Advanced JavaScript

Object Oriented JavaScript

ITI – Assiut Branch
Eng. Hany Saad

JavaScript Objects Recap

- manipulate, and JavaScript Objects fall into 3 categories:
  - Custom Objects
    - → Objects that you, as a JavaScript developer,create and use.
  - Built – in Objects
    - → Objects that are provided with JavaScript to make your life as a JavaScript developer easier
  - DOM Objects
    - → Objects provide the foundation for creating dynamic web pages.
    - → The DOM provides the ability for a JavaScript script to access, extend the content of a web page dynamically.

JavaScript Built-in Objects

- **JavaScript Built-in Objects:**
  - String
  - Number
  - Array
  - Date
  - Math
  - Boolean
  - RegExp
  - Error
  - Function
  - Object

Function Object

- **Functions are a special data type.**
- **Functions are actually objects that are invokable.**
- **There is a built-in constructor function called Function() which allows an alternative (but not recommended) way to create a function.**
- **Functions can be considered as values in JavaScript.**
- **Functions can be:**
  - Assigned to a variable, array element.
  - passed as an argument to another function
  - a value returned from a method call.
  - created on the fly.
- **This makes using functions a very handy and flexible, but also a confusing one.**

Function Object (cont.)

- There are three primary approaches to creating functions in JavaScript:
  - The most common functions **Declarative / Static/ Function Statement**

```
function AddNums(x , y)
{
        return (x+y);
}
alert(AddNums ( 2 , 3 ));
```

  - JavaScript functions are objects. They can be defined using the Function constructor (**Dynamic / Anonymous/ Function Constructor**)

```
var NumsSum= new Function( "x" , "y" , "return (x+y)" );
var r= NumsSum ( 2 , 3 ) ;
alert (r) ;
```

  - This is equivalent to the function literal, it is also known as Factory Function (**Literal / function expression**)

```
var result= function ( x , y ) { return x + y; };
alert( result ( 2 , 3 ) );
```

  - More: https://dmitripavlutin.com/6-ways-to-declare-javascript-functions

Function Object(Cont.)

Declarative / Static Function

```
function functionname (param1, param2, ..., paramn) { function statements }
```

- The most common type of function uses the declarative/static format.
- This approach begins with the
  - → function keyword,
  - → followed by function name,
  - → parentheses containing zero or more arguments,
  - → and then the function body:
- The declarative/static function is:
  - → parsed once, when the page is loaded
  - → Hoisted (useful for mutual recursion)
  - → the parsed result is used each time the function is called.
  - → It's easy to spot in the code,
  - → simple to read and understand,
  - → has no negative consequences (usually), such as memory leaks.

Function Object(Cont.)

Dynamic / Anonymous Function

```
var variable = new Function("param1”, "param2“,.. , "paramn", "function body");
```

- Anonymous: because the function itself isn't directly declared or named.
- Dynamic: The JavaScript engine creates the anonymous function dynamically, and each time it's invoked, the function is dynamically reconstructed.
- Functions created with the Function constructor do not create closures to their creation contexts.
- Example:

```
var fun= new Function(“x,“,”y”, “alert(x+y));
alert(fun(2,3);

var fun= new Function(“x,“,”y”, “alert(x-y));
alert(fun(2,3);
```

- More: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

Function Object(Cont.)

Function Literal

```
var func = (params) { statements; }
```

- Also known as function expressions because the function is created as part of an expression, rather than as a distinct statement type.
- They resemble anonymous functions in that they don't have a specific function name.
- They resemble declarative functions, in that function literals are parsed only once.
- Example:

```
//Ex.1:
document.getElementById("b1").click= function(){
  alert("test");
 };
//Ex.2:
setInterval(function(){
  //function body;
 },1000);
```

Anonymous Functions

- functions are like any other variable so they can also be used without being assigned a name (Anonymous).
- Anonymous functions are functions that are passed as arguments or declared inline and have no name.
- Advantages:
  - Can used in event handling.
  - Can passed as a parameter to another function.
- Example:

```
Window.onerror= function (msg,l,url) {
        alert (msg);};
```

```
var timer= setInterval(function(){
                .....
                .....
    },1000);
```

Functions Hoisting

- Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.
- Hoisting applies to variable declarations and to function declarations.
- Because of this, JavaScript functions can be called before they are declared

```
myFunction(5);

function myFunction(y) {
    return y * y;
}
```

- Functions defined using an expression are not hoisted.

