

# Объектно-ориентированное программирование в Java

Croc Java School

---

# Понятие класса



## First-class citizen

Entity that supports all operations generally available to other entities.

Can be:

- the argument of a function
- returned as a result of a function
- the subject of assignment
- tested for equality



## First-class citizens in Java

- Примитивы (boolean, char, byte, short, int, long, float, double)
- Объекты
- Массивы




# Класс как тип данных

## Тип данных

- множество значений
- множество операций над значениями

## Класс в Java

- поля
- методы



```
class Date {  
  
    // fields  
    int year;  
    int month;  
    int day;  
  
    // methods  
    boolean before(Date other) {}  
    void add(int days) {}  
}
```



## Состояние класса

Конкретные значения полей класса в текущий момент.

Состояние должно быть корректным.

Каждая операция переводит класс из одного корректного состояния в другое или завершается ошибкой, если переход невозможен.



## Корректность состояния

```
class Date {  
  
    void add(int days) {  
        this.days += days; // possibly incorrect state  
    }  
}
```





## Контроль за состоянием важен

```
class Date {  
  
    void add(int days) {  
        int d = getDaysSinceBase() + days;  
        setDaysSinceBase(d);  
    }  
  
    int getDaysSinceBase() {  
    }  
  
    void setDaysSinceBase(int days) {  
    }  
}
```



## Классы vs объекты

Классы описывают тип данных.

Объекты соответствуют конкретным значениям этих типов.

Объекты = экземпляры, инстансы.

- Тип данных *Date* — класс.
- Переменная *birthday* типа *Date* — объект.



## Конструирование объектов

Для создания объектов используем ключевое слово `new`.

```
Date birthday = new Date(1999, 10, 10);
```

Конструктор

```
class Date {  
    Date(int year, int month, int day) {  
        this.year = year;  
        this.month = month;  
        this.day = day;  
    }  
}
```



# Конструктор

Конструктор - это метод, у которого

- название совпадает с названием класса
- отсутствует возвращаемое значение

Не конструктор:

```
void Date() {  
    // ...  
}
```



## Конструктор

В конструкторе инициализируются поля класса.

Но он может включать и другой код.

Поля могут быть инициализированы и при определении:

```
class Date {  
    int year = 2019;  
    int month = 1;  
    int day = 1;  
}
```



## Конструктор по умолчанию

При отсутствии явного конструктора Java определяет конструктор по умолчанию, инициализирующий поля значениями по умолчанию и не имеющий аргументов.

При наличии хотя бы одного конструктора конструктор по умолчанию не добавляется.



## Package

Пакеты Java определяют пространства имен классов и влияют на доступ к данным.

В рамках одного пакета не может быть двух классов с совпадающими идентификаторами.

```
package ru.croc.wjs;  
  
class Date {  
    // class declaration  
}
```

В файловой системе класс должен быть расположен по пути

```
./ru/croc/wjs/Date.java
```



## Импорт классов

Для использования классов из пакетов, отличных от текущего, их необходимо импортировать.

```
import java.util.List;  
import ru.croc.wjs.*;
```





## Порядок элементов класса

Package declaration

Imports

Class declaration

Fields

Methods

---

# Принципы ООП



# Принципы ООП

Инкапсуляция

Наследование

Полиморфизм

Абстрагирование

# Инкапсуляция

---



## Инкапсуляция

Ограничение доступа к состоянию класса в целях сохранения его консистентности.

В Java имеет смысл “сокрытия”.

Пример

```
Date birthday = new Date();  
birthday.month = 13; // puts class in a non-valid state
```



## Инкапсуляция

Доступ к данным осуществляется через методы:

```
class Date {  
    private int month;  
  
    int getMonth() {  
        return this.month;  
    }  
  
    void setMonth(int month) {  
        if (month < 1 || month > 12)  
            // error  
        this.month = month;  
    }  
}
```



## Модификаторы доступа

### **private**

Доступ в пределах одного класса

### **default (package private)**

Доступ в пределах пакета

### **protected**

Доступ в пределах пакета и классов-наследников

### **public**

Доступ не ограничен



# Модификаторы доступа

Применимы к:

- полям класса
- методам класса
- классам



# Наследование

---



## Наследование

Описание нового класса на основе существующего с целью заимствования функциональности.

Позволяет устранить проблему дублирования кода через выстраивание иерархии классов.



## Наследование

```
public class Shape {  
  
    protected Color color;  
  
    public Color getColor() {  
        return this.color;  
    }  
  
    public void setColor(Color color) {  
        this.color = color;  
    }  
}
```



## Наследование

```
public class Rectangle extends Shape {  
  
    private double x1, y1, x2, y2;  
  
    // ...  
}
```

```
public class Circle extends Shape {  
  
    private double x, y, r;  
  
    // ...  
}
```



## Наследование

Shape — базовый класс

Rectangle, Circle — классы-наследники или дочерние классы

Rectangle и Circle получают доступ к полям и методам Shape в соответствии с ограничениями доступа.

Конструкторы не наследуются.

Java не поддерживает множественное наследование.

Для запрета наследования от класса его можно пометить модификатором `final`.



## this and super

Литералы доступа к полям текущего и базового класса.

```
// for some reason we may want this only for circles
public void hide() {
    super.color = Color.TRANSPARENT;
}
```

Адресуемое через super поведение может быть определено в любом из родителей класса.



