



Faculty of Software Engineering and Computer Systems

Programming

Lecture #4.
SOLID, X-classes

Instructor of faculty
Pismak Alexey Evgenievich
Kronverksky Pr. 49, 1331 room

pismak@itmo.ru

Saint-Petersburg

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;
```

```
    }
```

```
}
```

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

```
    System.out.println( new Cat() );
```

```
}
```

“Cat@382bf1c8”

Перегруженный метод

equals

```
public class Cat {  
    private String name = ...;                                cat == new Cat()  
    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();
    }
}
```

Принципы ОО проектирования

- DRY – Don't Repeat Yourself
- KISS – Keep It Simple, Stupid (Keep It Short & Simple)
- YAGNI – You Aren't Gonna Need It
- SOLID

S.O.L.I.D. принципы

- Принцип единственной ответственности
- Принцип открытости/закрытости
- Принцип подстановки Барбары Лисков
- Принцип разделения интерфейса
- Принцип инверсии зависимостей

SOLID (Single Responsibility Principle)



Проблемы при использовании ООП

SOLID (Single Responsibility Principle)

A module should have one, and only one, reason to change

Модуль **должен** иметь только одну причину для изменения



Модуль **должен** быть ответственным только за одного актора

A module should be responsible to one, and only one, actor

SOLID

```
1. class Person{
2.     public void eat(){ };
3.     public void walk(){ };
4.     public void run(){ };
5.     public void driveCar(Car car){ };
6.     public void createOcean() {};
7.     public void beSomething() {};
8. }
```

SOLID

```
1. class Book {  
2.     public String getName(){ };  
3.     public String getTitle(){ };  
4.     public String getText(int start, int limit){ };  
5.     public void printPage(int page){ };  
6. }
```

SOLID

```
1. class Book {
2.     public String getName(){ };
3.     public String getTitle(){ };
4.     public String getText(int start, int limit){ };
5. }
6.
7. class BookPrinter {
8.     public void printPage(Book book, int page){ };
9. }
10. class BookReader {
11.     public void nextPage(int page){ };
12.     public void prevPage(int page){ };
13. }
```

SOLID (Open-Closed principle)



SOLID (Open-Closed principle)

A software artifact should be open for extension but closed for modification

Элемент ПО **должен** быть открыт для расширения, но закрыт для изменения

SOLID

```
1. class Animal {  
2.     private int speed;  
  
3.     public int  getSpeed() { };  
4.     public void setSpeed() { };  
5.     public void run()  {};  
6. }
```

* в хорошо спроектированных программах новая функциональность вводится путем добавления нового кода, а не изменением старого, уже работающего

SOLID

```
1. class Book {
2.     public String getName(){ };
3.     public String getTitle(){ };
4.     public String getText(int start, int limit){ };
5. }
6.
7. class PaperBook extends Book {
8.     public int numberOfPages(){ };
9. }
10. class ElectornicBook extends Book {
11.     public void charge(){ };
12. }
```

SOLID (Liskov Substitution Principle)



SOLID (Liskov Substitution Principle)

Subclasses should be substitutable for their base classes

Подклассы **должны** подставляться на место их базового класса

SOLID

```
1. class Duck {  
2.     public int  swim() { }  
3. }
```

```
1. class ToyDuck extends Duck {  
2.     private Battery battery = ...  
3.     public int  swim(){  
4.         if (battery.isCharged())  
5.         }  
6. }
```

```
1. Duck[] ducks = // init array different ducks  
  
2. for(Duck duck : ducks) {  
3.     duck.swim();  
4. }
```

SOLID

```
1. class Rectangle {
2.     double a; double b;
3.     public double area() {
4.         return a * b;
5.     }
6.     public void setA(double a) {}
7.     public void setB(double b) {}
8. }
```

```
1. class Square extends Rectangle {
2.     public Square(double a) {
3.         this.a = this.b = a;
4.     }
5.     public void setA(double a) {
6.         super.setA(a);
7.         this.b = a; // ???????????
8.     }
9. }
```

```
1. Rectangle square = new Square(5.0);
2. square.setB(4.0);
3. square.setA(3.0);
4. assert (square.area() == 12.0);
```

SOLID (Interface Segregation Principle)



SOLID (Interface Segregation Principle)

Many client specific interfaces are better than one general purpose interface

Много специализированных интерфейсов **лучше** одного универсального

SOLID

```
1. interface Animal {
2.     public void eat();
3.     public void run();
4. }
5.
6. class Bat implements Animal {
7.     public void run() {
8.         // cannot run
9.     }
10.    public void fly() {...}
11.    public void eat() {...}
12.}
```

