# Содержание



# Понятия класса, объекта. Инкапсуляция.

## Определения

**Объект (object)** — самоописывающая структура данных, обладающая внутренними состояниями и способная обрабатывать передаваемые ей сообщения.

**Инкапсуляция (incapsulation)** — один из основных принципов ООП, заключающийся в том, что доступ к внутреннему состоянию объекта осуществляется только через механизм передачи сообщений.

**Класс (class)** — тип данных, значениями которого являются объекты, имеющие сходное внутреннее состояние и обрабатывающие одинаковый набор сообщений.

*Замечание*: объекты являются самоописывающими, так как содержат информацию о классе, к которому они принадлежат.

*Замечание*: класс можно рассматривать как шаблон для порождения объектов.

*Замечание*: объекты называют также **экземплярами класса** (class instanses).

*Пример*:

```
// Класс Point
(x, y, z) // координаты - это экземплярные поля
get_r(double x, double y, double z) // экземплярный метод
```

Типы классов:

1. Публичный
   Один на файл.
2. Непубличный
   В одном файле может быть больше одного.

*Замечание*: файл с исходным кодом описывающим публичный класс, именовывается так же, как и `public` (публичный) класс, с точностью до регистра.

## Структура публичного класса

1. Непубличные классы
   Эти классы видны только внутри этого файла (при объявлении непубличных классов можно не ставить модификатор `public`)
2. Члены класса
   - Экземплярные поля
     Хранение внутренних состояний объекта.
   - Статические поля
     Хранение данных, общих для всех объектов класса. Модификатор `static`.
   - Экземплярные методы
     Выполняют обработку передаваемых объекту сообщений.

- Экземплярные конструкторы
  Инициализирует только что созданный объект (Метод, который выполняется в момент инициализации объекта).
- Статический конструктор (набор статических блоков)
  Инициализирует статические поля класса.
- Статический метод
  Выполняют действия, для которых не нужен доступ к конкретному объекту класса.
- Вложенные классы
  Объекты, необходимые для реализации данного класса.

**Примеры**

1. Не нарушаем принцип инкапсуляции
   - Main.java:
     ```java
     public class Main {
         public static void main(String[] args) {
             Point A = new Point("Tochka A");
             A.setCoords(1.0, 1.0, 34.0);
         }
     }
     ```
   - Point.java:
     ```java
     public class Point {
         private double x;
         private double y;
         private double z;
         public Point(String Name) { // конструктор
             this.name = Name;
         }
         public void setCoords(double inX, double inY,
          ↪  double inZ) {
             this.x = inX;
             this.y = inY;
             this.z = inZ;
         }
     }
     ```
2. Нарушаем
   - Main.java:
     ```java
     public class Main {
         public static void main(String[] args) {
             Point A = new Point("Tochka A");
     ```

```
            A.x = 1.0;
            A.y = 1.0;
            A.z = 34.0;
        }
    }
```

- Point.java:

```
public class Point {
    public double x;
    public double y;
    public double z;
    public Point(String Name) { // конструктор
        this.name = Name;
    }
}
```

# Статические методы, статические поля.

## Определения статического поля, экземлярного метода

**Статическое поле (static field)** — это такое поле, принадлежащее некотрому классу, значение которого разделяется всеми объектами этого класса.

*Пример*:

```
class Point {
    public int x, y; // координаты точки
    public static int count; // общее количество точек
};
```

**Экземплярный метод (instance method)** — подпрограмма (функция), осуществляющая обработку переданного объекту сообщения.

- *Экземплярный метод* передает объекту сообщения.
- *Экземплярный метод* имеет доступ к внутреннему состоянию объекта (может читать/изменять значения экземплярных полей).

*Пример*:

```
class Person {
    public String name;
    public int yearOfBirth;
    private String address;
};
```

`name`, `yearOfBirth`, `address` — экземплярные поля. `public`, `private` — модификаторы.

## Виды модификаторов

1. `private`
   Доступ разрешен только из тела класса.
2. Без модификатора
   Доступ разрешен для самого класса и классов из того же пакета.
3. `protected`
   Доступ разрешен для самого класса, для классов из того же пакета, а также наследников класса.
4. `public`
   Доступ возможен откуда угодно.

## Перегрузка методов

Конструктор не указан. Создается конструктор по умолчанию:

```
public class Cat {
    public String name;
    public int age;
};
```

Явно указан конструктор по умолчанию:

```
public class Cat {
    public String name;
    public int age;
    public Cat() {}
};
```

Перегрузка конструктора:

```
public class Cat {
    public String name;
```

```
    public int age;
    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Cat() {}
};
```

`String name, int age` — **структура метода** (method structure).

