Advanced JavaScript

Dr. Niveen Nasr El-Den
iTi

Knowing doesn’t build Skills..
Practicing do!!

Day 1

These are the Golden Days of JavaScript

JavaScript is designed on a simple **object-based paradigm**

Objects are the Fundamental unit of JavaScript

An object is a collection of properties.
A property is an association between a name (or key) and a value.

JavaScript Objects

According to Client-side RTE

- JavaScript Objects fall into 4 categories:
  - **Custom Objects**
  - **Built - in Objects**
  - **BOM Objects "Browser Object Model" (Host)**
  - **DOM Objects "Document Object Model".**

- In addition to objects that are predefined in the browser

  → we can define our own objects.

JavaScript Built-in Objects

- String
- Number
- Array
- Date
- Math
- Boolean
- RegExp
- Error
- Function
- Object

Creating variables

Literal “Short-Hand”	Constructor
var str = “abc”;	var str = new String();
var arr = [];	var arr = new Array();
var reg = /[a-z]/gmi;	var reg = new RegExp('[a-z]', 'gmi');
var obj = {};	var obj = new Object();

var fn = function(a, b){
return a + b;
}
	var fn = new Function(
	'a’, ‘b’, 'return a+b');

	var fn = new Function('a, b’ ,'return a+b');

Object Object

Reminder: Object Object

- Object is the parent of all JavaScript objects, which means that every object you create inherits from it
  - Reminder : the Global object is window object

- To create an object
  - var obj = { }; → preferable way
  - var obj = new Object( );

- Object object has constructor property that used to return the constructor function of the created Object.

- Objects are considered Associative Arrays also called a hash (the keys are strings)

Reminder

JavaScript uses **Arrays** to represent **indexed** arrays

&

Objects to represent **associative** arrays.

Reminder: Object Object

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

// adding property to object obj
obj.name = “JavaScript”;//dot notation → preferable approach
//obj[“name”]= “JavaScript”;// subscript notation
```

```
var obj ={
            // adding property to object obj
            name : “JavaScript”,
            //”name” : “JavaScript”,
        };
```

Object Object Properties & Methods

- .hasOwnProperty(“prop”)
- .valueOf()
- .toString()
- Object.keys(obj)
- Object.entries(obj)
- Object.values(obj)
- Object.defineProperty(obj,”prop”,{})
- Object.defineProperties(obj,{})
- Object.create(obj [,{}])
- …

Facts #1 About JavaScript Function

- Every thing you can do with other data types can be done with functions
  - Function can be
    - assigned to
      - a variable,
      - an 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.

Example

A property's value can be a function, in this case the property is known as a method

When an Object has at least one method, is an Object with behavior

Adding Method to Object

- Method is an action performed by executing a function
- Method is added to an object by assigning a function object as a value of an object property

?

Adding Method to Object

- Method is an action performed by executing a function

- Method is added to an object by assigning a function object as a value of an object property

```
// adding method to object obj
obj.say = function myFun() {  //literal function
    alert( "hello" );
};

 // "say" is a property that contains a  function object
```

Adding Method to Object

- Method is an action performed by executing a function

- Method is added to an object by assigning a function object as a value of an object property

```
// adding method to object obj
obj.say = function () {  //literal function
    alert( "hello" );
};

 // "say" is a property that contains a  function object
```

Adding Method to Object

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

// adding property to object obj
obj.name = “JavaScript”;  // dot notation
//obj[“name”]= “JavaScript”;  // subscript notation

// adding method to object obj
obj.say = function() { //literal function
    alert( "hello" );
};

 // “say” is a property that contains a  function object
```

Adding Method to Object

```
var obj ={
        // adding property to object obj
        name : “JavaScript”,
        //”name” : “JavaScript”,

        // adding method to object obj
        say : function() { //literal function
            alert( "hello" );
        }
    };
```

Example!

ES6+ : Adding Method to Object

```
var obj ={
        // adding property to object obj
        name : “JavaScript”,
        //”name” : “JavaScript”,

        // adding method to object obj
        say() { //literal function
            alert( "hello" );
        }
    };
```

Example!

