = b + 1;
                System.out.println(b);

        }
}
```

8 Условные операторы

Зачастую программы обладают нелинейной логикой, то есть их работа зависит от тех данных, которые были получены на предыдущих шагах. Для того, чтобы выполнять отдельные куски кода только в случае удовлетворения условий, в языке есть условные операторы. Условия могут быть записаны по-разному, но первое условие всегда оформляется в виде блока if:

```
if(условие) {
    // действия выполняемые при верном условии
}
```

В качестве условия может быть указано любое выражение, имеющее тип boolean. Например, мы можем сравнивать численные значения переменных и в результате получать логический тип.

```
int a = 12;
int b = 14;
if(a > b) {
    System.out.println("a > b");
}
```

Между численными типами доступны следующие операции:

- $>$ - строго больше
- $<$ - строго меньше
- $>=$ - больше или равно
- $<=$ - меньше или равно
- $==$ - равенство
- $!=$ - неравенство

В качестве объектов для сравнения могут выступать как переменные численного типа, так и обычные константные значения:

```
int a = 15;
if(a % 2 == 0) {
    System.out.println("Число четное");
}
```

Бывают ситуации, когда после проверки условия нужно сделать одно из двух действий: для верного условия и для всех остальных случаев. Тогда на помощь приходит блок else:

```
int a = 15;
if(a % 2 == 0) {
    System.out.println("Число четное");
} else {
    System.out.println("Число нечетное");
}
```

Например, в данном случае у нас проверка работает так: если число четное (остаток ноль), то печатаем одну строку, в противном случае (else) печатаем другую строку. Важно обратить внимание, что эти блоки связаны между собой и программа просматривает их последовательно: сначала проверяет первое условие, если оно верное, то выполняет код из первых фигурных скобок, если же условия неверно, то выполняет код из вторых фигурных скобок. При такой записи всегда будет выполнен ровно один кусок кода.
Также важно понимать, что блок else можно написать только после блока if.

Кроме этого бывают случаи, когда необходимо проверить больше двух условий для одной переменной. Данный кусок кода обрабатывает результат http запроса. После обращения к какому-то адресу в сети, вы всегда получаете в ответ некоторое число - код возврата. В зависимости от того, что вам вернули из сети, можно сказать, успешен ли запрос или в ходе выполнения произошла ошибка.
После первого блока if вы можете добавлять любое количество блоков else if, которые будут последовательно проверяться одно за другим. Опять же обращаю внимание, что при такой записи условия связаны в цепочку и будет выполнен максимум один блок кода.

```
public class ConditionsExample {
    public static void main(String[] args) {
        int requestResult = 200;

        if(requestResult == 200) {
            System.out.println("Request is OK");
        } else if(requestResult == 400) {
            System.out.println("Bad request");
        } else if(requestResult == 403) {
            System.out.println("Request is forbidden");
```

```
        } else {
            System.out.println("Unknown return code");
        }
    }
}
```

Логика работы этой части кода следующая: если код возврата равен 200, печатаем 'Request is OK', иначе если результат равен 400, печатаем 'Bad request', иначе если код возврата равен 403, печатаем 'Request is forbidden', иначе во всех остальных случаях печатаем 'Unknown return code'.
Таким образом мы можем выстраивать цепочки условий, которые позволяют проверить, в каком состоянии находится программа.

8.1 Объединение нескольких условий

Часто возникает необходимость проверять несколько условий одновременно. Например, что число находится между двумя заданными границами. Для этого есть специальные операторы, позволяющие объединять несколько условий внутри одного блока if. Для того, чтобы записать "условие1 и условие2 нужно написать оператор :

```
int a = 15;
if(a > 10 && a < 20) {
        System.out.println("In ragne");
}
```

Тогда условие будет верно, когда верно и первое условие, и второе условие.

Для того, чтобы записать "условие1 или условие2 нужно написать оператор ||:

```
int value = 15;
if(a < 10 || a > 20) {
        System.out.println("In range");
}
```

Это условие будет верно тогда, когда верно либо певрое условие, либо второе.

Для того, чтобы понять, какой оператор нужно использовать, сформулируйте условие в голове: если используете "и то, и другое значит оператор (и), если используете "или то, или другое значит оператор || (или).

8.2 Сравнение между ссылочными типами

При использовании операции сравнения (==) нужно учитывать, что ее можно принять ТОЛЬКО если работаете с примитивными типами. (см. Примитивные типы данных) Рассмотрим пример:

```
String a = "hello";
String b = "hello";
```

```
if (a == b) {
        System.out.println("Strings are equals");
}
```

В этом случае, переменная a и b ссылаются на разные куски памяти, то есть хранят ссылки на разные объекты. При сравнении через двойное равно ссылочных объектов вы сравниваете не значения объектов (в нашем случае это символы строк), а значение ссылок. Таким образом, сравнение через двойное равно небезопасно и его следует избегать при работе со ССЫЛОЧНЫМИ типами.

Что же использовать вместо такой операции сравнения? Для этого есть метод equals, который применяется к сравниваемым строкам. Например:

```
String a = "Hello";
String b = "Hello";
if(a.equals(b)) {
    System.out.println("Strings are equal");
}
```

9 Более подробно о типе char

Среди примитивных типов данных есть один, о котором стоит поговорить отдельно - это тип char. Для языка переменная типа char имеет 16 разрядов и может хранить значения вплоть до 65536. Но в то же время никто не запрещает сложить в эту же переменную отдельный символ:

```
char number = 50;
char symbol = 'a';
```

Исходя из такой записи видно, что строчный тип String хранит в себе последовательность символов, каждый из которых является типом char. Об этом так же говорит метод charAt(), который возвращает символ в виде типа char. Так почему же char считается численным типом в Java? Это объясняется тем, что на самом деле char хранит в себе код символа, который в нем записан. Коды символов берутся из таблицы символов (ASCI таблица). В ней вы найдете код для любого печатного символа. Интересной особенностью значений кодов является то, что, например, символы английского алфавита находятся в таблице строго друг за другом.
Для того, чтобы получить из печатного символа его код, нужно преобразовать тип char, например, в int.

```
char symbol = 'a';
int codeOfSymbol = (int) symbol;
System.out.println(codeOfSymbol);
```

```
>> 97
```

Кроме того, можно выполнить и обратное преобразование из кода символа в сам символ:

```
int value = 99;
char symbol = (char) value;
System.out.println(symbol);

>> c
```

Для чего может быть полезным такое преобразование? Например, для обработки символов строки. Допустим, вы хотите проверить, что текущий символ строки является строчной латинской буквой. Для этого можно записать подобное условие:

```
char currentChar = 'r';
if(currentChar >= 'a' && currentChar <= 'z') {
        System.out.println("It is lower case character");
}
```

Как вы видите, в данном случае мы работаем с символом как с числом, а последовательное расположение символов алфавита в ASCI таблице позволяет нам проверить, что он находится между 'a' и 'z'.

10 Циклы

Часто задача программы, которую мы пишем - автоматизировать большое количество однотипных действий. Для описания повторения одного и того же действия заданное количество раз в языке есть специальная конструкция - цикл.

Циклы можно записать по-разному, но у всех есть общая схема, описание которой зачастую необходимо для работы. Сначала рассмотри описание цикла while.

```
int n = 10000; // количество повторений
int i = 0; // счетчик цикла
while (i < n) { // описание условия входа в цикл
    System.out.println(i); // любые действия с переменной i или любыми другими
    i = i + 1; // шаг цикла
}
System.out.println("Program finished");
```

Рассмотрим логику работы цикла. Когда мы доходим до строки while (i < n) мы проверяем выполнение условия. Если условие верно, мы заходим внутрь цикла и выполняем все действия, описанные внутри фигурных скобок. После завершения команд тела цикла, мы снова возвращаемся наверх к строке while (i < n) и проверяем условие. Если оно стало неверно, внутрь фигурных скобок мы больше не заходим, а продолжаем выполнение команд после цикла. Таким образом, для написания цикла необходимо задать несколько вещей:

- количество повторений
- начальное значение счетчика
- условие входа в цикл
- шаг цикла

Остановимся более подробно на последнем пункте. Для того, чтобы цикл когда-то завершился, необходимо, чтобы условие в какой-то момент перестало выполняться. Для этого мы изменяем значение счетчика (выполняем шаг цикла). Таким образом, мы начинаем перебирать значение счетчика от 0 и выполняем это до тех пор, пока он меньше n (количество повторений).

Аналогичную конструкцию можно написать в более удобном и коротком виде с помощью цикла for:

```
for(int i = 0; i < n; i = i + 1) {
    System.out.println(i);
}
```

В круглых скобках мы последовательно указываем: создание переменной-счетчика, условие входа в цикл, шаг цикла.

Рассмотрим применение цикла для перебора всех символов строки и подсчета количества буквы A в ней:

```
String value = "AbcAad";
int count = 0;
for(int i = 0; i < value.length(); i = i + 1) {
    char currentChar = value.charAt(i);
    if(currentChar == 'A') {
        count = count + 1;
    }
}
System.out.println(count);

>> 2
```

В данном случае счетчик цикла используется как индекс элемента, на который мы сейчас смотрим. Начинаем мы с нулевого элемента и продолжаем до тех пор, пока индекс строго меньше длины строки. То есть количество повторений в этом случае - длина строки.
Внутри цикла мы складываем символ на позиции i во временную переменную и далее сравниваем его с символом 'A'. Если условие верно, увеличиваем значение ответа на единицу.

10.1 Бесконечные циклы и прерывание циклов

Иногда в программе возникает необходимость написать цикл, который работает условное "бесконечно"долго. Для этого нужно обратить внимание на то, какое условие мы указываем в цикле. Если мы укажем в качестве условия что-то константное верное (например, true, 1 == 1), то условие цикла будет верно всегда. Рассмотрим пример с циклом while:

```
Scanner sc = new Scanner(System.in);
String value = sc.nextLine();
while(true) {
        System.out.println(value);
        value = sc.nextLine();
}
```

Такой кусок кода будет постоянно считывать ввод пользователя и печатать его же на экран. И такая программа может быть завершена только если мы "насильно"выключим ее (завершим исполнение).

Давайте модифицируем код до состояния, когда он будет прекращаться при вводе слове "exit":

```
Scanner sc = new Scanner(System.in);
String value = sc.nextLine();
while(true) {
        if(value.equals("exit") {
            break;
        }
        System.out.println(value);
        value = sc.nextLine();
}
```

Данная программа будет проверять каждую введенную строку на равенство со строкой "exit"и в случае равенства прекращать выполнение цикла. Это происходит благодаря команде break. Ее работа заключается в том, что он заканчивает выполнение цикла в тот же момент, как мы доходим до строки с этой командой. Это происходит вне зависимости от текущего значения условия, счетчика и чего-либо еще - мы сразу же перемещаемся за границы цикла и продолжаем выполнять следующий действия.

Если же у нас возникает ситуация, в которой нам нужно пропустить только текущую итерацию (повторение) цикла, можно использовать команду continue. Например, с помощью нее можно напечатать все четные числа до n:

```
int n = 10;
for(int i = 0; i < n; i++) {
        if(i % 2 != 0) {
```

```
                continue;
        }
        System.out.println(i);
}
```

Данный код будет работать следующим образом: если текущее значение счетчика - нечетное число, то мы пропускаем текущую итерацию цикла и сразу же возвращается на начало новой итерации. Обращу внимание, что при выполнении команды continue внутри цикла for, шаг цикла все равно выполняется. То есть операция i++ выполняется в любом случае.

11 Массивы

11.1 Что такое массивы и для чего они применяются

Представим, что у нас есть данные о средней температуре за каждый день года. То есть у нас есть 365 значений типа int и нам необходимо их обрабатывать. Если пользоваться тем, что у нас есть на текущем этапе, нам нужно создать 365 переменных. Этот этап можно пережить, но допустим нам необходимо посчитать среднее значение для всех этих чисел. Тогда это выражение будет выглядеть примерно так:

```
int sum = a1 + a2 + a3 + a4 + a5 + ... + a365;
```

Это выглядит не очень удобно и писать каждый раз такое выражение можно сильно устать. Аналогичные проблемы возникнут, если нам нужно изменить значения всех переменных.

Поэтому в языке есть специальные конструкции для хранения большого количество однотипных данных в одной переменной. Тип данных, который для этого необходим - это массив.

11.2 Создание массива

Схема создания массива, хранящего в себе элементы типа int, выглядит следующим образом:

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

В данном случае создается цепочка из 10 элементов, каждый из которых имеет тип int. Обращаю внимание, что int [] является отдельным типом в языке, который как раз и обозначает последовательную цепочку данных - массив. Имя переменной может быть задано любым, как и при создании любой другой переменной. В правой части равенства мы фактически создаем ячейки: указываем тип ячеек и в квадратных скобках их количество. Важно понимать, что после этой строки создается пустой массив, в котором лежат нули. В качестве количества элементов массива может быть указано любое выражени типа int (переменная, результат операции между переменными или просто константа). Типом ячеек массива может выступать любой

тип, существующий в языке. Можем создавать, массивы строк, чисел, символов и даже массивы массивов.
На практике часто необходимо создать массив из заданного набора элементов. Для этого есть специальная констукция:

```
int [] array = {1, 2, 3, 4, 5, 23, 21};
```

После этого у нас появляется массив состоящий из заданного набора элементов и с ним сразу же можно работать.

11.3 Обращение к элементам массива

После создания массива нам необходимо каким-то образом научиться работать с элементами, которые в нём лежат. Для этого каждой ячейке присваивается индекс. Индексы начинаются с 0 и соответственно индекс последнего элемента - длина массива минус 1.
Для того, чтобы обратиться к элементу, нужно написать имя переменной массива и далее в квадратных скобках индекс. Например:

```
int [] arr = {10, 20, 40, 20, 13};
System.out.println(arr[2]);

>> 40
```

Дальнейшие операции между любыми элементами массива выполняются точно так же, как между обычными переменными такого же типа. А именно:

```
int a = 12;
int b = 13;
int sum = a + b;

int [] arr = {12, 13};
sum = arr[0] + arr[1];
```

Массив не изменяет свойства объектов, которые в нем хранятся. Если вы создали массив int, то каждый элемент - это переменная типа int, соответственно вы можете изменять ее значение и использовать во всех действиях программы.

11.4 Обработка массивов

Вернемся к исходной цели изучения массивов - обработка большого количества данных. Допустим у нас будет создан массив на 365 элементов, в каждой ячейке которого лежит целое число. Давайте напишем программу, которая считает среднее значение для этих элементов. Для этого нам конечно же понадобятся циклы.

```
int [] data = new int[365];
// заполняем массив данными
```

```
// ....
int sum = 0;
for(int i = 0; i < data.length; i++) {
        sum = sum + data[i];
}
System.out.println(sum);
```

В качестве верхней границы цикла мы используем количество элементов в массиве. Для этого у каждого массива, есть свойство length, в котором собственно лежит число элементов в структуре.
Эта конструкция крайне важная для общего понимая происходящего. По сути обрабатывая массив, мы перебираем не сами элементы, а ИНДЕКСЫ элементов. Другими словами написанный цикл перебирает индексы от 0 до data.length - 1 (длина массива - 1) и затем уже внутри цикла обращается к каждому элементу по очереди с помощь data[i].
Таким образом, написанный цикл позволяет пройти, перебрать, просмотреть все элементы массива и выполнить с ними какое-либо действие.
При написании внутренней части таких циклов, важно понимать, что на каждом повторении вы работаете не сразу со всеми элементами, а с конкретной ячейкой data[i]. Это упростит общее понимание данной конструкции.

11.5 Многомерные массивы

Зачастую в программе нужно хранить данные не только в виде последовательной цепочки элементов (обычный массив), но и в виде "таблиц"или других многомерных объектов. Для этого мы можем использовать те же массивы, но в качестве элементов будут выступать уже не простые числа или строки, а целые массивы. Пример создания двумерного массива:

```
int [][] data = new int[4][5];
```

Описанная таким образом переменная создает двумерный массив, состоящий из 4 строк по 5 элементов в каждой строке. Обращаю внимание, что понятие "строк"массива является лишь устной договоренностью, так чаще всего считают при работе с ними. После создания во всех ячейках лежат значения по умолчанию, не забывайте об этом.
Как работать с такой структурой? Точно так же, как с обычным массив, только теперь у нас есть не один индекс, а два - индекс строки и стоблца. Для того, чтобы обратиться к элементу массива, нужно написать его имя и замет в первый квадратных скобках номер строки, во вторых квадратных скобках номер столбца.

```
int [][] data = new int[4][5];
data[1][2] = 123;
data[1][3] = 456;
data[1][4] = data[1][2] + data[1][3];
```

Дальнейшая работа с элементами опять же никак не отличается от работы с обычной переменной такого типа. То есть обратившись к элементу data[1][2] вы получаете просто объект типа int и дальше можете выполнять с ними все доступные действия. Как обрабатывать двумерные массивы? Для того, чтобы перебрать все элементы обычного одномерного массива, мы пишем цикл, перебирающий все индексы элементов. Для двумерного массива нам понадобится уже два цикла - один перебирает индексы строк, второй перебирает индексы столбцов.