Function object properties

- The typeof operator in JavaScript returns "function" for functions **But**, JavaScript functions can best be described as objects.
- JavaScript functions have both properties and methods.
- When a function receives parameter values from a caller, those parameter values are implicitly assigned to the arguments property of the function object.
- The arguments Object:
  - it looks like an array of arguments.
  - arguments, contains values of all parameters passed to the function.
  - The arguments.length property returns the number of arguments received when the function was invoked.

```
function myFunction(a, b) {
            return arguments.length;
}
alert (myFunction.argument.lengths);
```

More: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

Self-invoking Functions (IIFEs)

- **Self-invoking Functions**: You can define an anonymous function and execute it right away. By calling this function right after it was defined.
- It's also called: **IIFE "Immediate Invoke Function Expression" Pattern.**
- IIFE stands for Immediately Invoked Function Expression
- It is a function expression that is invoked immediately
- A common often used pattern.
- Can be invoked on the fly at the point it is created.
- Function expression is wrapped within () operator

Self-invoking Functions (IIFEs)

- Example:

```
(
function(){
        alert(‘hellooooo');
}
)();
```

- You can pass an anonymous function as a parameter to another function.

```
alert((function(n) {
        return (n*n);
})(10));
```

- Besides advantages and disadvantages of anonymous function, IIFEs are:
  - Suitable for initialization tasks
  - Work done without creating global variable
  - Its where the magical part happens in avoiding closures
  - Also, cant execute twice unless it is put inside loop or another function
  - Introduces a new scope that restrict the lifetime of a variable

Object object

- Object is the parent of all JavaScript objects, which means that every object you create inherits from it.
- To create an object:

```
//old way of creating an object
var obj = new Object( );
//new way of creating an object (Litral)
var obj = { };
```

Some of Object object properties & methods

- hasOwnProperty("prop"): A method returns true if the current object instance has the property defined in its constructor or in a related constructor function.

- toString(): A method that returns the object as string.

Value type & Reference Type in JavaScript

- Value type variables & Reference Type Variables:

```
//Value Type
var str="abc" //value type
var str2=str;
str="xyz";
alert (str2)//abc
//Objects are reference type
var str=new  String ("abc"); //reference type
var str2=str;
str="xyz";
alert (str2)//xyz
```

- Arguments are Passed by Value
  - JavaScript arguments are passed by value: The function only gets to know the values, not the argument's locations.
  - If a function changes an argument's value, it does not change the parameter's original value.
- Objects are Passed by Reference
  - In JavaScript, object references are values, because of this, it looks like objects are passed by reference
  - If a function changes an object property, it changes the original value.

Custom Objects "Classes"

- JavaScript is Object oriented language.
- Any language needs to have: encapsulation, polymorphism, and inheritance, so we can called it Object Oriented language.
- Strictly speaking, JavaScript is a class-less language. The Class keyword, although reserved, is not part of the language definition. JavaScript is built on Objects rather than Classes.
- In another words, there's no classes in JavaScript, but you can create an object which is equal to class in other languages.
- To create objects in JavaScript, there's 2 approaches:
  - Creating objects using Literal Notation
  - Creating Objects using Object Constructor and prototyping

1- Creating Objects using Constructor

- An object in JavaScript is a complex construct usually consisting of a constructor as well as zero or more methods and/or properties.
- **Functions are your First Class in JavaScript!**
- A constructor function looks like any other JavaScript function, but its purpose is:
  - to define the initial structure of an object
  - to define it's property and method names
  - It can populate some or all of the properties with initial values.
  - Values to be assigned to properties of the object are typically passed as parameters to the function,
  - Statements in the constructor function assign those values to properties.

1- Creating Objects using Constructor (Cont.)

