

Базовые понятия Java

Базовые понятия Java

- Типы данных
- Переменные
- Строки
- Печать результатов в консоль
- Математические и логические операции
- Оператор if
- Цикл for
- Классы + **Основы ООП**
- Функции
- Комментарии
- Импорты



Общие моменты

- Команды выполняются последовательно сверху вниз
- Важен **регистр** символов
- Каждая строка заканчивается ;
- После } точка с запятой не ставится
- Авто-форматирование: **Ctrl + Alt + L**
- Авто-удаление лишних импортов: Ctrl + Alt + O

Типы данных

- **Тип данных** – это вид данных с одинаковой структурой и поведением
- **int, long, short, byte** – целые числа
- **double, float** – дробные (вещественные) числа
- **char** – СИМВОЛ
- **boolean** – логический тип (выдает true/false)
- **String** – строки
- и др.
- Для целых чисел чаще всего используют **int**, для вещественных чисел **double**. В тестах чаще всего используется тип **String**

Переменные

- **Переменная** – ячейка памяти компьютера, которая может хранить в себе одно значение заданного типа
- Нужны, чтобы запоминать данные, а потом их использовать. Чтобы использовать переменную, нужно сначала ее объявить
- Переменные в Java имеют **тип** и **название** , а также часто содержат в себе **значение**
- Пример:
- `int a = 3 * 30;` `// переменная целого типа int a = 90`
`int b = a + 5;` `// переменная целого типа int b = 95`
`System.out.println(a + b);` `// печать в консоль 185`

Переменные

- Можно сначала объявить переменную, а лишь затем задать ей значение

- `int a;` `// переменная целого типа int`

`int b = a + 5;` `// ошибка компиляции –`
 `// переменной a еще не присвоено`
 `// значение`

`a = 3;` `// все ОК`

Имена переменных

- Важен регистр символов: `variable`, `Variable` и `VARIABLE` – это разные имена переменных
- **Допустимые имена:**
- Первый символ – буква, либо символ подчеркивания `_`
- Последующие символы – буквы, знак подчеркивания или цифры
- Языком допускается использовать русские символы, но этого лучше не делать
- В Java принято давать переменным имена в соответствии с нотацией «верблюды»: `maxNumber`, `helloWorld`

Оператор присваивания

- Переменная в левой части оператора получает значение результата выражения в правой части
- Совершенно нормальное дело:

```
int x = 5;
```

```
x = x + 6;
```

Если переменная используется в левой части от присваивания, то это значит положить в неё

В остальных случаях – это получение копии значения переменной

- Сначала вычисляется правая часть: из переменной `x` вытаскивается 5 и прибавляется к литералу 6, получается 11
- Затем переменной `x` присваивается новое значение 11

Строки

- Строки имеют тип `String`
`String s = "Hello";`
- Значения строк заключаются в двойные кавычки
- Могут содержать спецсимволы: `\n` – перевод строки (Enter), `\t` – табуляция
- **Конкатенация** – добавление второй строки в конец первой:
- `String s1 = "Hello " + "world"; // Hello world`
- Добавление значения к строке:
- `String s1 = "Hello " + 300; // Hello 300`

Печать результатов

- Предположим, вы посчитали результат некоторого выражения:
- `int x = 8;`
`int y = x * 2;`
- Далее вы хотите распечатать пользователю **y**
- Вот так делать **неправильно**, выведется просто 16, это непонятно:
- `System.out.println(y);` `// 16`
- **Правильно** с пояснением:
- `System.out.println("y = " + y);` `// y = 16`

Длина строки

- `String s = "Hello";`
`int a = s.length(); // 5`
- Length – с англ. длина

Математические операции

- Число считается вещественным, если в нем есть точка (1.0), либо символ экспоненты (1e34)
- Арифметические операторы: + - * / %
- % - остаток от целочисленного деления
`int r = 25 % 7; // 4`
- Приоритеты операторов – как в математике, можно использовать скобки

Логические операции

- **Логическим высказыванием** называется некоторое предложение, про которое можно сказать **ИСТИННО** оно или **ложно**
- Логический тип **boolean** имеет два возможных значения – **true** (истина) и **false** (ложь)
- Пример:
- `boolean a = true;`
`boolean b = false;`

Логические выражения

Выражение	Символ	Пример
Проверка на равенство	==	a == b
Проверка на неравенство	!=	a != b
Строгое сравнение	> и <	a > b a < b
Нестрогое сравнение	>= и <=	a >= b a <= b

- Пример:

```
double a = 6.5;
```

```
System.out.println(a > 5); // true
```

Не нужно путать проверку на равенство == с оператором присваивания =

Логические связи

Название	Синонимы	Операторы	Примеры
Логическое И	Конъюнкция	&&	<code>a >= 5 && b == 3</code>
Логическое ИЛИ	Дизъюнкция		<code>3 < 5 b > 4</code>
Логическое НЕ	Отрицание	!	<code>!(a == 4)</code>

- Пример:

```
int a = 6;
```

```
int b = 4;
```

```
System.out.println(a >= 5 && b == 3); // false
```

```
System.out.println(3 < 5 || b > 4); // true
```

```
System.out.println(!(a == 4)); // true
```

- Оператор && имеет больший приоритет, чем ||

Проверка на равенство

- Переменные численных типов можно сравнивать при помощи ==
- Для сравнения строк в Java оператор == не подходит
- Нужно использовать команду equals
- `String s1 = "Hello";`
`String s2 = "Hello";`

```
System.out.println(s1.equals(s2)); // true
```


Проверка boolean на true/false

- Часто бывает нужно проверить `boolean` переменную на равенство `true` или `false`
- Так делать не принято :
- `if (x == true) { // код }`
- `if (x == false) { // код }`
- Нужно делать так:
- `if (x) { // x равно true }`
- `if (!x) { // x равно false }`

Чтение с консоли

- Для чтения с консоли используется тип `Scanner`
- Ключевое слово `new` создает новый объект типа `Scanner`
- В скобках передаем ему `System.in` – источник чтения с консоли: `Scanner scanner = new Scanner(System.in);`
- Далее пользуемся сканнером – читаем при помощи него числа и строки с консоли
- `int a = scanner.nextInt();` // прочитать целое число
`double b = scanner.nextDouble();` // прочитать double
`String s = scanner.nextLine();` // прочитать строку

