Лекция №3

Основы java

Бобряков Дмитрий
Senior разработчик

Меня хорошо видно и слышно?

Проверить, идет ли запись

Ставим «+», если все хорошо
«-», если есть проблемы

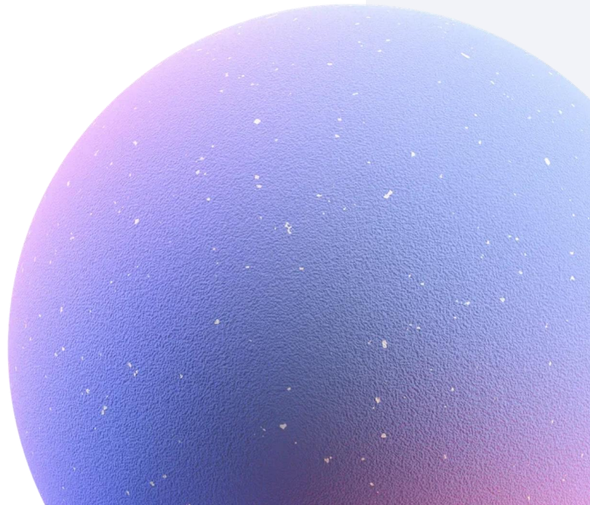
Дмитрий Бобряков

Senior разработчик

- 7 лет в IT
- Senior-разработчик в BigData Streaming
- Experience: NetCracker, AlphaBank, MWS
- Java-ментор в MWS
- Преподаватель в МФТИ

Содержание лекции

- Типы в java
- Ключевые слова
- Циклы
- Условия
- Классы/интерфейсы/прочее
- Исключения



Переменные

Переменная — это основной элемент хранения данных в программе. Это именованная область памяти, значение которой может изменяться в ходе выполнения программы.

Базовый синтаксис для объявления переменной выглядит так:
[модификаторы] тип имя_переменной [= значение];

- Тип (Type): Определяет, какие данные может хранить переменная (например, int, String, double). Это обязательная часть объявления.
- Имя (Identifier): Идентификатор, по которому вы обращаетесь к переменной. Должен соблюдать правила именования (начинаться с буквы, `$` или `_`, не быть ключевым словом).
- Значение (Value): Начальное значение, присваиваемое переменной. Это опционально, но крайне рекомендуется.

Переменные

```java
// 1. Простое объявление (без инициализации)
int age;
String name;

// 2. Объявление с инициализацией (присваиванием значения)
int count = 0;
double price = 199.99;
String greeting = "Hello, World!";
boolean isActive = true;

// 3. Объявление нескольких переменных одного типа
int x, y, z; // Все три переменные типа int, но не инициализированы.
int a = 5, b = 10, c = 15; // Объявление с инициализацией.
```

Значения по умолчанию

```
public class MyClass {
    private int defaultInt; // Будет = 0
    private String defaultString; // Будет = null

    public void printDefaults() {
        System.out.println(defaultInt); // Выведет 0
        System.out.println(defaultString); // Выведет null
    }
}
```

Ключевое слово final

Ключевое слово final используется для объявления переменной, которую можно инициализировать только один раз. После присваивания значения его нельзя изменить.

- Для final полей класса инициализацию можно отложить до работы конструктора.
- Имена констант принято писать в UPPER_SNAKE_CASE.

Ключевое слово final

```
    // Объявление константы
    public static final double PI = 3.14159;
    public final int maxConnections;

    public MyClass(int max) {
        this.maxConnections = max; // Инициализация final-поля в конструкторе
    }

    public void tryToChange() {
        // maxConnections = 100; // ОШИБКА КОМПИЛЯЦИИ! Cannot assign a value
to final variable
    }
```

Ключевое слово static

```
public final class CarUtils {

    public static String serialNumber1 = "test serial number";

}
```