- Creating new Instance from custom objects using Constructor function
  - Constructor Function for Employee Object.

```
function Employee (name, age)
{
        this.name = name;
        this.age = age;
}
```

  - To create instance (objects) with this constructor, invoke the function with the new keyword

```
//Creating an instance (object)
var emp1 = new Employee (“Aly", 23);
var emp2 = new Employee (“Hassan", 32);
alert(emp1.name) //Aly
```

1- Creating Objects using Constructor (Cont.)

- We can also generate a blank object and then populate it explicitly property by property:

```
var emp3 = new Employee( );
emp3.name = "Alice";
emp3.age = 23;
```

1- Creating Objects using Constructor (Cont.)

- Assign a default value to a Property:
  - In the Employee object constructor function, if the statement that invokes the function leaves the second parameter blank, the age parameter variable is initialized as a null value. To provide a valid but harmless default value (of zero) to that property, the syntax is as follows:

```
function Employee (name, age)
{
        this.name = name || “” ;
        this.age = age|| 0;
}
// creating instance
var emp1=new Employee();
var emp3=new Employee(“Ali”);
var emp4=new Employee(“Ali”,55);
alert(emp3.age)//0
```

1- Creating Objects using Constructor (Cont.)

- Assign a default value to a Property:
  - ES6 syntax for default value

```
function Employee (name="", age=0)
{
        this.name = name ;
        this.age = age;
}
// creating instance
var emp1=new Employee();
var emp3=new Employee("Ali");
var emp4=new Employee("Ali",55);
alert(emp3.age)//0
```

1- Creating Objects using Constructor (Cont.)

- Adding methods to the constructor function using Function Literal:

```
function Employee(name, age)
{
    this.name = name;
    this.age = age;

    this.show =  function ()
    {
        alert("Employee " + this.name + " is " + this.age + " years old.");
    }

}
```

Property

Function Literal

1- Creating Objects using Constructor (Cont.)

- Calling object Methods:

```
var emp1 = new Employee("Aly", 23);

var emp3 = new Employee();
emp3.name = "Laila";
emp3.age = "30";

emp1.show(); // Employee Aly is 23 years old.

emp3.show(); // Employee Laila is 30 years old.
```

1- Creating Objects using Constructor (Cont.)

- Private and public Members

```
function Employee (name, age)
{
        //Private Member
        var id;
        //Public Properties
        this.name = name;
        this.age = age;
}
```

- You can't access private variables(encapsulation)

```
var emp1 = new Employee ("Aly", 23);
var emp2 = new Employee ("Hassan", 32);
alert(emp1.name); //Aly
alert (emp1.id); //Error, id is private
```

1- Creating Objects using Constructor (Cont.)

- Adding private (inner) and public methods to the constructor function:

```
function Employee(name, age)
{
        var id; //Private Member
        this.name = name; //Public Property
        this.age = age; //Public Property
        function setID(x) //private Method
        {id=x;}
        setID(1);

        this.getID=function () //public Method
        {return id;};
}
```

1- Creating Objects using Constructor (Cont.)

- Accessing object Methods:

```
var emp1 = new Employee("Aly", 23);

emp1.getID(); // 1
emp1.setID(); //Error, setID() is private Method.
```

2- Creating Objects using literal notation

What is an Object?

- An object is an unordered list of primitive data types (and sometimes reference data types) that is stored as a series of name-value pairs.
- Each item in the list is called a property (functions are called methods).

```
//creating object with some properties
var emp1= { Name: "Richard", Age: 30};
var emp2={Name:”Ali”, Age:20};

// using object
//you don’t need to create instance
emp1. Name=“Mahmoud”;
. alert(emp1.Name);
```

2- Creating Objects using literal notation (Cont.)

