JavaScript Part 2

Functions

- Function is the group of code to perform particular task.
- Function can be used to reuse the code.

  Eg.

  ```
  function sum(a, b) {
  var c = a + b;
  return c;
  }
  ```

Following parts will construct a function

1. The ***function*** statement.
2. The ***name*** of the function.
3. Expected ***parameters*** (arguments)
4. A code block, also called the ***body*** of the function.
5. The ***return*** statement. A function always returns a value. If it doesn't return value explicitly, it implicitly returns the value ***undefined***.

- A function can be called by using its name followed by any parameters in parentheses or by using ***call*** built in function.
- When defining a function, you can specify what parameters the function expects to receive when it is called. A function may not require any parameters, but if it does and you forget to pass them, JavaScript will assign the value ***undefined.***
- In JavaScript a function can be a variable which contains a code and executable

Eg. var f = function(){return ‘a’;}

*typeof f = **"function"***

- JavaScript provides many built in functions such as isNaN(),pareseInt() etc..

Scope Of Variables

- Lifetime of a variable is known as scope.
- If a variable is defined inside a function it is known as ***local variable*** whose scope will end even after a function's completion

- If a variable is defined out side a function it is known as ***global variable*** whose scope will remain even after a function's completion

Types Of Functions

- **Anonymous Functions :** a function without any name.

  Eg. setTimeout(function() {console.log(‘a’);}, 1000);

- **Callback Functions :** a function which is passed as an argument to another function.

  Eg. setTimeout(function() {console.log(‘a’);}, 1000);

- **Immediately-invoked function expression(IIFE, pronounced as ‘iffy’) :**

  functions which will get executed immediately after they are defined

  Eg. (function() {

// the code here is executed once in its own scope

})();

Lexical Scope

- In JavaScript, functions have ***lexical scope***. This means that functions create their ***environment (scope)*** when they are defined, not when they are executed.

  Eg.

```
function f1(){ var a = 1; f2(); }
function f2(){ return a; }
f1();
```

Uncaught ReferenceError*: a is not defined.*

Closures

- a function within a function.
- a local variable for a function.
- an inner function whose scope continues even after the parent functions have returned.
- a function which will have access to variables in its own scope, plus the scope of its "parents". This is known as ***scope chain.***

  Eg.

```
var a = 1;
function f(){
   var b = 1;
  var n =  function() {
       var c = 3;
   }
```

n();

}

OOPS in JavaScript

Object

- An object is a collection of properties.
- Object properties will be arranged in key value pair.
- In an object every property has a key that is unique within the object.

  Eg. *var car = {'color':'black','brand':'Audi'};*

having quotes for keys in an object is optional but in some cases we have to use quotes such is if key has blank spaces or special characters.

- There are two ways to access a property of an object:

  Using square bracket notation, for example *car* [*"color"*]

  Using the dot notation, for example car.color

- Since in JavaScript functions are also data so a function can also be a property of an object

Eg. *var car = {'color':'black','brand':'Audi',get:function(){return 'color of car is' + car.color};*

Calling objects function is like : *car.get();*

- We can add properties to an object, remove properties from an object and also we can modify existing property of an object.

  Eg. car.weight = 50 // added a new property weight to the object *car.color = 'blue' // modified existing property*

delete car["color"] // removes color property from the object

- When we compare objects, you'll get true only if you compare two references to the same object. Comparing two distinct objects that happen to have the exact same methods and properties will return false.

  Eg. *var car= {color: 'blue'};*

  var car1= {color : 'blue'};

  benji === fido //returns false

  var mycar = car;

  car === mycar // returns true

- There is another way to create objects: by using constructor functions.

```
Eg.   function Car() {
        this.color = 'blue';
      }
```

In order to create an object using this function, you use the new operator, like this:

```
var car= new Car();
car.color;//returns blue
```

we can pass parameters also to constructor functions for creating objects.

```
Eg.   function Car(color) {
        this.color = color;
```

```
                }
            var car= new Car('blue');
```

“this” keyword

- Every line of JavaScript code is run in an “execution context.”
- The JavaScript runtime environment maintains a stack of these contexts, and the top execution context on this stack is the one that’s actively running.
- By default, this refers to the global object.
- When a function is called as a property on a parent object, this refers to the parent object inside that function.
- When a function is called with the new operator, this refers to the newly created object inside that function.

```
            Eg.    var counter = {
      val: 0,
      increment: function () {
      this.val += 1;
     }
    };
    counter.increment();
    console.log(counter.val); // 1
    counter['increment']();
    console.log(counter.val);//2
    var inc = counter.increment;
```

```
inc();
console.log(counter.val); // 2
console.log(val);//NaN
```

Prototype

- Prototype is a property of a function.
- Every JavaScript function has a property named prototype. Adding members to the function's prototype is implemented by adding them to the prototype property of the function.
- Prototype is a property that gets created as soon as you define the function. Its initial value is an empty object.

```
Eg.      var myFunction = function() {};
   console.log(myFunction.prototype); // logs object{}
```

- Private variables of a function aren't accessible through functions added to its prototype.
- You can add members to a function's prototype only after the function itself has been defined.