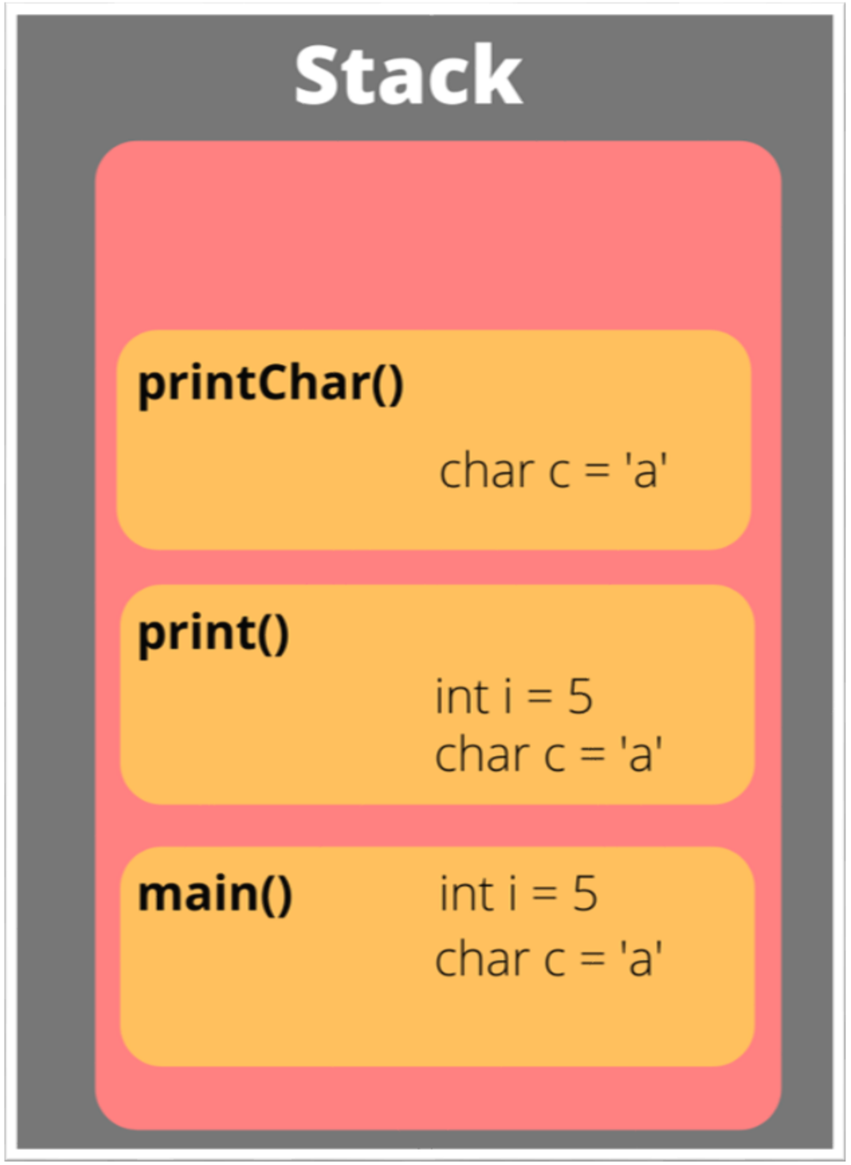
Ключевое слово static

```
import static org.example.CarUtils.SERIAL_NUMBER_2;


public final class StaticExample {

    void useStatic() {
        System.out.println(SERIAL_NUMBER_2);
    }
}
```

Примитивы и объекты

Все значения в Java делятся на два типа: примитивы и объекты.
К примитивам относятся следующие типы данных:

- byte — 8-битовое целое число со знаком.
  Может принимать значение от -128 до 127.
- short — 16-битовое целое число со знаком.
  Может принимать значение от -32768 до 32767.
- int — 32-битовое целое число со знаком.
  Может принимать значение от -2^31 до 2^31 - 1.
- long — 64-битовое целое число.
  Число со знаком может принимать значение от -2^63 до 2^63 - 1.
- float — 32-битовое число с плавающей запятой.
- double — 64-битовое число с плавающей запятой.
- boolean — логический тип данных, может
  иметь только 2 значения, true или false.
- char — один символ в формате Unicode.



Операторы

Арифметические: `+`, `-`, `*`, `/`, `%`
Операторы сравнения: `==`, `!=`, `>`, `<`, `>=`, `<=`
Логические: `&&` (и), `||` (или), `!` (не)

Подводный камень: Короткий цикл вычислений (short-circuit). В выражениях с `&&` и `||` правый операнд вычисляется только если это необходимо.

```
if (a != null && a.isValid()) { ... }
```

Если a == null, a.isValid() вызван не будет. Безопасно.