Приглашение для ввода

- Очень важно при чтении с консоли выводить пользователю приглашение для ввода
- Это некоторый текст, по которому пользователь поймет, что именно ему нужно ввести
 - И вообще поймет сам факт, что нужно что-то ввести
- Предположим, просто написан такой код:
`double a = scanner.nextDouble();`
- Когда программа дойдет до этой строки, она приостановится
- Пользователь никак не поймет, что от него что-то требуется

Приглашение для ввода

- Поэтому правильно делать, например, так:
- `System.out.print("Введите число: ");`
`double a = scanner.nextDouble();`
- Теперь программа напечатает пользователю сообщение до начала ввода с консоли, и пользователь поймет, что нужно ввести данные
- И по сообщению поймет что именно нужно ввести
- Естественно, сообщение должно быть более информативным

**Печать в
консоль**

Печать в консоль

- `System.out.println(аргумент);`
- Печатает переданный аргумент, а потом печатает перевод строки (Enter)
- `System.out.print(аргумент);`
- Печатает переданный аргумент, но не вставляет перевод строки
- `System.out.println();`
- Печатает перевод строки

Печать в консоль

- `System.out.printf(formatString, arg1...)`
- Принимает **строку форматирования** первым аргументом, а затем через запятую указываются аргументы, которые будут вставлены в эту строку
- Пример:
- `int a = 7 * 5;`
`System.out.printf("Result = %d", a); // Result = 35`

Printf

- `int x = 35;`
`int y = 34;`
`System.out.printf("X = %d, Y = %d", x, y); // X = 35, Y = 34`
- Строка форматирования – строка, некоторые части которой имеют особый смысл
- Такие части начинаются с символа % и называются **спецификаторами формата**
- Аргументы, которые мы передаем в printf, подставляются вместо этих спецификаторов
- Например, вместо первого %d вставится значение `x = 35`, вместо второго %d – значение `y = 34`

Зачем нужен printf?

- Он позволяет формировать сложные строки из нескольких строк и переменных в читаемом виде
- Пример:
 - **Конкатенация:**
 - `System.out.println("X = " + x + ", y = " + y);`
 - **Printf:**
 - `System.out.printf("X = %f, y = %f", x, y);`

Зачем нужен printf?

- Он позволяет вывести данные в разном формате
- Например, у вещественного числа может быть много знаков после запятой
- `double x = 33.3333333;`
- Можно указать сколько знаков вывести, нужно ли вывести в экспоненциальной форме, нужно ли добавлять пробелы для форматирования и т.д.
- `System.out.printf("%f", x);` `// 33.333333`
`System.out.printf("%e", x);` `// 3.333333e+01`
`System.out.printf("%.2f", x);` `// 33.33`

Спецификаторы

- Для каждого типа данных используется свой спецификатор, начинающийся с %

Спецификатор	Тип	Пример
%d	Десятичное целое число	159
%x	Шестнадцатеричное целое число	9f
%o	Восьмеричное целое число	237
%f	Вещественное число с точкой	15.9
%e	Вещественное число в экспоненциальной форме	1.59e+01
%s	Строка	Hello
%%	Символ процента	%
%n	Перевод строки	

Примеры использования printf

- Примеры:
- `int result = 50;`
`System.out.printf("Result = %d%%", result);`
`// Result = 50%`
- `String s = "Hello";`
`System.out.printf("Result = %s.", s);`
`// Result = Hello.`
- `System.out.printf("First line%nSecond line");`
`// First line`
`// Second line`

Практика 1

- Написать программу, которая просит ввести ваше имя, а затем выводит в консоль приветствие. Для чтения использовать `nextLine()` `Scanner`'а

- Пример:

Введите ваше имя:

// это печатает программа

Елена

// это ввожу я

Привет, Елена!

// это печатает программа

Условный оператор if

- Нужен для выполнения кода по некоторому условию **(условный оператор)**
- `if` (логическое выражение) действие
- `if (a > 5) {
 System.out.println("Внутри if");
}`
- Код внутри блока `if` выполняется только если условие в круглых скобках истинно (равно `true`)
- Следует всегда использовать блок в фигурных скобках, даже для одной команды, чтобы не ошибиться

if-else

- `if` может содержать необязательную часть `else` (иначе) – какой код выполнить, если условие в `if` является ложным
- `if` (логическое выражение)
 { действие 1 }
`else`
 { действие 2 }
- `if (a > 5) {`
 `System.out.println("a > 5");`
`} else {`
 `System.out.println("a <= 5");`
`}`
- Условие вычисляется 1 раз, и если оно оказалось `true`, то мы заходим в ветку `if`, иначе – в ветку `else`

Вложенный if и цепочки if'ов

- Условные операторы **if** можно вкладывать друг в друга любым образом
- Если нужно больше вариантов, чем 2, то можно составить цепочку **if**'ов
- **if** (a > 5) {
 System.out.println("a > 5");
} **else if** (a == 5) {
 System.out.println("a = 5");
} **else** {
 System.out.println("a < 5");
}
- Ветка **else**, должна быть последней, но ее может и не быть

Практика 2

- Написать программу, которая просит ввести ваше имя, а затем выводит в консоль приветствие. Для чтения использовать `nextLine()` `Scanner`'а
- Добавьте проверку на то, что пользователь ввел «пустое» имя
- Пример:

Введите ваше имя: `// это печатает программа`

`// я не ввожу ничего и нажимаю Enter`

Привет, неизвестный! `// это печатает программа`

Циклы

Циклы

- Позволяют выполнять один и тот же блок кода, пока выполняется некоторое условие
- В Java существует 4 вида циклов:
 - `while`
 - `do-while`
 - 2 разновидности цикла `for`

Цикл while

- **while** (логическое выражение)
инструкция
- Как работает:
 - Шаг 1. Вычисляется значение логического выражения (**условие цикла**)
 - Шаг 2. Если оно ложно, то цикл завершается
 - Шаг 3. Если оно истинно, то выполняется **тело цикла (инструкция)**. Затем переход на шаг 1

Сумма чисел от 0 до 9

- Часто цикл используют, чтобы пройти по диапазону чисел

- `int i = 0;` `// счетчик цикла`
`int sum = 0;`

```
while (i <= 9) {  
    sum = sum + i;  
    ++i; // i = i + 1  
}
```

Названия счетчиков

- Для названий переменных-счетчиков цикла часто используют короткие имена, состоящие из одной буквы
- Общепринято называть переменную-счетчик буквой *i*
- Если имя *i* уже занято, то использовать *j, k, m, n* и так далее
- Букву *l* пропускают, т.к. она похожа на 1

