OOPs

Created	@August 3, 2022 10:27 PM
Class	
Type	
Materials	
Reviewed	☐

Let’s say we are building an e-commerce application. In an e-commerce app, we have wide variety of `Products` . When we say the word `Products` we refer to a wide range of real life instances or commodities. For example: `Iphone 12` `macbook pro` etc these are real life products.

So in computer science we sometimes need to define a blueprint of real life instances. As in the above example `Products` is kinda like a blue print / template because every product will be having common properties i.e. every product has a name, has a price, has an image etc. So this blue print is not an actual real life instance but just an idea of what properties real life instances will be having. And things like `Iphone 12` `Iphone 13` `macbook pro` etc these are real life instances / entity .

Similarly for a building, an engineer first prepares the blue print of a flat, the blue print is not an actual flat, but just a concept that how flat will look, but when the workers actually complete the building, then we get multiple actual real flats.

So these products are concepts, but iphone, macbook are real life entitites.

These blueprints are called as `Class` and real life entity is called `objects`

Class

We can create classes in JS using keyword `class`

Inside class we can define two things,

- Properties of the real life entity
- Behaviour of the real life entity

For example let's think about what can be the properties of a product. So a product will be having a name, description, price, rating etc.

And products can have the following behaviour, display the product, buy the product, update the product, etc.

To define the properties of a real life entities we use `data members` and to define the behaviours we use `member functions` .

To define member functions we can just write functions in the class. But to define data members we have to define them inside something called as `constructor`.

What is a constructor ?

when we try to create a new object (new real life entity), we can do it by calling the constructor of the class. So, we can say, constructor is the first function that gets called while creating a new object. In languages like JAVA, C++ constructor function has the same name as that of the class. But in JS to define a constructor we have to use `constructor` keyword.

```
class Product {
  // constructor will be the first function to be called for creating an object
  constructor() {

  }

  // member function
  displayProduct() {

  }

  buyProduct() {

  }

}
```

To define data members in the class we will initialise them inside the constructor using `this` keyword. We can write `this.propertyName` to define a new data member of the class.

```
class Product {
  // constructor will be the first function to be called for creating an object
```

```
  constructor(n, p) {
    this.name = n;
    this.price = p;
  }

  // member function
  displayProduct() {

  }

  buyProduct() {

  }

}
```

How to create instance of a class or we can say object ?

For a class in order to create an object we can use `new` keyword. The `new` keyword is followed by the class name which actually calls the constructor. So all the parameters the constructor is expecting we can pass it there.

```
const iphone = new Product("Iphone", 100000); // calls the constructor to create obj
console.log(iphone); // prints the product
console.log(typeof iphone); // type of iphone is a object
```

So when we write `this.name = n` it becomes a property / data member of the class.

And data members can be access inside member functions.

```
class Product {
  // constructor will be the first function to be called for creating an object
  constructor(n, p) {
        console.log("Constructor called !!!!")
    this.name = n;
        this.price = p;
  }

  // member function
  displayProduct() {
        console.log("Name of the product is", this.name);
        console.log("Price of the product is", this.price);
  }
```

```
  buyProduct() {

  }

}

const iphone = new Product("Iphone", 100000);
// member functions can be called by . operator
iphone.displayProduct();
```

To call a member function we can use `.` operator. For example: `iphone.displayProduct()` will actually call the member method.

Objects

So Javascript is not actually purely Object oriented programming language. It's better to instead call it as Objects oriented programming. When I say Objects oriented then I mean that JS objects are going to be the central focus of this system and the whole OOPs concept will revolve around it. Classes as features were released way later in JS i.e. classes were not always present in JS, but objects were.

Objects are one of the ecma-script language types.

There are multiple ways to create objects in JS, using classes or using `{}` etc.

this in JS

Behaviour of `this` keyword in JS is different than other languages like `C++` `Java` etc. What does it mean in JS ? In javascript, `this` references to the execution context i.e. this refers to the context from where function or class is called. The context is determined entirely by how function was called.

Let's write a complex number class.

We need to define a class which resembles a real life complex number with a real and imaginary part. And let's try to define a member function that can help us to add two complex numbers.

```
class Complex {
  constructor(r, i) {
    this.real = r;
    this.imag = i;
  }

  add(c) {
    this.real += c.real;
    this.imag += c.imag;
  }
  display() {
    console.log(this.real, " + i", this.imag);
  }

}
```

new Keyword

It looks like new keyword just calls the class constructor. But it does much more. But new keyword has nothing to do with classes. In fact classes are just a syntactic sugar done over function to make the JS syntax consistent with other languages. Whenever a new keyword is used with a function it does 4 steps . These 4 steps actually helps the new keyword to initialise a JS object with `this` keyword associated with it.

```
function product(n, p) {
    this.name = n;
    this.price = p;
}

const iphone = new product("iphone", 100000);
console.log(iphone)
```