Adding Methods to the object

```
var emp1={ name:“Aly", age: 23,
                    show:function ( )
                    {
                          alert(this.name + " is " + this.age + " years old.“);
                    }
          };

// calling method, and again you don’t need to create instance
emp1.show();
```

Adding New property to specific object

- After an object exists, you can add a new property to that instance by simply assigning a value to the property name of your choice.
- After an object exists, you can add a new property to that instance by simply assigning a value to the property name of your choice.

```
var emp1 = { name:“Aly", age: 23};
var emp2 = { name: “Hassan",  age: 32};
var emp3=new Employee();
emp2.salary = 320;
emp3.salary=500;
alert(emp2.salary); //320
alert(emp3.salary); //500
alert (emp1.salary); //undefined, emp1 hasn’t salary property
```

- After that assignment, only emp2 has that property.
- There is no requirement that a property be pre-declared in its constructor or shortcut creation code.

Factory Function pattern

- Factory Function for Employee Object :

```
var Employee = function (e_nm, e_ag)
    { return {
            name : e_nm,
            age : e_ag
            }
    }
```

```
var Employee = function (e_nm, e_ag)
    { var emp = { name : e_nm,
                  age : e_ag
                };
      return emp;
    }
```

- **Creating object instances using Factory Function Method**

```
var emp1 = Employee (“Aly", 23);

var emp2 = Employee (“Hassan", 32);

var emp3 = Employee ();
```

Scope

- A scope is the lifespan of a variable
- In ES5; Only functions have scope. Blocks (if, while, for, switch) do not.
- ES6 presenting let for block scoping
- All variables are within the global scope unless they are defined within a function.
- All global variables actually become properties of the window object.
- When variables are not declared using the var keyword, they are declared globally

- Variables inside a function are
  - Free var: if they are not declared inside function scope and belong to another scope
  - Bound var: if they are declared inside function
- In JavaScript function scope is lexical/static scope; where free variables belongs to parent scope .
- Other language may have dynamic scope where free variables belongs to calling scope.
- JavaScript doesn't have dynamic scope

Scope(Cont.)

- Example 1:

```
// myVar is a global variable
var  myVar = “Hello”;
// create function to modify its own myVar variable
function test ()
{
        var myVar; // new local variable to test(), //this is called shadowing
        myVar=“Test1” // Change local variable
        window.myVar=“Test2” //Change global varibale
}
test();
// Global myVar equals “Test2”
alert( myVar);
```

Shadowing
occurs when a scope declares a variable that has the same name as one in a surrounding scope; the outer variable is blocked in the inner scope

Scope(Cont.)

- Example 2:

```
// myVar is a global variable
var  myVar = “Hello”;
// create function to modify its own myVar variable
function test ()
{
        myVar = “Bye”; // Global myVar’s value has changed
        myVar2=“Test” // Creating new global varibale
        var myVar3; //Creating new local variable.
}
test();
// Global myVar equals “Bye”
alert( myVar);
```

Scope(Cont.)

- Example 3:

```
var x1 =10; //make glabal variable “x1”
x2=10; //make glabal variable “x2”
function test ()
 {
        y=10; //global variable “y”
       var z=10; //local varibale “z”
}
```

Inner Functions

Inner Functions:

- Functions can be defined within one another
- The nested (inner) function is private to its containing (outer) function.
- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function can't use the arguments and variables of the inner function.
- In other words, The inner function contains the scope of the outer function.
- This provides a great deal of utility in writing more maintainable code. If a function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside the function that will be called from elsewhere. This keeps the number of functions that are in the global scope down, which is always a good thing.
- This is also a great to decrease using of global variables as Nested functions can share variables in their parent, so you can use that mechanism to couple functions together.

Inner Functions (cont.)


```
function getRandomInt(max)
{
        var randNum = Math.random() * max;
        function calcCeil()
        {
                var r= Math.ceil(randNum);
                return r;
        }
        var res= calcCeil();
        return res; // Notice that no arguments are passed
        //return r; //will not work
}
// Alert random number between 1 and 5
alert(getRandomInt(5));
```

Inner Function; only accessible within getRandomInt

Private methods

- Private methods are functions that are only accessible to methods inside the object and cannot be accessed by external code.
- Is the same as inner functions, but mostly we called it private method when it's inside function constructor (object), and inner function when it's inside ordinary function.

```
var User = function (name) {
    this.name = name;
    function welcome () {
    alert( "Welcome back, " + this.name + ".");}
    welcome();
}
// Create a new User
var me = new User( "Aly" ); // alerts: "Welcome back, Aly."

me.welcome();  // Fails because welcome is not a public method
```