```
int [][] data = new int[4][5];
for(int i = 0; i < 4; i++) { // перебираем индексы строк
    for(int j = 0; j < 5; j++) { // перебираем индексы столбцов
        data[i][j] = i + j;
    }
}
```

Обычно при переборе всех элементов массива мы получали их количества с помощью data.length. Сейчас это значение будет возвращать количество строк. Как получить длину строки? data[0] - это первая строка (массив), поэтому если мы напишем data[0].length, мы получим количество столбцов в массиве. И код написанный выше будет выглядеть так:

```
int [][] data = new int[4][5];
for(int i = 0; i < data.length; i++) { // перебираем индексы строк
    for(int j = 0; j < data[i].length; j++) { // перебираем индексы столбцов
        data[i][j] = i + j;
    }
}
```

12 Методы внутри класса

Зачастую программа, выполняющая какую-либо работу разростается достаточно быстро и содержит большое количество кода. Кроме того, отдельные действия (считывание данных, обработка полученной строки или массива) могут повторяться в программе много раз. Все это приводит к тому, что код становится неаккуратным, трудным для понимания и отладки.
Для того, чтобы бороться с этой проблемой, в языке есть возможность создавать методы. Сам по себе метод представляет собой косок кода, вынесенный в отдельную часть вне метода main(). Давайте рассмотрим общую схему создания метода на текущем этапе:

```
class Main {
    public static void main(String [] args) {
        // код главного метода
    }
```

```
    public static ВОЗВРАЩАЕМЫЙ_ТИП ИМЯ(АРГУМЕНТЫ) {
        // код вспомогательного метода
    }
}
```

Вначале нужно написать public static для того, чтобы метод можно было использовать в методе main(), далее мы указываем тип значения, которое возвращает функция. Например, фукнция проверяющая, что число является четным, может возвращать тип boolean.
Далее идет имя функции. Можно указывать любые названия, но обычно функции называются с маленькое буквы в стиле camelCase.
После имени в обязательном порядке идут круглые скобки, в которых при необходимости мы указываем аргументы через запятую.
Рассмотрим функцию, проверяющую четность числа:

```
public static void main(String [] args) {
    boolean b = isEven(123);
    System.out.println(b);
}

public static boolean isEven(int number) {
    boolean result = false;
    if(number % 2 == 0) {
        result = true;
    }
    return result;
}
```

В данном фрагменте кода реализована функция isEven, которая принимает число, проверяет, является ли оно четным и возвращает в зависимости от результата true или false.
Важно понимать, что если вы указали возвращаемый тип boolean, то внутри вашего метода обязательно должна быть строка return РЕЗУЛЬТАТ;. По факту эта команда присваивает результат выполнения функции.

Другой важный момент - агрументы. Когда вы указываете в аргументе число (int number), то это означает, что при выполнении функции у вас уже есть переменная number, в которой лежит какое-то число. Указывать значения аргументов функции внутри самого метода как правило не нужно.

Теперь давайте разберемся с тем, как использовать написанную функцию. Во фрагменте кода выше в методе main() есть строка *boolean b = isEven(123);*. В этой строке происходит вызов функции isEven. Вызвать функцию - значит написать имя функции и далее круглые скобка (либо пустые, если аргументо нет, либо со списком значений через запятую). При выполнении этой строки (что удобно посмотреть

в режиме отладки), мы переходим внутрь метода isEven, описанного после main(), выполняем все команды, находящиеся в нём и возвращаем значение false. В текущем вариант при выполнении isEven аргумент number будет равен 123, так как при вызове функции мы указали это значение.
После завершения выполнения вспомогательного куска кода, вместо isEven(123) подставляется результат выполнения функции. Формально строка превращается для Java в *boolean b = false;*.

13 ООП

При проектирование любой программы полезно придерживаться некоторых правил и рекомендаций по построению кода. Существует несколько подходов к написанию программ, одним из которых является объектно ориентированное программированию.

В основе этого подхода лежит понятие объекта. Это можно объяснить тем, что в большинстве программ мы стараемся отразить некоторую модель, созданную мысленно либо взятую из реальной жизни. Описание любого процесса - это описание сущностей, участвующих в работе, и выполняемых ими действий. Например, если будет задача запрограммировать конвейер, принимающий объект и упаковывающий его в коробку, то можно будет выделить следующие сущности: конвейер, объект для упаковки, упаковка. Кроме этого у конвейера будет описано действие "упаковать".

Продолжая разговор об объектах нужно сказать, что кроме действий у каждой сущности есть набор свойств. Например, у машины есть численные значения габаритов, цвета, марки, модели двигатели и так далее. Все это присуще любому объекту.

Для описания объектов в Java существуют классы. Тот самый public class Main на самом деле является отдельной сущностью и в нашем случае описывал объект, выполняющий некоторые преобразования с данными. Принято выносить каждый класс в отдельный файл для избежания путаницы. Итак, давайте создадим класс для описания машины:

