



Faculty of Software Engineering and Computer Systems

Programming

Lecture #4.

X-classes, exceptions,
functional programming.

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

Saint-Petersburg

S.O.L.I.D. принципы

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

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 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 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 (Interface Segregation 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 lada = new Car(new Wheel());
3.     Car notLada = new Car(new SportWheel());

4. }
```

Вложенные классы

```
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() {
```

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

```
        }
```

```
    }
```

```
}
```

Вложенные классы

// создание экземпляра внешнего класса

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

// Создание экземпляра внутреннего класса

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

Вложенные классы

```
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()`?

Вложенные классы

```
public class Car {
```

```
    private Wheel backRightWheel;
```

```
    private Wheel backLeftWheel;
```

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

```
    public class Wheel {
```

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

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

```
        }
```

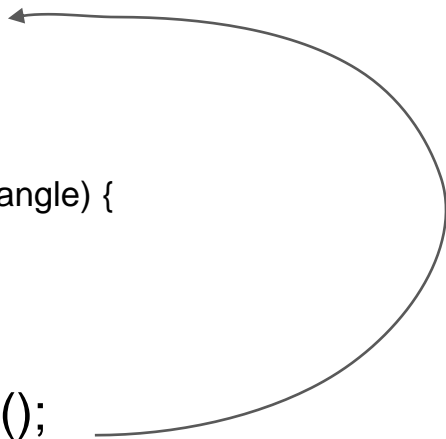
```
        public void crash() {
```

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

```
        }
```

```
    }
```

```
}
```



Внутренние классы

```
public class Car {
```

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

```
    public static class BadAir {
```

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

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

```
        }
```

```
    }
```

```
}
```

Внутренние классы (static)

```
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

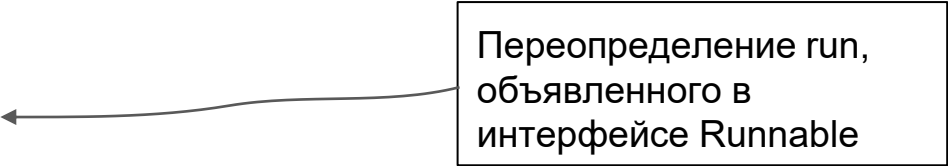
```
public interface Runnable {  
    void run();  
}
```

```
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() );  
}
```

Record'ы

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

```
@Override  
public int hashCode() {  
    return name != null ? name.hashCode() : 0;  
}  
  
@Override  
public String toString() {  
    return "Animal{" +  
        "name='" + name + '\\'' +  
        "'}";  
}
```

Record'ы

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

- 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.

С. Д. Руднев, [@s_d_rudnev](#) “final by default”

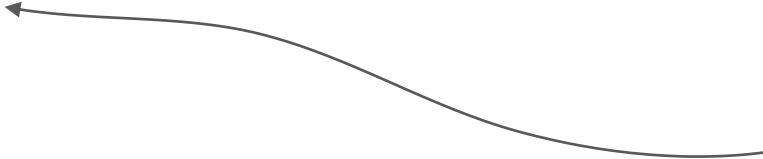
Исключения (exceptions)

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

```
    int x = .... // читаем из файла
```

```
    int y = 5 / x;
```

```
}
```



Если вдруг **x** окажется равным **0**, то что нам предложит объектно-ориентированная Java ?

Исключения (exceptions)

```
public static void main(String[] s) {  
  
    try {  
  
        int x = .... // читаем из файла  
        int y = 5 / x;  
  
    } catch (ArithmeticException e) {  
        // обработка  
    }  
  
}
```

Java предлагает порождать объекты определенных типов данных, которые будут содержать состояние и поведение возникшей ошибки.

А еще в языке имеется ряд ключевых слов для того, чтобы оперировать такими объектами, обрабатывать их генерацию и т.д.

Исключения (exceptions)

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

```
try {
```

Ключевое слово,
предваряющее блок кода

```
int x = .... // читаем из файла  
int y = 5 / x;
```

Блок кода, который необходимо
выполнить

```
} catch (ArithmeticException e) {
```

```
    // обработка
```

Если в блоке возникнет
ошибка, то управление
будет передано блоку *catch*

```
} finally {  
    // обработка  
}
```

Блок *finally* выполняется в
любом случае: была
ошибка или нет. Он не
является обязательным

```
}
```


Исключения (exceptions)

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