Цикл while

- Тело цикла может не выполниться **ни разу**, если условие сразу было ложным
- Если условие всегда истинно, то цикл выполняется бесконечно. Это называется **зацикливанием** и обычно является ошибкой
- В примере зацикливание может произойти если забыть сделать `++i`;

Цикл do-while

- `do`
инструкция
`while` (логическое выражение);
- Как всегда, инструкция – 1 команда или блок кода в фигурных скобках
- Как работает:
 - Шаг 1. Выполняется тело цикла
 - Шаг 2. Проверятся условие. Если истинно, то возвращаемся на шаг 1. Если ложно, то конец цикла

Цикл do-while: сумма чисел от 0 до 100

- ```
int i = 0;
int sum = 0;

do {
 sum += i;
 ++i;
} while (i <= 100);
```

# Цикл do-while

- Отличие от `while`: тело цикла `do-while` всегда выполняется хотя бы 1 раз, т.к. первая проверка условия происходит после 1 итерации цикла

# Цикл for

- Цикл предназначен для прохода по диапазону чисел
- Используется счетчик цикла
- Синтаксис:
- `for` (инициализация; лог. выражение; модификация) {  
    // код, который выполняется для каждой итерации цикла  
}
- **Инициализация** – это установка начального значения счетчика
- **Логическое выражение** – это условие продолжения цикла. Пока оно true, цикл будет выполняться
- **Модификация** – это изменение счетчика после каждой итерации

# Пример цикла for

- ```
for (int i = 1; i <= 10; i++) {  
    System.out.print (i + " ");  
}
```
- **Инициализация** – создается целый счетчик *i* с начальным значением 1
- **Логическое выражение** – цикл будет продолжаться пока *i* <= 10
- **Модификация** – после каждой итерации значений счетчика будет увеличиваться на 1 (*i*++). После увеличения снова проверяется условие
- Результат выполнения цикла:
1 2 3 4 5 6 7 8 9 10

Практика 3

- Написать программу, которая просит ввести ваше имя, а затем выводит в консоль приветствие. Для чтения использовать `nextLine()` `Scanner`'а
- Добавьте проверку на то, что пользователь ввел «пустое» имя. Пусть программа запрашивает имя до тех пор, пока пользователь не введет не-пустое имя

Введите ваше имя: `// это печатает программа`

`// я не ввожу ничего и нажимаю Enter`

Введите ваше имя: `// это печатает программа`

Елена `// это ввожу я`

Привет, Елена! `// это печатает программа`

Классы

- Весь код программ на Java находится внутри **классов**
- Класс – это отдельный файл с именем в «верблюжьей нотации», с большой буквы: HelloWorld, CircleCalculation
- В классе есть поля и методы (рассмотрим подробнее на практике)
- **Объявление класса:**
- `public class HelloWorld { // код }`
- `public` – это модификатор доступа, он означает, что данный класс будет доступен во всем проекте
- Далее идет ключевое слово `class`, а после него – имя класса. Оно должно полностью совпадать с именем файла
- Затем идет **блок кода** в фигурных скобках `{ }`

Функции

- Функция – это часть программы, к которой можно обращаться как к одной команде
- У функции могут быть входные и выходные данные, а может их не быть
- Функция без выходных данных – void

До преобразования в функцию

- Пример – вычисление среднего арифметического для двух целых чисел
- До преобразования в функцию:

```
public static void main(String[] args) {  
    int a = 5;  
    int b = 6;  
  
    double average = (double) (a + b) / 2;  
    System.out.println("Average = " + average);  
}
```


После преобразования в функцию

```
public static void main(String[] args) {  
    int a = 5;  
    int b = 6;  
  
    double average = getAverage(a, b);  
    System.out.println("Average = " + average);  
}
```

Вызов функции

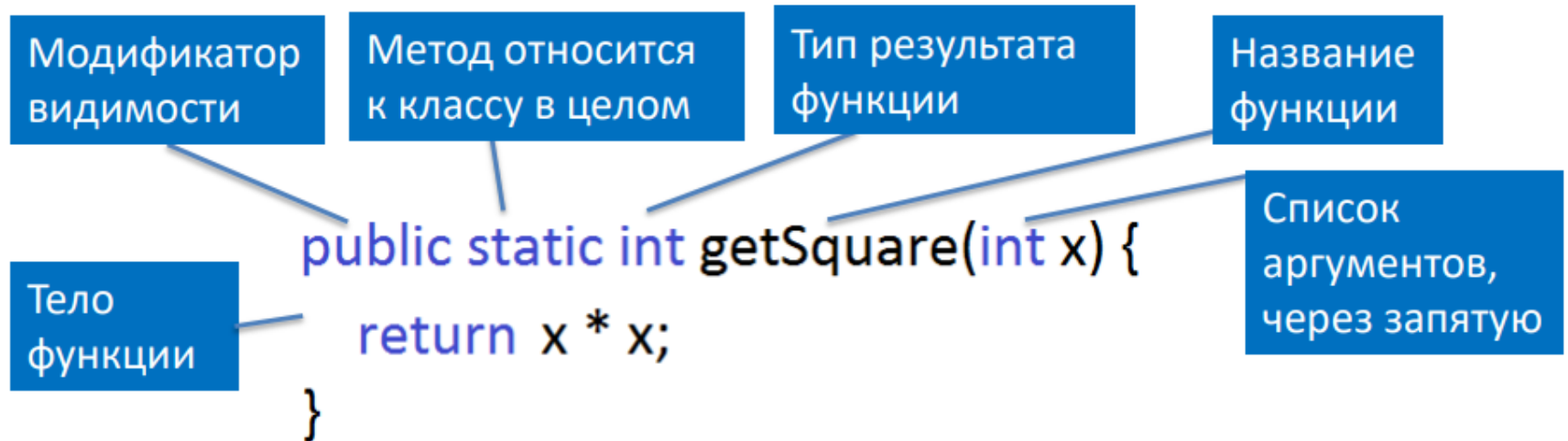


```
public static double getAverage(int a, int b){  
    return (double) (a + b) / 2;  
}
```

Объявление функции



Объявление функции



- У функции есть название, тип возвращаемого значения (если есть), список аргументов в формате [тип] [имя] (если есть), тело функции (выполняемый код)
- Оператор `return` завершает функцию и выдает результат
- Если функция `void`, то оператора `return` не будет

Вызов функции

- Вызов функции из того же класса:

```
int y = getSquare(3);           //9
```

- Вызов функции другого класса:

```
int y = Main.getSquare(3);      //9
```

- При вызове функции сначала вычисляются аргументы, а затем выполняется функция
- После выполнения функции исполнение программы возвращается к тому месту, откуда была вызвана функция

