Введение в программирование на Java

Лекция 8. ООП (часть 3). Интерфейсы. Абстрактные классы.

Виталий Олегович Афанасьев

10 марта 2025

Класс Object

Неявное наследование от класса Object

Каждый класс неявно наследуется от класса Object.

Таким образом, две следующих записи идентичны:

```
public class Car {
    ...
}
public class Car extends Object {
    ...
}
```

Object можно использовать как обычный тип:

```
Object car = new Car("BMW", 250);
Object truck = new Truck("KAMAZ", 100, 0);
```

Методы класса Object

У Object определён ряд методов:

1. `public boolean equals(Object obj)`
2. `public int hashCode()`
3. `public String toString()`
4. `public final Class<?> getClass()`
5. `public final void notify()`
6. `public final void notifyAll()`
7. `public final void wait()`
8. `public final void wait(long timeoutMillis)`
9. `public final void wait(long timeoutMillis, int nanos)`
10. `protected Object clone()`
11. `protected void finalize()`

Таким образом, эти методы можно вызывать у любого класса (а некоторые — и переопределять).

Методы класса Object: toString (1)

Метод `toString` используется для преобразования объекта в строку, описывающаю его. Строка должна быть человекочитаемой.

По умолчанию метод определён как: `имя_класса @ hashCode`.

```
1 Object car = new Car("BMW", 250);
2
3 System.out.println(car.toString()); // Car@7b23ec81
```

Методы класса Object: toString (2)

Рекомендуется переопределять метод `toString` у каждого класса.

Полезно составлять строку из имени класса и значений всех полей.

```
1  public class Car {
2      @Override
3      public String toString() {
4          return "Car{" +
5                  "model='" + model + '\'' +
6                  ", speed=" + speed +
7                  ", maxSpeed=" + maxSpeed +
8                  '}';
9      }
10 }
```

```
1 Car car = new Car("BMW", 250);
2 car.accelerate();
3 System.out.println(car.toString());
4   // Car{model='BMW', speed=1, maxSpeed=250}
```

Методы класса Object: getClass (1)

Метод `getClass` возвращает информацию о реальном типе объекта.

```
1 Object car = new Car("BMW", 250);
2
3 Object truck = new Truck("KAMAZ", 100, 0);
4
5 System.out.println(car.getClass()); // class Car
6 System.out.println(truck.getClass()); // class Truck
```

Методы класса Object: getClass (2)

При помощи него можно проверять, имеют ли объекты одинаковый тип во время выполнения.

```
1 Object car = new Car("BMW", 250);
2
3 Object truck1 = new Truck("KAMAZ", 100, 0);
4 Object truck2 = new Truck("MAN", 120, 0);
5
6 System.out.println(car.getClass() == truck1.getClass()); // false
7 System.out.println(truck1.getClass() == truck2.getClass()); // true
```

Подробнее с `getClass` и Reflection API мы разберёмся в будущем.

Методы класса Object: equals (1)

Метод `equals` определяет отношение эквивалентности между объектами.

По умолчанию `equals` определён как сравнение ссылок на объекты.

```
1 Car bmw1 = new Car("BMW", 250);
2 Car bmw2 = new Car("BMW", 250);
3 Car toyota = new Car("Toyota", 100500);
4
5 System.out.println(bmw1.equals(bmw2)); // false
6 System.out.println(bmw1.equals(toyota)); // false
7 System.out.println(bmw1.equals(bmw1)); // true
```

Методы класса Object: equals (2)

Классическое переопределение `equals` — сравнение типов объектов и значимых полей.

```java
public class Car {
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        if (getClass() != o.getClass()) return false;
        Car other = (Car) o;
        return maxSpeed == other.maxSpeed && model.equals(other.model);
    }
}
```

```java
Car bmw1 = new Car("BMW", 250);
Car bmw2 = new Car("BMW", 250);
Car toyota = new Car("Toyota", 100500);
System.out.println(bmw1.equals(bmw2)); // true
System.out.println(bmw1.equals(toyota)); // false
System.out.println(bmw1.equals(bmw1)); // true
```

Методы класса Object: equals (3)

Переопределять `equals` нужно только в тех случаях, когда сравнение объектов **действительно имеет смысл**.