Adding Method to Object

```
var obj ={
        // adding property to object obj
        name : “JavaScript”,
        //”name” : “JavaScript”,

        // adding method to object obj
        say : say
        };

function say() {
    alert( "hello" );
}
```

Example!

In JavaScript we have **property** that contains **Function Object**

A **function** always returns a value

Facts #2 About JavaScript Function

- JavaScript is a first-class function.
  - *Functions* are treated as first class citizens since it can be considered as *values* in JavaScript and treated like any variable. i.e. being passed as an argument, returned from a function, modified, and assigned to a variable
- Functions in JavaScript are first-class objects
  - Functions are a special data type.
  - Function objects have properties and functions
- 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.

Function Object

Function Object

- JavaScript functions are objects. They can be defined using the Function constructor
  (Dynamic / Anonymous / **Function Constructor**)
  - var sum = new Function('a', 'b', 'return a + b;');
  - alert(sum (2 , 3));

- JavaScript functions using the function literal, it is also known as Factory Function
  (Literal / Anonymous / **Function expression**)
  - var sum = function(a, b){return a + b;};
  - alert(sum (2 , 3));

- The more common traditional way:
  (Declarative / Static / **Function Statement**)
  - function sum(a, b){return a + b;}
  - alert(sum (2 , 3));

Declarative/ Static Function

- The most common traditional type of function uses the declarative/static format.

```
function funNm (par1, par2,.. , parn){
    //fun body;
}
```

- This approach begins with
  - function keyword,
  - followed by function name,
  - parentheses containing zero or more arguments,
  - and then the function body
  - Parsed once when the page is loaded
    - The parsed result is used each time the function is called
  - Hoisted (useful for mutual recursion)
- Simple to read and understand
- Its a function statement that does some work

Dynamic/Anonymous Function

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

- The Dynamic/Anonymous Function:
  - *Anonymous:* because the function itself isn't directly declared or named.
  - *Dynamic*: The JavaScript engine creates the anonymous function dynamically,
    - each time it's invoked, the function is dynamically reconstructed.
  - Uses Function object constructor
  - Do not create closures to their creation contexts; they are always created in the global scope
- Example:

```
var sayHi = new Function("toWhom","alert('Hi ‘ + toWhom);”);
sayHi("World!");
```

Literal Declaration

```
var func = function [fun_nm]  (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:

```
var func = function (x, y) { return x * y; }
alert(func(3,3));
```

Anonymous Function

- functions are like any other variable so they can also be used without being assigned a name.
- Anonymous functions are functions that are passed as arguments or declared inline and have no name
- Example:
  - 1;
  - [1,2,”str”];
  - “Hello!!”
  - function(a){return a;}

Anonymous Function

- Advantage:
  - Used as callback function
  - Used as IIFEs
  - Hide Variables from the Global Scope
- Disadvantage:
  - Cant execute twice unless it is put inside loop or another function

Calling anonymous Functions

- You can pass an anonymous function as a parameter to another function. The receiving function can do something useful with the function that you pass.
- Self-invoking Functions
  - You can define an anonymous function and execute it right away. By calling this function right after it was defined. This is called IIFE “Immediate Invoke Function Expression”
  - Example

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

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

IIFE

- IIFE stands for Immediately Invoked Function Expression
- It is a function expression that is invoked immediately
- A common often extra ordinary used pattern
- Can be invoked on the fly at the point it is created
- Function expression is wrapped within () operator

IIFE

- Prefix operators may be used

```
void function () {
  console.log("inside IIFE") }();
```

```
!function () {
  console.log("inside IIFE") }();
```

- Trailing semicolon is required between two IIFEs