Комментарии

- **Комментарии** – это текст внутри программы, который отбрасывается компилятором
- Используются для пояснения, для временно отключения кода, в качестве заметок TODO, для документирования кода
- Однострочные комментарии:
`int i = 4; // это комментарий`
- Многострочные комментарии:
`int a = 4; /* первая строка комментария
 Вторая строка комментария */`
- TODO заметки: `//TODO: реализовать метод позже`

Импорт классов

- Чтобы использовать некоторые классы и их методы, нужно импортировать эти классы в наш класс



- Можно также обращаться по длинному имени, но это неудобно:
io.github.bonigarcia.wdm.WebDriverManager.chromedriver().setup();

Импорт классов в IDEA

- IDEA позволяет автоматически импортировать класс
- Для этого нужно:
 1. Поставить курсор на имя класса, выделенное красным
 2. Нажать Alt + Enter
 3. В выпадающем списке выбрать **Import class**
 4. Если там несколько вариантов, то надо выбрать нужный

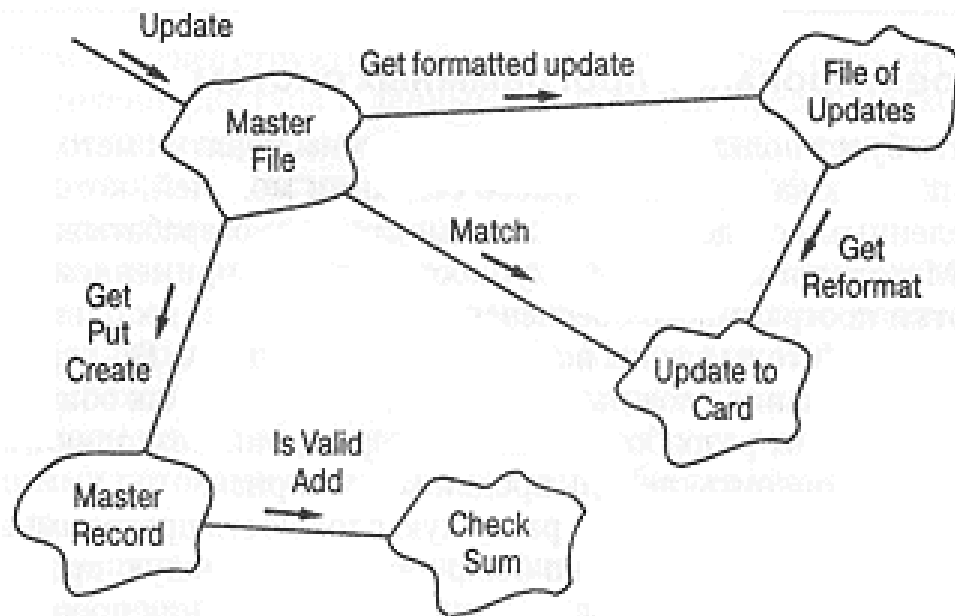
Основы ООП

Объектно-ориентированное программирование

- ООП - одна из самых распространенных «промышленных» парадигм программирования
- **Парадигма программирования** - это совокупность идей и понятий, определяющих стиль написания компьютерных программ
- Программы в ООП пишутся в терминах **объектов** и **классов**

Объектно-ориентированное программирование

- Java является **объектно-ориентированным языком**
- Это означает что программа на Java представляет собой набор взаимодействующих **объектов**



Объект

- **Объект** – это некоторая конкретная сущность (предмет, явление)
- **Примеры:**
Стол, кошка, гроза, ноутбук, человек и т.д.
- Каждый объект обладает **состоянием, поведением и уникальностью**

Состояние объекта

- **Состояние** – это набор значений характеристик объекта в данный момент времени
- В программировании состояние объектов задается при помощи переменных-полей
- **Пример для ноутбука:**
 - Высота, длина, ширина (вещественные числа)
 - Процент заряда (целое число)
 - Название модели (строка)
 - Заряжается сейчас или нет (**boolean**) и т.д.
- Состояние может меняться под внешним воздействием, либо сам объект может менять свое состояние
- Например, со временем процент заряда падает



Поведение объекта

- **Поведение** – это действия, которые может совершать объект и как объект может реагировать на воздействие со стороны других объектов
- Пример для ноутбука – его можно поставить на зарядку, и тогда процент будет расти
- Или выключатель – его можно включить или выключить
- В программировании поведение объекта задается при помощи функций-методов



Уникальность объекта

- **Уникальность** объекта – это то, что отличает его от других объектов
- Например, каждый человек уникален
- Или есть два одинаковых стула, но это все равно два стула, а не один. То есть стулья уникальны – это 2 отдельных объекта
- В программировании уникальность задается расположением объекта в памяти компьютера



Классы

- **Класс** – это совокупность всех **объектов** с одинаковой структурой и поведением
- Проще говоря **класс** – это вид одинаковых объектов
- Каждый объект обязательно **принадлежит** некоторому **классу**
- Если объект принадлежит некоторому классу, то говорят, что он является **экземпляром класса**
- То есть **объект** – это **экземпляр класса**

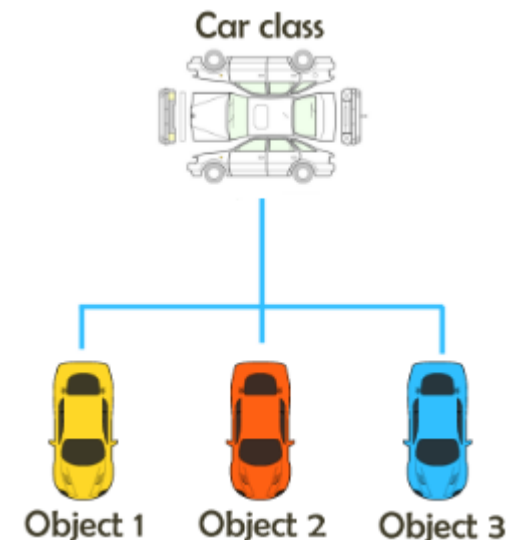
Пример класса

- Пусть, у нас есть кошка **Мурка**
- Она является объектом **класса Кошка**
- Все кошки (то есть объекты класса **Кошка**) устроены и ведут себя схожим образом
- У всех них есть свое состояние – цвет, положение в пространстве, размеры и др.
- У них есть свое поведение – они могут ходить, бегать, реагировать на воздействия и т.д.