При этом нужно соблюсти законы равенства:

- Рефлексивность: `x.equals(x)` должно возвращать `true` для любого не-`null` объекта `x`.
- Симметричность: `x.equals(y)` и `y.equals(x)` должны возвращать одинаковые результаты.
- Транзитивность: если `x.equals(y)` истинно и `y.equals(z)` тоже истинно, то `x.equals(z)` тоже должно быть истинно.
- Консистентность: если объекты `x` и `y` не менялись, то `x.equals(y)` должен возвращать один и тот же результат при нескольких вызовах.
- `x.equals(null)` должно возвращать `false` для любого не-`null` объекта `x`.

Массивы как объектный тип

Любой массив тоже является подтипом класса Object.

Тем не менее, методы `toString` и `equals` для них не переопределены, поэтому для этих операций небходимо использовать вспомогательный класс `java.util.Arrays`.

```
1  int[] arr1 = new int[] { 1, 2, 3 };
2  int[] arr2 = new int[] { 1, 2, 3 };
3  Object arr2AsObject = arr2;
4
5  System.out.println(arr.toString()); // [I@15aeb7ab
6  System.out.println(Arrays.toString(arr1)); // [1, 2, 3]
7
8  System.out.println(arr.equals(arr2)); // false
9  System.out.println(arr.equals(arr1)); // true
10 System.out.println(Arrays.equals(arr1, arr1)); // true
11 System.out.println(Arrays.equals(arr1, arr2)); // true
12 System.out.println(Arrays.equals(arr1, (int[]) arr2AsObject)); // true
```

Интерфейсы

Постановка решаемой задачи (1)

Добавляя слои абстракции при наследовании можно прийти к тому, что некоторые классы не содержат никаких данных и конкретного поведения, а просто описывают некоторый концепт.

Как реализовать методы таких классов?

```
1  public class Circle extends Shape {
2      @Override
3      public double perimeter() { return 2 * Math.PI * radius; }
4  }
5  public class Rectangle extends Shape {
6      @Override
7      public double perimeter() { return 2 * (width + height); }
8  }
9  public class Shape {
10     public double perimeter() {
11         return ???; // Что должен возвращать данный метод?
12     }
13 }
```

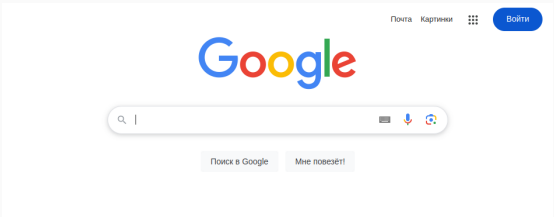
Постановка решаемой задачи (2)

Инстанциировать такие "неконкретные" объекты тоже довольно странно.

```
1 Circle circle = new Circle(10);
2 Rectangle rectangle = new Rectangle(4, 5);
3 Shape shape = new Shape(); // ???
```

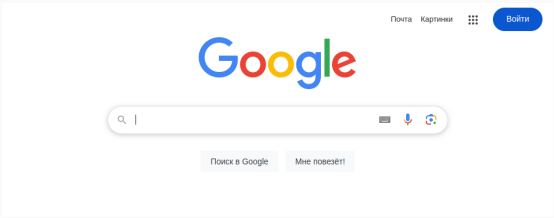
Интерфейсы в реальной жизни

Интерфейсы в реальной жизни

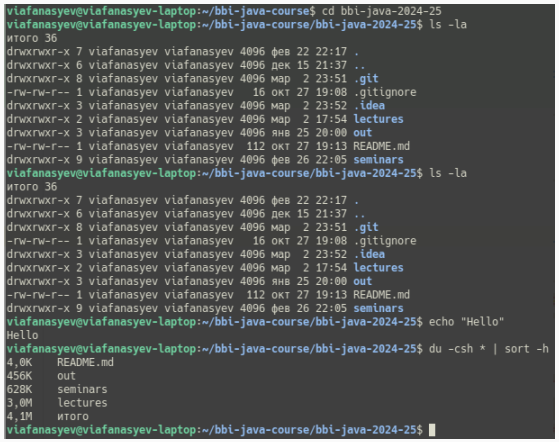


Графический интерфейс (GUI)

