

Введение в программирование на Java

Лекция 6. ООП (часть 1). Базовые понятия.

Виталий Олегович Афанасьев

16/17 декабря 2025

Algorithms + Data Structures = Programs (Н. Вирт, 1976)

Процедурное программирование:

1. Программа в первую очередь определяется своим поведением (т.е. алгоритмами).
2. Структуры данных выполняют вспомогательную роль, передавая данные между подпрограммами.

Объектно-ориентированное программирование:

1. Объекты (т.е. данные) первичны. Они описывают то, с чем работает программа.
2. Методы — способ передачи информации между объектами.

Объектно-ориентированное программирование пытается спроецировать реальный мир на программный код.

Пример:

На Новый Год Ваня отправил подарок Юле,
которая живёт в другом городе.

В данном случае объектами могут являться:

- Ваня
- Юля
- Подарок (и каждая его составляющая, в т.ч. упаковка)
- Служба доставки
- Курьер

Абстракция

Абстракция (в случае ООП) — определение тех свойств объекта, которые важны для решения задачи, и при этом исключение из рассмотрения несущественных.

То, где проходит граница "важно/неважно", **зависит от задачи**.

С точки зрения водителя, автомобиль — средство передвижения. У автомобиля есть замок зажигания и несколько педалей.

С точки зрения автомеханика, автомобиль — набор электромеханических компонент, которые взаимодействуют с друг другом.

С точки зрения физика, автомобиль — куча элементарных частиц, связанных фундаментальными взаимодействиями.

Рекомендую к ознакомлению: [Гради Буч: Объектно-ориентированный анализ и проектирование. Глава 4: Классификация](#)

Поведение объекта — какие операции с ним важно проводить?

Состояние объекта — какие данные он хранит и как они изменяются?

Поведение может изменять состояние.

Состояние может влиять на поведение.

Понятия класса и объекта (1)

Класс — это описание данных, которые хранит объект, и его поведения.

```
1 // Файл: Car.java
2 public class Car {
3 // Поля класса. Описывают данные, которые хранит каждый объект.
4     public String model;
5     public String color;
6     public int maxSpeed;
7     public int currentSpeed;
8
9 // Методы класса. Описывают операции, которые можно сделать с объектом.
10    public void makeSound() {
11        System.out.println("Beeeeeep!");
12    }
13
14    public void accelerate() {
15        if (currentSpeed < maxSpeed) {
16            currentSpeed += 1;
17        }
18    }
19 }
```

Понятия класса и объекта (2)

Объект — конкретный экземпляр класса (**instance**).

Для создания экземпляров класса используется оператор `new`.

```
1 // Файл: Main.java
2 public class Main {
3     public static void main(String[] args){
4         Car blackBmw = new Car();
5         blackBmw.model = "BMW M5";
6         blackBmw.color = "black";
7         blackBmw.maxSpeed = 250;
8
9         blackBmw.accelerate();
10        System.out.println(blackBmw.currentSpeed); // 1
11    }
12 }
```

Понятия класса и объекта (3)

У одного класса может быть несколько экземпляров.

```
1 // Файл: Main.java
2 public class Main {
3     public static void main(String[] args){
4         Car blackBmw = new Car();
5         blackBmw.model = "BMW M5";
6         blackBmw.color = "black";
7         blackBmw.maxSpeed = 250;
8
9         Car whiteSupra = new Car();
10        whiteSupra.model = "Toyota Supra";
11        whiteSupra.color = "white";
12        whiteSupra.maxSpeed = 100500;
13
14        whiteSupra.accelerate();
15        System.out.println(whiteSupra.currentSpeed); // 1
16        System.out.println(blackBmw.currentSpeed); // 0
17    }
18 }
```


Конструкторы (1)

Чтобы упростить заполнение полей объекта, в классах можно определить **конструкторы** — специальные методы, вызывающиеся при создании объекта.

```
1 // Файл: Car.java
2 public class Car {
3     public String model;
4     public String color;
5     public int maxSpeed;
6     public int currentSpeed;
7
8     public Car(String carModel, String carColor, int carMaxSpeed) {
9         model = carModel;
10        color = carColor;
11        maxSpeed = carMaxSpeed;
12        currentSpeed = 0;
13    }
14 }
15 // main:
16 Car blackBmw = new Car("BMW M5", "black", 250);
```