## this() and super()

```
public class Shape {  
    protected Color color;  
    public Shape(Color color) {  
        this.color = color;  
    }  
}
```

```
public class Circle extends Shape {  
    public Circle(Color color) {  
        super(color);  
    }  
}
```

# Полиморфизм

---





# Полиморфизм

Позволяет использовать уникальное поведение объектов с общим предком (базовым классом) без спецификации типов этих объектов.



## Сигнатура метода

```
public void draw(Graphics g) throws Exception {  
    // draw a shape  
}
```

Сигнатура метода = название + список параметров.

В классе не может быть определено несколько методов с одинаковой сигнатурой.



## Variable Arguments

```
public int max(int... numbers) {  
    int max = Integer.MIN_VALUE;  
    for (int number : numbers) {  
        if (number > max)  
            max = number;  
    }  
    return number;  
}
```

Сигнатура эквивалентна методу

```
public int max(int[] numbers) {  
    // ...  
}
```



## Overloading

Названия методов совпадают, но списки параметров различаются.

```
public class Shape {  
  
    boolean intersects(Shape shape) {}  
  
    boolean intersects(Polygon polygon) {}  
  
    boolean intersects(Point[] points) {}  
  
    boolean intersects(Point... point) {} // duplicate signature  
}
```

При вызове метода Java выбирает наиболее специализированный вариант.



## Overriding

Сигнатура метода наследника совпадает с сигнатурой метода базового класса.

```
public class HighContrastShape extends Shape {  
  
    public void setColor(Color color) {  
        Color contrastColor = makeContrast(color);  
        super.setColor(contrastColor);  
    }  
}
```

Переопределенный метод не может иметь более строгий модификатор доступа.



## Базовый метод

```
public class Shape {  
  
    public void draw(Graphics g) {  
        // do nothing  
    }  
}
```



## Реализации в дочерних классах различаются

```
public class Rectangle extends Shape {  
  
    public void draw(Graphics g) {  
        // draw rectangle  
    }  
}
```

```
public class Circle extends Shape {  
  
    public void draw(Graphics g) {  
        // draw circle  
    }  
}
```



## Вызов виртуального метода

```
Shape shape = new Circle(0, 0, 10);  
shape.draw(Graphics.get());
```

### **compile-time**

Метод `draw(Graphics g)` должен быть определен в классе `Shape`.

### **run-time**

В процессе исполнения вызывается метод класса, соответствующий конкретному типу объекта в процессе выполнения.





## Гетерогенные коллекции

```
public class Renderer {  
  
    private Graphics g;  
  
    public void render(Shape[] shapes) {  
        for (Shape shape : shapes)  
            shape.draw(g);  
    }  
}  
  
new Renderer().render(new Shape[] {  
    new Rectangle(10, 10, 20, 20),  
    new Circle(15, 15, 5)  
});
```



## instanceof and casting

```
Shape shape;  
if (shape instanceof Circle) {  
    Circle circle = (Circle)shape; // narrow class reference  
    System.out.println("Radius = " + circle.getRadius());  
}
```

# Абстрагирование

---



## Абстракция (данных)

Выделение значимых характеристик классов и исключение незначимых.

Уровни абстракции в Java:

1. объекты
2. классы
3. абстрактные классы
4. интерфейсы



## Абстрактные классы и методы

Выступают в качестве шаблонов для дочерних классов.

Экземпляры абстрактных классов создавать нельзя.

Не могут быть `private` и `final`.

Наследники абстрактного класса должны переопределить все его абстрактные методы.



## Shape — абстрактный класс

```
public abstract class Shape {  
  
    protected Color color;  
  
    public Color getColor() {  
        return this.color;  
    }  
  
    public void setColor(Color color) {  
        this.color = color;  
    }  
  
    public abstract void draw(Graphics g);  
}
```



## Интерфейсы

The person writing the interface says, *"hey, I accept things looking that way"*, and the person using the interface says *"OK, the class I write looks that way"*.

```
public abstract interface Drawable {  
    public static final Color CLEAR_COLOR = Color.BLACK;  
    public abstract void draw(Graphics g);  
}
```



## Наследование и реализация

У интерфейсов нет состояния, они определяют только абстрактные методы.

Начиная с Java 8 интерфейс может предоставить реализацию метода по умолчанию.

Классы наследуются, но интерфейсы реализуются.

Наследовать несколько классов нельзя, но реализовать несколько интерфейсов можно.





## Shape наследует интерфейс Drawable

```
public abstract class Shape implements Drawable {  
  
    protected Color color;  
  
    public Color getColor() {  
        return this.color;  
    }  
  
    public void setColor(Color color) {  
        this.color = color;  
    }  
  
    public abstract void draw(Graphics g);  
}
```



## default (Java 8+)

В Java 8 появилась возможность определить реализацию интерфейсных методов по умолчанию.

```
interface Drawable {  
  
    void draw(Graphics g);  
  
    default Color getClearColor() {  
        return Color.BLACK;  
    }  
}
```

---

**Зачем все это нужно?**



Проекты на практике развиваются несколько лет.

Со временем стоимость поддержки и развития растет нелинейно.

Разумно спроектированные системы проще развивать.

Плохо спроектированные системы развивать и поддерживать очень дорого или вовсе невозможно.



# Object-Oriented Design

SOLID

<https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

---

# Object class