Интерфейсы в реальной жизни

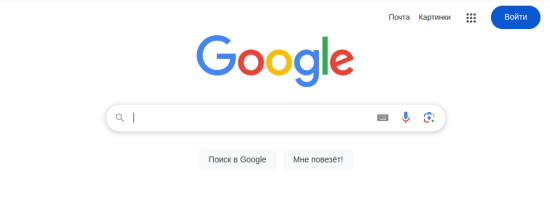


Графический интерфейс (GUI)

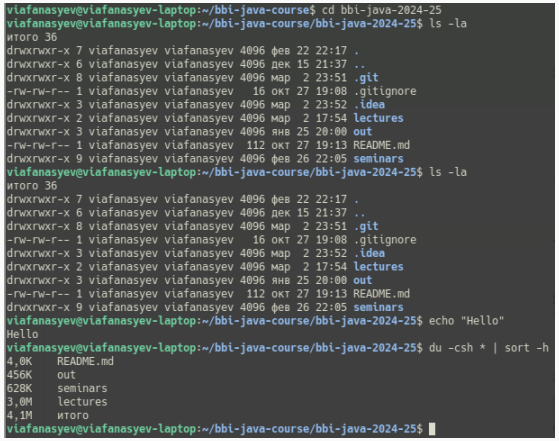


Интерфейс командной строки (CLI)

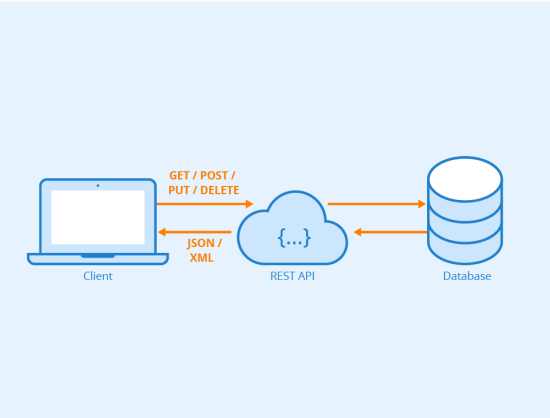
Интерфейсы в реальной жизни



Графический интерфейс (GUI)

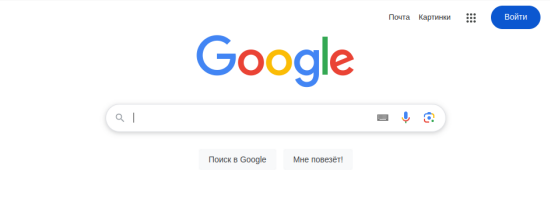


Интерфейс командной строки (CLI)

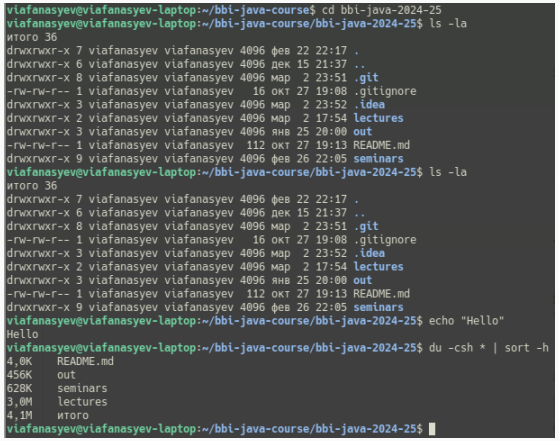


Программный интерфейс (API)

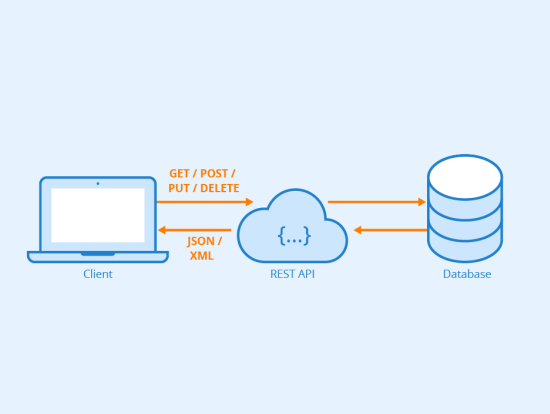
Интерфейсы в реальной жизни



Графический интерфейс (GUI)



Интерфейс командной строки (CLI)



Программный интерфейс (API)

Аппаратный интерфейс

Понятие интерфейса в ООП

Интерфейс — программная конструкция, при помощи которой можно описать "что" должен уметь делать объект, при этом не говоря "как".