```
    try {
```

```
        int x = .... // читаем из файла
```

```
        int y = 5 / x;
```

```
    } catch (ArithmeticException e) {
```

```
        // обработка
```

```
    } finally {
```

```
        // обработка
```

```
    }
```

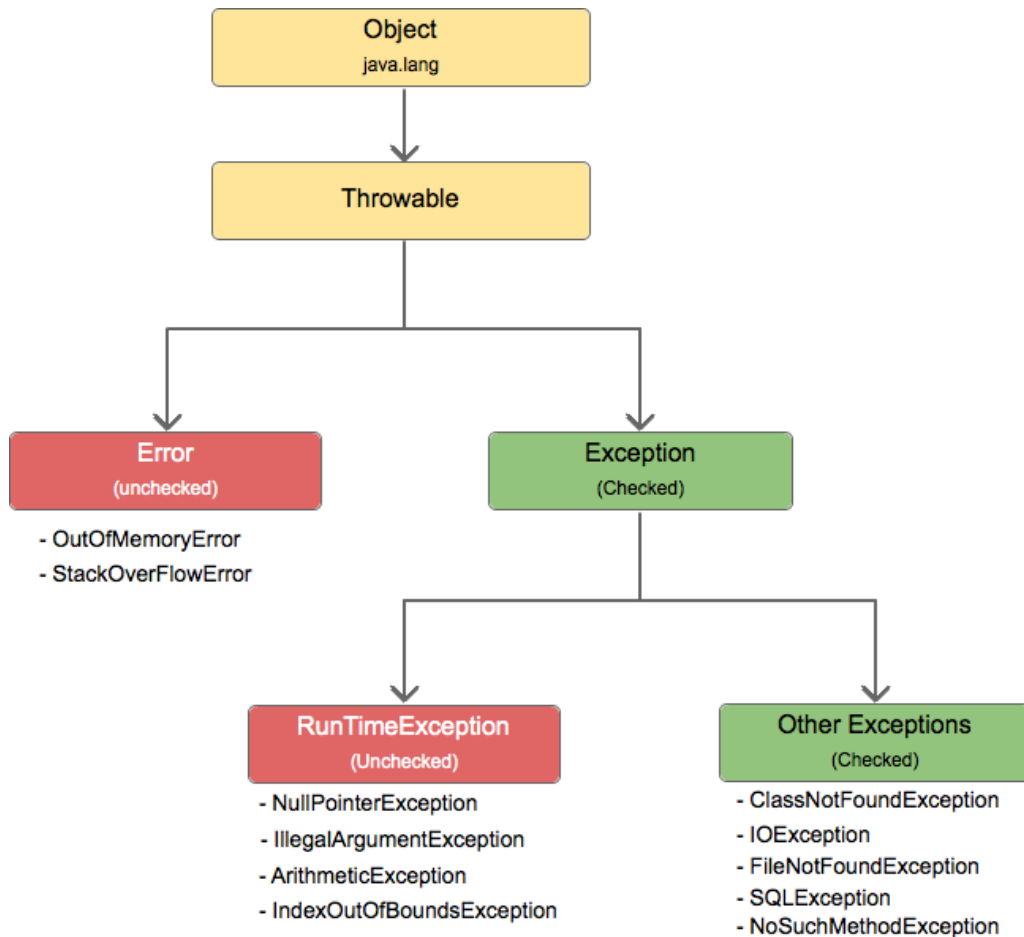
```
}
```

Каждая такая исключительная ситуация в программе генерирует объект своего типа

Если посмотреть внимательно на то, как устроен блок catch, то можно заметить, что он похож на объявление метода с именем catch и одним аргументом с именем e

Как Java понимает какие типы данных являются исключениями, а какие нет?

Исключения и их классификация



Собственные исключения

```
public class TheBestException extends Exception {
```

```
// можно переопределить ранее объявленные методы
```

```
// можно добавить свои поля, методы, конструкторы
```

```
}
```

Определяем класс, который
будет наследником класса
Throwable

Затем используем его в
нужном месте
оператором throw

```
public void someMethod() {
```

```
TheBestException tbe =  
    new TheBestException();  
    throw tbe;
```

```
}
```

Исключения (exceptions)

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

```
    try {
```

```
        someMethod();
```

```
    } catch (TheBestException e) {
```

```
        // обработка
```

```
    } finally {
```

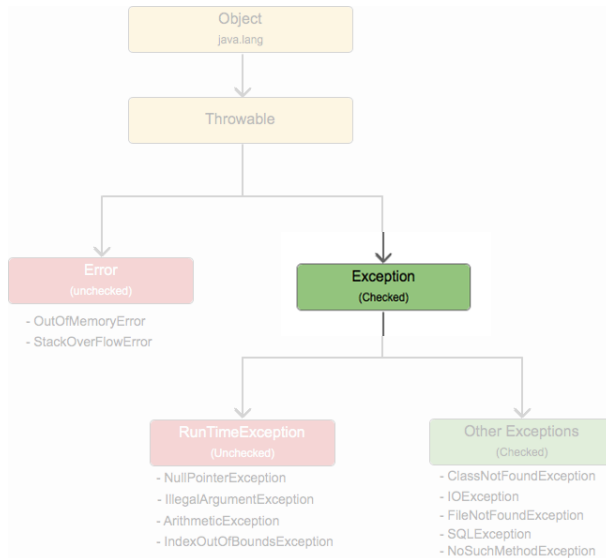
```
        // обработка
```

