



Faculty of Software Engineering and Computer Systems

Programming

Lecture #3.
OOP. Continue.

Instructor of faculty
Pismak Alexey Evgenievich
Kronverksky Pr. 49, 374 room
pismak@itmo.ru


Saint-Petersburg

Блоки инициализации

```
1. public class A {  
2.     public A() {  
3.         System.out.println("A  
         constr");  
4.     }  
  
5. {  
6.     System.out.println("A block");  
7. }  
8. }
```

```
1. public class B extends A {  
2.     public B() {  
3.         System.out.println("B  
         constr");  
4.     }  
  
5. {  
6.     System.out.println("B block");  
7. }  
8. }
```

```
public  
    static  
        void main(String ...s) {  
  
            B b = new B();  
  
        }
```

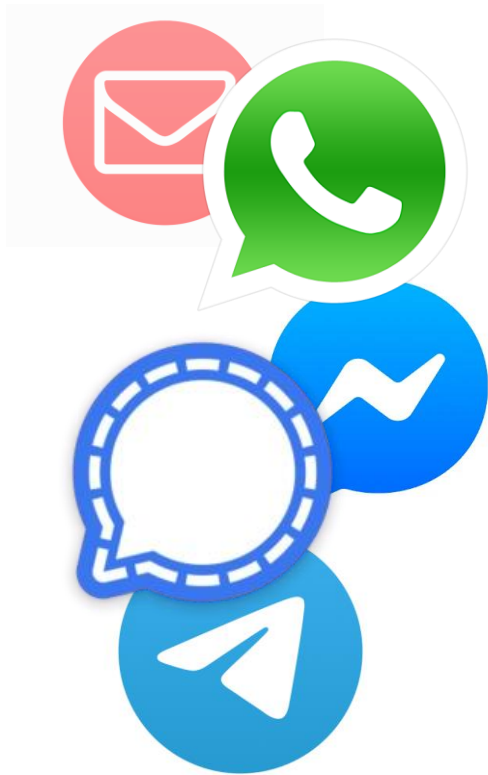


A block
A constr
B block
B constr

Блоки инициализации

```
1. public class Main {  
2.     private static int field;  
3.     public static void main (String... s) {  
4.         System.out.println("main");  
5.     }  
  
6.     static {  
7.         field = 10;  
8.         main(null);  
9.         System.exit(-1);  
10.    }  
  
11. }
```

Полиморфизм



это возможность использовать
объекты с одинаковым
протоколом взаимодействия
без информации о типе и
внутренней структуре объекта

1. Переопределение методов
2. Перегрузка методов
3. Коварианты возвращаемых типов
4. Абстрактные классы

Переопределение методов

```
public class Vector {  
  
    protected float x, y;  
  
    public void multiply(float value) { }  
    public void move(Vector v) { }  
    public Vector normalize() { ... }  
}
```

```
public class Vector3 extends Vector {  
    protected float z;  
  
    @Override  
    public void multiply(float value) { }  
}
```

Перегрузка методов

```
public class Vector {  
    ...  
    public void move(Vector v) { }  
}
```

```
public class Vector3 extends Vector {  
  
    protected float z;  
  
    public void move(Vector3 vector3) { //это другой метод move  
        ...  
    }  
}
```

Коварианты

```
public class Vector {  
    ...  
    public Vector normalize() { ... }  
}
```

```
public class Vector3 extends Vector {  
  
    protected float z;  
  
    @Override  
    public Vector3 normalize() { ... }  
}
```

Абстрактные классы

```
public abstract class Animal {  
  
    private String name;  
    private int health;  
  
    public Animal() { ... }  
  
    public void eat(Food food) {  
        health +=  
food.getCalories();  
    }  
  
    public abstract void run();  
    public abstract void jump();  
  
}
```

~~Animal animal =
new Animal();~~

Даже если этих методов нет

Абстрактные классы

```
public class Crocodile extends Animal {  
  
    @Override  
    public void run() {  
        // do nothing  
    }  
  
    @Override  
    public void jump() {  
        // do nothing too  
    }  
}
```

Абстрактные классы

```
public class Kangaroo extends Animal {
```

```
    @Override
```

```
    public void run() {  
        while(true) jump();  
    }
```

```
    @Override
```

```
    public void jump() {  
        // code of jump  
    }
```

```
}
```