Классы

- В программировании мы описываем **классы**
- В классе мы описываем, что в нем есть – какие поля, каких типов, какие есть методы, пишем их код
- А потом создаем сколько нам нужно **объектов (экземпляров)** этих классов и работаем с ними
- Т.е. **класс** – это как бы чертеж, описание, по которому потом можно создавать объекты, а **объект** – конкретная деталь, сделанная по этому чертежу

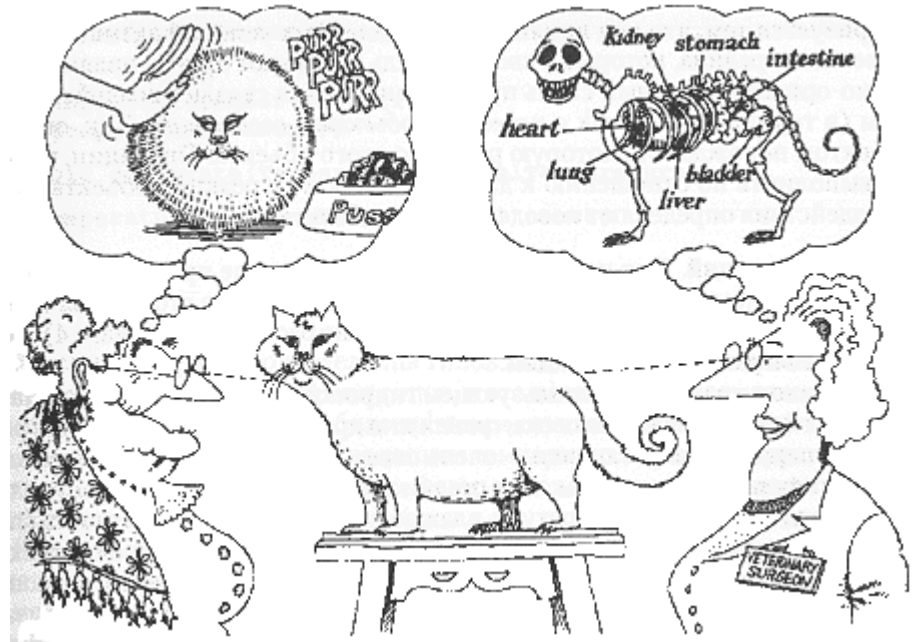


Принципы ООП

- В ООП программы пишутся в терминах классов и объектов
- Принципы ООП:
 - Абстракция
 - Инкапсуляция
 - Наследование
 - Полиморфизм

Абстракция

- **Абстракция** – выделение существенных характеристик объекта и существенного поведения, и отбрасывание несущественных характеристик и поведения
- Один и тот же объект реального мира для разных задач может быть представлен по-разному
- Важно выбирать абстракцию как можно более простую, но достаточную для задачи



Пример абстракции 1

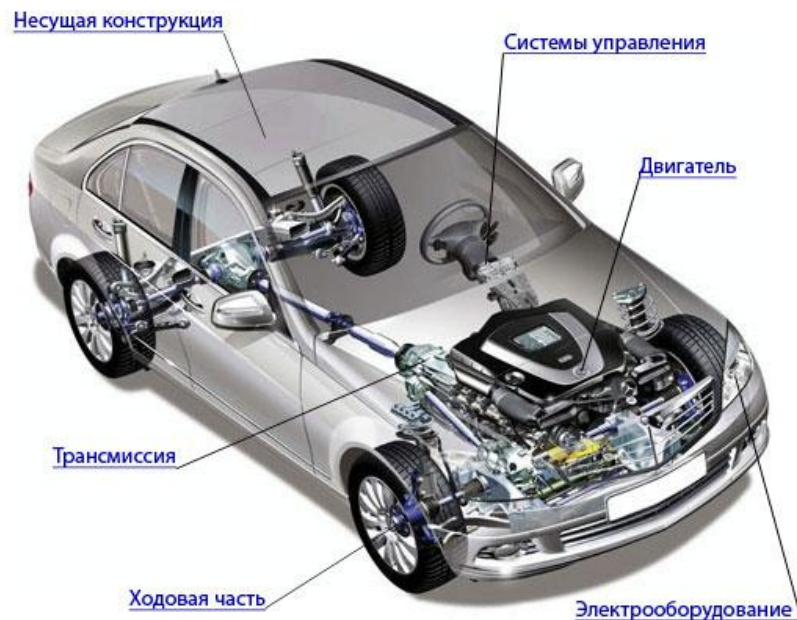
- Допустим, мы деканат и у нас такая задача – хранить список всех студентов
- Для этого выделим класс **Студент**
- Понятно, что студент является человеком, то есть у него есть пол, размеры, вес, возраст, у каждого человека много присущих ему черт
- Но для нашей задачи нам достаточно знать о студентах только ФИО, дату рождения, контактные данные, номер зачетки, номер группы

Инкапсуляция

- **Инкапсуляция** – механизм языка, позволяющий объединить данные и методы, работающие с этими данными, в единый объект, и скрыть детали реализации от пользователя кода
 - Здесь видно 2 роли инкапсуляции – объединение в объект и сокрытие реализации
- В соответствии с принципом инкапсуляции, мы хотим максимально скрыть от **пользователя кода (т.е. программиста)** реализацию классов, а дать им только возможность работать с публичным интерфейсом
- Это позволяет легко подменять одну реализацию другой (можно спокойно менять то, что скрыто)

Инкапсуляция

- Пример из жизни – устройство автомобиля
- Автомобиль состоит из огромного количества деталей, которые как-то друг с другом взаимодействуют
- Но чтобы водить автомобиль, не нужно знать многого – нужно только уметь работать с интерфейсом – руль, педали, коробка передач