Обертки над примитивами

- → Могут быть null
- → Занимают места на порядок больше чем их примитивные версии
- → Имеют богатый API для работы с ними
- → Неотъемлемая часть при работе с коллекциями (List, Map, Set, Queue, etc..)

int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean

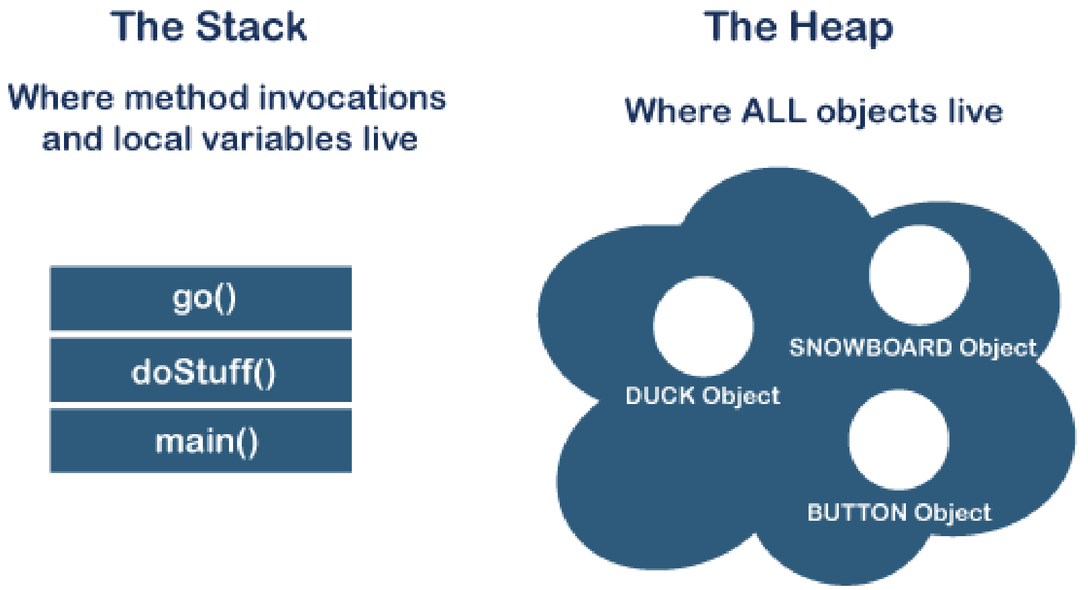
Примитивы и объекты

→ Любые другие значения являются объектами.

→ Объекты имеют неявного наследника под названием Object

→ Объекты хранятся в хипе

Переменные этих типов хранят не сам объект, а ссылку (адрес) на объект в куче (heap). К ним относятся все классы, массивы, интерфейсы, перечисления (enums).



Ссылочные типы

Переменные этих типов хранят не сам объект, а ссылку (адрес) на объект в куче (heap). К ним относятся все классы, массивы, интерфейсы, перечисления (enums).

```
String text = "Hello"; // 'text' - ссылка на объект String в куче
int[] numbers = new int[10]; // numbers - ссылка на массив
MyFirstClass obj = new MyFirstClass(); // obj - ссылка на объект
```

Значение null

Это особое значение, которое может быть присвоено любому объекту (примитивы не могут быть null). Значение null используется в Java для того, чтобы записать отсутствующие данные какого-либо типа.

```
public class Person {
  private final String firstName;
  private final String lastName;
  private final String patronymic;

  /* конструктор */
}
```

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person("Петр", "Иванов", null);
        System.out.println(person);
    }
}
```

Значение null

```java
public class Main {
    public static void main(String[] args) {
        Airplane airplane = new Airplane(1980, "TU-154");
        airplane = null;
        airplane.print();
    }
}
```

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "org.example.Airplane.print()" because "airplane" is null at org.example.Main.main(Main.java:8)

Значение null

```
public class MyClass {
    private int defaultInt; // Будет = 0
    private String defaultString; // Будет = null

     public void printDefaults() {
         System.out.println(defaultInt); // Выведет 0
         System.out.println(defaultString); // Выведет null
     }
}
```

Циклы for

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