*Примеры* создания объекта класса `Cat`:

```
Cat A = new Cat();
Cat B = new Cat("Meow", 5);
```

## Статические методы

**Статический метод (static method)** — метод, не имеющий доступа к внутреннему состоянию этого объекта. Другими словами, статический метод может обратиться только к статическим переменным и методам.

`private static` vs `public static`:

- `public static`
  Статическое поле можно определить через любой объект класса или имя класса.
- `private static`
  Можно определить только внутри класса.

*Примеры*:

```
public class Point {
    private double x;
    private double y;
    private static int n;
    public static int val;
    public Point() {
        this.n = 10;
        this.val = 100;
    }
    public void setCoords(double varX, double varY) {
        this.x = varX;
        this.y = varY;
```

```
    }
    public double getN() {
        return this.n;
    }
    public void setN() {
        this.n = 100;
    }
};

public class Main {
    public static void main(String[] args) {
       Point PointA = new Point(); // n = 10, val = 100
        PointA.n = 242; // ошибка, т.к. поле private
        Point.n = 100; // ошибка, т.к. поле private
        Point.val = 200; // верно, обращение к static полю
         ↪  через имя класса
        // n = 10, val = 100

        Point PointB = new Point(); // n = 10, val = 100
        PointA.setN(); // n = 100, val = 100
    }
};
```

## Статические блоки

**Статический блок (static-блоки)** — код, расположенный в статическом блоке, будет выполнен во время запуска программы, или при первой загрузке класса, еще до того, как этот класс будет использоваться в программе (т.е. до создания его экземляров, вызова статических методов и обращения к ним и т.д.).

```
public class A {
    static Date timeStart; // время запуска программы
    Date timeStartObj; // время инициализации объекта
    static {
        timeStart = new Date();
    }
    public A() {
        timeStartObj = new Date();
    }
};
```

```
public class Main {
    public static void main(String[] args) {
        A a = new A(); // timeStart != timeStartObj
    }
};
```

# Статические поля, методы, блоки.

## Доступ к статическим полям

**Утверждение**: если экземплярное поле является статичным, то это значение глобально для всех объектов.

**Утверждение**: статический метод может обратиться только к статическим переменным. => любой статический метод не имеет доступа к внутреннему состоянию объектов этого класса.

*Пример*:

```
public class Point {
    // ... объявления полей
    private static double y;
    // ...

    public static void setY(double varY) {
        y = varY; // this.y = varY - нельзя
    }
};
```

## Пример использования массивов объектов

```
public class Main {
    public static void main(String[] args) {
        // Объекты класса Point
        Point A = new Point("A", 5, 4);
        Point B = new Point("B", 100, 100);

        // Определение массива объектов из 100 элементов
        Point[] points = new Point[100];
        //      ~~~~~~ - имя массива объектов
        points[0] = new Point("AA", 2, 1);
```

```
        points[1] = new Point("BB", 34, 48);
        // ...
    }
};
```

*Замечание*: если в объявлении класса содержится несколько статических блоков, то код в них выполняется последовательно.

*Замечание*: совокупность всех `static`-блоков играет роль статического конструктора.

## Перегрузка методов

**Сигнатура метода (method signature)** — информация о количестве и типах формальных параметров в методе.

*Замечание*: если у метода нет формальных параметров, то говорят, что у метода пустая сигнатура.

**Перегрузка метода (method overloading)** — это объявление для заданного класса двух или более методов, имеющих одинаковое имя, но различные сигнатуры.

*Замечание*: решение о том, куда передается управление при вызове перегруженного метода X, принимается на этапе компиляции на основе сопоставления типов фактических параметров вызываемого метода и сигнатуры, имеющей имя X.

## Виртуальные методы

**Раннее связывание (early binding)** — определение адреса вызываемого экземлярного метода во время компиляции программы.

**Позднее связывание (late binding)** — определение адреса вызываемого экземлярного метода на основе информации о классе объекта во время выполнения программы.

**Виртуальные экземплярные методы** – экземплярные методы, для вызова которых выполняется позднее связывание называются **виртуальными**.

*Замечание*: в языке `Java` все методы являются **виртуальными**.

