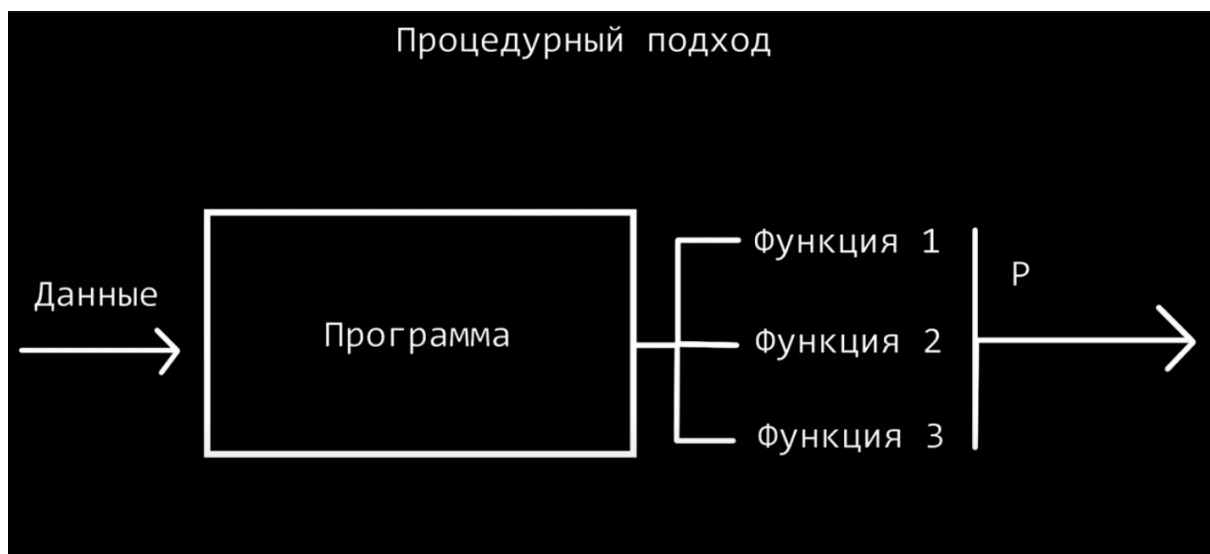




# ООП

Подходы программирования.

Процедурный.

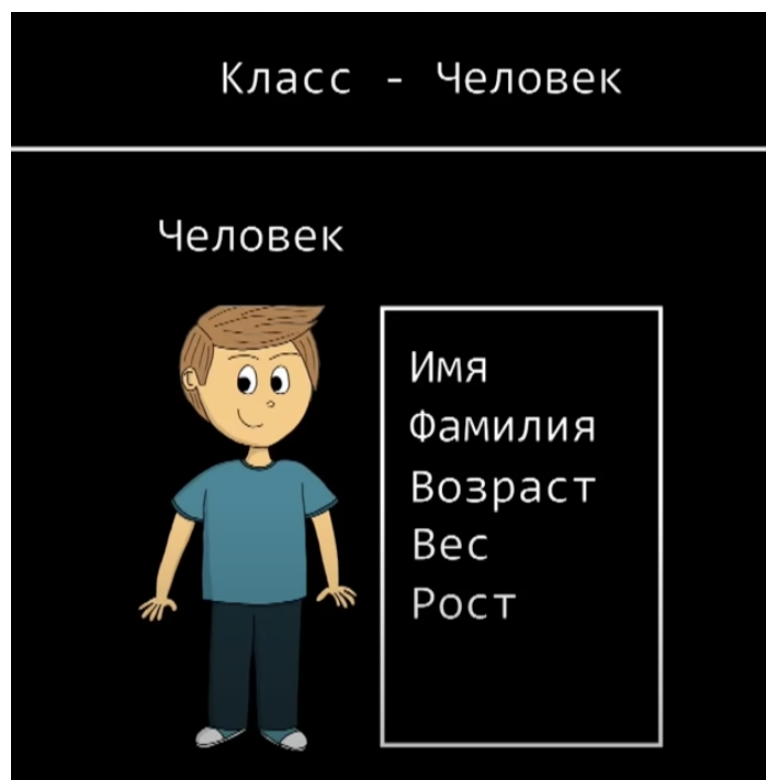


В процедурном подходе программа ожидает на вход некоторые данные, выполняет ряд процедур, они же функции, и на выходе возвращает какой то результат вычислений этих функций.