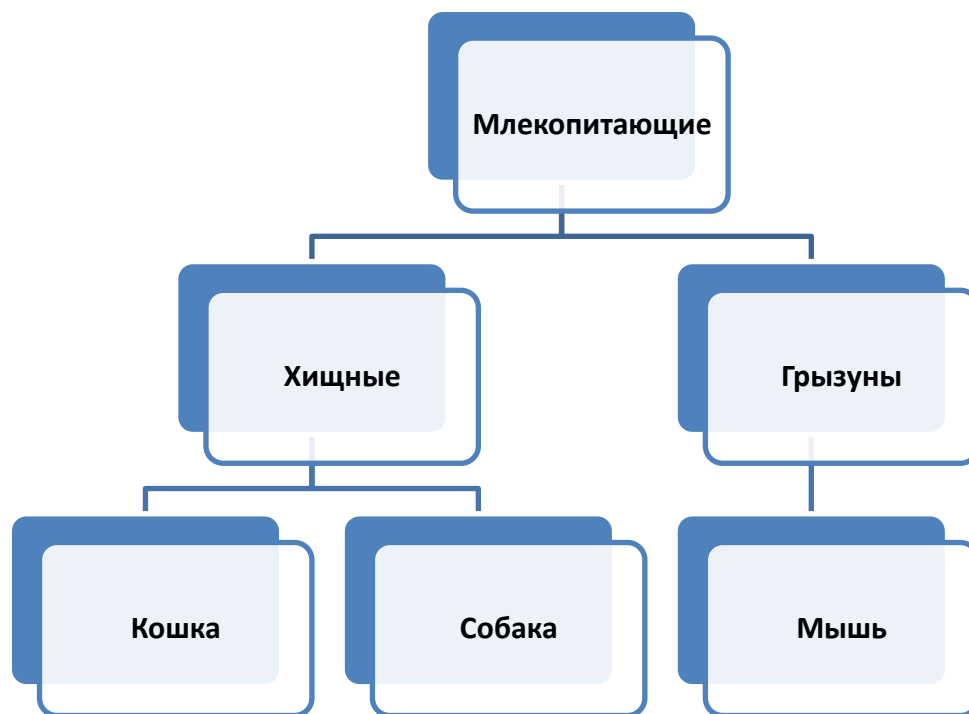
Инкапсуляция

- Так же и в коде – класс может быть очень сложно устроен, иметь вспомогательные функции и поля, но наружу предоставлять только функции, нужные другим
- И пользователем кода даже не нужно знать как этот класс устроен внутри
- Надо только знать как этим классом пользоваться
- Пример – класс `Scanner` в Java

Наследование

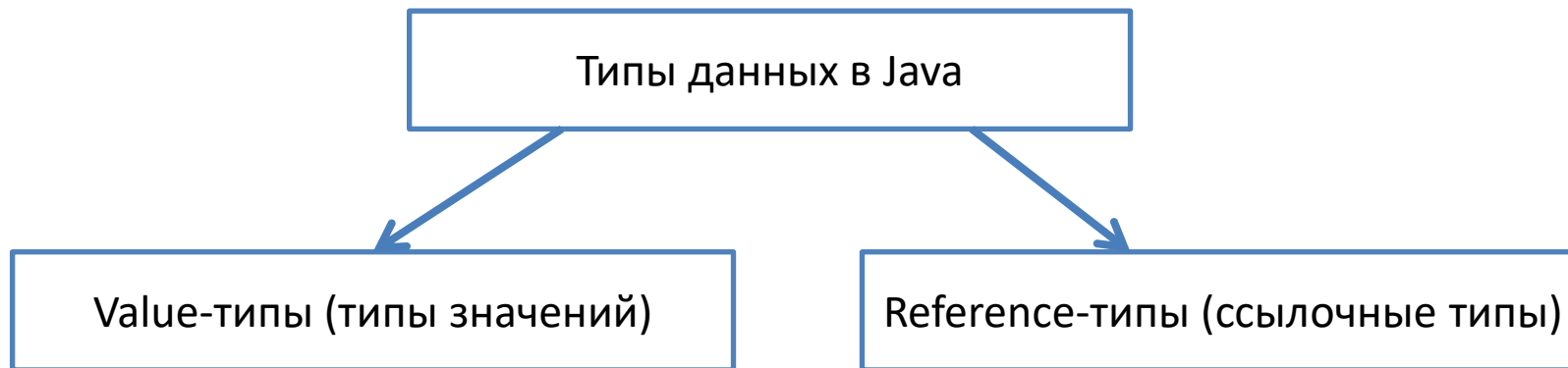
- Классы могут образовывать иерархию **наследования**
- Класс-наследник получает все свойства класса-родителя, может переопределять его черты, либо добавлять новые черты

Наследование



- Пример: биологическая классификация, рассмотрим кошку
- Так как кошка принадлежит классу млекопитающих, то она наследует свойства, присущие этому классу – кормление детей молоком
- Так как принадлежит классу хищников, то ест мясо и т.д.

Наследование



Числовые целые:

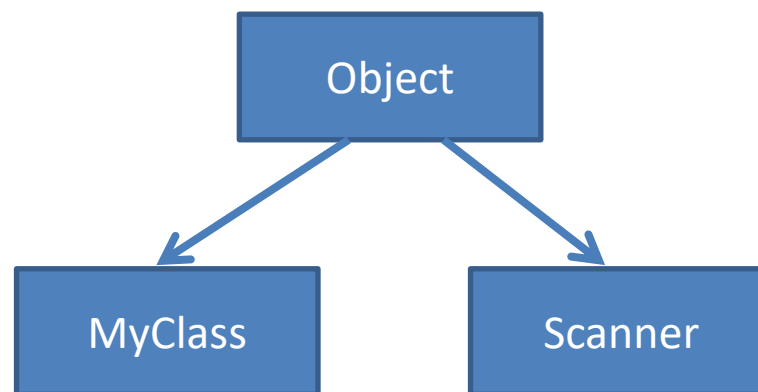
byte, short, int, long

Вещественные:

float, double

Логический: boolean

Символьный: char



**Все классы наследуются
от класса Object**

Синтаксис класа

Классы в Java

- `class Point {`
 `// члены класса: поля и методы`
}
- Каждый класс в Java может содержать поля (переменные) и методы (функции)
- Поля определяют структуру класса, а методы – поведение класса

Классы в Java

Имя класса

- class Point {
 private double x;
 private double y;

Поля (переменные)

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

Конструктор
(специальная функция),
вызываемая при
создании объекта

```
public void print() {  
    System.out.printf("(%.2f, %.2f)", x, y);  
}  
}
```

Метод (функция)

Порядок объявления членов класса

- ```
class Point {
 private double x;
 private double y;

 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
 public void print() {
 System.out.printf("(%f, %f)", x, y);
 }
}
```
- Порядок членов класса неважен, но обычно поля пишут вместе вверху, ниже пишут вместе конструкторы, а ниже - методы

# Классы в Java

- class Point {  
 private double x;  
 private double y;

- public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
}

- public void print() {  
 System.out.printf("(%.2f, %.2f)", x, y);  
}

Если имя поля конфликтует с именем переменной, то обращаемся к нему через **this**

