4️⃣

Лаба 4

Теория:

Вложенные классы

Исключения

Рефлексия

- Throwable и Exception и все их наследники (за исключением наследников Error-a и RuntimeException-a) — checked
- Error и RuntimeException и все их наследники — unchecked
- класс Class, integer Hashcode

Вложенные классы

Вложенные классы (те которые находятся внутри другого класса) выглядят так:

```
class OuterClass { //внешний класс
    ...
        //два вложенных класса
    class InnerClass { //вложенный класс, но не статический,
        ...           //экземпляр, который нужно создавать
    }
    static class StaticNestedClass { //статический вложенный,
        ...                          //продолжением внешнего
    }
}
```

- Смысл вложенных классов:
    - **группируешь все классы которые используешь только в одном месте**
    - **увеличиваешь инкапсуляцию**, потому что **из вложенного класса можно обращаться ко всем приватным переменным внешнего класса**
    - **код ближе друг к другу и не разбросан по файлам**
- Вложенные классы могут иметь свои **модификаторы доступа**
- **НЕ** являются наследниками внешнего класса, а своя отдельная единица, спрятанная во внешнем классе
- могут быть как **абстрактными**, так и **интерфейсами**, то есть как и любой другой класс

Если вложенный класс не статичный

- могут быть **статические** поля, но не методы (зачем статический метод если смысл локального класса в экземпляре?)
- Обращаться к ним нужно только через внешний класс (если позволяет модификатор доступа):

    - `OuterClass.StaticNestedClass…`
- чтобы создать экземпляр вложенного класса нужны танцы с бубном:

```
OuterClass outerObject = new OuterClass(); //сначала экзем
OuterClass.InnerClass innerObject = outerObject.new InnerC
//        ^                                    ^
//задаем тип переменной,                //создаем экземпля
//нужно обращаться через точку          //обращаясь через
```

Если вложенный класс статичный

- как будто отдельный класс из файла засунули во внешний класс
- можно обращаться напрямую без внешнего класса
- используется как вспомогательный, для каких-нибудь стремных функций

Локальные классы

Является **вложенным не статичным классом**, который объявляется в блоке кода (имеет все свойства не статичного **вложенного класса**)

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

        class Test { //локальный класс, который объявлен в фу
            int someInt = 0;

            public Test(int newInt) {
                ...
            }

        Test test = new Test();
        ...
        }
    }
}
```

- Так как локальные классы объявляются в блоке, они будут доступны только в этом блоке, так что у **локальных классов нет модификаторов доступа**
- локальным классом могут выступать **абстрактные** классы и **интерфейсы** (то есть локальный класс является таким же, как и любой другой класс)

Анонимные классы

Используются как локальные, но когда локальный класс нужен только один раз

Весь смысл в создании экземпляра и переопределении: обычно используется чтобы сократить код и не создавать отдельный класс для переопределения.

При создании анонимного класса мы создаем **наследника** класса, который мы взяли за основу. К примеру

```
public interface Callable {
    void call();
}

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

        Callable callableInstance = new Callable() { //"экзем
            @Override
            public void call() {
                ...//код
            }
        }; //обязательно точка с запятой после создания анони
    }
}
```

Выглядит так, как будто мы вызываем конструктор класса (круглые скобки возле названия интерфейса), хотя конструктора у интерфейса быть не может. (эти скобки нужны просто чтобы выглядеть как другие классы), однако если класс имеет конструктор можно создать сразу используя конструктор.

Создатели джавы пошли еще дальше и еще больше сократили написание:

```
Callable callableInstance = () -> { //лямбда выражение являет
    ...//код
};
```

Исключения

Исключение (exception) это такой класс, который используется с ключевым словом `throw`

Исключения делятся на **проверяемые** и **не проверяемые**

Checked (проверяемые)

```
public class MyException extends Exception { //проверяемое
    public String info;

    public MyException(String info) {
        this.info = info;

        System.out.println("Вот информация по поводу исключен
    }
}

public class Main {

    public void susMethod() throws MyException {
        //какой-то стремный метод
        //метод который помечен throws не обязательно имеет в
        throw new MyException("на допсу"); //кидаем только эк
        //при вызове throw сразу покидаем метод
        System.out.println("Никогда не выведется");
    }

    public static void main(String[] args) {
        try {
            //тут все методы которые могут кинуть исключение
            susMethod();
            //как только любой метод кинул исключение сразу о
        }
        catch (MyException e) { //тут может быть сам класс ис
                                                    //или
```

```
            ...//работаем над исключением
        }
        finally {
            ...//код исполняется вне зависимости поймали иск
        }
    }
}
```

Unchecked (не проверяемые)

1. ***ArrayIndexOutOfBoundsException***
2. ***NullPointerException***
3. ***ArithmeticException***
4. Все классы наследуемые от Error, RuntimeException

когда программа кидает unchecked мы знаем что просто так не восстановить все с помощью try-catch. То есть ошибка именно в разработчике 😔

```
1.0F / 0.0F //RuntimeException -> ArithmeticException -> Divi

String test = null;

test.anyMethod(); //RuntimeException -> NullPointerReference

int test[] = new int[2];

test[10000]; //RunTimeException -> ArrayIndexOutOfBound
```

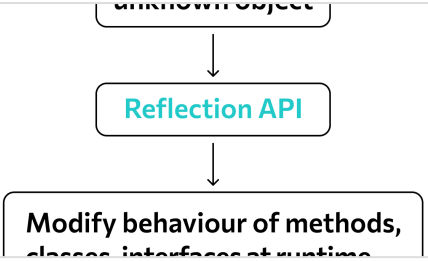
Рефлексия

2 часа ночи…

Java Reflection API (рефлексия кода) - что это

Что такое рефлексия кода и для чего она нужна. Введение в Java Reflection API. Механизм исследования данных о программе во время её выполнения.

https://blog.skillfactory.ru/glossary/java-reflection-api/



Using Java Reflection

Tech article migrated from java.sun.com

https://www.oracle.com/technical-resources/articles/java/javareflection.html

Из класса Class любого класса (`нужный_класс.class`) можно получить все интерфейсы, методы, поля которые реализует класс

getInterfaces()

getMethods()

getFields()