ES6 Classes

Let us create a person object,

Ex: 01

```
const yash = {
  name: 'Yash',
  age: 24
}

const joseph = {
  name: "Joseph",
  age: 34
}
```

If we have multiple objects to be defined, we need to repeat this process. Now, if our requirement changes and we have to add height, then we have to add that property to every object. This will not work in complicated scenarios. So, instead of repeating we use ***functions***.

Notice the reason we are using the first letter as capital is because this is going to be object creator function, which is also called has constructor function.

```
function Person(name, age) {}
```

Here we can use the properties inside a return in the function but rather than doing bad I am going to show you another way, by passing the properties inside the parenthesis (name & age).

Instead of using 'return' we can use the keyword 'this'. When we are defining a constructor function, ‘this’ keyword will be referring to the object which is going to be created behind the scene. So,

```
function Person(passedName, passedAge) {
  this.name = passedName
  this.age = passedAge
}
```

So, when we are creating a constructor function, the keyword **`'this'`** actually takes value depending on where it is called . this will be referring to the object function that is going to be created from this function.

Ex: 02

```
function Person(passedName, passedAge) {
  this.name = passedName
  this.age = passedAge
}

const joseph = new Person("Joseph", 34)
console.log(joseph)
```

Output:

Person {name: "Joseph", age:34}

The variable joseph will have the name and age property attached to it. So, when we console.log it, we can see it Person object. The name Person is coming exactly like the name we have given it. When we have console.log the 'joseph' variable, this 'joseph' variable is a person object, that is why inside it, it has two properties age and name.

It is mandatory to use **'new'**, because if we don't then the **'this'** keyword would be referring to the window object instead of the new object that is going to be created from the function. In this case if we do not put the **'new'** keyword it would be into the window object and therefore it would show an output as undefined. In console if we enter window and search in its properties, we can find age & name in it. **'age'** property has been assigned to the window object. So, if we don't; use **new** keyword, javascript is treating this function as a regular function and it is not giving any special allowances to it. In such case, when you are not using the **'new'** keyword, the keyword **'this'** is actually referring to the **window object**. But moment you add the **new** keyword, before the invocation of the **Person**() function, then javascript treats this function a little bit special. In that case what javascript is doing is, the keyword **'this'** will now be are referring to the new object that is being creative.

There are four steps happening (behind the scenes):

Step 1: JavaScript creates an Empty object into the memory. Every object has two properties prototype and constructor which would be referred to the same object that has been created.

Step 2: The keyword '**this**' is been attached to the newly created object in in the memory.

Step 3: Then it whatever is executes whatever is put inside the constructor function.

Step 4: Finally, returns that entire object which was created in the memory from this function.

Inheritance:

Ex: 03

```
function Person(passedName, passedAge) {
  this.name = passedName
  this.age = passedAge
}

Person.prototype.hello = function() {
  console.log("Hello")
}

function Programmer(passedName, passedProgrammingLang) {
  this.name = passedName
  this.passedProgrammingLang = passedProgrammingLang
}
```

We have constructor functions one is Person and the other one is Programmer. But if you notice, if we want to define some function which is common to Programmer but not common to Person. The idea is all the features the Person has. We can link one function to another so that it will have the features of both the classes.

Now, we want to write some program which links them. Every programmer is a person, but every person is not a program. All the functions are objects in JavaScript.

```
Programmer.prototype = Object.create(Person.prototype)
Programmer.prototype.constructor = Programmer

Programmer.prototype.writeCode = function() {
  console.log(`${this.name} is coding`)
}

const prgm = new Programmer("ABC", "JavaScript")
prgm.hello()
prgm.writeCode()
```

Output:

Hello
ABC is coding

We defined a Person constructor and a Programmer constructor. Both of them, logically speaking all Programmer will be of Person only. now we want to mimic the inheritance behaviour. in order to show that, prototype or object type of that function. This is another way of referencing the Person object only.

We are defining a function **`hello`**() on the person prototype. similarly, we defined another function on the programmer constructor function. all the objects created with programmer constructor function should have a **`writeCode`**() function also so that we can call it. but want the **`hello`**() function attached to the programmer constructor. in order to do that,

```
Programmer.prototype = Object.create(Person.prototype)
```

Everything in JavaScript is inherited from Object. **`Object.create`**() is one function and inside this function we have just typed in where it should link to, the super class. after doing that, we are telling what should be the constructor function of the programmer function. now when we look into the output and say **`'prgm'`** which is an object of Programmer, but I am calling **`hello`**() I am getting the output accordingly. this is only possible when my Programmer is linked to the Person constructor class.

this is how we would in ES5, inherit create link between constructor functions end use sir path of different constructor functions another function.