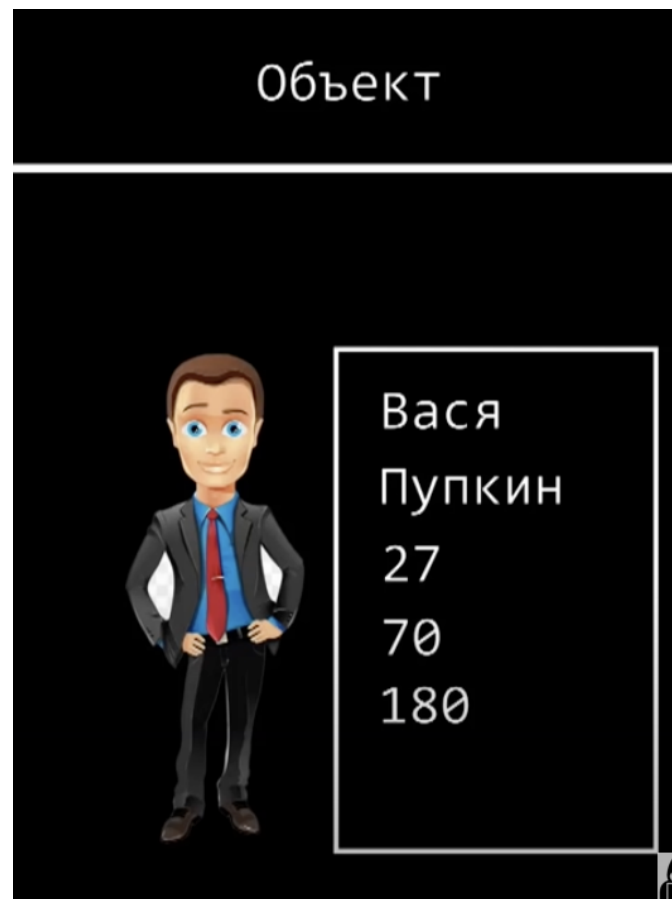
Когда программы стали большими, их стало тяжело писать в процедурном подходе и придумали объектно-ориентированный подход.



Например есть человек, можно описать его какие то общие характеристики, имя, рост, вес, возраст, фамилия. в ООП Человек - это КЛАСС.



Конкретный представитель КЛАССА - называют объектом. КЛАСС - это некоторое описание характеристик, а объект это ЭКЗЕМПЛЯР класса. У которого общие характеристики, которые описаны в классе представлены конкретными значениями.





На базе описанного класса мы можем создавать сколько угодно объектов. Но хорошая практика - это создавать определенный класс, под свои задачи.

```
class Rectangle {  
    width;  
    height;  
  
    constructor(w:number ,h: number) {  
        this.width = w;  
        this.height = h;  
    }  
    calculateArea(){  
        return this.width * this.height  
    }  
}  
  
const rect = new Rectangle(10, 5)  
const rect2 = new Rectangle(12, 2)  
const rect3 = new Rectangle(15, 14)  
  
// создаем 3 экземпляра (объекта) на основе описанного класса Rectangle
```

**3 основных столпа ООП это Инкапсуляция, Наследование, Полиморфизм.**

Инкапсуляция тесно пересекается с понятием сокрытие. Суть инкапсуляция в том, что сам класс является капсулой, которая содержит в себе св-ва и методы, для работы со свойствами.