**this** – ключевое слово, обозначающего текущий объект (для которого вызвана функция)

Можем всегда обращаться к полям и методам через **this**

# Конструкторы

- ```
class Main {  
    public static void main(String[] args) {  
        Point point = new Point(3, 2);  
        point.print();  
    }  
}
```
- **Конструктор** – специальная функция, которая позволяет создать и инициализировать экземпляр класса
- Конструктор нельзя вызвать явно, но он вызывается если создавать объект при помощи оператора **new**

Конструкторы

- ```
class Point {
 private double x;
 private double y;

 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
}
```
- При объявлении функции-конструктора не указывается возвращаемый тип. Конструктор ничего не возвращает
- Имя конструктора всегда совпадает с именем класса



# Конструкторы

- Класс может иметь несколько конструкторов

- ```
class Point {  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point() {  
    }  
}
```

Конструкторы

- Конструктор может не иметь аргументов
- Если при объявлении класса вообще не создавать конструктор, то компилятор Java сам генерирует **конструктор по умолчанию** (он без аргументов), который ничего не делает, а только вызывает конструктор класса-родителя
- Если в классе создать конструктор с аргументами, то компилятор не создает дополнительный конструктор без аргументов


Обращение к полям и методам классов

- ```
class Main {
 public static void main(String[] args) {
 Point point = new Point(3, 2);
 point.print();
 }
}
```
- Обращение к полям и методам объекта осуществляется через оператор точка
- Для членов класса могут иметься разные **права доступа**. Они задаются при объявлении класса при помощи **модификаторов видимости**, например, `public` и `private`. Еще есть `protected` и package видимость
- Если прав недостаточно, то обращение к члену класса приведет к ошибке компиляции

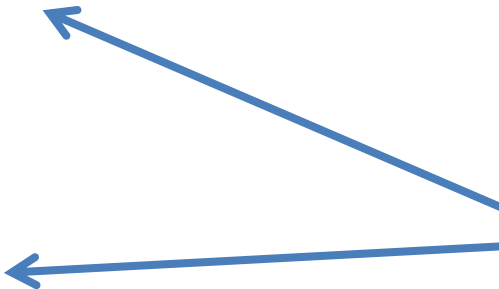
# Классы в Java

- class Point {  
 private double x;  
 private double y;  
  
 public Point(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
 public void print() {  
 System.out.printf("(%f, %f)", x, y);  
 }  
}

private члены класса  
видны только  
функциям внутри  
класса



public члены класса  
видны всем



# Обращение к полям и методам классов

- Модификаторы видимости и есть средство инкапсуляции в Java – они позволяют скрыть реализацию класса, а наружу выставлять только то, что должны использовать пользователи класса
- **Поля всегда должны быть `private`!! Если к ним все же нужен доступ, то для этого должны использоваться методы**

# Обращение к полям и методам классов

- Поля всегда должны быть **private**!! Если к ним все же нужен доступ, то для этого должны использоваться методы

- ```
class Point {  
    private double x;  
  
    public double getX() {  
        return x;  
    }  
  
    public void setX(double x) {  
        this.x = x;  
    }  
}
```

Соглашение именования –
методы для получения
значений должны
начинаться с **get**, а для
установки значения – с **set**

Методы **get**
называют геттерами,
методы **set** -
сеттерами

Не обязательно
иметь оба

Зачем поля private?

- **Достоинства:**

- Пользователи кода теперь не могут вмешиваться во внутренние дела класса, например, присвоить полю недопустимое значение
- Если имя поля изменится, или поле вообще исчезнет, то метод можно оставить с прежним именем, и тогда это изменение не затронет код, который использовал этот метод
- Метод может выполнять дополнительную работу: проверять корректность данных, сохранять сообщения в лог и т.д.

- **Недостатки:**

- Некоторое падение производительности т.к. получить значение поля дешевле, чем вызвать метод. Но производительность часто не важна

Нестатические члены класса

- ```
class Person {
 private String name;

 public Person(String name) {
 this.name = name;
 }

 public String getName() {
 return this.name;
 }
}
```

Поле name будет  
свое у каждого  
экземпляра класса  
Person

Методы класса  
могут работать с  
полями объекта. На  
сам объект можно  
сослаться при  
помощи слова this



# Статические члены

Статические члены относятся не к конкретным экземплярам, а к классу в целом

- class Person {  
 private String name;  
 public static final int MAX\_NAME\_LENGTH = 100;  
 public Person(String name) {  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
 public static String formatName() {  
 // код  
 }  
}

Статические поля существуют в единственном экземпляре

Чтобы работать со статическими членами не нужно создавать объекты класса

# Статические члены класса

- ```
class Person {  
    public static final int MAX_NAME_LENGTH = 100;  
    public static String formatName(String name) {  
        // возвращает имя с инициалами  
    }  
}
```
- Как обращаться к статическим методам и полям:
- ```
public static void main(String[] args) {
 int maxLength = Person.MAX_NAME_LENGTH;
 String formattedName
 = Person.formatName("Ivan Ivanovich");
}
```

# Статические члены класса

- Мы уже много работали со статическими методами и полями
- Например, мы использовали класс `Math` и его статические члены:
  - `Math.PI` – статическое поле-константа
  - `Math.random()` – получение случайного числа
  - `Math.abs(x)` – получение модуля числа и т.д.
- Такие классы, как `Math`, которые содержат только статические методы и статические константы, называются **классами-утилитами**