Конструкторы (2)

Можно объявлять несколько конструкторов (т.е. перегружать их).

```
1 // Файл: Car.java
2 public class Car {
3     public String model;
4     public String color;
5     public int maxSpeed;
6     public int currentSpeed;
7
8     public Car(String carModel, String carColor, int carMaxSpeed) {
9         model = carModel;
10        color = carColor;
11        maxSpeed = carMaxSpeed;
12        currentSpeed = 0;
13    }
14    public Car(String carModel) {
15        model = carModel;
16        color = "black";
17        maxSpeed = 200;
18        currentSpeed = 0;
19    }
20 }
```

Конструкторы (3)

При отсутствии конструкторов, у класса неявно генерируется конструктор по умолчанию без параметров.

```
1 // Файл: Car.java
2 public class Car {
3     public String model;
4     public String color;
5     public int maxSpeed;
6     public int currentSpeed;
7
8     // Т.к. нет ни одного конструктора, неявно присутствует следующий:
9     // public Car() {
10    // }
11 }
```

Ключевое слово this (1)

Ключевое слово `this` обозначает ссылку на текущий объект (т.е. объект, на котором был вызван текущий метод).

Полезно, если параметры/переменные метода называются так же, как поля класса.

```
1 // Файл: Car.java
2 public class Car {
3     public String model;
4     public String color;
5     public int maxSpeed;
6     public int currentSpeed;
7
8     public Car(String model, String color, int maxSpeed) {
9         // this.model - поле класса, model - параметр конструктора
10        this.model = model;
11        this.color = color;
12        this.maxSpeed = maxSpeed;
13        this.currentSpeed = 0;
14    }
15 }
```

Ключевое слово this (2)

У ключевого слова `this` также есть второе значение — конструктор текущего класса.

Полезно при перегрузке конструкторов со значениями по умолчанию.

```
1 // Файл: Car.java
2 public class Car {
3     ...
4
5     public Car(String model, String color, int maxSpeed) {
6         this.model = model;
7         this.color = color;
8         this.maxSpeed = maxSpeed;
9         this.currentSpeed = 0;
10    }
11
12    public Car(String model) {
13        // Вызывает конструктор Car(String, String, int)
14        this(model, "black", 200);
15    }
16 }
```

Неизменяемые поля

Поля, которые никогда не изменяются во время жизни объекта (устанавливаются в нужное значение только при создании) рекомендуется объявлять как `final`.

```
1 // Файл: Car.java
2 public class Car {
3     // Полям model, color, maxSpeed можно присвоить значение единожды
4     // Поле currentSpeed можно будет изменить в любом методе
5     public final String model;
6     public final String color;
7     public final int maxSpeed;
8     public int currentSpeed;
9
10    public Car(String model, String color, int maxSpeed) {
11        this.model = model;
12        this.color = color;
13        this.maxSpeed = maxSpeed;
14        this.currentSpeed = 0;
15    }
16 }
```

Значения по умолчанию

Полям, которым не было присвоено значение в конструкторе, при создании устанавливается значение по умолчанию (см. лекцию 4).

```
1 // Файл: Car.java
2 public class Car {
3     public final String model;
4     public final String color;
5     public final int maxSpeed;
6     public int currentSpeed; // 0 по умолчанию
7     public String originCountry; // null по умолчанию
8     public String owner = "Manufacturer"; // Manufacturer по умолчанию
9
10    public Car(String model, String color, int maxSpeed) {
11        this.model = model;
12        this.color = color;
13        this.maxSpeed = maxSpeed;
14    }
15 }
```

Модификатор static (1)

Модификатором `static` помечаются члены класса, которые связаны не с конкретными экземплярами, а с самим классом.

```
1 // Файл: MathHelpers.java
2 public class MathHelpers {
3     public static int sum(int a, int b) {
4         return a + b;
5     }
6 }
7 // Файл: Main.java
8 public class Main {
9     public static void main(String[] args) {
10         // Для вызова метода sum не нужно создавать сам объект класса
11         int twoPlusTwo = MathHelpers.sum(2, 2); // == 4
12     }
13 }
```