1. The new keyword creates a brand new empty JS object first.
2. It does some linking (we will learn later)
3. Calls the function with `this` property set to the new empty object we created in step 1.
4. At the last of the function execution, if the function doesn't return an object, it assumes we want to return `this` and hence if no object is returned at the last of the

function it returns `this`. Instead of an object if we will let's say return a number then also it will return `this`

Arrays are objects in JS, so if we return arrays, then also `this` won't be returned.

```
function product(n, p) {
    console.log("this at start", this);
    this.name = n;
    this.price = p;
    console.log("this at end", this);
    // return {marks: 100}
    return 100
}

const iphone = new product("iphone", 100000);
console.log(iphone)
```

Difference in function based constructor and class based constructor

Class based constructors cannot be used as normal functions. They are bound to be used with `new`.

Whereas normal function based constructors can be called without `new` as well .

Arrow Functions and this

Arrow functions don't work in the same way as normal functions and don't assign this to the calling context.

So what does `this` resolves to inside arrow functions ? So they behave like they don't have any this. And it lexically resolves `this`. So that means it one by one look on the outer scope for the presence of this just like normal lexical scope resolution.

Pillars of OOP

- Inheritance
- Encapsulation
- Polymorphism

- Abstraction

Inheritance

If you visit book my show, then there are different type of shows, example: movies, comedy shows, concerts etc.

Every show has got some common properties like name of the show - movies → movie name, comedy show → set name , concert → concert name

Similarly every different type of show will be having some description, some price etc.

But there will be some non common properties also, like movies will be having details of theatre , whereas comedy show will be having venue, and concert will be having venue.

If you want to represent everything in form of classes what can be a good design ? One way is we can prepare separate movie class, show class and concert class. There will be problem with this because not only they have common properties but they have common behaviours also, example booking. So if we will create separate classes we need to re-write the logic of booking. So what can we do ?

For these situations, inheritance comes to the rescue. Inheritance ensures code reusability. Inheritance says that we can have parent classes and child classes, so that child classes can inherit properties from the parent class. So we can prepare a show class, and then movie class, concert class and comedy class will inherit the common properties from show class example name, description, rating and behaviours like booking.

We can segregate out common properties in a separate class and make it the parent. Then we can have child classes inherit the properties and behaviours from parent class.

```
class A {
    constructor() {
        this.x = undefined;
    }
    display() {
        console.log(this.x);
    }
}

class B extends A { // B-> child, A -> parent
    constructor() {
        super(); // calls the parent constructor
```

```
        this.y = 10;
    }
}

class C extends A {
    constructor() {
        super();
        this.z = 9;
    }
}

const bObj = new B();
console.log(bObj)
const cObj = new C();
console.log(cObj)
```

So the above example, can help us to understand inheritance in JS. When a child class calls it's constructor, we need to first of all call the super constructor i.e constructor of the parent, so that we can define the reference of `this`

Abstraction

Abstraction refers to hiding the internal details of an implementation and only exposing relevant details using which user can actually use the features developed by us. For example: In js we don't know internally what is the implementation of `arr.sort()` we just know how to use it that's it. We don't care about algorithmic implementation going behind the functions.

Encapsulation

Encapsulation provides a protective shield to our code so that only relevant stakeholders can access and modify the internal details of a class.

We can use access modifiers to limit the access of the properties outside of the class.

- public
- private

```
class A {
    #y; // declare the private variable
    constructor() {
        this.x = undefined;
```

```
        this.#y = null; // private
    }
    display() {
        console.log(this.x);
    }

    getY() { // getter
        console.log(this.#y);
    }

    putY(el) { // setter
        if(typeof el != 'string') {
            return;
        }
        this.#y = el;
    }
}

const aObj = new A();
console.log(aObj);

aObj.x = 10;
// aObj.#y = 9;

console.log(aObj)
aObj.putY(10)
aObj.getY();
```

```
class Complex {
  #real;
  #imag;
  constructor(r, i) {
    this.#real = r;
    this.#imag = i;
  }

  getReal() {
    return this.#real;
  }

  getImag() {
    return this.#imag;
  }

  setReal(r) {
    this.#real = r;
  }

  setImag(i) {
    this.#imag = i;
  }
```

```
  display() {
    console.log(`${this.#real} + i${this.imag}`);
  }

  add(c) {
    this.#real += c.getReal();
    this.#imag += c.getImag();
  }


}
```

Polymorphism

Polymorphism means multiple forms. There are two types of polymorphism, compile time and run time.

Prototypes

By understanding the prototype system of JS, we will be able to decipher most of the logic around creation and working of the JS objects.

Objects are created by constructor calls via `new` keyword.
Now we know that in terms of OOP, classes are the blueprints and objects / instances are the working real life model of those blueprints. This is the most common understanding. Now, this class based coding is kind of creating copies. Why I am saying, we create new copies, because let's say we have the product class, we created new objects out of it, and then in the class we made a change around some behaviour or some property, do you think that change will reflect in an already created object ? And let's say we made some changes in the new product object, do you think that change will reflect in the class ? Because in languages like C++ and java, the relationship between class and object exist for a split of second and the moment we have our new object with us, there is no relation left between them. Can we say even the inheritance operation involves copying.