Inner Function = Private Methods

Privileged methods

- The term privileged method is not a formal construct, but rather a technique.
- It's coined by Douglas Crockford
- Privileged methods essentially have one foot in the door:
  - Then can access private methods and values within the object.
  - They are also publicly accessible (like getters and setter in C++).

```
var User = function (name, age) {
var year = ((new Date().getFullYear() )– age); //local variable → private member
this.getYearBorn = function () { return year;};
};
// Create a new User
var user_1 = new User( "Aly", 25 );
// Access privileged method to access private year value
alert( user_1.getYearBorn());
alert( user_1.year); // undefined because year is private
```

Context

- Your code will always be running within the context of another object

Context is maintained through the use of the this variable.

- The this variable will always refer to the object that the code is currently inside of.

Prototype property

- Prototype: is a property that allows you to add more properties and methods to any created object.
- It gets created as soon as you define a function.
- Attaching new properties to a prototype will make them a part of every object instantiated from the original prototype, effectively making all the properties public (and accessible by all).
- This is another way to add more functionality to already created objects.
- It is also used for inheritance
- Detailed explanation for prototype property: https://hackernoon.com/prototypes-in-javascript-5bba2990e04b

Prototype property (cont.)


```
function User( name, age )
{
        this.name = name;
        this.age = age;
}
// Add a public accessory method for name
User.prototype.printEmp = function()
                        {return this.name + “his age” + this.age;};
User.prototype.job=“Engineer”;
var user1=new User(“Ahmed”,55);
alert( user1. printEmp()); //Ahmed his age 55
alert(user1.job)//Engineer.
user1.job=“Developer”;
alert(user1.job);//Developer
```

Prototypal Inheritance

- JavaScript uses a unique form of object creation and inheritance called prototypal inheritance.
- The premise behind this is that an object constructor can inherit methods from one other object, creating a prototype object from which all other new objects are built.
- This process is facilitated by the prototype property.
- Prototypes do not inherit their properties from other prototypes or other constructors; they inherit them from physical objects.

Prototypal Inheritance (cont.)


```
// Create the constructor for a Person object
function Person( name ) {
        this.name = name||"No Name";
}
// Create a new User object constructor
function User(password) {
    // Notice that this does not support graceful overloading/inheritance
    // e.g. being able to call the super class constructor
    this.password = password;
};
// The User object inherits all of the Person object's methods
User.prototype = new Person();
var NewUser = new User("pwd");
alert("New User Name:" + NewUser.name);//No Name, inherited from person
alert("New User Password: " + NewUser.password); //Member variable
NewUser.name="Ahmed";//inherited from person
alert("New User Name:" + NewUser.name);
```

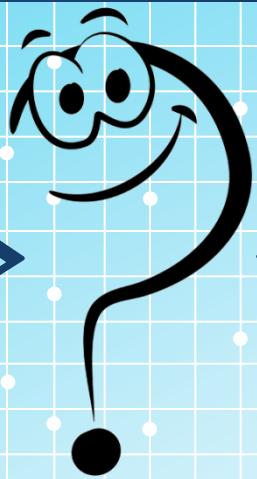
Prototypal Inheritance (Cont.)

- User.prototype = new Person();.
  - Let's look in depth at what exactly this means.
  - User is a reference to the function constructor of the User object.
  - new Person() creates a new Person object, using the Person constructor.
  - You set the result of this as the value of the User constructor's prototype.
  - This means that anytime you do new User(), the new User object will have all the methods that the Person object had when you did new Person().

Parasitic Inheritance

- Parasitic inheritance was first proposed as a way of handling inheritance in JavaScript by **Douglas Crockford** in 2005.
- •In this methodology you create an object by using a function that copies another object, augments it with the additional properties and methods you want the new object to have and then returns the new object.
- var childObj= Object.create (parentObjProto)

<SCRIPT >



</SCRIPT>

```
<script>document.writeln("Thank You!")</script>
```