Модификатор static (2)

Нестатические методы могут использовать статические члены класса.

```
1 // Файл: Person.java
2 public class Person {
3     public static int numberOfPersons = 0;
4     public final String name;
5     public Person(String name) {
6         this.name = name;
7         ++numberOfPersons;
8     }
9 }
10 // Файл: Main.java
11 public class Main {
12     public static void main(String[] args) {
13         Person person1 = new Person("John");
14         Person person2 = new Person("Bob");
15         System.out.println(Person.numberOfPersons); // 2
16     }
17 }
```

Модификатор static (3)

Но статические методы **не** могут использовать нестатические члены класса.

```
1 // Файл: Person.java
2 public class Person {
3     public static int numberOfPersons = 0;
4     public final String name;
5     public Person(String name) {
6         this.name = name;
7         ++numberOfPersons;
8     }
9
10    public void sayName() {
11        System.out.println("My name is " + name);
12    }
13
14    public static void badMethod() {
15        System.out.println(numberOfPersons); // OK
16        System.out.println(name); // ERROR: name - нестатическое поле
17        sayName(); // ERROR: name - не статический метод
18    }
19 }
```

Инкапсуляция

Инкапсуляция — фундаментальный принцип ООП.

Инкапсуляция подразумевает связывание данных с операциями, которые на них можно производить, при этом детали реализации скрываются от пользователя. При вызове метода класс взаимодействует с внутренними данными, **не нарушая их согласованности**.

Таким образом, пользователи класса работают с ним как с "чёрным ящиком".

Пример из жизни: команды отдают собаке, а не её лапам. Собака сама разберётся со своими лапами.

Пример из Java: класс `Scanner`. Для считывания данных с клавиатуры не нужно знать, **как** реализован данный класс. Важно лишь знать, **что** этот класс умеет делать.

Если реализация поменяется, это никак не отразится на коде, использующем класс!

Соккрытие (1)

Инкапсуляция тесно связана с понятием **сокрытия**.

Соккрытие — механизм, позволяющий "спрятать" некоторые члены классов от его пользователей.

В Java это достигается модификаторами доступа: `public`, `private`, `protected`, `package-private`.

Соккрытие (2)

К членам класса с модификатором `public` можно получить доступ из других классов.

```
1 // Файл: Car.java
2 public class Car {
3     public final String model;
4     public final String color;
5     public final int maxSpeed;
6     public int currentSpeed = 0;
7
8     public void accelerate() {
9         if (currentSpeed < maxSpeed) {
10             currentSpeed += 1;
11         }
12     }
13 }
14 // main:
15 Car car = new Car(...);
16 car.accelerate(); // OK: public
17 car.currentSpeed += 1_000_000; // OK: public, но нарушили согласованность
```

Соккрытие (3)

К членам класса с модификатором `private` можно получить доступ только изнутри класса.

```
1 // Файл: Car.java
2 public class Car {
3     public final String model;
4     public final String color;
5     public final int maxSpeed;
6     private int currentSpeed = 0; // Поле доступно только в этом классе
7
8     public void accelerate() {
9         if (currentSpeed < maxSpeed) {
10             currentSpeed += 1; // OK: private, но код внутри класса
11         }
12     }
13 }
14 // main:
15 Car car = new Car(...);
16 car.accelerate(); // OK: public
17 car.currentSpeed += 1_000_000; // ERROR: private
```

Соккрытие (4)

`private` ограничивает доступ классом, а не объектом. Т.е. можно влиять на приватные члены других экземпляров данного класса.

```
1 // Файл: Car.java
2 public class Car {
3     public final String model;
4     public final String color;
5     public final int maxSpeed;
6     private int currentSpeed = 0;
7
8     public void crashInto(Car other) {
9         this.currentSpeed = 0;
10        other.currentSpeed = 0;
11        // OK: private доступен у другого экземпляра этого класса
12    }
13 }
```

Инкапсуляция \neq Соккрытие (1)

Понятия инкапсуляции и соккрытия не являются тождественными.

