Siddharth Singha Roy

Winter 19-20

Objects and Functions

In JavaScript we have the following primitive datatypes

1. Numbers
2. String
3. Boolean
4. Undefined
5. Null

Everything else is an object (Like functions, objects, arrays etc).

JavaScript is an object oriented programming language that supports inheritance. Inheritance allows us to follow the DRY Principle and keep our code short

Unlike other programming languages where we use the word class, in JavaScript we use constructor or prototypes.

Inheritance follows IS A Relationship

Declarations inside a class (constructor or prototype) follows HAS A Relationship.

JavaScript is a prototype based language. Inheritance feature in JavaScript is supported due to prototype. Each and every object in JS has a prototype property.

The prototype property of an object is where we put methods and properties that we want other objects to inherit.

If we use .prototype we actually get an object.

The constructor's prototype property is not the property of the prototype itself, but the property of the object. Simply speaking the prototype property is in the object of that class and not the class itself.

Every constructor is the child of an object object (at the topmost level). This also has its own prototype.

When a certain method or property of the object is called, the search starts in the method and property of the object itself, if it is not found then it moves on to the object's prototype until it is found. This is known as prototype chain.

Function Constructor

new keyword

Using the new keyword we are able to create an **empty object.**

var John = new Person("john", 19):

In this code, first a brand new empty object is created. Then after that the constructor person, which in this case is Person is called with the arguments.

Now we know that calling a function will create a new execution constant which will have a this variable.

Now we know that for any regular function call, the this variable will point towards the global object. But this is now what we want. This is taken care by the new variable. It make the this variable point towards the empty object that has been created.

If the function constructor does not return anything, the the object created by the new function will be passed to John.

If our function constructor has some functions and we create objects of that constructor, then all the objects will have the functional constructor with them. Thus all the objects will have copy of the same function which is not efficient.

Declaring a function in the prototype property will allow the objects to inherit it.

```
Person.prototypes.calcAge = function(){
        this.age = 2019-this.birthYear;
}
```

In the prototype we can also add property (instead of methods).

To analyse a particular object we can simply use the web browser for it. We get something called __proto__ in the object property list. This is in fact the prototype of the particular object.

john.__proto__ === Person.prototype

This is true.

We see that inside the __proto__ we have another __proto__ . This is the prototype of the object function constructor. As stated earlier each function constructor is an instance of the object function constructor.

So we can use john.isPrototypeOf as isPrototypeOf is in the prototype of the Object function constructor. First isPrototypeOf is searched in the given object property. Then it is searched in the prototype of that object. Now it will be searched in the function constructor prototype since it is an instance of the object prototype.

This is the prototype chain.

One of the useful methods here is the hasOwnProperty()

```
John.hasOwnProperty('age');   // True
```

Any property that is present in the prototype of that object will be shown as False.

```
John instanceof Person.  // True
```

We can even see that an array belongs to the function constructor Array which in turn belongs to the function constructor Object.

Object.create

```
var personProto = {
   caculateBirthYear : function() {
      consoole.log(2019 - this.age);
   }
};

var john = Object.create(personProto); // The first argument is the prototype of that object
john.name = "John";
john.age = 18;

var mark = Object.create(personProto,
{
   name : { value : "Mark"},
   age : { value : 19 },
});
```

The create method takes two arguments

1. The object prototype
2. The properties of the object in the form of an object with the syntax mentioned above.

If we do not pass the second argument then an empty object with the prototype will be returned.

```
var sid = Object.create(null);
```

If we want to create an object with no prototype.

Primitives vs Objects

Variables containing primitives hold the data inside of the variable itself.

Variables containing objects do not hold the data inside the variable, instead contains a reference of the memory where object is stored.

Functions also follow the same principle. If we modify an object inside the function, the changes are reflected in the original function. For the primitive datatypes any changes made are not reflected back in the original variable.

For objects we just pass a reference of that object to the function.

Functions

1. Functions are also objects in JavaScript. It is an instance of the Object function constructor.
2. A function can behave like any other object.
3. Functions can also be stored inside variables
4. We can pass a function as argument to another function
5. We can return a function from a function

A **callback function** is the one which we pass to another function instead of executing it. This ensures that the function can be executed later.

It is a function passed into function as an argument which is then invoked inside the outer function to complete some kind of routine or action.

Any function that does not have a name is not known as anonymous function.

Functions are JavaScript are first class function. It means that they can be treated like objects for example stored, passed, retuned like objects.

Immediately Invoked Function Expressions (IIFE)

```
(
   function() {
      var score = Math.random() * 50;
      console.log(score>5);
   }
)();
```

This is an IFFE. Instead of having a separate function declaration that can be reused, what we do instead is create a function that will be executed immediately. Also this can be called once since we do not assign a name for it.

Now since we don't want to assign a name, if we write the function normally it will throw an error since name of the function is not defined. The parser will treat it as a function declaration. We can trick the parser into thinking it is a function expression by wrapping it with brackets.

IIFE is used to write functions that will only be used once and also when data privacy is required.

Closures

An inner function has always access to the variables and parameters of its outer function even after the outer function has returned (ie. its execution is over).

Even after the execution has been completed, the variable object and scope chain isn't removed. They still remain in the memory and can be accessed. Hence we can access the parent function value from inside the function.

Scope chains a pointer to the variable object.

Closure means that the scope chain will always stay intact. Hence even after executing the outer function we were able to access that values of the outer function from the inner function.

```
function retirement(retirementAge) {
    var a = " years left till retirement";
    return function(yearOfBirth){
        var age = 2016-yearOfBirth;
        console.log((retirementAge-age)+a);
    }
}

var retirementUS = retirement(65);
retirementUS(1992);

retirement(66)(1992);
```

Bind, Call and Apply

Used in functions to mutate the value of the this variable.

CALL

```
john.presentation.call(emily, 'friendly','morning');
// The call method is used for method borrowing
```

The first argument to the call method is the reference of this and the following arguments are the data that we want to pass to the function. Since we pass Emily as this reference this.age will now refer to the Emily object.

APPLY

This is similar to the call method but the difference is the second argument is an array. So use this when the function expects array as input.

```
john.presentation.apply(emily, [‘friendly','night']);
```

BIND

The bind function is also similar to the call function in the way that both allow us to explicit state the value of this.

However unlike call and apply where the function is invoked immediately, here a new copy of the function is returned and we can store this new copy in a variable.

The bind function also allows us to set the value of one of the parameter of the function and then return the new function. This is known as Currying.

Currying is the method by which if have a function taking multiple arguments, we return a new function with few of the parameters set.

Binding a function

```
function isFullAge(limit, age) {
    return age>=limit;
}

isFullAge.bind(this,20);
```

Arrays

For Of Loop

This loop combines the forEach and the for loop. The problem with forEach and map is that we cannot use break or continue.

```
const arr = [1,2,3,4,5];
for ( const ele of arr) {
    if(ele%2==0)
        continue;
    console.log(ele);
}

ages.findIndex(cur => cur>=18);
```

This returns the index of the array ages which has age > 18.

This returns the index. If we want the value

```
ages.find(cur => cur>=18);
```