Подготовка к сдаче лабы 4

Оглавление

[Исключения 2](#_Toc92989660)

[Типы исключений 2](#_Toc92989661)

[Требования для выброса исключений 2](#_Toc92989662)

[Вложенные классы 2](#_Toc92989663)

[Inner classes 2](#_Toc92989664)

# Исключения

Исключение – возникновение ошибки или непредвиденной ситуации. Для того, чтобы программа знала, как поступать придумали обработку исключений.

Блок **try –** определяет блок, в котором может произойти исключение (возникнуть ошибка)

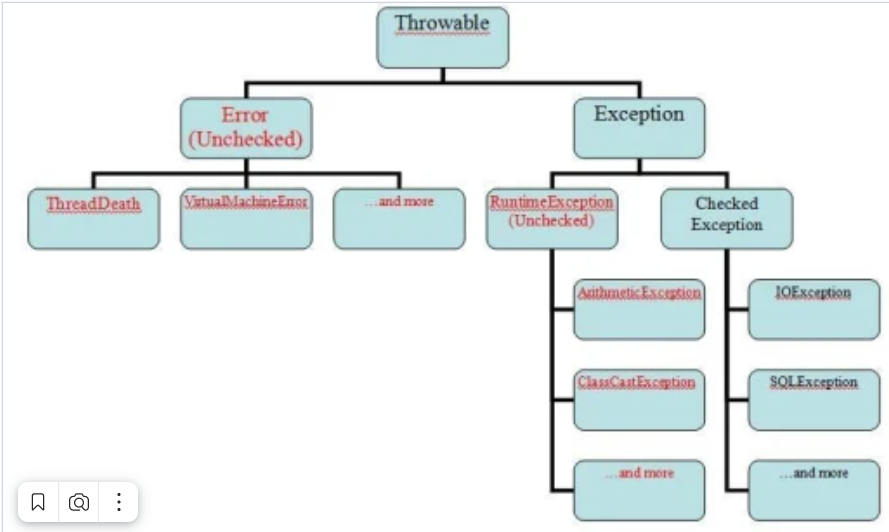
Блок **catch –** определяет блок кода, в котором происходит отлов определённого исключения (их может быть несколько)

Блок **finaly** **–** определяет блок кода, который однозначно должен быть выполнен (является необязательным, то есть может существовать только конструкция try-catch)

**Throw** – используется для вброса исключений

**Throws –** предупреждает о том, что метод может (должен) выбросить исключение (прописывается в сигнатуре)

## Типы исключений



Throwable – самый базовый класс всех исключений. В этом классе содержится код, который записывает текущий стрек-трейс вызовов функции в массив. Очевидно, что в оператор throw можно передать только наследника класса Throwable. Теоретически можно написать throw new Throwable(), но так никто не делает, ибо зачем оно как бы надо.

**Трек-трейс** – Например, в любой момент работы программы можно узнать класс и имя метода, который сейчас выполняется. И даже не одного метода, а получить информацию о всей цепочке вызовов методов от текущего метода до метода main().

Класс **Error** – прямой наследник класса Throwable. Объекты типа Error (и его классов наследников) создаются в случае каких-то серьёзных проблем непосредственно в java-машине. Например, сбой в работе или нехватка памяти. Зачастую программист ничего не может сделать если случилась такая ошибка.

Класс **Exception** (и **RuntimeException**) – это обычные ошибки, которые возникают во время работы многих методов. Цель каждого выброшенного исключения – быть захваченным блоком catch.  
Другими словами, если какая-то переменная оказалась равна null, метод выкинет **NullPointerException**, если в метод передали неверные аргументы — выкинет **InvalidArgumentException**, если в методе случайно было деление на ноль — **ArithmeticException**.

Класс **RuntimeException –** это разновидность (подмножество) исключений **Exception**. Можно даже сказать, что этот класс — это версия обычных исключений: на эти исключения накладывается меньше требований и ограничений.

Непроверяемые исключения (unchecked) наследуются от классов **Error и RuntimeException**, все остальные исключения являются проверяемыми (checked).

## Требования для выброса исключений

1. Если метод выбрасывает проверяемое исключение, то в своей сигнатуре он должен содержать информацию об этом

public PhoneNumber(String phoneNumber) throws InvalidPhoneNumberException

Непроверяемые исключения же могут работать без такого объявления.

1. Если был вызван метод, которые содержит проверяемое исключение в своей сигнатуре, то это нельзя оставить просто так, компилятор просто не даст вам этого сделать. Вы должны проверить данное исключение блоком try-catch или try-catch-finaly. Но это не всегда нужно делать. Можно пробросить исключение выше, по методам. Можно обработать только одно исключение, а второе пробросить дальше. Ну или попросту перехватить все исключения.

# Вложенные классы

class OuterClass {

    ...

    static class StaticNestedClass {

        ...

    }

    class InnerClass {

        ...

    }

}

**Java** позволяет создавать одни классы внутри других:

Именно такие классы называются вложенными. Они делятся на два типа:

1. Non-static nested classes – нестатические вложенные классы. По-другому их ещё называют inner classes – внутренние классы.
2. Static nested classes – статические вложенные классы.