Интерфейсы также иногда называют **контрактами**.

Использование интерфейсов в Java (1)

Определяются интерфейсы так же, как и классы, но при помощи ключевого слова `interface`.

Методы интерфейсов не имеют тела.

```
1 public interface Shape {
2     public double perimeter();
3 }
```

Использование интерфейсов в Java (2)

Если класс реализует интерфейс, то это указывается при помощи `implements` при объявлении класса.

Класс **обязан** реализовать все методы интерфейса. Иначе — ошибка компиляции.

```
1 public class Circle implements Shape {
2     @Override
3     public double perimeter() {
4         return 2 * Math.PI * radius;
5     }
6 }
```

Использование интерфейсов в Java (3)

Интерфейсы можно использовать как обычные типы: при определении переменных, параметров и указании типов возвращаемых значений.

Но при этом интерфейсы нельзя инстанциировать.

```
1 public static void main(String[] args) {
2     Shape circle = new Circle(10);
3     int halfPerim = halfPerimeter(new Circle(5));
4     Shape shape = new Shape(); // Ошибка компиляции!
5 }
6
7 private static double halfPerimeter(Shape shape) {
8     return shape.perimeter() / 2;
9 }
```

Использование интерфейсов в Java (4)

Члены интерфейса по умолчанию являются публичными, поэтому модификатор `public` можно (и рекомендуется) опускать.

```
1 public interface Shape {
2     int perimeter();
3 }
```

Множественное наследование (1)

Класс не может наследоваться от нескольких классов одновременно.

```
1  public class Person {
2      public int age() { ... }
3  }
4  public class Named {
5      public String name() { ... }
6  }
7  public class Student extends Person, Named { // Ошибка компиляции!
8      @Override
9      public int age() { ... }
10     @Override
11     public String name() { ... }
12 }
```

Множественное наследование (2)

Но при этом класс может реализовывать несколько интерфейсов.

```
1  public interface Person {
2      int age();
3  }
4  public interface Named {
5      String name();
6  }
7  public class Student implements Person, Named { // OK
8      @Override
9      public int age() { ... }
10     @Override
11     public String name() { ... }
12 }
```

Множественное наследование (3)

```
1 public interface Swimming { ... }
2 public interface Walking { ... }
3 public interface Flying { ... }
4
5 public class Fish implements Swimming { ... }
6 public class Dog implements Swimming, Walking { ... }
7 public class Duck implements Swimming, Walking, Flying { ... }
```

Множественное наследование (3)

```
1 public interface Swimming { ... }
2 public interface Walking { ... }
3 public interface Flying { ... }
4
5 public class Fish implements Swimming { ... }
6 public class Dog implements Swimming, Walking { ... }
7 public class Duck implements Swimming, Walking, Flying { ... }
```

Множественное наследование (4)

Класс может одновременно наследоваться от другого класса и реализовывать несколько интерфейсов.

```
1 public interface Swimming { ... }
2 public interface Walking { ... }
3 public interface Flying { ... }
4
5 public class Animal { ... }
6
7 public class Duck extends Animal implements Swimming, Walking, Flying {
8     ...
9 }
```

Интерфейсы как дополнительный уровень абстракции

Интерфейсы позволяют абстрагироваться от конкретных реализаций.

```
1  public interface IntStack {
2      void push(int value);
3      int pop();
4  }
5
6  public class DynamicArrayIntStack implements IntStack {
7      // Реализация на динамическом массиве
8  }
9  public class LinkedListIntStack implements IntStack {
10     // Реализация на связанном списке
11 }
12
13 public class PostfixCalculator {
14     private final IntStack stack = ...;
15                     // Важна не реализация, а контракт!
16 }
```

Таким образом спроектирован Collection API в Java.

