Основы ООП

Немного истории

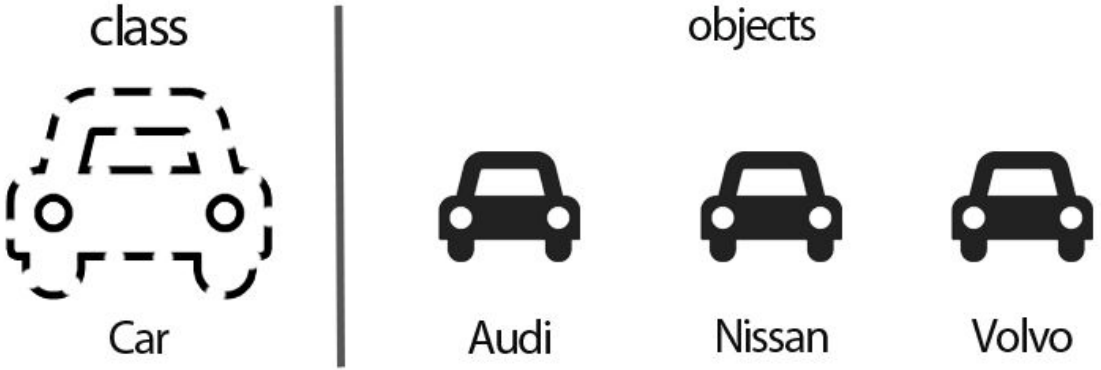
- Первым языком программирования, в котором были предложены основные понятия ООП, была **Си́мула-67 (Simula 67)**. В момент его появления в 1967 году в нём были предложены революционные идеи: **объекты**, **классы**, **виртуальные методы** и др.

- Первым широко распространённым объектно-ориентированным языком программирования стал **Smalltalk**. Здесь понятие **класса** стало основообразующей идеей для всех остальных конструкций языка.

Так что же такое ООП?

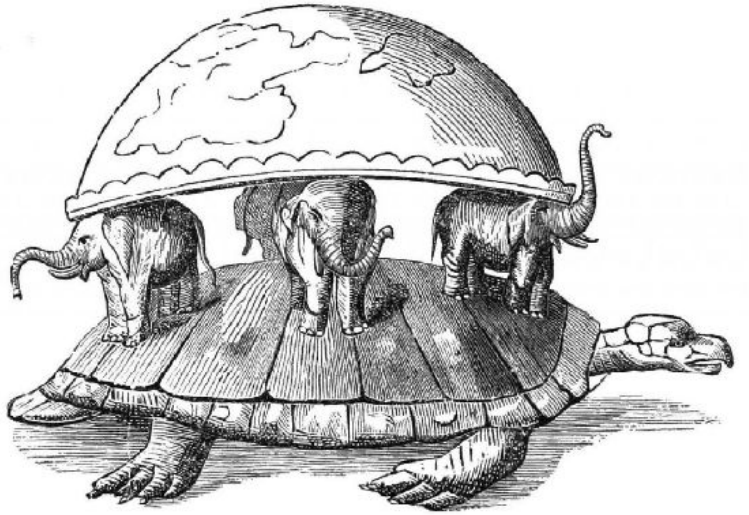
Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Понятия класс и объект



Класс - это шаблон, на основе которого может быть создан конкретный программный объект, он описывает свойства и методы, определяющие поведение **объектов** этого класса. Каждый конкретный **объект**, имеющий структуру этого класса, называется **экземпляром класса**.

Принципы ООП



1. Инкапсуляция
2. Абстрагирование
3. Наследование
4. Полиморфизм

Принцип 1. Инкапсуляция

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Изначальное значение слова «инкапсуляция» в программировании — объединение данных и методов работы с этими данными в одной упаковке («капсуле»).

Предоставление открытых методов для работы с объектом также является частью механизма инкапсуляции, так как если полностью закрыть доступ к объекту – он станет бесполезным.

Принцип 1. Инкапсуляция

```
public class Cat {

    public String name;
    public Integer age;
    public Integer weight;

    public Cat(String name, Integer age, Integer weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }

    public Cat() {
    }

    public void sayMeow() {
        System.debug('Meow!');
    }
}
```

Принцип 1. Инкапсуляция

```
public class Cat {

    public String name;
    public Integer age;
    public Integer weight;

    public Cat(String name, Integer age, Integer weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }

    public Cat() {
    }

    public void sayMeow() {
        System.debug('Meow!');
    }
}
```

```
Cat cat = new Cat();
cat.name = '';
cat.age = -1000;
cat.weight = 0;
```

Принцип 1. Инкапсуляция

```
public class Cat {

    private String name;
    private Integer age;
    private Integer weight;

    public Cat(String name, Integer age, Integer weight) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }

    public Cat() {
    }

    public void sayMeow() {
        System.debug('Meow!');
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
}

    public Integer getWeight() {
        return weight;
    }

    public void setWeight(Integer weight) {
        this.weight = weight;
    }
}
```

Принцип 1. Инкапсуляция / Преимущества

- Контроль за корректным состоянием объекта. Примеры этому были выше: благодаря сеттеру и модификатору private, мы обезопасили нашу программу от котов с весом 0.