```
(
    function(){
      alert('hello IIFE1');
    }
)();
(
    function(){
      alert('hello IIFE2');
    }
)()
```

Function Object Properties

- prototype
  - This is another way to add more functionality to already created objects
- length
  - Specifies the number of parameters the function expects.
    - i.e. set in its signature
- caller (obsolete)
  - This returns a reference to the function that called our function
- name
  - This returns the name of the function if it has a name after function keyword.

arguments Object

- When a function receives parameter values from a caller, those parameter values are implicitly assigned to the *arguments* property of the function object.
  - i.e. arguments, contains values of all parameters passed to the function.
  - Missing parameters are treated as undefined values
  - We can pass more arguments than expected.
- It looks like an array, but its not an array although it contains indexed elements and has length property that returns number of arguments passed to the function
- It has callee property. (deprecated)
  - This contains a reference to the function being called.
  - arguments.callee allows Self-invoking anonymous functions to call themselves recursively.

Example!

- It is used inside function body implementation.

arguments.callee Example

```
function accumlateSum (n){

    if(n!=0)
        return n + accumlateSum (--n);

    else
        return n;

}
```

```
function accumlateSum (n){
    if(n!=0)
        return n + arguments.callee(--n);
    else
        return n;
}
```

Reminder: Function Default arguments

```
function myFun(){
	var x = arguments[0] || 10;
	var y = arguments[1] == undefined ? 11 :
arguments[1]

	return x+y;
}

myFun(); //21
myFun(1); //12
myFun(1,2); //3
```

```
function myFun(x=10,y=11){ /*ES6*/
	return x+y;
}
```

Function Object Methods

- Function borrowing:
  - apply(this_obj, params_array)
    - Allows you to call another function while overwriting its this value.
    - The first parameter that apply() accepts is the object to be bound to this inside the function and the second is an array of parameters to be passed to the function being called.
  - call(this_obj, p1, p2, p3, ...)
    - Same as apply() but accepts parameters one by one, as opposed to as one array.
  - bind(obj)
    - Allows you to call a function into another object.
    - We can achieve function currying via bind

http://javascriptissexy.com/javascript-apply-call-and-bind-methods-are-essential-for-javascript-professionals/

Function Object Methods

```
var myStr = “this is an example of using Function Methods”;
var arr = [];

//borrowing using apply
arr.join.apply (myStr, [“*”]);

//borrowing using apply
arr.join.call (myStr, “*”);

//currying using bind
var newBind = arr.join.bind(myStr);
newBind(“*”);

var newBind = arr.join.bind(myStr , “*”);
newBind();
```

Note: arr can be replaced by [] or (new Array)

Result: "t*h*i*s* *i*s* *a*n* *e*x*a*m*p*l*e* *o*f* *u*s*i*n*g* *F*u*n*c*t*i*o*n* *M*e*t*h*o*d*s"

call() and apply() Example

```
var myObj = {
      nm: "myObj Object",
      myFunc: function(){
                        alert(this.nm)
                  },
      myFuncArgs: function(x,y){
                        alert(this.nm + " " + x + " “ + y)
            }
};

var obj1 = {nm: "obj1 Object"};
```

```
myObj.myFuncArgs(1,2);//myObj Object 1 2

myObj.myFuncArgs.apply(obj1,[1,2]);//obj1 Object 1 2

myObj.myFuncArgs.call(obj1,1,2);//obj1 Object 1 2
```

Inner Functions

- Functions can be defined within one another
- Inner functions have access to the outer function's variables and parameters.

```
function getRandomInt(max) {
var randNum = Math.random() * max;
        function ceil() {
        return Math.ceil(randNum);
        }
return ceil(); // Notice that no arguments are passed
}

// Alert random number between 1 and 5
alert(getRandomInt(5));
```

Inner Function; only accessible within getRandomInt

Inner Functions