```
1. interface Bird {
2.     public void sing();
3.     public void fly();
4. }
5.
6. class Emu implements Bird {
7.     public void sing() {
8.         // cannot sing
9.     }
10.    public void fly() {
11.        // cannot fly
12.    }
13.    public void run() {...}
```

SOLID

```
1. interface Animal{
2.     public void eat();
3.     public void run();
4. }
5.
6. class Bat implements Animal {
7.     public void run() {
8.         // cannot run
9.     }
10.    public void fly() {...}
11.    public void eat() {...}
12.}
```

```
1. interface Eatable { void eat(); }
2. interface Runnable { void run(); }
3. interface Flyable { void fly(); }
4. interface Singable { void sing(); }
```

```
1. interface Bird{
2.     public void sing();
3.     public void fly();
4. }
5.
6. class Emu implements Bird {
7.     public void sing() {
8.         // cannot sing
9.     }
10.    public void fly() {
11.        // cannot fly
12.    }
13.    public void run() {...}
```

SOLID (Dependency Inversion Principle)

High-level modules should not depend on low-level modules.

Both should depend on abstractions.

Abstractions should not depend on details.

Details should depend on abstractions.

Модули верхнего уровня **не должны** зависеть от модулей нижнего уровня.

И те, и другие **должны** зависеть от абстракций.

Абстракции **не должны** зависеть от реализации.

Реализации **должны** зависеть от абстракций.

SOLID (Dependency Inversion Principle)



SOLID (Dependency Inversion Principle)

```
1. public class Car {  
2.     private Wheel wheel = new Wheel();  
3.     public void go() {  
4.         wheel.rotate();  
5.     }  
6. }
```

SOLID (Dependency Inversion Principle)

```
1. public class Car {  
2.     private Wheel wheel;  
3.     public Car(Wheel wheel) {  
4.         this.wheel = wheel;  
5.     }  
6.  
7.     public void go() {  
8.         wheel.rotate();  
9.     }  
10. }
```

SOLID (Dependency Inversion Principle)

```
1. class SportWheel extends Wheel {
2.     @Override
3.     public void rotate() {
4.         // burn ground
5.     }
6. }

1. public static void main(String[] args) {

2.     Car car = new Car(new Wheel());
3.     Car sportcar = new Car(new SportWheel());

4. }
```

SOLID (Dependency Inversion Principle)

```
1. class BachelorStudent {  
2.     public void doProgrammingExam(Programming prog) {  
3.         prog.getExam().getTasks();  
4.     }  
5. }  
6. class Programming { public Exam getExam() }  
7. class Control { public Task[] getTasks() }  
8. class Task { ... }
```


SOLID (Dependency Inversion Principle)

```
1. class Student {  
2.     public void doFinalControl(Subject subject) {  
3.         subject.getControl().getTasks();  
4.     }  
5. }  
6. class Subject { public Control getControl() }  
7. class Control { public Task[] getTasks() }  
8. class Task { ... }
```

```
1. class BachelorStudent extends Student { }  
2. class MasterStudent extends Student { }  
3. class Programming extends Subject { }  
4. class Exam extends Control { }  
5. class MultipleChoice extends Task { }
```

If we don't use DI we are limited by small possibilities



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() { }  
}
```



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

Record'ы

```
public record Animal (String name) {}
```

Record'ы

```
public record Animal(String name){}
```

```
public final class Animal {  
    private final String name;  
    public Animal(String name) {  
        this.name = name;  
    }  
    public String name() {  
        return name;  
    }  
    public int hashCode() {  
        return Objects.hashCode(name);  
    }  
    public boolean equals(Object obj) {  
        return Objects.equals(this, obj);  
    }  
    public String toString() {  
        return getClass().getName() + "(name=" + name + ")";  
    }  
}
```

Record'ы

```
public record Animal(String name){}
```

```
Animal myCat = new Animal("Barsik");  
System.out.println(myCat.name());  
System.out.println(myCat);
```

- Record не может наследоваться от какого-нибудь класса, но может реализовывать интерфейсы
- У Record не может быть других полей объекта, кроме тех, которые объявлены в конструкторе при описании класса (да, это конструктор по умолчанию, кстати). Статические — можно.
- Поля неявно являются финальными. Объекты неявно являются финальными. Со всеми вытекающими, вроде невозможности быть абстрактными.
- От Record нельзя наследоваться, т.к. он “final class”⁰

Элементы класса (members)

```
classClazz {
```

```
private int field;
```

```
protected void method() { }
```

```
static int staticField;
```

```
public static void staticMethod() { }
```

```
}
```

members

Переменные внутри метода

```
classClazz {  
    protected void method(int parameter) {  
        double localVar1 = 0.0;  
        var lovalVar2 = "Hello";  
        for (var count = 0; count < 10; count++) {  
            int answer = 42;  
        }  
    }  
}
```

local variables

Вложенные классы (Nested classes)

```
class Outer {  
    class Nested { }  
}
```