```
int[] numbers = {1, 2, 3};
for (int num : numbers) { // Для каждого элемента в numbers
    System.out.println(num);
}
```

Циклы while

```
while (condition) {
    //  ...
}

do {
    // Выполнится хотя бы один раз
} while (condition);
```

Улсловие if

```
if(1 + 1 > 3) {
   …
} else if (1 + 1 == 3) {
   …
} else {
   …
}
```

Условие switch

```java
// Старый стиль (до Java 14) - подвержен ошибкам из-за 'break'
int day = 3;
String dayName;
switch (day) {
    case 1:
        dayName = "Monday";
        break; // Если забыть break, выполнение "провалится" дальше!
    case 2:
        dayName = "Tuesday";
        break;
    default:
        dayName = "Unknown";
}

// Новый стиль (Java 14+, выражение →, yield)
String dayName = switch (day) {
    case 1 → "Monday"; // Нет проваливания, break не нужен
    case 2 → "Tuesday";
    case 3 → {
        // Для сложной логики можно использовать блок с yield
        String fullName = "Wednesday";
        yield fullName;
    }
    default → "Unknown";
};
```

Что такое класс?

- Базовое понятие: Класс — это прежде всего чертеж или шаблон, на основе которого создаются объекты (экземпляры). Он описывает:
- Состояние (State): Данные, которые будет хранить объект. Это поля класса (переменные).
- Поведение (Behavior): Действия, которые объект может выполнять. Это методы класса (функции).
- Простая аналогия: Представьте класс CookieCutter (формочка для печенья).
- Класс — это сама формочка (ее форма, например, звезды).
- Объект — это конкретное печенье, вырезанное этой формочкой.
- Поле класса — это, например, size (размер формочки).
- Метод класса — это cut() (вырезать печенье).

```
package org.example;

public class Car {
}
```

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

```java
package org.example;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Car extends AbstractCar implements Vehicle {
    private String brand;
    private int year;
    private String color;

    public Car(String brand, int year, String color) {
        this.brand = brand;
        this.year = year;
        this.color = color;
    }

    @Override
    public void describeCar() {
        System.out.println("Бренд: " + this.brand);
        System.out.println("Год выпуск: " + this.year);
        System.out.println("Цвет: " + this.color);
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return year == car.year && Objects.equals(brand, car.brand) &&
Objects.equals(color, car.color);
    }

    @Override
    public int hashCode() {
        return Objects.hash(brand, year, color);
    }

    @Override
    public String toString() {
        return String.format("Car[brand=%s, year=%d, color=%s]",
this.brand, this.year, this.color);
    }
}
```

Поля и конструктор

```
public class Car {

    private String brand;
    private int year;
    private String color;

    public Car(String brand, int year, String color) {
        this.brand = brand;
        this.year = year;
        this.color = color;
    }
}
```

Геттеры и Сеттеры

```
public String getBrand() {
        return brand;
}

public void setBrand(String brand) {
        this.brand = brand;
    }
```

Equals & HashCode

```
 @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return year == car.year && Objects.equals(brand,
car.brand) && Objects.equals(color, car.color);
    }

    @Override
    public int hashCode() {
        return Objects.hash(brand, year, color);
    }
```

ToString

```
 @Override
    public String toString() {
        return String.format("Car[brand=%s, year=%d, color=%s]",
this.brand, this.year, this.color);
    }
```

Абстрактные Классы: Недоопределенные чертежи

Базовое понятие: Это класс, помеченный ключевым словом abstract. Он представляет собой неполный чертеж. Его главная цель — быть родителем для других классов, объединяя их общую логику.

Зачем они нужны?
- Нельзя создать экземпляр: new AbstractClass() — ошибка компиляции.
- Может содержать абстрактные методы: Методы без реализации (без тела `{}`). Они говорят: "Все мои потомки обязаны уметь это делать, но как именно — решат они сами".
- Может содержать обычные методы: С готовой реализацией, которая наследуется потомками.

Аналогия: Представьте абстрактный класс `Shape` (Фигура).
- У любой фигуры есть площадь. Но как ее вычислить? Для круга — одна формула, для квадрата — другая.
- Мы обязываем все фигуры иметь метод `calculateArea()`, но не реализуем его на уровне фигуры.

Абстрактный класс