В классе могут быть как публичные свойства, так и приватные. Публичными мы можем пользоваться из вне, в любой части нашего кода. Для разделения публичных и приватных свойств, существуют модификторы свойства public и private. Приватные свойства можно использовать только внутри класса, вызывать их из вне невозможно.

Зачастую свойства делают приватными, мы можем установить их только тогда, когда создаем экземпляр на основе класса. А чтобы получить свойства или переназначить другое значение, существуют геттеры и сеттеры.

```
class Rectangle {
    private _width;
    private _height;

    constructor(w:number ,h: number) {
        this._width = w;
        this._height = h;
    }

    get width(){
        return this._width
    }
    set width(value){
        if (value <= 0) {
            this._width = 1
        } else {
            this._width = value
        }
    }

    calculateArea(){
        return this._width * this._height
    }
}
```

Таким образом вызывая метод get у экземпляра объекта, мы получим значение его width.

А для установки нового значения `width` мы можем использовать `set`, передав новое значение, но в сеттере класса происходит некая проверка, на основе которой происходит изменение `width`. В данном примере мы имеем свойство `height` как приватное, для него мы не указали геттеры и сеттеры, поэтому из вне мы не можем с ним работать.

**По умолчанию свойства и методы в классе устанавливают модификатор доступа в `public`, но хорошей практикой считается явное указание `private` или `public`.**

В следующем примере мы задали все свойства как приватные. Но для `userName` и `userPassword` у нас есть геттеры и сеттеры, мы можем влиять на них из вне, а `userId` должен быть уникальным и никогда не меняться, поэтому для `userId` у нас существует только геттер, что бы мы могли получить его только для каких то проверок и чтения. Таким образом мы никак не сможем поменять ID и оно всегда будет уникальным у каждого экземпляра класса.

```
class User {

    private _userName
    private _userPassword
    private _userId

    constructor(name: string, password: string) {
        this._userName = name
        this._userPassword = password
        this._userId = generateRandomId()
    }
    get userName() {
        return this._userName;
    }

    set userName(value) {
        this._userName = value;
    }

    get userPassword() {
        return this._userPassword;
    }

    set userPassword(value) {
        this._userPassword = value;
    }

    get userId() {
        return this._userId;
    }
}
```

Таким образом инкапсуляция и сокрытие деталей в том, что мы можем работать со свойствами объектов, только с помощью определенных методов, которые описаны в классе и работают с какой то логикой, о которой мы можем не знать всё, но знать например, что если нам нужно что то добавить мы должны вызвать метод `user.add()` и передать в него что либо, или `user.delete()` , зная, что это удалит у юзера какие либо поля.

## Наследование

Наследование позволяет расширять функционал и добавлять новые свойства, которых нет в родительском классе. Например у нас есть класс `Animal` и мы хотим расширить его созданием нового класса `Dog`.

```
class Animal {
    // .....
    // тут описаны свойства и методы класса, как в примерах выше
    // .....
}

class Dog extends Animal {
    // .....
    // тут описаны свойства и методы класса, как в примерах выше
    // .....
}
```

**Таким образом, экземпляры объектов, созданные от класса `Dog` будут иметь доступ к свойствам и методам из класса `Animal`.**

```
class Animal {
    private _isBig
    private _isFlying

    constructor(isBig: boolean, isFlying: boolean) {
        this._isBig = isBig
        this._isFlying = isFlying
    }

    sayIsBigOrNot() {
        this._isBig ? console.log('Im big!') : console.log('Im little!')
    }
}
```

```

class Dog extends Animal {
  private _name
  private _age
  private _voice

  constructor(isBig: boolean, isFlying: boolean, name: string, age: number,
voice: string) {
    super(isBig, isFlying); // <- вызываем конструктор родительского класса
    this._age = age
    this._name = name
    this._voice = voice
  }
}

const nord = new Dog(true, false, 'Nord', 3, 'bark')
nord.sayIsBigOrNot()

```

Мы расширили класс `Animal` классом `Dog`. Так экземпляр класса `Dog`, который мы назвали `nord` (объект) теперь имеет доступ к методу `sayIsBigOrNot()`, и к конструктору родительского класса `Animal`. Вызов метода `super()` в конструктора класса `Dog`, вызывает конструктор родительского класса, в который мы должны как параметры передать те свойства, которые находятся в родительском классе `Animal`.