В свою очередь нестатические вложенные классы (inner classes) бывают двух видов:

1. Локальными (local class)
2. Анонимными (anonymous class)

Изображение выглядит как текст, знак, снимок экрана, визитка

Автоматически созданное описание

## Inner classes

Эти классы создаются внутри других классов, всё достаточно просто.

package src;

import src.AbsClass.FairyPerson;

public class Karlson extends FairyPerson {

    public Vint vint = new Vint();

    public Karlson(String name, Location loc) {

        super(name, loc);

    }

    public class Vint {

        private String vintStatus = "Off";

        public void setVintStatus(String vintStatus) {

            this.vintStatus = vintStatus;

        }

        public String getVintStatus() {

            return vintStatus;

        }

        public void turnOn() {

            this.vintStatus = "On";

            System.out.println("Vint on");

        }

        public void turnOff() {

            this.vintStatus = "Off";

            System.out.println("Vint off");

        }

    }

}

Здесь у нас есть некий класс Karlson – карлсон. У него есть поля имя и локация (они определены в классе родителя). У него есть внутренний класс Vint (винт). Это полноценный класс, у него есть собственные методы.

Зачем же мы всё это делали, ведь можно было бы просто прописать отдельно класс Vint и не выносить себе мозг новыми приколами. Такая реализация тоже имеет право на существование, но всё же сложно представить ситуацию, когда какой-то отдельный винт нам может понадобиться без Карлсона.

Создавать потом отдельно объект класса Vint было бы логично, если бы у нас, допустим, был бы вертолёт или ещё какой-нибудь объект, которому нужен был бы винт. Но так как в нашей ситуации только одному классу нужен этот Vint мы можем создавать его прям внутри нашей программы и не париться.

**Внутренние классы** – это класс выделенный в программе некой сущности и неразрывно связанный с ней.

Это как например руль, сидение и педали – это части велосипеда, без которых его тяжело будет представить.

**Особенности внутренних классов**

1. Объект внутреннего класса не может существовать без объектов внешнего класса.

Это довольно очевидная особенность, ведь если бы её не было, тогда бы это всё не имело смысла.

1. У объекта внутреннего класса есть доступ к переменным внешнего класса (это работает с любым типом приватности, опять же очевидная особенность).
2. Объект внутреннего класса нельзя создать в статическом методе внешнего класса.

Это уже вытекает из первой особенности внутренних классов. Если вы пытаетесь создать объект в статическом методе то как бы привязываете объект внутреннего класса к самому внешнему классу, а не к его объекту, а это как бы фиговенько.

1. Внутренний класс не умеет в статику от слова совсем. Он не может содержать статические переменные и методы. Логика здесь такая же, как и в прошлом пункте.
2. При создании объекта внутреннего класса важную роль играет его модификатор доступа.

Как в принципе и везде, но тут особенности заключаются в месте где мы может создать объект данного класса. Если стоит любимый **public**, то использовать и создавать мы его может везде. С остальными логика такая же.

1. Модификаторы доступа для переменных работают также как и везде.

## Method local inner classes

Такие классы объявляются внутри методов других классов.

## Анонимные классы

Анонимные классы также являются подвидом вложенных классов.

Анонимные классы решают проблему написания большого количества одноразовых классов. Эти классы могут быть созданы на основе интерфейсов и абстрактных классов.

Interface test = new Interface(){};

1. Создаётся безымянный класс, реализующий интерфейс Interface
2. Компилятор, увидев такой класс, требует от тебя реализовать все методы интерфейса Interface
3. Создается один объект этого класса

Это то, что случилось в java машине когда она увидела такую запись.

В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным

И еще одно важное ограничение, которое досталось анонимным классам от их «предков» — внутренних классов: анонимный класс не может содержать статические переменные и методы.

# Reflection API

//Обращение к private field

        try {

            Field field = telephone.getClass().getDeclaredField("countryCode");// Получения поля с именем countryCode

            field.setAccessible(true); // Изначально поле не было публичным из-за чего приходится выдавать доступ на работу с ним

            field.set(telephone, "+8");

            number = (String) field.get(telephone);//сохранение значения этого поля. Изначально оно имеет тип Object, поэтому мы переносим его в String

        } catch (NoSuchFieldException | IllegalAccessException e) {

            e.getStackTrace();//Получение стек-трейса на текущий момент

        }

**.getDeclaredField(String)** – возвращает поле **String,** даже если оно private или protected

**.setAccessible(boolean)** - даёт доступ к полю, работа с которым изначально запрещена

**.set(Object, new value)** – меняет значение поля

**field.get(Object)** – возвращает значение данного поля некоего Object

//Создание объекта класса с помощью рефлексии

        Telephone telephone2 = null;

        try {

            Class clazz = Class.forName(Telephone.class.getName());

            Class[] parametrs = {String.class,Location.class};

            telephone2 = (Telephone) clazz.getDeclaredConstructor(parametrs).newInstance("test",room);

        } catch (Exception e) {

            e.getStackTrace();

        }

Изображение выглядит как текст

Автоматически созданное описание

Class.forName(Object.class.getName()) – подгружает определённый класс, если он до этого не использовался в программе. Object.class.getName() – возвращает полное имя файла, которое как раз нужно для forName(). В моём случае это src.Telephone

Class[] param = {String.class, Location.class} – массив с параметрами, которые далее будет использоваться при получении конструктора класса.

.getDeclaredConstructor(param) – получает конструктор с определёнными параметрами

.newInstance() – создаёт объект с такими параметрами.

    //обращение к private method

    public static void printCountryCode(Object telephone){

        try {

            Method method = telephone.getClass().getDeclaredMethod("printCountryCode");

            method.setAccessible(true);

            method.invoke(telephone);

        } catch (NoSuchMethodException | IllegalAccessException | InvocationTargetException e) {

            e.printStackTrace();

        }

    }