Инкапсуляция — соккрытие (абстракция, изоляция) деталей реализации.

Соккрытие (в случае модификаторов доступа) — **конструкции языка**, запрещающие доступ одного кода к другому.

Соккрытие — один из способов обеспечения инкапсуляции.

В некоторых языках нет механизма соккрытия (например, Python), но это не мешает инкапсулировать логику работы с данными.

Инкапсуляция \neq Соккрытие (2)

Даже при наличии соккрытия легко нарушить инкапсуляцию.

Например, метод `accelerate` в правильной реализации класса `Car` следит, чтобы скорость не превышала максимальной. Можно переложить эту проверку на пользователей класса. В таком случае, `Car` уже не инкапсулирует свои данные и объект может оказаться в некорректном состоянии.

Или же представьте себе такой класс `Scanner`, что после каждого пятого вызова метода `nextInt` приходилось бы вызывать метод `fixState`. Если же этот метод не вызвать — BOOM!

Пример нарушения инкапсуляция (1)

```
1 // Файл: Group.java
2 public class Group {
3     private final Person[] people;
4     ...
5
6     public void printPeopleNames() {
7         for (int i = 0; i < people.length; ++i) {
8             System.out.println(people[i].name);
9         }
10    }
11    public Person[] getPeople() {
12
13        return people;
14    }
15 }
16 // main:
17 Group group = new Group(...);
18
19
20 group.printPeopleNames();
```

Пример нарушения инкапсуляция (2)

```
1 // Файл: Group.java
2 public class Group {
3     private final Person[] people;
4     ...
5
6     public void printPeopleNames() {
7         for (int i = 0; i < people.length; ++i) {
8             System.out.println(people[i].name);
9         }
10    }
11    public Person[] getPeople() {
12        // Возвращается ССЫЛКА на массив
13        return people;
14    }
15 }
16 // main:
17 Group group = new Group(...);
18 Person[] people = group.getPeople(); // Этот массив ссылается на поле
19 people[0] = null; // Поменяли состояние поля people
20 group.printPeopleNames(); // Ошибка при обращении к person[0].name
```

Пример нарушения инкапсуляция (3)

```
1 // Файл: Group.java
2 public class Group {
3     private final Person[] people;
4     ...
5
6     public void printPeopleNames() {
7         for (int i = 0; i < people.length; ++i) {
8             System.out.println(people[i].name);
9         }
10    }
11    public Person[] getPeople() {
12        // Возвращается КОПИЯ массива
13        return Arrays.copyOf(people, people.length);
14    }
15 }
16 // main:
17 Group group = new Group(...);
18 Person[] people = group.getPeople(); // Этот массив НЕ ссылается на поле
19 people[0] = null; // Состояние поля в объекте group не меняется
20 group.printPeopleNames(); // ОК
```

Accessor-методы (1)

В случае, если у класса есть поля, к которым необходимо получить доступ извне, рекомендуется делать сами поля `private`, при этом создавая **accessor-методы**.

Getter — метод, возвращающий значение поля.

Setter — метод, меняющий значение поля.

```
1 // Файл: User.java
2 public class User {
3     private String name;
4
5     public String getName() {
6         return name;
7     }
8     public void setName(String name) {
9         this.name = name;
10    }
11 }
12 // main:
13 user.setName("Vasya Pupkin");
```

Accessor-методы (2)

Если реализация класса поменяется, то не придётся менять весь код, который обращался к полям — дополнительную логику можно реализовать в getter'ах и setter'ах.

```
1 // Файл: User.java
2 public class User {
3     private String firstName;
4     private String lastName;
5
6     public String getName() {
7         return firstName + " " + lastName;
8     }
9     public void setName(String name) {
10         String[] firstAndLastName = name.split(" ");
11         this.firstName = firstAndLastName[0];
12         this.lastName = firstAndLastName[1];
13     }
14 }
15 // main:
16 user.setName("Vasya Pupkin"); // Код не поменялся!
```