Внутренние, локальные, анонимные

```
class Outer {  
    static class StaticNested { }  
    class Inner { }  
    void method() {  
        class Local { }  
        (new Outer()  
            { void method() { println(); } } // Anonymous  
        ).method();  
    }  
}
```

Внутренние классы (Inner)

```
public class Car {
```

```
    private Wheel backRightWheel;
```

```
    private Wheel backLeftWheel;
```

```
    private void crash() { }
```

```
    public class Wheel { // доступны все модификаторы доступа
```

```
        public void rotate(float angle) {
```

```
            // ....
```

```
        }
```

```
        public void crash() {
```

```
            // ....
```

```
        }
```

```
    }
```

```
}
```

Внутренние классы (Inner)

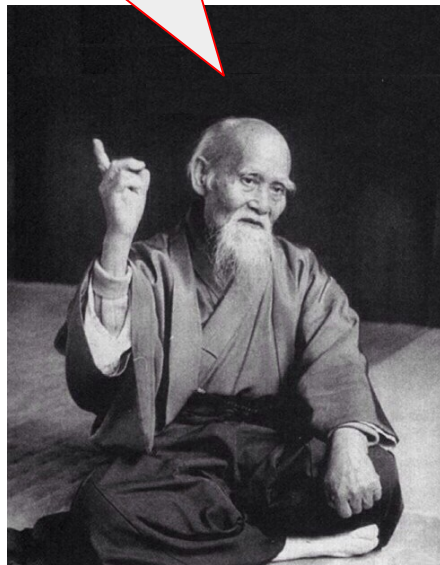
// создание экземпляра внешнего класса

```
Car car = new Car();
```

// Создание экземпляра внутреннего класса

```
Car.Wheel wheel = car.new Wheel();
```

Всё, что не static, требует существования экземпляра класса!



Внутренние классы (Inner)

```
public class Car {
```

```
    private Wheel backRightWheel;  
    private Wheel backLeftWheel;
```

```
    private void crash() {}
```

```
    public class Wheel {
```

```
        public void rotate(float angle) {  
            // ....  
        }
```

```
        public void crash() {  
            // ....  
        }
```

```
    }
```

```
}
```

Пусть поведение машины
“ломаться” будет закрытым

А колесо можно повредить чем-то
снаружи, и по этой причине сделаем
это поведение открытым

Как тогда из метода `Wheel.crash()` вызвать метод `Car.crash()`?

Внутренние классы (Inner)

```
public class Car {
```

```
    private Wheel backRightWheel;
```

```
    private Wheel backLeftWheel;
```

```
    private void crash() {}
```

```
    public class Wheel {
```

```
        public void rotate(float angle) {
```

```
            // ....
```

```
        }
```

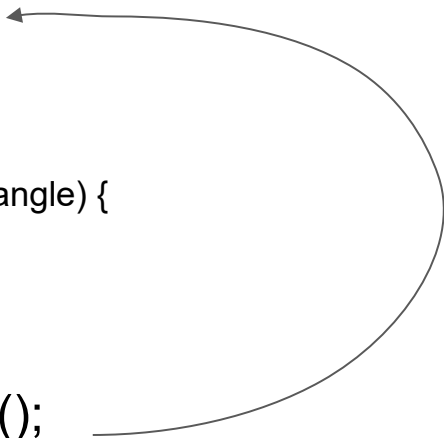
```
        public void crash() {
```

```
            Car.this.crash();
```

```
        }
```

```
    }
```

```
}
```



Статические вложенные классы

```
public class Car {
```

```
    // ....
```

```
    public static class BadAir {
```

```
        public void generate() {
```

```
            // ....
```

```
        }
```

```
    }
```

```
}
```

Статические вложенные классы



Для создания объекта внутреннего класса не нужен объект внешнего класса

Из объекта вложенного класса нельзя обращаться к не статическим членам внешнего класса, Карл

Статические вложенные классы

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Car.Wheel wheel = new Car.Wheel();  
  
    }  
  
}
```


Локальные классы

```
public class Main {
```

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

```
        class TheBestPlaceForUselessClassDeclaration {
```

```
            // ....
```

```
        }
```

```
        TheBestPlaceForUselessClassDeclaration tbpfucd =  
            new TheBestPlaceForUselessClassDeclaration();
```

```
    }
```

```
}
```

Локальные классы 2

```
public class Main {  
  
    public static void main(String[] args) {  
  
        if (args.length == 0) {  
            class TheBestPlaceForUselessClassDeclaration {  
                // ....  
            }  
            TheBestPlaceForUselessClassDeclaration tbpfucd =  
                new TheBestPlaceForUselessClassDeclaration();  
        }  
  
