



Faculty of Software Engineering and Computer Systems

Programming

Lecture #3.
OOP. Continue.

Instructor of faculty
Pismak Alexey Evgenievich
Kronverksky Pr. 49, 1331 room
pismak@itmo.ru

Saint-Petersburg

В предыдущей серии

- Собственные типы данных - классы
- Наследование
- Реализация классов
- Создание экземпляров класса
- Конструкторы
- Модификаторы доступа



В этой серии

- static
- this
- super
- Блоки инициализации
- Полиморфизм
- Инкапсуляция
- Интерфейсы
- Класс Object
- Интерфейсы/перечисления
- Абстрактные классы



‘static’

```
public class Guy {
```

```
    public static Guy createClone(Guy guy) {  
        Guy clone = new Guy();  
        clone.name = guy.name;  
        clone.age = guy.age;  
        return clone;  
    }
```

```
// где-то в main методе
```

```
Guy myGuy = new Guy();  
Guy cloneOfMyGuy = Guy.createClone(myGuy);
```

```
// new Guy().createClone()
```

```
}
```

Why we wrote *'static'*

```
public class Main {  
  
    public static void main(String args[]) {  
  
    }  
}
```

JVM:
Main app = new Main();
app.main(args)

Why we wrote *'static'*

```
public class Main {
```

```
    public Main(int something) { ... }
```

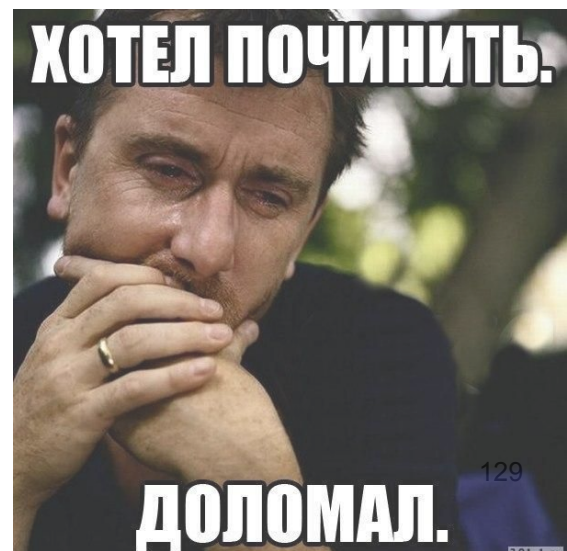
```
    public static void main(String args[]) {
```

```
    }
```

```
}
```

JVM:

```
Main app = new Main(); // Error
```



Why we wrote *'static'*

```
public class Main {  
  
    public Main(int something) { ... }  
  
    public static void main(String args[]) {  
    }  
}
```

JVM:
Main.main(args)

static fields / static methods

Math.*PI* / Math.*E* / Math.*sin*() / Math.*log*()

System.*out* / System.*in* / System.*currentTimeMillis*()

Byte.*MIN_VALUE* / Byte.*valueOf*() / Byte.*parseValue*()

```
class Item {
```

```
    private static int itemID = 0;
```

```
    public Item() { itemID++; }
```

```
}
```


‘this’ and ‘super’

```
package ru.ifmo.se.prog.examples;
```

```
public class Human {
```

```
    String name;                // имя  
    boolean isMale;            // пол. true если муж.  
    int age;                    // возраст  
    // ... остальные свойства
```

```
    public Human() {  
        // сначала бы вызвать родительский конструктор...  
    }  
    public void applySkill(int age) { }  
    // ... остальные методы
```

```
}
```

‘this’ and ‘super’

```
package ru.ifmo.se.prog.examples;
```

```
public class Human {
```

```
    String name;                // имя
    boolean isMale;             // пол. true если муж.
    int age;                    // возраст
    // ... остальные свойства
```

```
    public Human() {
        // сначала бы вызвать родительский конструктор...
    }
```

```
    public void applySkill(int age) { this.age = age; }
```

```
}
```

‘this’ and ‘super’

```
package ru.ifmo.se.prog.examples;
```

```
public class Human {
```

```
    String name;
```

```
// имя
```

```
    boolean isMale;
```

```
// пол. true если муж.
```

```
    int age;
```

```
// возраст
```

```
    // ... остальные свойства
```

```
    public Human() {}
```

```
    public Human(int age) {
```

```
        this();
```

```
    }
```

```
    public void applySkill(int age) { this.age = age; }
```

```
}
```

‘this’ is a first implicit method parameter

```
package ru.ifmo.se.prog.examples;
```

```
public class Human {  
    String name;
```

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

```
    public void setName(Human this, String name) {  
        this.name = name;
```

```
    }
```

```
}
```

‘this’ and ‘super’

```
package ru.ifmo.se.prog.examples;
```

```
public class BadBoy extends Human {
```

```
    public BadBoy() {  
        super("Name");  
    }
```


```
}
```