```java
// Абстрактный класс
public abstract class Shape {

    // Обычное поле - есть у всех фигур
    private String color;

    // Конструктор - он нужен, чтобы инициализировать поля предка
    public Shape(String color) {
        this.color = color;
    }

    // Абстрактный метод - не имеет тела, только сигнатура.
    // Класс, содержащий абстрактный метод, ДОЛЖЕН быть абстрактным.
    public abstract double calculateArea();

    // Обычный метод с реализацией - есть у всех фигур
    public String getColor() {
        return color;
    }

    // Еще один обычный метод
    public void printInfo() {
        System.out.println("I'm a " + color + " shape.");
    }
}
```

Абстрактный класс

```
// Конкретный класс-наследник (Concrete Class)
public class Circle extends Shape { // extends - ключевое слово для
наследования

    // Свое, специфичное поле
    private double radius;

    // Конструктор
    public Circle(String color, double radius) {
        super(color); // super() вызывает конструктор родительского класса
(Shape)
        this.radius = radius;
    }

    // ОБЯЗАТЕЛЬНАЯ реализация абстрактного метода родителя
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    // Свой специфичный метод
    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}
```

Интерфейсы: Контракты на поведение

Базовое понятие: Интерфейс — это чистейший контракт. Он определяет *что* класс должен делать, но не *как*. До Java 8 он мог содержать только абстрактные методы. Сейчас возможности сильно расширились.

Зачем они нужны?
- Обеспечивают множественное "наследование" (реализацию нескольких контрактов одним классом).
- Позволяют достичь слабой связанности (loose coupling) компонентов системы. Код зависит от абстракции (интерфейса), а не от конкретной реализации.
- Определяют роль, которую может играть объект.

Аналогия: Интерфейс `USB`.
- У него есть строгий контракт: разъем, напряжение, protocol передачи данных.
- Неважно, кто производитель устройства (флешка, клавиатура, мышь), если оно реализует контракт `USB`, оно гарантированно заработает при подключении к USB-порту.

Интерфейс

```
// С Java 8+ (добавились default и static методы)
public interface Drawable {

    void draw(); // abstract метод - все еще основа контракта

    // default метод - реализация по умолчанию для всех, кто реализует
интерфейс.
    // Нужен для обратной совместимости. Можно переопределить.
    default void setColor(String color) {
        System.out.println("Setting color to " + color);
        // Реальная логика установки цвета была бы здесь
    }

    // static метод - принадлежит самому интерфейсу, вызывается как
Drawable.printInfo()
    // Не наследуется реализующими классами.
    static void printInfo() {
        System.out.println("This is the Drawable interface.");
    }
}
```

Интерфейс

```
// Класс может реализовать (implements) множество интерфейсов
public class Button implements Drawable, Clickable { // Два контракта сразу!

    private String label;

    // ОБЯЗАТЕЛЬНАЯ реализация абстрактного метода из Drawable
    @Override
    public void draw() {
        System.out.println("Drawing a button with label: " + label);
    }

    // Реализация метода из Clickable (предположим, что такой интерфейс есть)
    @Override
    public void onClick() {
        System.out.println("Button was clicked!");
    }

    // Метод setColor() НЕ ОБЯЗАТЕЛЕН к реализации, т.к. есть default
реализация.
    // Но мы можем его переопределить, если захотим:
    // @Override
    // public void setColor(String color) { ... }
}
```

Сводка: Ключевые различия и что когда использовать

Характеристика	Класс	Абстрактный Класс	Интерфейс (Modern)
Экземпляры	Можно создавать (`new`)	**Нельзя** создавать	**Нельзя** создавать
Поля	Любые (обычно `private`)	Любые (обычно `private`)	Только `public static final` (константы)
Методы	Любые	Абстрактные и/или реализованные	Абстрактные, `default` , `static`
Наследование	Один класс (`extends`)	Один абстрактный класс (`extends`)	**Много** интерфейсов (`implements`)
Суть	**Реализация** и **состояние**	**Частичная реализация** для иерархии "is-a"	**Контракт** (поведение) для ролей "can-do"
Когда использовать?	Для создания объектов с конкретными свойствами и поведением.	Когда несколько классов тесно связаны иерархией и имеют общую логику.	Когда нужно определить роль, которую могут играть **несвязанные** классы. Для достижения слабой связанности.

Enum