*Выводы*:

1. Все методы в Java виртуальные, следовательно, для любых методов выполняется позднее связывание.
2. В Java метод класса может быть определен в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
3. Таким образом, программисту не обязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно знать, что объект принадлежит классу или наследнику класса, в котором объявлен метод.

## Перегрузка экземплярного конструктора

**Экземплярный конструктор (instance constructor)** — экземплярный метод, предназначенный для инициализации только что созданного объекта.

*Замечание*: в Java и других объектно-ориентированных языках программирования операция создания объекта объединена с вызовом конструктора, что гарантирует отсутствие неинициализированных объектов.

*Замечание*: перегрузка конструктора осуществляется аналогично перегрузке методов, поскольку конструктор это такой же метод.

**Конструктор по умолчанию (default constructor)** — экземплярный конструктор с пустой сигнатурой.

*Замечание*: во всех языка программирования конструктор по умолчанию всегда создается даже если для класса не определен ни один конструктор.

**Статический конструктор (static constructor)** — статический метод, предназначенный для инициализации статических полей класса. Статический конструктор не имеет параметров и вызывается до вызова любого статического метода или конструктора класса.

*Замечание*: в Java нет статических конструкторов, т.к. конструктор по определению не может быть статическим. Не может существовать конструктора для класса, т.к. класс не является его экземпляром.

# Наследование, субтипизация

## Понятие субтипизации

Пусть тип данных `A` является **подтипом (subtype)** для типа данных `B`, тогда программный код, который рассчитан на обработку значений типа `B`, может быть корректно использован для обработки значений типа `A`.

*Пример*:

```
for (int i = 0; i < 10; i++) {}
for (double i = 0; i < 10; i++) {}
```

В данном случае, пример программного кода рассчитанный на обработку значений типа `double`, также предназначен для обработки значений типа `int`.

**Субтипизация (subtyping)** — это свойство языка программирования, означающее возможность использования подтипа в программе.

*Замечание*: для личного понимания субтипизации можно рассматривать **типы данных** как **множества значений**, тогда тип `A` является подтипом для `B` означает `B` подмножество `A`.

*Замечание*: семантика языков программирования, поддерживающих субтипизацию и ориентированных на статическую проверку допускает, что одно значение может иметь сразу несколько типов.

## Явная и неявная типизация

**Неявная типизация** — выбор типа на основе анализа структуры значения. (или выведение типа в статически типизированных языках)

**Явная типизация** — тип указывается программистом явно. (Используется в `Java`)

## Наследование как механизм явной субтипизации

**Наследование (inheritance)** — это способ получения нового класса на основе уже существующего класса, сочетающего усложнение

внутреннего состояния объектов, расширение интерфейса обрабатывающего сообщения и изначальные реакции на некоторые из них.

Если класс B наследуется от класса A, то A — **базовый класс (base class, superclass)**, B — **производный класс (derived subclass)**.

*Замечание*: при наследовании производный класс получает все экземплярные поля базового класса и все экземплярные методы, кроме конструкторов.

*Замечание*: тело любого конструктора производного класса должно начинаться с вызова одного из конструкторов базового класса.

**Множественное наследование** — производный класс может иметь более одного базового класса.

Java, C# не поддерживают множественное наследование. C++, Python, Perl6 — поддерживают.

## Абстрактные классы и интерфейсы

Альтернативой множественному наследованию в Java являются:

- **Абстрактные классы**
- **Интерфейсы**

*Пример*:

```
class ПроизводныйКласс extends БазовыйКласс {
    ...
}
```

*Замечание*: по умолчанию любой класс в Java является наследником встроенного класса Object.

*Замечание*: примеры методов класса Object: clone(), toString(), ...

*Замечание*: если конструктор базового класса является конструктором по умолчанию, то его вызов добавляется компилятором Java в конструктор производного класса автоматически. В противном случае необходимо явно вызвать конструктор базового класса в конструкторе производного класса.

*Пример*:

```
class Animal {
    private String species;
    public Animal(String species) {
        this.species = species;
    }
};

class Dog extents Animal {
    private String breed;
    public Dog(String breed) {
        super("Sobaka");
        this.breed = breed;
    }
};
```

## Неявное и явное приведение типов

*Замечание*: **неявное** приведение типов происходит тогда, когда два типа автоматически конвертируются.

*Замечание*: **неявное** приведение типов возможно, когда два типа данных совместимы. А также, когда мы присваиваем значение меньшего типа к большему.

*Замечание*: `char` и `boolean` несовместимы.

*Замечание*: числовые типы данных совместимы друг с другом. Например `int` и `double`.