Init blocks

```
1. public class A {  
2.     public A() {  
3.         System.out.println("A constr");  
4.     }  
5.  
6.     {  
7.         System.out.println("A block");  
8.     }  
9. }
```

```
1. public class B extends A {  
2.     public B() {  
3.         System.out.println("B constr");  
4.     }  
5.  
6.     {  
7.         System.out.println("B block");  
8.     }  
9. }
```

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



A block
A constr
B block
B constr

Init blocks

```
1. public class Main {  
2.  
3.     private static int field;  
4.  
5.     public static void main (String... s) {  
6.         System.out.println("main");  
7.     }  
8.  
9.     static {  
10.         field = 10;  
11.         main(null);  
12.         System.exit(-1);  
13.     }  
14.  
15. }
```

Polymorphism



Кто умеет пользоваться этой штукой?

Polymorphism



Кто умеет пользоваться этой штукой?

Polymorphism



Кто умеет пользоваться этой штукой?

Polymorphism



Кто умеет пользоваться этой штукой?

Polymorphism



Кто умеет пользоваться этой штукой?

Polymorphism



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

Method overriding

```
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) { }  
}
```

Method overloading

```
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  
        ...  
    }  
  
}
```

Covariance

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

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


Abstract classes

```
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

Abstract classes

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

Abstract classes

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

```
    @Override
```

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

```
    @Override
```

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

```
}
```

Abstract classes

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

Abstract classes

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

ПОЛИМОРФИЗМ

Final modifier

`final class` // cannot be **extended**

`final method()` // cannot be **overridden**

`final variable` // cannot be **reassigned**

Encapsulation



Encapsulation



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

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

Interfaces

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

Interfaces

```
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

Felines (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;  
    }  
  
}
```

Felines (Кошачьи)

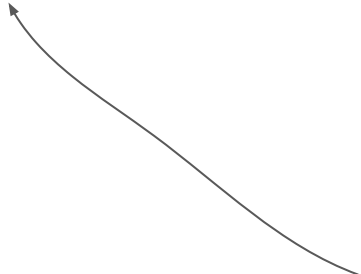
```
public class Lion implements Felidae {  
  
    private String name = "Симба";  
    private Size size = new Size(100, 200, 400, 220);  
  
    @Override  
    public void feed() {  
        this.size.increase();  
    }  
  
    @Override  
    public Reaction stroking() {  
        return new Reaction.TO_KILL_AND_EAT;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

Advantages of interfaces

1) Абстрагирование от реализации

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

Чем это отличается от использования абстрактных классов?



Advantages of interfaces

2) Множественное наследование

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

Interface methods modifiers

1. abstract
2. default
3. static
4. private


Check objects type

```
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());  
}
```



Check objects type

```
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. }
```

Check objects type

```
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



ВОПРОСЫ НА
ЛАБОРАТОРНЫХ

toString

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

“Cat@382bf1c8”

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

← Перегруженный метод

equals

```
public class Cat {  
  
    cat == new 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()) );  
}
```

hashCode()

```
public class Cat {  
  
    private String name = ...;  
    private int age = ...;  
  
    public int hashCode() {  
        return (name.hashCode() << 4) + age;  
    }  
}
```

a.equals(b) == true

a.hashCode == b.hashCode()

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

empty class ?

```
public class MyClass
```

```
}
```


implicit code

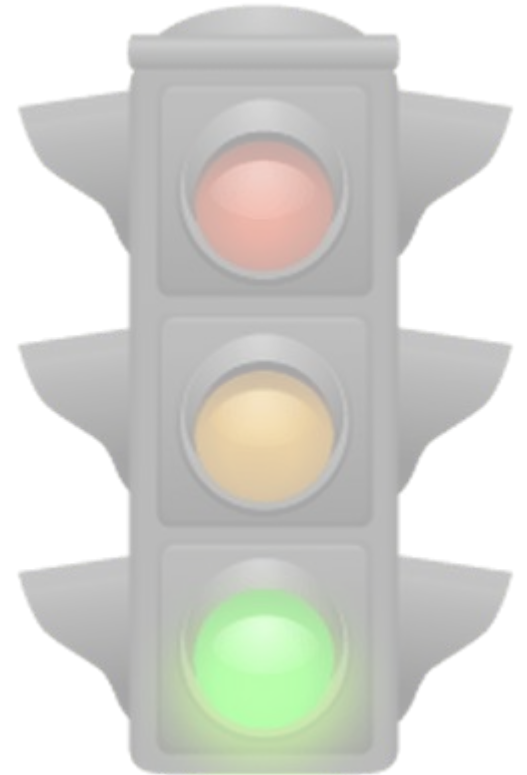
```
import java.lang.*;

public class MyClass extends Object {

    public MyClass() {
        super();
    }
}
```

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() { }  
}
```

A diagram consisting of two curved arrows. The first arrow starts 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 starts from the `implements` keyword and points to the `public void glow()` method at the bottom of the enum, illustrating that the enum implements the `Glowable` interface.

Enum “under the hood”

```
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