```
    }
```

```
}
```

Классификация в действии (checked)

```
public class ExceptionExample {  
  
    public void someMethod() throws Exception {  
  
        Exception ex = new Exception();  
        throw ex;  
  
    }  
}  
  
public static void main(String[] s) {  
    ExceptionExample ee = new ExceptionExample();  
    try {  
        someMethod();  
    } catch (Exception e) {  
        // обработка  
    }  
}
```



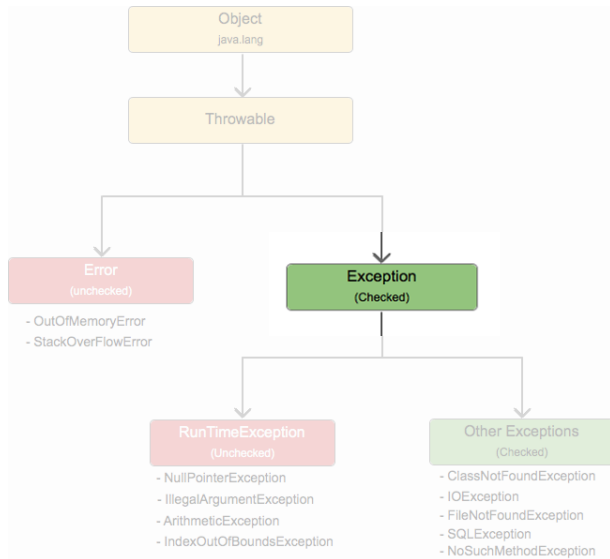
Коварианты инструкции throws

```
public class MyException extends Exception { }
```

```
public class ExceptionGen extends ExceptionExample {
```

```
    public void someMethod() throws MyException {  
        MyException ex = new MyException();  
        throw ex;  
    }  
}
```

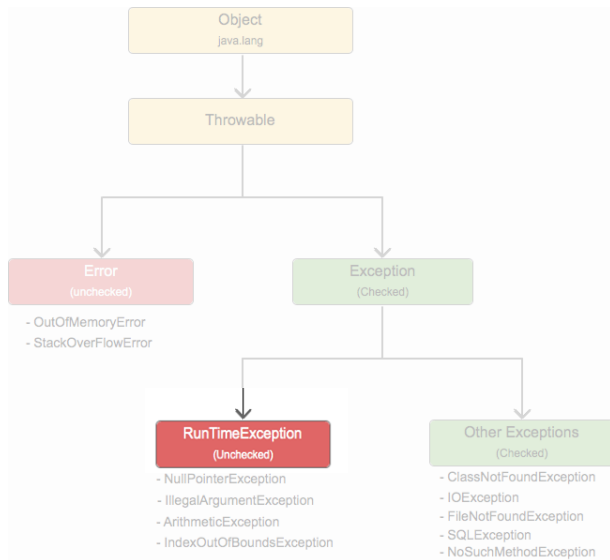
```
public static void main(String[] s) {  
    ExceptionExample ee = new ExceptionGen();  
    try {  
        someMethod();  
    } catch (Exception e) {  
        // обработка  
    }  
}
```



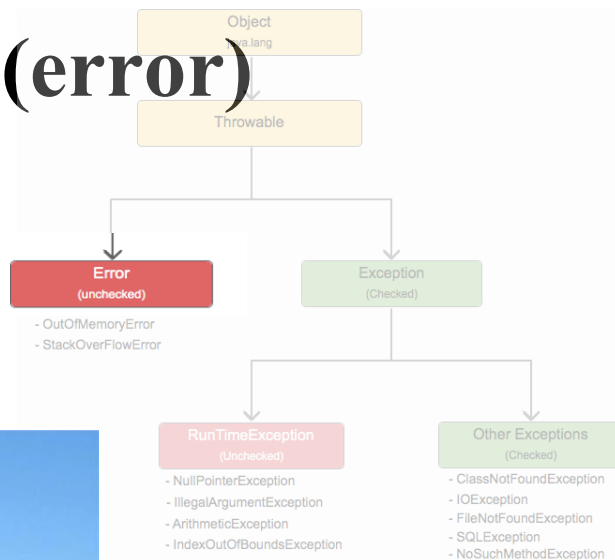
Классификация в действии (unchecked)

```
public class MyException extends RuntimeException { }
```

```
public static void main(String[] s) {  
    ExeptionExample ee = new ExceptionGen();  
    try {  
        someMethod();  
    } catch (Exception e) {  
        // обработка  
    }  
}
```



Классификация в действии (error)



Multiple catch

```
public class ExceptionExample {  
  
    public void method() {  
        try {  
            method();  
        } catch (Exception1 | Exception2 ex) {  
            // do nothing  
        }  
    }  
}
```

```
public class ExceptionExample {  
  
    public void method() throws Exception1, Exception2 {  
        method();  
    }  
}
```