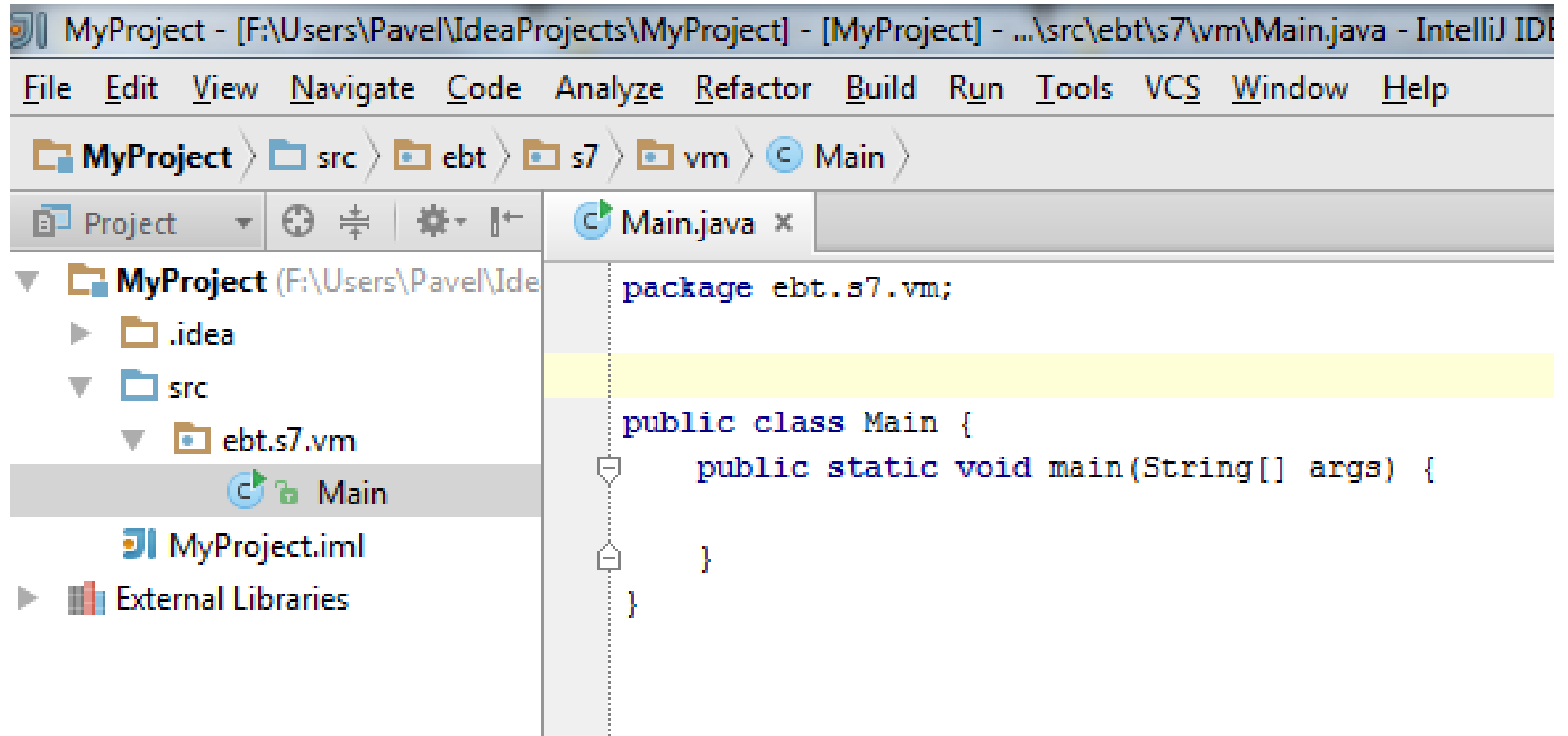
# Static и не-static

|                | He static                             | Static                                                                            |
|----------------|---------------------------------------|-----------------------------------------------------------------------------------|
| <b>В целом</b> | Относится к объекту                   | Относится к классу в целом                                                        |
| <b>Поля</b>    | Это поле будет у каждого объекта свое | Поле будет одно на весь класс. Оно хранится не в объектах, а отдельно в программе |
| <b>Методы</b>  | Метод вызывается только от объекта    | Метод вызывается от класса в целом                                                |

# Структура программ на Java

- Программы на Java обычно состоят из многих файлов
- В каждом файле находится один или более классов
- Классы можно группировать по так называемым **пакетам**
- В них стараются помещать близкие друг к другу типы
- Например, в один пакет можно поместить классы GUI – графического интерфейса, а в другой пакет – классы логики программы

# Пакеты



# Пакеты

- `package ru.academits.java;`
- Пакеты могут вкладываться друг в друга. Запись **ru.academits.java** означает что есть пакет **ru**, в нем есть вложенный пакет **academits**, а в нем есть вложенный пакет **java**
- Структура папок проекта должна повторять структуру пакетов. В противном случае будет ошибка компиляции
- Класс **Main** находится в пакете **ru.academits.java**. Это означает, что в проекте на верхнем уровне должна быть папка **ru** (имя совпадает с именем пакета), внутри неё – папка **academits**, внутри неё – папка **java**, внутри неё – файл **Main.java**, в котором обязательно есть класс с именем **Main** и в этом файле указан `package ru.academits.java.`

# Зачем нужны пакеты?

- Пакеты позволяют:
  - лучше структурировать файлы проекта
  - не делать вспомогательные классы доступными вне пакета, внутри которого они объявлены. То есть пакеты также являются средством инкапсуляции
  - избежать конфликтов имен. Благодаря пакетам можно давать разным классам одинаковые имена, если эти классы лежат в разных пакетах
- В Java именем класса является не просто имя, которое мы указываем при объявлении класса, а имя пакета + имя класса
- Наш класс **Main** на самом деле: **ru.academits.java.Main**



# Практика 4. Основы ООП

- Создать класс Person
- В нём: объявить три строковых поля name, middleName, familyName
- Описать конструктор, при помощи которого заполняются поля
- Реализовать геттеры и сеттеры для полей
- Добавить метод toString
- После этого написать небольшую программу с использованием этого класса

# Задача на дом «Среднее арифметическое»

- Написать программу, вычисляющую среднее арифметическое чисел из некоторого диапазона чисел (например, от 3 до 17)
- Концы диапазона задать переменными, начальное число берите  $> 1$ , чтобы было посложнее
- **Среднее арифметическое чисел** – нужно сумму всех чисел поделить на количество этих чисел
- В этом же классе - найти среднее арифметическое только четных чисел из этого диапазона чисел

## Задача на дом «Цикл for»

- Распечатать числа, кратные четверке от 1 до 100, причем в обратном порядке (то есть, начиная с 100)
- Определение кратности четверке вынести в отдельную функцию
- Использовать цикл for

# Задача на дом «Цифры числа\*»

- Прочитать с консоли целое число
- Найдите сумму его цифр
- Найдите сумму только тех цифр числа, которые являются нечетными числами
- Найдите максимальную цифру числа
- В качестве ответа на ДЗ отправить ссылку на репозиторий на почту [academits.autotest@gmail.com](mailto:academits.autotest@gmail.com) (каждую задачу сохранить в виде отдельного класса)

# Задача на дом «Person»

- Выполнить практику 4 с урока
- В класс Person добавить поле **age**
- Реализовать геттер и сеттер для поля **age**
- Сделать метод для получения года рождения человека
- В классе Main создать объект класса Person, установить ему значение поля age и вывести на экран год рождения человека
- В качестве ответа на ДЗ отправить ссылку на репозиторий на почту [academits.autotest@gmail.com](mailto:academits.autotest@gmail.com)