Class and Constructors

Constructor

You can create objects in JavaScript using curly braces { ... } syntax. But what if you need to create multiple similar objects. You can either write the same syntax for every object, or you can use the **constructor** to create objects.

Using { ... } syntax to create multiple objects can create certain inconsistencies in code-

- There can be spelling mistakes.
- The code can become difficult to maintain
- Changes to all the objects will be difficult.

To overcome all the above inconsistencies, JavaScript provides a **function constructor**. The constructor provides a **blueprint/structure for objects.** You use this same structure to create multiple objects.

Constructors technically are regular functions. There is one convention to constructors -

❖ The first letter of the function is capital.

Objects can be created by calling the constructor function with the **new** keyword.
Using a constructor means that -

- All of these objects will be created with the same basic structure.
- We are less likely to accidentally make a mistake than if we were creating a whole bunch of generic objects by hand.

It is important always to use the **new keyword** when invoking the constructor.
If new is not used, the constructor may clobber the 'this', which was accidentally passed in most cases as the global object (window in the browser or global in Node.). Without the **new** function will not work as a constructor.

```
function Student(first, last, age) {
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        }

var stu1 = new Student("John", "Doe", 50);
var stu2 = new Student("Sally", "Rally", 48);
```

The **new keyword is the one that is converting the function call into constructor call,** and the following things happen -

1. A brand new empty object gets created.
2. The empty object gets bound as this keyword for the execution context of that function call.
3. If that function does not return anything, then it implicitly returns this object.

NOTE: this referred in the constructor bound to the new object being constructed.

Classes

Classes are introduced in ECMAScript 2015. These are **special functions that allow one to define prototype-based classes** with a clean, nice-looking syntax. It also introduces great new features which are useful for object-oriented programming.

Example :

```
class Person {
        constructor(first, last) {
                this.firstName = first;
                this.lastName = last;
        }
}
```

The way we have defined the class above is known as the **class declaration** in order to create a new instance of class Person by using the new keyword.

```
let p1 = new Person("Rakesh", "Kumar") ;
```

Some points you need to remember -

- You define class members inside the class, such as methods or constructors.
- By default, the body of the class is executed in strict mode.

- The constructor method is a special method for creating and initialising an object.
- You cannot use the constructor method more than once; else, SyntaxError is thrown.
- Just like the constructor function, a **new** keyword is required to create a new object.

NOTE: The type of the class is a **function**

```
typeOf(Person) ; // function
```

Getter and Setter

You can also have a **getter/setter** to **get the property value** or **set the property values** respectively. You have to use the **get** and **set** methods to create a getter and setter, respectively.

```
class Vehicle {

        constructor(variant) {
                this.model = variant;
                }
        get model( ) {
                return this.model;
         }
        set model(value) {
                this.model = value;
        }
 }

let v = new Vehicle(“dummy”);
console.log(v.model) ; // get is invoked
v.model = “demo” ;      // set is invoked
```

Class Expression

A class expression is another way to define a class, which is similar to a function expression. They can be **named and unnamed both**, like -

```
let Person = class { };
            OR
let Person = class Person2 { };
```

NOTE: The name given to the class expression is local to the class's body.

Hoisting

Class declarations are **not hoisted**. If you try to use hoisting, it will return **not defined** error.

Instance

It is made using the blueprint of the class having its behaviours and properties.

Example: Let's say Human is a class, and we are all its instances

```
class Human{
        constructor(name , height , weight){
                this.name = name;
                this.height=height;
                this.weight=weight;
        }
}

let Sam = new Human("Sam" , "6" , "80");
```

- Sam is an instance of Human class.

Encapsulation

The process of wrapping property and function within a single unit is known as encapsulation.

```
class Person{
  constructor(name,age){
     this.name = name;
     this.age = age;
  }
  add_Address(add){
     this.add = add;
  }
  getDetails(){
     console.log("Name is",this.name,"Address is:", this.add);
  }
}

let person1 = new Person('Sam',22);
person1.add_Address('Delhi');
person1.getDetails();
```

In the above example, we simply created a person instance using the constructor and Initialized its properties.
Encapsulation refers to the **hiding of data** or data Abstraction which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript but there are certain ways by which we can restrict the scope of variable within the Class/Object.

```
function person(firstname,lastname){

  let firstname = firstname ;
  let lastname = lastname ;
   let getDetails_noaccess = function( ){
       return (“Name is:”, this.firstname , this.lastname) ;
  }
   this.getDetails_access = function( ){
      return (“Name is:”, this.firstname , this.lastname) ;
  }
}

let person1 = new person(‘Sam’, ‘Cumberbatch’);

console.log( person1.firstname);  //Undefined
console.log( person1.getDetails_noaccess); // Undefined
console.log( person1.getDetails_access( ) );   // Name is Sam Cumberbatch
```

In the above example we try to access some property **person1.firstname** and functions **person1.getDetails_noaccess** but it returns undefined while their is a method which we can access from the person object **person1.getDetails_access( )** and by changing the way to define a function we can restrict its scope.