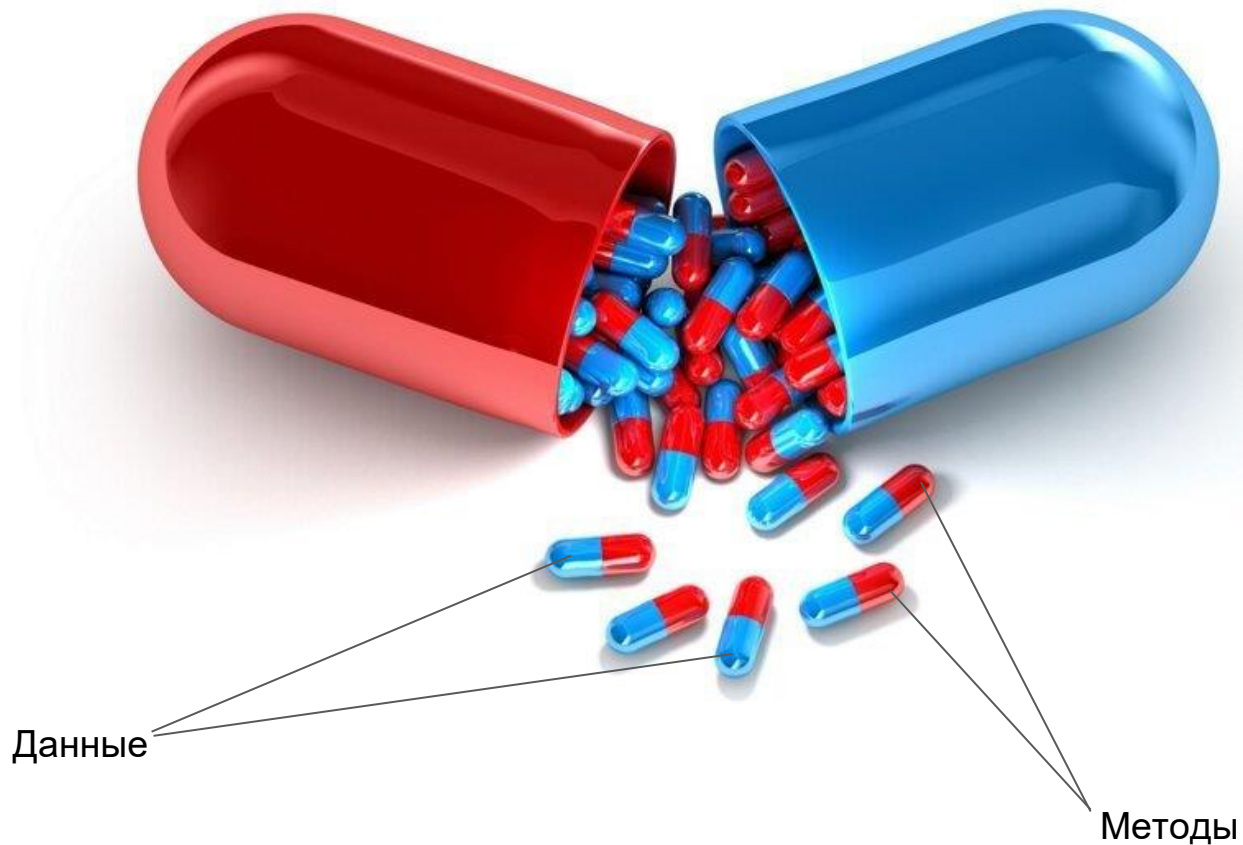
Абстрактные классы

```
public abstract class Felidae extends Animal {  
  
    public abstract void hunt();  
  
}
```

Абстрактные классы

```
public static void main(String[] args) {  
  
    Animal[] animals = new Animal[5];  
  
    // initialization array here  
    // animals[0] = new Crocodile();  
  
    for(Animal animal : animals) {  
        animal.run();  
    }  
}
```

Инкапсуляция



Это свойство объекта, объединяющее данные и методы, работающие с ними, в классе и скрыть детали реализации.

Инкапсуляция связана с понятием интерфейса класса. Всё, что не входит в интерфейс, инкапсулируется в классе.

Интерфейсы

1. Поведение объектов
2. Реализация интерфейса
3. Зачем интерфейсы, если есть классы?

Интерфейсы

```
public interface Felidae {  
    // кошачьи  
  
    public static final String DEFAULT_NAME = "Барсик";  
  
    void feed();  
    Reaction stroking();  
}
```

1. Что это вообще такое?
2. Все методы абстрактные
3. Все методы public
4. Все "поля" интерфейса могут быть только public static

Кошачьи

```
public class Cat implements Felidae {  
  
    private String name = Felidae.DEFAULT_NAME ;  
    private Size size = new Size(10, 20, 40, 22);  
  
    @Override  
    public void feed() {  
        this.size.increase();  
    }  
  
    @Override  
    public Reaction stroking() {  
        return new Reaction.POSITIVE;  
    }  
  
}
```


Кошачьи

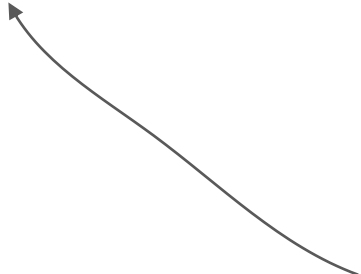
```
public class Lion implements Felidae {  
  
    private String name = "Симба";  
    private Size size = new Size(100, 200, 400, 220);  
  
    @Override  
    public void feed() {  
        this.size.increase();  
    }  
  
    @Override  
    public Reaction stroking() {  
        return new Reaction.TO_KILL_AND_EAT;  
    }  
  
}
```

Преимущества интерфейсов

1) Абстрагирование от реализации

```
class Human {  
  
    public void feedTo ( Felidae felidae ) {  
        felidae.feed();  
    }  
  
}
```

Чем это отличается от использования абстрактных классов?