О нём — на будущих лекциях.

Расширение интерфейсов

Интерфейс может расширять любое количество других интерфейсов.

Класс, реализующий интерфейс, должен реализовать все методы из всех интерфейсов в иерархии.

```
1  public interface Collection {
2      int size();
3  }
4  public interface IntStack extends Collection {
5      void push(int value);
6      int pop();
7  }
8  public class DynamicArrayIntStack implements IntStack {
9      @Override
10     public void push(int value) { ... }
11     @Override
12     public int pop() { ... }
13     @Override
14     public int size() { ... }
15 }
```

default-методы в интерфейсах (1)

Интерфейсы имеют свойство эволюционировать.

Но добавить новый в метод в существующий интерфейс не так просто. Если этот интерфейс уже реализуют какие-то классы, то все классы придётся переписать.

Это особенно критично в случае публичных библиотек.

```
1 public interface Collection {
2     int size();
3     boolean isEmpty();
4         // Как добавить этот метод и не сломать существующий код?
5 }
```

default-методы в интерфейсах (2)

Для решения этой проблемы методы в интерфейсах можно объявлять с модификатором `default`.

`default`-методы имеют реализацию по умолчанию.

В классах-наследниках можно переопределить эту реализацию (например, сделать более эффективной), но это не обязательно.

```
1 public interface Collection {
2     int size();
3     default boolean isEmpty() {
4         return size() == 0;
5     }
6 }
```

Множественное наследование при наличии default-методов (1)

При наличии default-методов могут возникать коллизии в потомках.

```
1  public interface Intrfc {
2      default void foo() {
3          System.out.println("Interface");
4      }
5  }
6  public class Parent {
7      public void foo() {
8          System.out.println("Parent class");
9      }
10 }
11 public class Child extends Parent implements Intrfc {
12 }
13
14 public static void main(String[] args) {
15     Child child = new Child();
16     child.foo(); // Какой из методов вызывается?
17 }
```

Множественное наследование при наличии default-методов (2)

Для разрешения этих коллизий есть два простых правила:

1. Реализация в классе всегда "побеждает" реализацию из интерфейса.
2. При наличии одинаковых методов в интерфейсах, один из которых `default`, требуется явно выбрать реализацию.

Множественное наследование при наличии default-методов (3)

1. Реализация в классе всегда "побеждает" реализацию из интерфейса.

```
1  public interface Intrfc {
2      default void foo() {
3          System.out.println("Interface");
4      }
5  }
6  public class Parent {
7      public void foo() {
8          System.out.println("Parent class");
9      }
10 }
11 public class Child extends Parent implements Intrfc {
12 }
13
14 public static void main(String[] args) {
15     Child child = new Child();
16     child.foo(); // Parent class
17 }
```

Множественное наследование при наличии default-методов (4)

2. При наличии одинаковых методов в интерфейсах, один из которых default, требуется явно выбрать реализацию.

```
1  public interface FirstIntrfc {
2      default void foo() { System.out.println("First interface"); }
3  }
4  public interface SecondIntrfc {
5      default void foo() { System.out.println("Second interface"); }
6  }
7  public class Child implements FirstIntrfc, SecondIntrfc {
8      // Ошибка компиляции: несколько default
9  }
10
11 public static void main(String[] args) {
12     Child child = new Child();
13     child.foo();
14 }
```

Множественное наследование при наличии default-методов (5)

2. При наличии одинаковых методов в интерфейсах, один из которых `default`, требуется явно выбрать реализацию.

```
1  public interface FirstIntrfc {
2      default void foo() { System.out.println("First interface"); }
3  }
4  public interface SecondIntrfc {
5      default void foo() { System.out.println("Second interface"); }
6  }
7  public class Child implements FirstIntrfc, SecondIntrfc {
8      @Override
9      public void foo() {
10         FirstIntrfc.super.foo(); // <имя интерфейса>.super.<имя метода>
11     }
12 }
13
14 public static void main(String[] args) {
15     Child child = new Child();
16     child.foo(); // First interface
17 }
```