- Удобство для пользователя за счет интерфейса. Мы оставляем «снаружи» для доступа пользователя только методы. Ему достаточно вызвать их, чтобы получить результат, и совсем не нужно вникать в детали их работы.

- Изменения в коде не отражаются на пользователях. Все изменения мы проводим внутри методов. На пользователя это не повлияет: если мы меняем что-то в работе метода, для него останется незаметным: он, как и раньше, просто будет получать нужный результат.

Принцип 2. Абстрагирование

Абстрагирование – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, **абстракция** – это набор всех таких характеристик.

Принцип 2. Абстрагирование

!!!

При создании программы также очень важно помнить, что с точки зрения разных задач один и тот же объект может обладать совершенно разными характеристиками.

Принцип 1. Абстрагирование

```
public class Model {
    private String name, height, hair, eyes;

    public Model(String name, String height, String hair, String eyes) {
        this.name = name;
        this.height = height;
        this.hair = hair;
        this.eyes = eyes;
    }

    //.....
}

public class Employee {
    private String name;
    private Integer socialInsuranceNumber, taxNumber;

    public Employee(String name, Integer socialInsuranceNumber, Integer taxNumber) {
        this.name = name;
        this.socialInsuranceNumber = socialInsuranceNumber;
        this.taxNumber = taxNumber;
    }

    //.....
}
```

Принцип 3. Наследование

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.

Принцип 3. Наследование/ Правила наследования

- Дочерний класс может быть родительским для другого класса, тот родительским еще для одного и т.д. Образуется цепочка наследования, которая может быть любой длины.

- Класс расширяет другой класс с помощью ключевого слова **extends** в определении класса. Класс может расширять только один другой класс, но он может реализовывать несколько интерфейсов.

- Класс, расширяющий другой класс, наследует все методы и свойства расширенного класса.

- Кроме того, расширяющий класс может переопределять существующие виртуальные методы с помощью ключевого слова **override** в определении метода.

Принцип 3. Наследование

```
public abstract class Animal {
    public virtual void makeSound() {}
    public void run() {
        System.debug('Animal is running');
    }
}
```


```
public class Dog extends Animal {
    public String name;
    public Dog(String name) {
        this.name = name;
    }
    public override void makeSound() {
        System.debug('Woof!');
    }
}
```

Dog

Cat

```
public class Cat extends Animal {
    public String name;
    public Cat(String name) {
        this.name = name;
    }
    public override void makeSound() {
        System.debug('Meow!');
    }
}
```

Принцип 3. Наследование/ Преимущества и недостатки

- Повторное использование кода.
- Устанавливается логическое отношение «is a» (является кем-то, чем-то). Например: Dog is an animal. (Собака является животным).
- Модуляризация кода.

Принцип 4. Полиморфизм

Полиморфизмом в объектно-ориентированных языках чаще всего называют механизмы, позволяющие работать с разными данными и разными функциями используя одно и то же имя.

«Один интерфейс — много реализаций».

Принцип 4. Полиморфизм

- **Перегрузка функций.** Позволяет создавать несколько функций с одним и тем же именем в одном классе. Это как правило используется для того, чтобы дать одно имя группе функций, выполняющих сходные задачи, но работающих с разными исходными данными.

- **Переопределение (overriding).** Используется в том случае, если какая-то функция объявлена в родительском классе, и в дочернем нужна функция с тем же именем и параметрами, но выполняющая другие действия.

Принцип 4. Полиморфизм

Перегрузка функций:

```
public Integer sum(Integer x, Integer y) {
    return x + y;
}

public Double sum(Double x, Double y) {
    return x + y;
}

public Integer sum(Integer x, Integer y, Integer z) {
    return x + y + z;
}
```