Eg. Below is a constructor function Gadget() which uses this to add two properties and one method to the objects it creates.

```
function Gadget(name, color) {
            this.name = name;  this.color = color;
            this.whatAreYou = function(){ return 'I am a ' +
        this.color + ' ' + this.name   };
```

- Adding methods and properties to the prototype property of the constructor function is another way to add functionality to the objects that a constructor produces.

Eg.
```
Gadget.prototype = {
price: 100,
rating: 3,
getInfo: function() {
return 'Rating: ' + this.rating + ', price: ' + this.price;
}};
```

- All the methods and properties you have added to the prototype are directly available as soon as you create a new object using the constructor

Eg.
```
var newtoy = new Gadget('webcam', 'black');
newtoy.name; ///returns "webcam"
newtoy.color; /// returns "black"
newtoy.price; ///returns 100
```

In the example mentioned when we try to access a property of ***newtoy****, say* ***newtoy.name*** *the JavaScript engine will look through all of the properties of the object searching for one called name and, if it finds it, will return its value.*

And when we try to access a property ***price,*** *The JavaScript engine will examine all of the properties of newtoy and will not find the one called rating. Then the script engine will identify the prototype of the constructor function used to create this object (same as* *newtoy.constructor.prototype). If the property is found in the prototype, this property is used.*

- Following is the way to get the prototype object.

  newtoy.constructor.prototype.constructor.prototype

  ///returns ***Object price=100 rating=3***

- When a constructor function has its own property and a property in its prototype object with same name then function's own property will take the priority.

Eg.

```
function Gadget(name) {
this.name = name;
}
Gadget.prototype.name = 'foo';
var toy = new Gadget('camera');
                    toy.name; //returns "camera"
                    delete toy.name;
                     toy.name; //returns "foo"
```

Inheritance in JavaScript

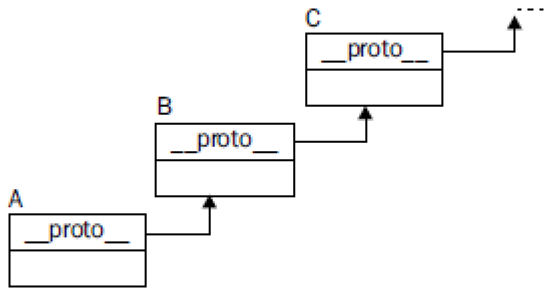
- Inheritance is the concept of one object gaining the properties or behaviors of some other object.
- In JavaScript inheritance happens through prototype chaining. Before we know about prototype chaining we have to know about the _proto_ link.

The __proto__ link

- At any time when an object is created from a constructor function using the **new** keyword, a hidden link is added between the object instance created and the prototype property of the constructor function used to create it. This link is known inside the instance as **__proto__.**

Prototype Chaining

Prototype Chaining happens through the __proto__ link, when a function is invoked using the new operator, an object is created and this object will have **_proto_** link which links to the prototype object. And this link allows methods and properties of the prototype object to be used as if they belong to the newly-created object.



- In the above illustration **A** contains a number of properties. One of the properties is the hidden __proto__ property, which points to another object, **B**. **B**'s __proto__ property points to **C**. This chain ends with the **Object** object, which is the highest-level parent, and every object inherits from it.

- The practical side is that when object **A** lacks a property but **B** has it, **A** can still access this property as its own. The same applies if **B** also doesn't have the required property, but **C** does. This is how inheritance takes place.

Eg.
```
function Shape(){
this.name = 'shape';
this.toString = function() {return this.name;};}
function TwoDShape(){ this.name = '2D shape';}
function Triangle(side, height) {
this.name = 'Triangle';
this.side = side;
this.height = height;
this.getArea = function(){return this.side * this.height / 2;};
}
```

The code that performs the inheritance magic is as follows:

```
TwoDShape.prototype = new Shape();
Triangle.prototype = new TwoDShape();
my = new Triangle(5, 10);
my.getArea(); ///returns 25
my.toString() //returns ”Trangle”
```

The execution var happens as below

- *It loops through all of the properties of my and doesn't find a method called* ***toString().***
- *It looks at the object that my.****__proto__*** *points to; this object is the instance* ***new TwoDShape()*** *created during the inheritance process.*
- *Now the JavaScript engine loops through the instance of* ***TwoDShape*** *and doesn't find a* ***toString()*** *method. It then checks the* ***__proto__*** *of that object. This time __proto__ points to the instance created by new Shape().*
- *The instance of new Shape() is examined and* ***toString()*** *is finally found.*
- *This method is invoked in the context of my, meaning that this points to my.*