Введение в программирование на Java

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

Виталий Олегович Афанасьев

17 февраля 2025

Процедурное программирование VS ООП

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

Процедурное программирование VS ООП

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.

Процедурное программирование VS ООП

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.

ООП и реальный мир

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

ООП и реальный мир

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

Пример:

> На День всех влюблённых Ваня отправил букет из роз Юле, которая живёт в другом городе.

В данном случае объектами могут являться:

- Ваня
- Юля
- Букет (и каждая роза в нём)
- Флорист(ы)
- 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**.

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.

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.

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.

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

Характеристики объектов

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.

```java
// Файл: Car.java
public class Car {
// Поля класса. Описывают данные, которые хранит каждый объект.
    public String model;
    public String color;
    public int maxSpeed;
    public int currentSpeed;

// Методы класса. Описывают операции, которые можно сделать с объектом.
    public void makeSound() {
        System.out.println("Beeeeeep!");
    }

    public void accelerate() {
        if (currentSpeed < maxSpeed) {
            currentSpeed += 1;
        }
    }
}
```

Понятия класса и объекта (2)

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

Для создания экземпляров класса используется оператор `new`.

```java
// Файл: Main.java
public class Main {
    public static void main(String[] args){
        Car blackBmw = new Car();
        blackBmw.model = "BMW M5";
        blackBmw.color = "black";
        blackBmw.maxSpeed = 250;

        blackBmw.accelerate();
        System.out.println(blackBmw.currentSpeed); // 1
    }
}
```

Понятия класса и объекта (3)

У одного класса может быть несколько экземпляров.

```java
// Файл: Main.java
public class Main {
    public static void main(String[] args){
        Car blackBmw = new Car();
        blackBmw.model = "BMW M5";
        blackBmw.color = "black";
        blackBmw.maxSpeed = 250;

        Car whiteSupra = new Car();
        whiteSupra.model = "Toyota Supra";
        whiteSupra.color = "white";
        whiteSupra.maxSpeed = 100500;

        whiteSupra.accelerate();
        System.out.println(whiteSupra.currentSpeed); // 1
        System.out.println(blackBmw.currentSpeed); // 0
    }
}
```

Конструкторы (1)

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

```java
// Файл: Car.java
public class Car {
    public String model;
    public String color;
    public int maxSpeed;
    public int currentSpeed;

    public Car(String carModel, String carColor, int carMaxSpeed) {
        model = carModel;
        color = carColor;
        maxSpeed = carMaxSpeed;
        currentSpeed = 0;
    }
}
// main:
Car blackBmw = new Car("BMW M5", "black", 250);
```

Конструкторы (2)

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

```java
// Файл: Car.java
public class Car {
    public String model;
    public String color;
    public int maxSpeed;
    public int currentSpeed;

    public Car(String carModel, String carColor, int carMaxSpeed) {
        model = carModel;
        color = carColor;
        maxSpeed = carMaxSpeed;
        currentSpeed = 0;
    }
    public Car(String carModel) {
        model = carModel;
        color = "black";
        maxSpeed = 200;
        currentSpeed = 0;
    }
}
```

Конструкторы (3)

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

```java
// Файл: Car.java
public class Car {
    public String model;
    public String color;
    public int maxSpeed;
    public int currentSpeed;

    // Т.к. нет ни одного конструктора, неявно присутствует следующий:
    // public Car() {
    // }
}
```

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.

```java
// Файл: Car.java
public class Car {
    public String model;
    public String color;
    public int maxSpeed;
    public int currentSpeed;

    public Car(String model, String color, int maxSpeed) {
        // this.model - поле класса, model - параметр конструктора
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
        this.currentSpeed = 0;
    }
}
```

Ключевое слово this (2)

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

Полезно при перегрузке конструкторов со значениями по умолчанию.

```java
// Файл: Car.java
public class Car {
    ...

    public Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
        this.currentSpeed = 0;
    }

    public Car(String model) {
        // Вызывает конструктор Car(String, String, int)
        this(model, "black", 200);
    }
}
```