In JS, the copy operation doesn't exist for these classes and objects. Instead there exist something called as Linking.

A constructor call makes an object `linked to` its own prototype.

Prototypes are the mechanism by which JS objects inherit features from one another. If you will create a new object inside browser, then we are able to see a Prototype property as well. So if you do, `obj.` then you will get a list of properties associated with obj, few of them we defined ourselves but other like `toLocaleString`, `isPrototypeof` these method we didn't create. So these are somehow inherited from somewhere. Where they come from ? Every object in JS has an inbuilt property called it's `prototype`

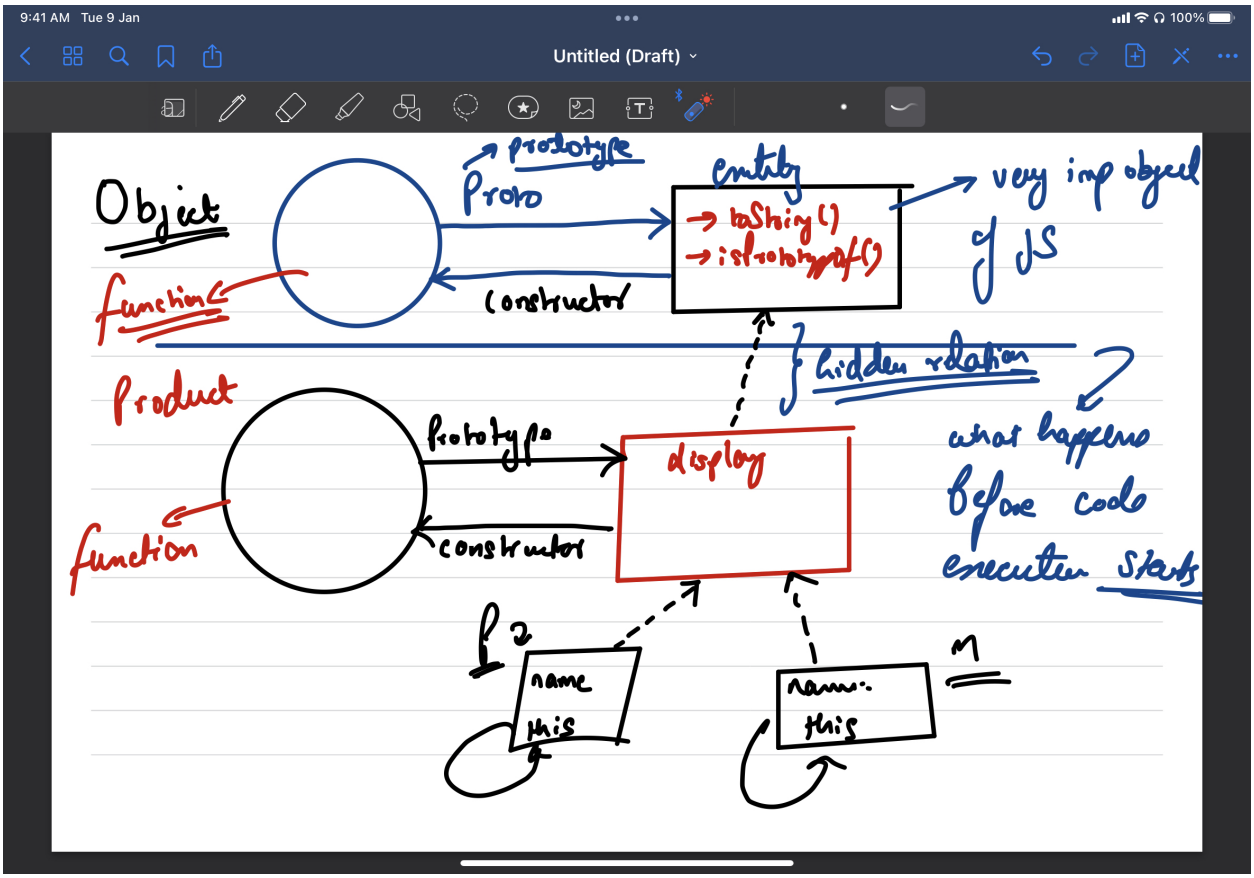
The prototype is itself an object, and it is an object then it will have it's own prototype. It leads to a chain of prototypes. The chain ends, when we reach a prototype that has null for it's own prototype.

When we try to access a property of an object, if the property is present then we access it, but if it is not present then we search for the property in this prototype chain.

```
function Product(n) {
  this.name = n;
}

Product.prototype.display = function (msg) {
  console.log(this.name, msg);
}

let d = new Product("iphone");
d.display("in stock");
```



Prototypal Inheritance