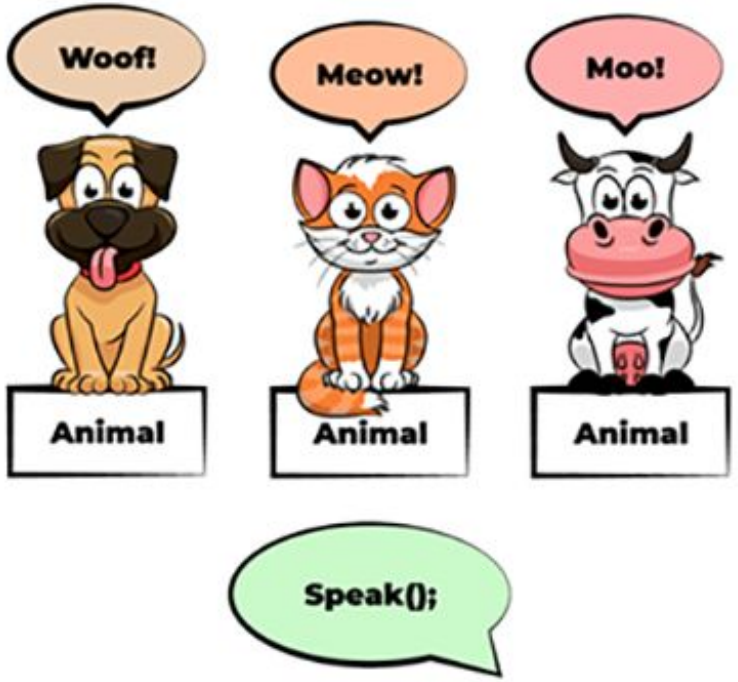
У перегружаемых функций могут отличаться типы параметров, или их количество и типы возвращаемых значений. Но параметры должны отличаться обязательно. Двух функций с одинаковым именем, одинаковым количеством и одинаковыми типами параметров в одном классе быть не может!

Принцип 4. Полиморфизм

Переопределение (overriding):

```
public abstract class Animal {
    public virtual void makeSound() {}
    public void run() {
        System.debug('Animal is running');
    }
}

public class Cat extends Animal {
    public String name;
    public Cat(String name) {
        this.name = name;
    }
    public override void makeSound() {
        System.debug('Meow!');
    }
}
```



Принцип 4. Полиморфизм / Преимущества

- Позволяет записывать методы лишь однажды и затем повторно их использовать для различных типов данных, которые, возможно, еще не существуют (обобщенные действия или алгоритмы).
- Возможность работать с несколькими типами так, как будто это один и тот же тип.
- Один и тот же интерфейс может быть использован для создания методов с разными реализациями.

Ключевое слово Static

Переменные, методы, код инициализации

Характеристики Static-методов, переменных, кода инициализации:

- **Связаны с классом.**

  Модификатор **static** в Java напрямую связан с классом, если поле статично, значит оно принадлежит классу, если метод статичный, аналогично — он принадлежит классу.

  Исходя из этого, можно обращаться к статическому методу или полю используя имя класса:

```
1 Integer i = -42;
2 Integer i2 = math.abs(i);
3 system.assertEquals(i2, 42);
```

Характеристики Static-методов, переменных, кода инициализации

- Статические методы имеют преимущество в применении, т.к. **отсутствует необходимость каждый раз создавать новый объект для доступа** к таким методам. Статический метод можно вызвать, используя тип класса, в котором эти методы описаны. Class Math — замечательный пример со статическими переменными и методами.

```
public static Decimal getCicleCircumference(Decimal radius) {
    return 2 * Math.PI * radius;
}
```

- **Инициализируются только когда класс загружен**. Порядок инициализации сверху вниз, в том же порядке, в каком они описаны в исходном файле Apex класса.

- Статические методы и переменные **можно использовать только с внешними классами.** Внутренние классы не имеют статических методов или переменных.

Блок инициализации

```
01 public class MyClass {
02
03     class RGB {
04
05         Integer red;
06         Integer green;
07         Integer blue;
08
09         RGB(Integer red, Integer green, Integer blue) {
10             this.red = red;
11             this.green = green;
12             this.blue = blue;
13         }
14      }
15
16    static Map<String, RGB> colorMap = new Map<String, RGB>();
17
18     static {
19         colorMap.put('red', new RGB(255, 0, 0));
20         colorMap.put('cyan', new RGB(0, 255, 255));
21         colorMap.put('magenta', new RGB(255, 0, 255));
22     }
23 }
```

Класс может иметь любое количество блоков кода инициализации экземпляра. Они могут появляться в любом месте тела кода. Блоки кода выполняются в том порядке, в котором они появляются в файле, так же, как и в Java.

Статический код инициализации можно использовать для инициализации статических конечных переменных и объявления статической информации.

Транзакции Apex

```apex
public class P {
   public static boolean firstRun = true;
}
```

```apex
trigger T1 on Account (before delete, after delete, after undelete) {
        if(Trigger.isBefore){
           if(Trigger.isDelete){
              if(p.firstRun){
                  Trigger.old[0].addError('Before Account Delete Error');
                   p.firstRun=false;
               }
            }
         }
}
```

Статическая переменная является статической только в рамках транзакции Apex. Она не является статической на сервере или во всей организации. Значение статической переменной сохраняется в контексте одной транзакции и сбрасывается через границы транзакции. Например, если запрос Apex DML приводит к многократному запуску триггера, статические переменные сохраняются в этих вызовах триггера. Рекурсивный триггер может использовать значение переменной класса для выхода из рекурсии.

СПАСИБО
ЗА ВНИМАНИЕ!