```
public enum VehicleType {
    CAR,
    SHIP
}
```

Анонимные классы

```
Vehicle vehicle = new Vehicle() {
            @Override
            public void print() {
                System.out.println("Машина Ford 1983 года");
            }
        };
```

Лямбда-выражения

```
Vehicle vehicle = (() -> System.out.println("Машина Ford 1983
года");

vehicle.print();
```

Аннотации

Аннотации — это форма метаданных, которые предоставляют данные о программе, но не являются частью самой программы. Они не имеют прямого влияния на операцию кода, который они аннотируют

Простая аналогия: Представьте, что вы читаете книгу с заметками на полях. Эти заметки:

1. Не меняют основной текст книги
2. Несут дополнительную информацию для читателя
3. Могут давать инструкции ("обрати внимание на это", "это важно")

Аннотации

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.CONSTRUCTOR,
ElementType.METHOD})
public @interface CarAnnotation {
}

@CarAnnotation
public class Car {}
```

Аннотации

@Retention:

→ RetentionPolicy.SOURCE — visible by neither the compiler nor the runtime

→ RetentionPolicy.CLASS — visible by the compiler // default

→ RetentionPolicy.RUNTIME — visible by the compiler and the runtime

Аннотации

`@Override` — указывает, что метод переопределяет метод суперкласса.

```
public class Parent {
    public void doSomething() {
        // ...
    }
}

public class Child extends Parent {
    @Override // Убедитесь, что это действительно переопределение
    public void doSomething() {
        // ...
    }
}
```

Аннотации

`@FunctionalInterface` — указывает, что интерфейс предназначен быть функциональным интерфейсом.

```
@FunctionalInterface
public interface SimpleFunction {
    void execute();
    // boolean equals(Object obj); // Разрешено - метод из Object
    // void anotherMethod(); // ОШИБКА! Функциональный интерфейс должен иметь
ровно один абстрактный метод
}
```

Зачем нужно: Гарантирует, что интерфейс имеет ровно один абстрактный метод и может быть использован с лямбда-выражениями.

Reflection API

```
Car car = new Car("electro engine");


Class<? extends Car> aClass = car.getClass();
aClass.getName();  // org.example.Car
```

На практике полезно при работе с:

- → Аннотациями
- → AOP
- → При обработке исключений
- → Когда нужно подменить значение иммутабельного поля в объеке

Метаинформация об объектах
и классах включает себя:

- → Классы
- → Методы
- → Поля
- → Типа
- → Аннотации
- → Пакеты
- → Модули

Record

```
public record RecordCar(
        String brand,
        int year,
        String color) implements Vehicle {

    @Override
    public void print() {
    }
}
```

- → is a restricted form of a class
- → It’s ideal for "plain data carriers"
- → for classes that contain data not meant to be altered
- → have only the most fundamental methods such as constructors and accessors

- → final
- → private
- → full constructor
- → getters
- → ~~setters~~
- → equals & hashCode
- → toString
- → ~~extends~~

Sealed class

```
public abstract sealed class SealedCar permits
Toyota {
}


public final class Toyota extends SealedCar {
}
```