Неизменяемые поля

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

```java
// Файл: Car.java
public class Car {
    // Полям model, color, maxSpeed можно присвоить значение единожды
    // Поле currentSpeed можно будет изменить в любом методе
    public final String model;
    public final String color;
    public final int maxSpeed;
    public int currentSpeed;

    public Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
        this.currentSpeed = 0;
    }
}
```

Значения по умолчанию

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

```java
// Файл: Car.java
public class Car {
    public final String model;
    public final String color;
    public final int maxSpeed;
    public int currentSpeed; // 0 по умолчанию
    public String originCountry; // null по умолчанию
    public String owner = "Manufacturer"; // Manufacturer по умолчанию

    public Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

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.

```java
// Файл: Person.java
public class Person {
    public static int numberOfPersons = 0;
    public final String name;
    public Person(String name) {
        this.name = name;
        ++numberOfPersons;
    }
}
// Файл: Main.java
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("John");
        Person person2 = new Person("Bob");
        System.out.println(Person.numberOfPersons); // 2
    }
}
```

Модификатор static (3)

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

```java
// Файл: Person.java
public class Person {
    public static int numberOfPersons = 0;
    public final String name;
    public Person(String name) {
        this.name = name;
        ++numberOfPersons;
    }

    public void sayName() {
        System.out.println("My name is " + name);
    }

    public static void badMethod() {
        System.out.println(numberOfPersons); // OK
        System.out.println(name); // ERROR: name - нестатическое поле
        sayName(); // ERROR: name - не статический метод
    }
}
```

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".

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.

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.

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!

Сокрытие (1)

Инкапсуляция тесно связана с понятием **сокрытия**.

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

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

Сокрытие (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
```

Сокрытие (3)

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

```java
// Файл: Car.java
public class Car {
    public final String model;
    public final String color;
    public final int maxSpeed;
    private int currentSpeed = 0; // Поле доступно только в этом классе

    public void accelerate() {
        if (currentSpeed < maxSpeed) {
            currentSpeed += 1; // OK: private, но код внутри класса
        }
    }
}
// main:
Car car = new Car(...);
car.accelerate(); // OK: public
car.currentSpeed += 1_000_000; // ERROR: private
```

Сокрытие (4)

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

```java
// Файл: Car.java
public class Car {
    public final String model;
    public final String color;
    public final int maxSpeed;
    private int currentSpeed = 0;

    public void crashInto(Car other) {
        this.currentSpeed = 0;
        other.currentSpeed = 0;
            // OK: private доступен у другого экземпляра этого класса
    }
}
```

Инкапсуляция $\neq$ Сокрытие (1)

Понятия инкапсуляции и сокрытия не являются тождественными.

Инкапсуляция — сокрытие (абстракция, изоляция) деталей реализации.

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

Сокрытие — один из способов обеспечения инкапсуляции.

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

Инкапсуляция $\neq$ Сокрытие (2)

Даже при наличии сокрытия легко нарушить инкапсуляцию.

Инкапсуляция $\neq$ Сокрытие (2)

Даже при наличии сокрытия 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.

Инкапсуляция ≠ Сокрытие (2)

Даже при наличии сокрытия 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)

```java
// Файл: Group.java
public class Group {
    private final Person[] people;
    ...

    public void printPeopleNames() {
        for (int i = 0; i < people.length; ++i) {
            System.out.println(people[i].name);
        }
    }
    public Person[] getPeople() {
        // Возвращается КОПИЯ массива
        return Arrays.copyOf(people, people.length);
    }
}
// main:
Group group = new Group(...);
Person[] people = group.getPeople(); // Этот массив НЕ ссылается на поле
people[0] = null; // Состояние поля в объекте group не меняется
group.printPeopleNames(); // OK
```

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.

```java
// Файл: User.java
public class User {
    private String firstName;
    private String lastName;

    public String getName() {
        return firstName + " " + lastName;
    }
    public void setName(String name) {
        String[] firstAndLastName = name.split(" ");
        this.firstName = firstAndLastName[0];
        this.lastName = firstAndLastName[1];
    }
}
// main:
user.setName("Vasya Pupkin"); // Код не поменялся!
```