Преимущества интерфейсов

2) Множественное наследование

```
public class Cat implements Felidae, Toy {  
  
    ...  
  
    @Override  
    public void play() {  
        this.size.decrease();  
    }  
  
}
```


Проверка типов объектов

```
public interface Toy {  
    void play();  
}
```

```
public class Cat implements Toy {  
  
    ...  
  
    @Override  
    public void play() {  
        this.size.decrease();  
    }  
  
    public void play(Cat cat) {  
        // ...  
    }  
}
```

```
Toy toy = new Cat();  
method(toy);
```

```
void method(Toy toy) {  
    toy.play();  
  
    toy.play(new Cat());  
}
```



Проверка типов объектов

```
Toy toy = new Cat();  
method(toy);
```

```
1. void method(Toy toy) {  
2.     toy.play();  
3.  
4.     if (toy instanceof Cat)  
5.         ((Cat)toy).play(new Cat());  
6. }
```

```
1. void method(Toy toy) {  
2.     toy.play();  
3.  
4.     if (toy.getClass() == Cat.class)  
5.         Cat cat = (Cat)toy;  
6.         cat.play(new Cat());  
7. }
```

Проверка типов объектов

```
Toy toy = new Cat();  
method(toy);
```

```
1. void method(Toy toy) {  
2.     toy.play();  
3.  
4.     if (toy instanceof Cat cat) {  
5.         cat.play(cat);  
6.     }  
7. }
```

Object

1. `int` hashCode
2. `boolean` equals
3. `Object` clone
4. `void` finalize
5. `String` toString
6. wait / notify / notifyAll



toString

```
public class Cat {  
  
    private String name = ...;  
  
    public String toString() {  
        return "Cat_" + name;  
    }  
}
```

```
public static void main (...) {  
    System.out.println( new Cat() );  
}
```

← Перегруженный метод

equals

```
public class Cat {  
  
    private String name = ...;  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Cat cat = (Cat) o;  
        return Objects.equals(name, cat.name);  
    }  
}
```

```
public static void main (...) {  
    Cat cat = new Cat();  
    System.out.println( cat.equals( new Cat()) );  
}
```

Enum

```
public class SignalTraffic {  
  
    private String color;  
    private boolean state;  
  
    public SignalTraffic(String color) {  
        this.color = color;  
    }  
  
    public void changeState() {  
        this.state = !state;  
    }  
}
```



Enum

```
public class Main {  
  
    public static void main(String ... args) {  
        SignalTraffic red = new SignalTraffic("RED");  
        red.change();  
    }  
}
```

Как переписать **SignalTraffic** так, чтобы существовало только три объекта с фиксированно заданными полями?

Enum

```
public enum SignalTraffic {  
  
    RED,  
    YELLOW,  
    GREEN  
  
}
```

```
public class Main {  
  
    public static void main(String ... args) {  
        SignalTraffic red = SignalTraffic.RED;  
  
    }  
  
}
```

Enum (upgrade#1)

```
public enum SignalTraffic {  
    RED ("красный"),  
    YELLOW ("желтый"),  
    GREEN ("зеленый");
```

```
    private String name;
```

```
    SignalTraffic(String name) { this.name = name; }
```

```
    public String getName();
```

```
}
```

поля
конструкторы
методы

```
public static void main(String ... args) {
```

```
    SignalTraffic red = SignalTraffic.RED;
```

```
    System.out.println("Color of signal = " + red.getName());
```

```
}
```

Enum (upgrade#2)

```
public enum SignalTraffic {  
    RED ("красный"),  
    YELLOW ("желтый"),  
    GREEN ("зеленый") {  
        public void blink() {}  
    };  
  
    private String name;  
    SignalTraffic(String name) { this.name = name; }  
    public String getName();  
  
        public void glow() { }  
}
```

Enum (upgrade#3)

```
public enum SignalTraffic {  
    RED ("красный"),  
    YELLOW ("желтый") {  
        @Override  
        public void glow() { }  
    },  
    GREEN ("зеленый");  
  
    ...  
  
    public void glow() { }  
}
```

Enum (upgrade#4)

```
public enum SignalTraffic implements Glowable {  
    RED ("красный"),  
    YELLOW ("желтый") {  
        @Override  
        public void glow() { }  
    },  
    GREEN ("зеленый");  
  
    ...  
  
    public void glow() { }  
}
```

The diagram consists of two curved arrows. The first arrow originates from the `implements` keyword in the first line of the code and points to the `@Override` annotation in the `YELLOW` enum constant's `glow()` method. The second arrow originates from the `implements` keyword and points to the `public void glow()` method of the unnamed enum constant at the bottom of the list.

Enum под капотом

```
public class SignalTraffic extends Enum {  
    SignalTraffic RED = new SignalTraffic("красный");  
    SignalTraffic YELLOW = new SignalTraffic("желтый");  
    SignalTraffic GREEN = new SignalTraffic("зеленый");  
  
    ...  
  
}
```

Как следствие:

- от enum нельзя наследоваться
- Enum уже притаскивает некоторые методы по наследству:
 - name
 - ordinal
 - valueOf
 - values