Множественное наследование при наличии default-методов (6)

2. При наличии одинаковых методов в интерфейсах, один из которых default, требуется явно выбрать реализацию.

```
1  public interface FirstIntrfc {
2      default void foo() { System.out.println("First interface"); }
3  }
4  public interface SecondIntrfc {
5      void foo(); // Этот метод не default
6  }
7  public class Child implements FirstIntrfc, SecondIntrfc {
8      // Всё ещё ошибка компиляции:
9      //   метод void foo() в нескольких интерфейсах, один из них default
10 }
11
12 public static void main(String[] args) {
13     Child child = new Child();
14     child.foo();
15 }
```

Вспомогательные методы в интерфейсах (1)

Помимо публичных методов в интерфейсах можно задавать вспомогательные приватные методы.

Приватные не-статические методы должны быть объявлены как `default`.

```
1  public interface Intrfc {
2      default void foo() {
3          doSomethingCommon();
4      }
5      default void bar() {
6          doSomethingCommon();
7      }
8
9      private default void doSomethingCommon() {
10         System.out.println("Common");
11         nonDefaultMethod();
12     }
13
14     void nonDefaultMethod();
15 }
```

Вспомогательные методы в интерфейсах (2)

Также можно объявлять статические методы (как private, так и public).

```
1 public interface Intrfc {
2     static void staticHelper() { // Неявно public
3         System.out.println("Static helper");
4         privateStaticHelper();
5     }
6     private static void privateStaticHelper() {
7         System.out.println("Private Static helper");
8     }
9 }
```

Константы в интерфейсах

В интерфейсах также можно задавать константы.

По умолчанию константы имеют модификаторы `public static final`.

Сделать их не-`static` нельзя! Т.е. они не являются полями класса.

Также нельзя сделать их не-`public`.

```
1 public interface Intrfc {
2     int ULTIMATE_ANSWER = 42;
3 }
4
5 public static void main(String[] args) {
6     System.out.println(Intrfc.ULTIMATE_ANSWER); // 42
7 }
```

Абстрактные классы

Абстрактные классы (1)

Иногда абстракции описывают не только поведение "неконкретных" объектов, но ещё и какой-то набор данных, которые хранит каждый класс.

Интерфейсов в таком случае недостаточно, ведь они описывают только поведение.

Абстрактные классы (2)

В таком случае можно использовать абстрактные классы.

Абстрактные классы эквивалентны обычным классам за тем исключением, что некоторые их методы можно помечать `abstract`.

```
1  public abstract class BinaryOperation {
2      private final String name;
3      public BinaryOperation(String name) {
4          this.name = name;
5      }
6      public String getName() {
7          return name;
8      }
9      public abstract int execute(int l, int r);
10 }
```

Абстрактные классы (3)

Наследники обязаны реализовать абстрактные методы, либо сами быть абстрактными.

```
1 public class PlusOperation extends BinaryOperation {
2     public PlusOperation() {
3         super("Plus");
4     }
5     @Override
6     public int execute(int l, int r) {
7         return a + b;
8     }
9 }
```

Абстрактные классы (4)

Абстрактные классы, как и интерфейсы, можно использовать как обычные типы.

Так же как интерфейсы их нельзя инстанциировать.

```
1 public static void main(String[] args) {
2     BinaryOperation plus = new PlusOperation();
3     int sumOf2And2 = executeFor2And2(new PlusOperation());
4     BinaryOperation binOp = new BinaryOperation(); // Ошибка компиляции!
5 }
6
7 private static int executeFor2And2(BinaryOperation op) {
8     return op.execute(2, 2);
9 }
```

Абстрактные классы VS интерфейсы

Т.к. интерфейсы и абстрактные классы довольно похожи, может возникнуть вопрос — когда пользоваться одним, а когда другим.

- При наследовании родителем может выступать максимум один класс, но любое количество интерфейсов.
- Полей в интерфейсах быть не может, в отличии от абстрактных классов.

Если вам не нужно выносить данные в абстрактную сущность — лучше всегда использовать интерфейсы.