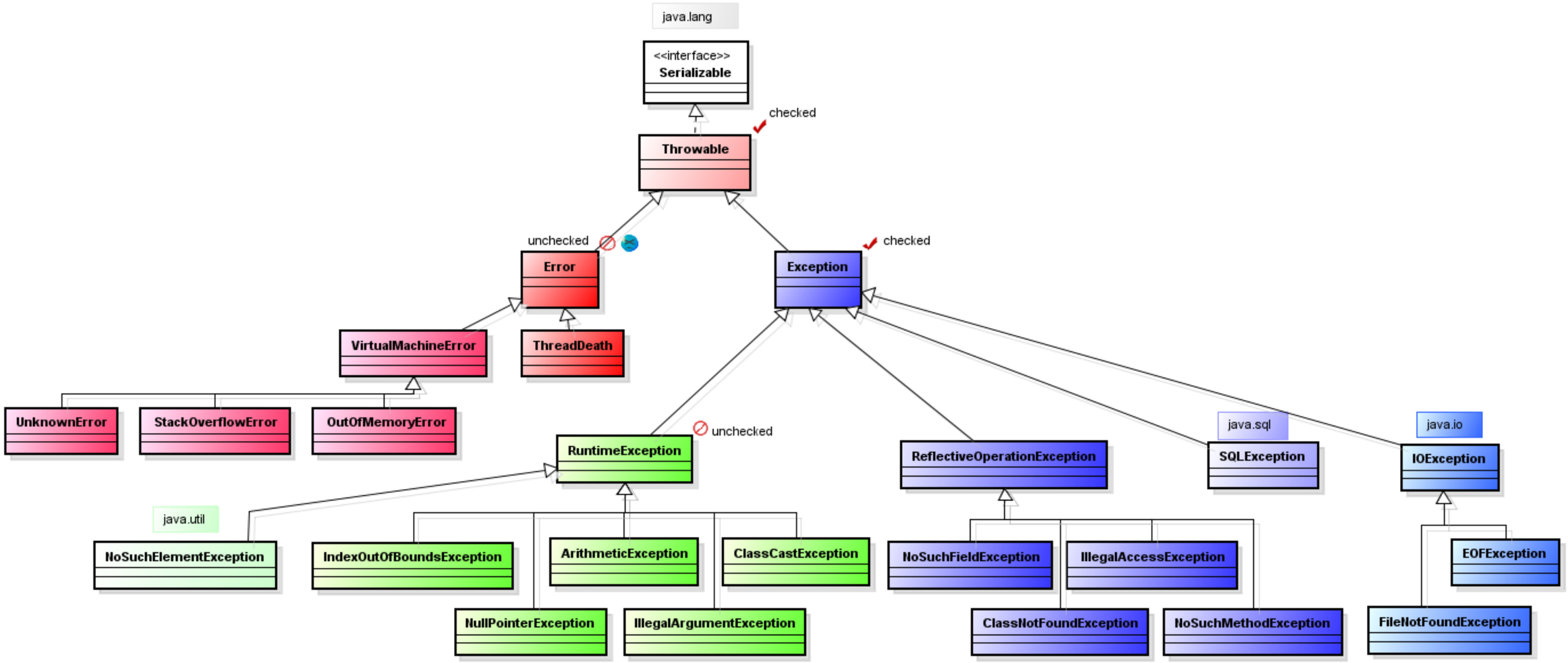
Метод main

Точка входа в программу. JVM ищет именно его, чтобы начать выполнение.

```
public static void main(String[] args) {
    // Создание объекта
    MyFirstClass obj = new MyFirstClass(42);
    obj.myMethod(); // Вызов метода
}
```

- `static` : метод принадлежит классу, а не экземпляру. Для вызова не нужен объект.
- `String[] args` : аргументы командной строки.

Исключения


java.lang
<<interface>>
Serializable
checked
Throwable
unchecked
Error
checked
Exception
VirtualMachineError
ThreadDeath
UnknownError
StackOverflowError
OutOfMemoryError
unchecked
RuntimeException
java.sql
java.io
ReflectiveOperationException
SQLException
IOException
java.util
NoSuchElementException
IndexOutOfBoundsException
ArithmeticException
ClassCastException
NullPointerException
IllegalArgumentException
NoSuchFieldException
IllegalAccessException
ClassNotFoundException
NoSuchMethodException
EOFException
FileNotFoundException

Выбрасывание и обработка исключений

```
public class Airplane implements Vehicle {
    /* поля и конструктор */

    public Airplane(int year, String color) {
        if (color == null) {
            throw new RuntimeException("Color cannot be null");
        }
        this.year = year;
        this.color = color;
    }
}
```

Выбрасывание и обработка исключений

```
try {
    Airplane airplane = new Airplane(1980, null);
    airplane.print();
} catch (RuntimeException e) {
    System.out.println("Ошибка при создании Airplane");
} finally {
    System.out.println("Произошла попытка создания");
}
```