If we write it in ES6 then,

```
class Person {

  constructor (name, age) {
    this.name = name
    this.age = age
  }
}
const personObj = new Person('Yash', 88)
console.log(personObj)
```

Output:

Person {name: 'Yash', age: 88}

Here encapsulated everything into a class declaration. Behind the scene script is converting the entire think into the Person() function. The output the same but since the ES5 version i was not making much sense to most of the developers ES6 came with this syntactic sugar. Javascript is changing the whole code to the previous one so that it can understand. If we have to define a method, we would not define a function, but directly call the function.

```
class Person {

  constructor (name, age) {
    this.name = name
    this.age = age
  }

  sayHello() {
    console.log(`${this.name} says hello. My age is ${this.age}`)
  }
}

const personObj = new Person('Yash', 88)
console.log(personObj)
```

Inside __proto__ (this is what is referring to .prototype), we can find class constructor and sayHello() function. JS added a function onto the prototype of the function class. If we expand it, we can see the Object() function as well. It will created for all functions.

The first prototype we see is referring to the **`Person`** class. If we create another class Programmer(), my idea was to add one more attribute called **programminglanguage**. We would have to write a constructor function as it is also a class and will tell JavaScript to be added or to be made accessible from another class. We are inheriting the properties of class into another class. Ultimately, whatever is inside the Person class, will be made accessible from the Programmer class. However, it would require name, age and programming language.

The issue is although I have linked all the properties inside my programmer class, but I need to call the constructor function of the Person class, so that its properties are also initialized. In order to do this, there is another keyword called 'super'. This super referring to the parent class where ever this class is extending from. So when we input in the parenthesis it will call the constructor of the Person class. Since constructor requires name and age so we will be passing passedName and passedAge. This will initialize the property which is inside my Person class and then in the Programmer class we can say,

```
class Person {

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

```
  sayHello() {
    console.log(`${this.name} says hello.`)
  }
}
```

```
class Programmer extends Person {
  constructor(passedName, passedAge, passedProglLang) {
    super (passedName, passedAge) // refering to parent class
    this.programmingLang = passedProglLang
  }

  writeCode() {
    console.log(`${this.name} is coding in ${this.programmingLang} & the
age is ${this.age}`)
  }
}

const personObj = new Person('Yash', 88)
console.log(personObj)

const programmer = new Programmer('Bob', 45, 'Java')
console.log(programmer)
```

Now, if we add personObj.sayHello(), it would console log the statement it would give the output. But if we try, personObj.writeCode(), it will show error, because it does not have the function defined onto it. It is defined in the programmer class. So, if we call programmer.wirteCode(), it will display both the statements from sayHello() and wirteCode() as it is defined in the Programmer class.

Getters and Setters:

All the properties in the objects when we instantiate the class; we were directly able to say personObj.name and change the value. But instead of directly accessing the property name, we will creating functions for individual properties.

Ex: 0

```
class Vehicle {
  constructor(year, mfg) {
    this._year = year
    this._mfg = mfg
  }
}
```

In order to get a property, we will use the function. We can write,

```
class Vehicle {
  constructor(year, mfg) {
    this._year = year
    this._mfg = mfg
  }

  get year() {
  return this._year
  }

  set year(newYear){
    this._year = newYear
  }

  get manufacture() {
    return this._mfg
  }

  set manufacture() {
    this._mfg = newMfg
  }
}
const vehicleObj = new Vehicle(1997, 'Bajaj')

console.log(vehicleObj)
console.log(vehicleObj.year)
vehicleObj.year = (2000)
console.log(vehicleObj.year)
```

The only difference is when you write get and set, whatever the name you give the function becomes the property name. Similar to this, we are not calling the getter and setter function. It is getting called by JS behind the scenes.

Static Variables & Functions:

In our constructor if you remember, the more objects I create all the values are tied to its object property. Static variables are common across all the objects. Meaning that variable is defined on the class itself rather than the objects of class. And this variable is accessible by all the objects. If you make any change in that class static variable, that change would be available to see for all the other objects

The idea behind this is as the objects are getting created using the vehicle class, we would be incrementing The Count of it. when constructor is executed,

```
class Vehicle {
  static count = 0

  constructor(year, mfg) {
    this._year = year
    this._mfg = mfg
    vehicle.count +=1

  }

  get year() {
  return this._year
  }

  set year(newYear){
    this._year = newYear
  }

  get manufacture() {
    return this._mfg
  }

  set manufacture() {
    this._mfg = newMfg
  }
}
const vehicleObj1 = new Vehicle(1997, 'Bajaj')
console.log(Vehicl.count)
const vehicleObj2 = new Vehicle(1875, 'ASD)
console.log(Vehicl.count)
const vehicleObj3 = new Vehicle(1957, 'Chevrolet')
console.log(Vehicl.count)

console.log(vehicleObj1)
console.log(vehicleObj2)
console.log(vehicleObj3)
```

At the end, we can see 3 objects were created. If we expand any of them, we can see the inside the constructor function, we can see the static variable 'count'. If I expand other objects, we can see the 'count' value is same as it is shared and it has only single source of truth which is inside the class itself. Whatever changes have been made; those changes are reflected for all the objects for that classes.