```
public class Car {

}
```

Вот так выглядит пустой класс. Важно понимать, что когда мы создали свой класс, мы по сути добавили новый тип данных, который можно использовать в других частях программы. Для создания переменной типа Car нужно написать следующую строку:

```
Car audi = new Car();
```

Здесь мы создаем переменную с именем audi, которой присваиваем новый экземляр класса Car. Это означает, что по описанному классу создается нужный нам объект. Давайте наполним класс содержимым. Для описания свойств объекта существуют поля класса. Они представляют собой переменные, находящиеся внутри объекта.

```
public class Car {
    int currentSpeed;
    int maxSpeed;
    String model;
}
```

Теперь внутри каждого объекта типа Car будет находится три переменные, описанные выше. Как обращаться к этим переменным:

```
Car audi = new Car();
audi.currentSpeed = 10;
audi.maxSpeed = 200;
audi.model = "RS 6";
```

На самом деле длина массива length также является полем класса массив и обращались мы к ней через точку. То есть для того, чтобы обратиться к переменной, находящейся внутри класса, нужно написать имя переменной нужного типа и затем через точку имя поля класса. Далее с этим значением вы можете выполнять все действия, доступные для данного типа данных.

```
Car audi = new Car();
audi.model = "RS 6";
System.out.println(audi.model.length());
```

Для полного описания объекта нам необходимо добавить действия внутрь класса. Для этого мы можем описать методы. Например, метод увеличивающий текущую скорость машины:

```
public class Car {
    int currentSpeed;
    int maxSpeed;
    String model;

    public void increaseSpeed(int value) {
        currentSpeed += value;
    }
}
```

Теперь мы можем вызывать этот метод у любого объекта типа Car.

```
Car audi = new Car();
System.out.println(audi.currentSpeed);
```

```
audi.increaseSpeed(50);
System.out.println(audi.currentSpeed);

>> 0
>> 50
```

Кроме обычных функций есть специальный метод - конструктор. Он вызывается сразу после создания объекта и необходим для того, чтобы инициализировать объект начальными данными. Типичный конструктор выглядит следующим образом:

```
public class Car {
    int currentSpeed;
    int maxSpeed;
    String model;

    public Car(int curSpeed, int maximumSpeed, String m) {
        currentSpeed = curSpeed;
        maxSpeed = maximumSpeed;
        model = m;
    }

    public void increaseSpeed(int value) {
        currentSpeed += value;
    }
}
```

Теперь при создании объекта мы можем сразу же указать начальные значения внутренних переменных, что намного удобнее, чем вручную задавать значения полей объекта.

```
Car audi = new Car(0, 200, "RS 8");
System.out.println(audi.maxSpeed);
System.out.println(audi.model);

>> 200
>> RS 8
```

Конструкторов может быть любое количество, в зависимости от того, какие данные мы хотим инициализировать. Обратите внимание, что конструкторы считаются одинаковыми, если совпадает их набор агрументов по типам.

```
public class Car {
    int currentSpeed;
    int maxSpeed;
    String model;
```

```
    public Car(String m) {
        model = m;
    }

    public Car(int maximumSpeed, String m) {
        currentSpeed = 0;
        maxSpeed = maximumSpeed;
        model = m;
    }

    public Car(int curSpeed, int maximumSpeed, String m) {
        currentSpeed = curSpeed;
        maxSpeed = maximumSpeed;
        model = m;
    }

    public void increaseSpeed(int value) {
        currentSpeed += value;
    }
}

Car audi = new Car(0, 200, "RS 8");
System.out.println(audi.maxSpeed);
System.out.println(audi.model);

Car bmw = new Car(220, "M5");
System.out.println(bmw.maxSpeed);
System.out.println(audi.model);

Car mercedes = new Car("E33");
System.out.println(bmw.maxSpeed);
System.out.println(audi.model);

>> 200
>> RS 8
>> 220
>> M5
>> 0
>> E33
```