@pyplane_code

Javascript OOP

JS

@pyplane_code

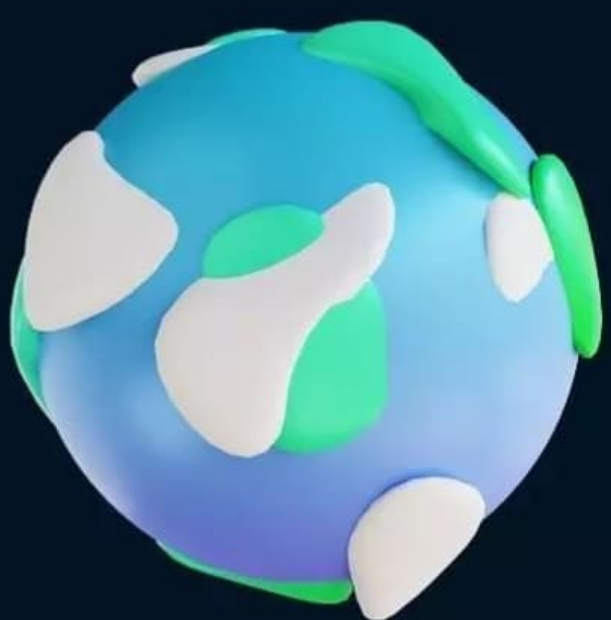
01

Introduction

In this tutorial we will explore the fundamentals for **Object Oriented Programming** (OOP) in Javascript in a practical way with examples

Table of contents

- first class
- first instances
- inheritance
- encapsulation
- polymorphism
- abstract classes



First class

On this slide we are creating our first class. Remember to add the constructor method that will enable the creation of instances (objects) based on our class

```
class Animal {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    getInfo() {
        return (
            `The name of the animal is
            ${this.name} and age is ${this.age}`
        )
    }
}
```

First instances

On the previous slide we created a new class, added a constructor method that takes in 2 arguments - name and age. To create an object from this constructor method we need to use the new keyword and pass in the given name and age of the animal object that we are creating. We also have added a getInfo method that returns the information about the object

```
const firstAnimal = new Animal('Rex', 2)
console.log(firstAnimal)
console.log(firstAnimal.getInfo())

const secondAnimal = new Animal('Barney', 5)
console.log(secondAnimal)
console.log(secondAnimal.getInfo())


> Animal {name: 'Rex', age: 2}
  The name of the animal is Rex and age is 2
> Animal {name: 'Barney', age: 5}
  The name of the animal is Barney and age is 5
```

Inheritance

Class inheritance is a feature that enables certain classes to take all the methods and properties of another one (parent class) and makes it possible to extend the parent class by adding more

```
class Dog extends Animal {
    constructor(name, age, breed) {
        super(name, age)
        this.breed = breed
    }

    bark(){
        return 'woof'
    }
}

class Cat extends Animal {
    constructor(name, age, weight) {
        super(name, age)
        this.weight = weight
    }
}

const myDog = new Dog('Rex', 2, 'German Shepard')
console.log(myDog.getInfo())
console.log(myDog.breed)
console.log(myDog.bark())

const myCat = new Cat('Whiskers', 5, '5kg')
console.log(myCat.getInfo())
console.log(myCat.weight)
```

```
The name of the animal is Rex and age is 2
German Shepard
woof
The name of the animal is Whiskers and age is 5
5kg
```

Encapsulation

Encapsulation is a restriction mechanism making accessing the data impossible without using special methods dedicated for this. In the example below we marked weight as a private property and in order to get and set a value we need to use the getter and setter method

```
class Cat extends Animal {
    #weight;                          → mark as private
    constructor(name, age, weight) {
        super(name, age)
        this.#weight = weight
    }

    getWeight() {                     → getter
        return this.#weight
    }

    setWeight(weight) {               → setter
        this.#weight = weight
    }
}

const myCat = new Cat('Whiskers', 5, '5kg')
console.log(myCat.getWeight())
myCat.setWeight('6kg')
console.log(myCat.getWeight())
```

Output:

```
5kg
6kg
```

Polymorphism

Polymorphism is a concept that utilizes inheritance for reusing methods multiple times with a different behaviour depending on class types. To understand this lets look at our example - in the dog class we will remove the bark method and in the animal class we'll add a makeSound method which will be overridden by cat and dog classes

```
class Animal {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    makeSound() {
        return (
            `Some nice sound made`
        )
    }
}

class Dog extends Animal {
    constructor(name, age, breed) {
        super(name, age)
        this.breed= breed
    }

    makeSound() {
        return 'woof'
    }
}
```

```
class Cat extends Animal {
    constructor(name, age, weight) {
        super(name, age)
        this.weight = weight
    }

    makeSound() {
        return 'meow'
    }
}

const myDog = new Dog('Rex', 2, 'German Sh.')
const myCat = new Cat('Whiskers', 5, '5kg')

console.log(myDog.makeSound())
console.log(myCat.makeSound())
```

output:

```
woof
meow
```

Abstract class p1

Abstract class is a class which can't be instantiated and require subclasses which inherit from a particular abstract class to provide implementations. We will change the Animal class to an abstract class. It will not be possible to create a instance of this class anymore like on slide number 3 and we will mark makeSound as an abstract method - in order to use it, a subclass must declare its own implementation of this method

```
class Animal {
    constructor(name, age) {
        this.name = name;
        this.age = age;
        if(this.constructor == Animal){
            throw new Error("Can't create a instance of Abstract class");
        }
    }

    makeSound() {
        throw new Error("abstract method doesn't have an implementation");
    }
}
```

Abstract class p2

```
class Dog extends Animal {
    constructor(name, age, breed) {
        super(name, age)
        this.breed= breed
    }

    makeSound() {
        return 'woof'
    }
}

class Cat extends Animal {
    constructor(name, age, weight) {
        super(name, age)
        this.weight = weight
    }

    // makeSound() {
    //     return 'meow'
    // }
}

// const myAnimal = new Animal('Barney', 3)
const myDog = new Dog('Rex', 2, 'German Sh.')
const myCat = new Cat('Whiskers', 5, '5kg')

console.log(myCat.makeSound())
```

→ can't instantiate anymore

→ this will give us an error since makeSound() method isn't avaliable