## Полиморфизм

выделяют 2 типа полиморфизма Параметрический(истинный) и ad-hoc (Мнимый)

Полиморфизм это принцип, который позволяет одному фрагменту кода работать с разными типами данных.



# ad-hoc

## (МНИМЫЙ)

```
class Calculator {  
  
    add(a: number, b: number): number {  
        return a + b;  
    }  
  
    add(a: string, b: string): string {  
        return a + b  
    }  
}  
  
add(5,5)      -> result = 10;  
add("5", "5") -> result = "55";
```

Вызывая один и тот же метод , в зависимости от переданных в него параметров, мы получим разный результат, в случае с числами мы получим их сумму, а в случае со строками, метод сделает конкатенацию. Это происходит из за перегрузки методов.

```
class Human {  
    protected _name  
    private _lastName  
  
    constructor(name: string, lastName: string) {  
        this._name = name;  
        this._lastName = lastName;  
    }  
  
    public greeting(){  
        console.log(`Hello my name is ${this._name} and I'm Human!`)  
    }  
}  
  
class Builder extends Human {  
    private _age
```

```

    constructor(name: string, lastName: string, age: number) {
        super(name, lastName);
        this._age = age
    }

    greeting(){
        console.log(`Hello my name is ${this._name} and I'm Builder!`)
    }
}
const human = new Human('Petr', 'Ololoshin')
const builder = new Builder('Roman', 'Chuchval', 29)

human.greeting()
builder.greeting()

```

Мы переписали одноименный метод `greeting()` в дочернем классе, теперь при вызове метода у объектов, созданных от разных классов. Будет выводиться разный текст, зависящий от того, какой класс мы использовали при создании нового объекта.

```

Hello my name is Petr and I'm Human!
undefined
Hello my name is Roman and I'm Builder!
undefined

```

Разная работа одного и того же метода в разных классах и называется полиморфизмом.

## Взаимодействие между классами. Агрегация и Композиция

### Композиция

```
// Композиция и агрегация
class Engine {
}

class Wheel {
}

class Car {
    engine: Engine;
    wheels: Wheel[]
}
```

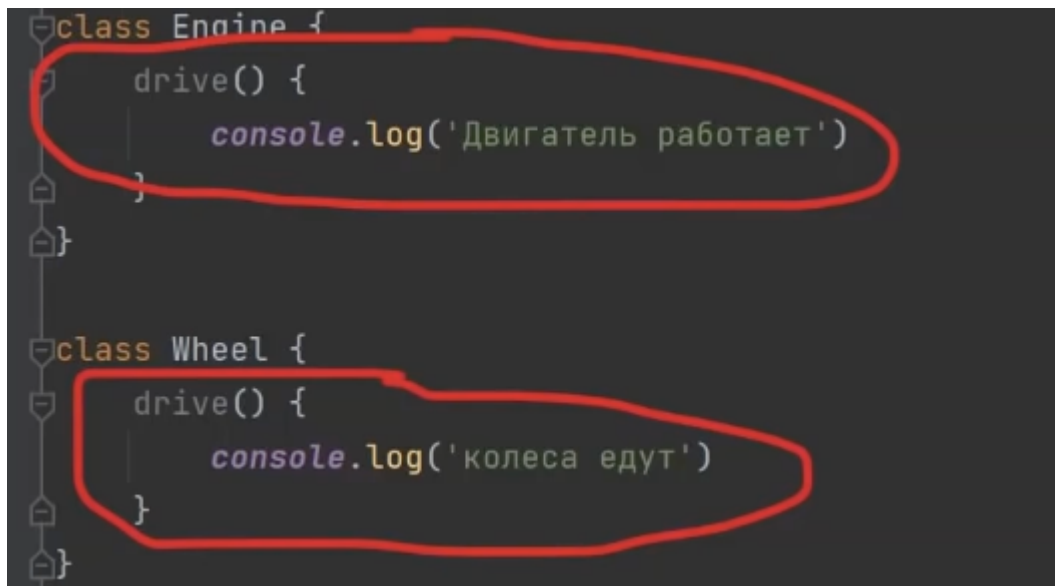
В классе Car объявляем что у нас будет один двигатель Engine и массив Колес.

```
class Car {
    engine: Engine;
    wheels: Wheel[]

    constructor() {
        // Композиция
        this.engine = new Engine()
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
    }
}
```