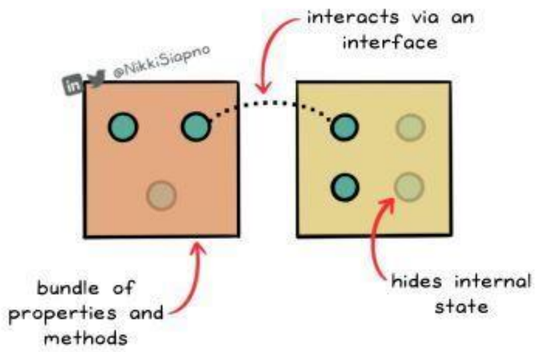
StackTrace

```
java.lang.RuntimeException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
        at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
        at org.codehaus.groovy.reflection.CachedConstructor.invoke(CachedConstructor.java:77)
        at org.codehaus.groovy.runtime.callsite.ConstructorSite$ConstructorSiteNoUnwrapNoCoerce.callConstructor(ConstructorSite
        at org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCallConstructor(CallSiteArray.java:52)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:192)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:196)
        at newifyTransform$_run_closure1.doCall(newifyTransform.gdsl:21)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
        at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:893)
        at org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.callCurrent(PogoMetaClassSite.java:66)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callCurrent(AbstractCallSite.java:151)
        at newifyTransform$_run_closure1.doCall(newifyTransform.gdsl)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
        at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:893)
        at org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.call(PogoMetaClassSite.java:39)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:121)
        at org.jetbrains.plugins.groovy.dsl.GroovyDslExecutor$_processVariants_closure1.doCall(GroovyDslExecutor.groovy:54)
        at sun.reflect.GeneratedMethodAccessor61.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
```

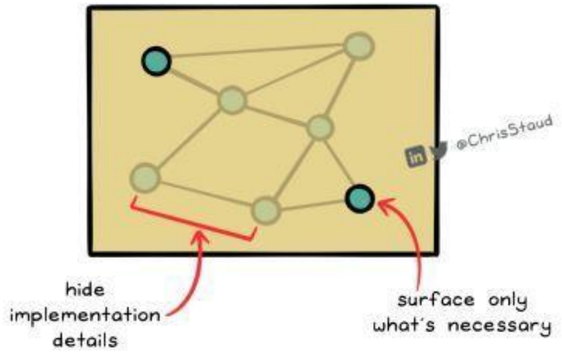
Principles of OOP

by levelupcoding.co

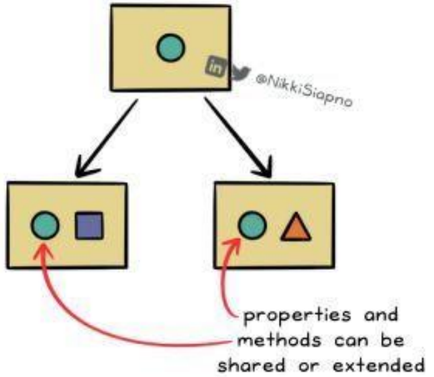
Encapsulation



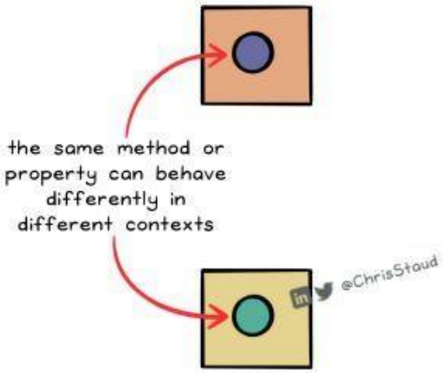
Abstraction



Inheritance



Polymorphism



Let me know you like it

@NikkiSiapno @ChrisStaud

Share it with your friends

Инкапсуляция (сокрытие)

Modifier	Class	Package	Subclasses	World
public	+	+	+	+
protected	+	+	+	-
default	+	+	-	-
private	+	-	-	-

Полиморфизм
Абстрактный класс и интерфейс

```
public class Car extends AbstractCar implements Vehicle {

    @Override
    public void print() {
        System.out.println("Бренд: " + this.brand);
        System.out.println("Год выпуск: " + this.year);
        System.out.println("Цвет: " + this.color);
    }
```

MTC True Tech

СПАСИБО ЗА ВНИМАНИЕ

Дмитрий Бобряков

Senior big data developer

dmitrybobryakov@gmail.com

@DmitryBobryakov