    }  
  
}
```

Локальные классы 3

```
public class Main {  
  
    public static void main(String[] args) {  
  
        while (true) {  
            class TheBestPlaceForUselessClassDeclaration {  
                // ....  
            }  
            TheBestPlaceForUselessClassDeclaration tbpfucd =  
                new TheBestPlaceForUselessClassDeclaration();  
        }  
    }  
}
```

Локальные классы 4

```
public class Main {
```

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

```
    }
```

```
    // блок инициализации
```

```
    {
```

```
        class TheBestPlaceForUselessClassDeclaration {
```

```
            // ....
```

```
        }
```

```
        TheBestPlaceForUselessClassDeclaration tbpfucd =  
            new TheBestPlaceForUselessClassDeclaration();
```

```
    }
```

```
}
```

Локальные классы

- Модификатор доступа не указывается
- Невозможно объявление статических методов (и любых иных статических членов), но
- Возможно использование статических констант
- Захват внешних локальных переменных возможен, если они определены, как **effectively final**
- Не могут быть статическими



Анонимные классы

```
public interface Runnable {  
    void run();  
}
```

```
public class Runner {
```

```
    public void start (Runnable instance) {  
        instance.run();  
    }
```

```
    public void test() {
```

```
        // тут хочется вызвать метод start, но реализацию  
        // Runnable писать лениво и нет необходимости
```

```
    }
```

```
}
```

Анонимные классы

```
public interface Runnable {  
    void run();  
}
```

```
public class Runner {
```

```
    public void start (Runnable instance) {  
        instance.run();  
    }
```

Вызов метода start

```
    public void test() {  
        start ( new Runnable() {
```

```
            public void run() { }
```

```
        } );
```

```
    }
```

```
}
```

Анонимные классы

```
public interface Runnable {  
    void run();  
}
```

```
public class Runner {
```

```
    public void start (Runnable instance) {  
        instance.run();  
    }
```

```
    public void test() {  
        start ( new Runnable() {
```

```
            public void run() { }
```

```
        } );
```

```
    }
```

```
}
```

Вызов метода start

Создание объекта Runnable на лету и передача его в качестве параметра методу start

Анонимные классы

```
public interface Runnable {  
    void run();  
}
```

```
public class Runner {
```

```
    public void start (Runnable instance) {  
        instance.run();  
    }
```

```
    public void test() {  
        start ( new Runnable() {
```


```
            public void run() { }
```

```
        } );
```

```
    }
```

```
}
```

Переопределение run,
объявленного в
интерфейсе Runnable



Почему анонимные?

```
public interface Runnable {  
    void run();  
}
```

```
public class Runner {
```

```
    public void start (Runnable instance) {  
        instance.run();  
    }  
}
```

```
    public void test() {  
        start ( new Runnable() {  
            public void run() { }  
        } );  
    }  
}
```

Анонимный класс - это аналог локального класса. Разница только в наличии у класса имени для повторного использования

```
    public void test() {
```

```
        class X implements Runnable {  
            public void run() { }  
        };  
        start ( new X() );  
    }
```



Анонимные классы

```
public class Car {  
    void go() { }  
}
```

```
public class Main {
```

```
    public static void main(String[] s) {  
        start ( new Car() {
```

```
        {  
            // инициализация нового объекта  
            // внедряя код в блок инициализации  
        }  
    }  
};
```

```
    }
```

```
}
```

Это не обязательно
интерфейсы

возможно не только переопределение
методов, но и использование блоков
инициализации, “**добавление**” полей
и методов

Анонимные классы

```
public class Car {  
    void go() { }  
}
```

```
public class Main {
```

```
    public static void main(String[] s) {  
        start ( new Car() {  
            {  
                // инициализация  
                // нового объекта  
                // внедряя код в блок  
                // инициализации  
            }  
        } );  
    }
```



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

```
    class X extends Car {  
        {  
            // инициализация  
            // нового объекта  
            // внедряя код в блок  
            // инициализации  
        }  
    };  
    start ( new X() );  
}
```

Вложенные классы (и интерфейсы)

- Nested – объявлен внутри класса (или интерфейса)
 - доступны элементы окружающего (даже приватные)
- Static nested – аналогичен классу верхнего уровня
 - доступны статические элементы окружающего
- Inner = non-static nested
 - Для создания нужен экземпляр окружающего
- Local – объявлен внутри метода или блока
 - Доступен только внутри блока
- Anonymous – local без имени

Вложенные классы (и интерфейсы)

- Anonymous – local без имени
 - Объявление с созданием единственного экземпляра
 - Расширяет класс или реализует интерфейс
 - Может переопределить методы класса или интерфейса
 - Не может быть статическим
 - Только неявный конструктор

???