После этого в конструктора автомобиля мы добавляем новый объект двигателя и в массив добавляем 4 объекта колеса. Самое важное здесь, что эти объекты

создаются внутри класса Car , а не где то снаружи.



```
class Engine {  
  drive() {  
    console.log('Двигатель работает')  
  }  
}  
  
class Wheel {  
  drive() {  
    console.log('колеса едут')  
  }  
}
```

The image shows a code editor with two JavaScript classes. The first class is `Engine`, which has a `drive()` method that logs 'Двигатель работает' to the console. The second class is `Wheel`, which also has a `drive()` method that logs 'колеса едут' to the console. Both `drive()` method blocks are circled in red.

далее у двигателя и колеса создаем метод, который выводит что то в консоль.

```
class Car {  
  engine: Engine;  
  wheels: Wheel[]  
  
  constructor() {  
    // Композиция  
    this.engine = new Engine()  
    this.wheels.push(new Wheel())  
    this.wheels.push(new Wheel())  
    this.wheels.push(new Wheel())  
    this.wheels.push(new Wheel())  
  }  
  // делегирование  
  drive() {  
    this.engine.drive();  
    for (let i = 0; i < this.wheels.length; i++) {  
      this.wheels[i].drive()  
    }  
  }  
}
```

потом в классе Car делаем метод, который будет вызывать метод drive у двигателя ( который создан в классе car) и итерируясь по массиву колес, вызывать их метод drive.

```
325
326   const bmw = new Car()
327   bmw.drive()
328
```

Car > constructor() > wheels

Terminal: Local × + ▾

Двигатель работает  
колеса едут  
колеса едут  
колеса едут  
колеса едут

соответственно если мы создадим новый объект на основе класса Car и вызовем метод drive() то увидим такой результат.

## Агрегация.

```

class Freshener {
}

class Car {
    engine: Engine;
    wheels: Wheel[];
    freshener: Freshener;

    constructor(freshener) {
        // Агрегация
        this.freshener = freshener;
        // Композиция
        this.engine = new Engine()
        this.wheels = []
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
        this.wheels.push(new Wheel())
    }
}

```

Главная идея в том, что на основе класса Freshener внутри конструктора Car мы не создаем объекты. Он живет отдельно, в отличии от колес и двигателя.

## Абстрактные классы и интерфейсы

## Интерфейсы

```
interface Reader {  
    read(url);  
}  
  
interface Writer {  
    write(data);  
}  
  
class FileClient implements Reader, Writer {  
    read(url) {  
        // логика  
    }  
    write(data) {  
        // логика  
    }  
}
```

Создаем 2 интерфейса, где описаны некие методы, только описаны, но не реализованы.

Затем создаем класс FileClient где имплементируем наши интерфейсы. В классе мы обязаны РЕАЛИЗОВАТЬ все методы, которые описаны в имплементируемых интерфейсах.

Мы можем имплементировать сколько угодно интерфейсов. И один и тот же интерфейс могут имплементировать сколько угодно классов, каждый внутри себя может реализовать метод из интерфейса по своему, со своей логикой.