```
function a(param) {
    function b(theinput)    {
        return theinput * 2;
    };

    return 'The result is ' +
        b(param);
};
```

```
var a = function(param) {
    var b = function(theinput) {
        return theinput * 2;
    };

    return 'The result is ' +
        b(param);
};
```

a(2); →"The result is 4“

a(8); →"The result is 16“

b(2); →b is not defined

Example!

Inner Functions

- 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 cannot use the arguments and variables of the inner function
    - i.e. The inner function contains the scope of the outer function.
  - When two arguments or variables in the scopes of a closure have the same name, there is a name conflict. More inner scopes take precedence
  - According to scope chain, the inner-most scope is the first on the chain.

Inner Functions

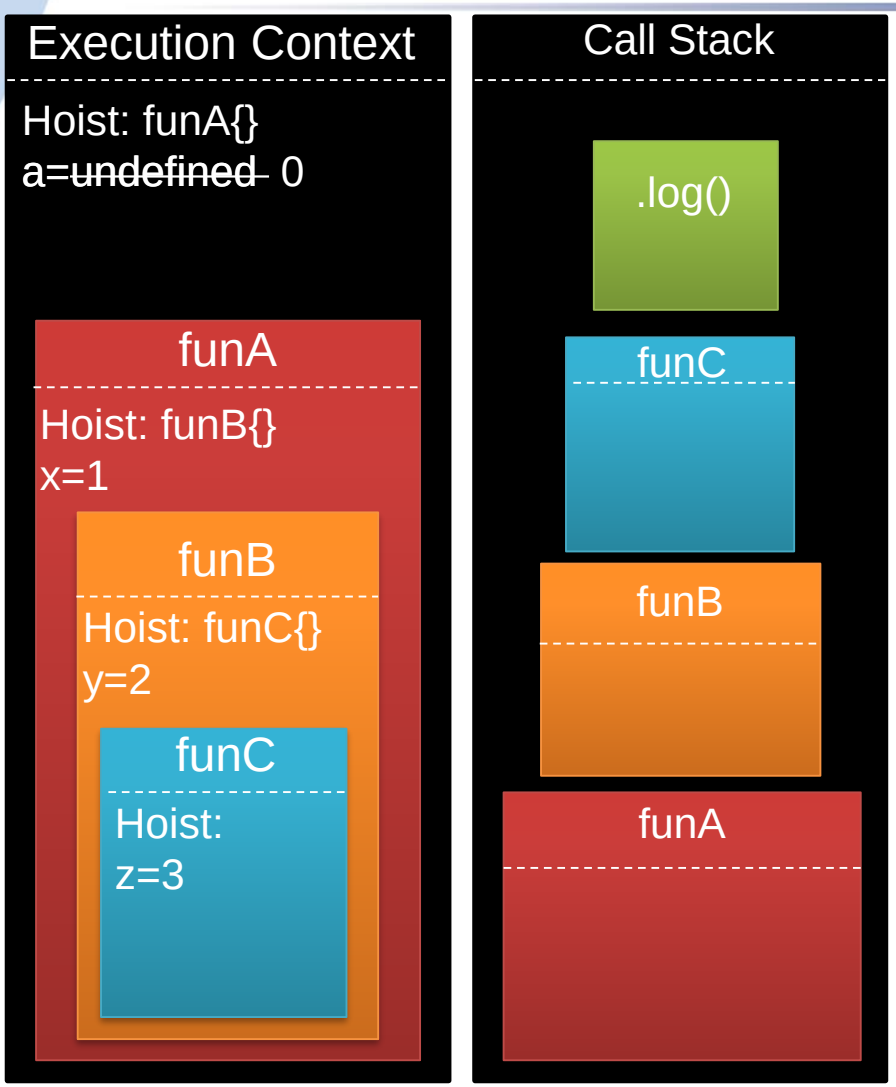
```
function myFun(x) {
    var z = 10;

    function innerFun(y) {
        return x + y + z;
    }

    return innerFun;
}
```

```
var myFun = function (x) {
    var z = 10;

    return function (y) {
        return x + y + z;
    };

}
```

```
var fun = myFun(5);
var result = fun(10);
```

```
var result = myFun(5)(10);
```

Inner Functions




```
function funA(x) {
  function funB(y) {
    function funC(z) {
      console.log(x + y + z + a);
    }
    funC(3);
  }
  funB(2);
}
var a=0;
```

```
funA(1);        //6
```

Closures

- Closure is one of the most powerful features of JavaScript.

  “A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).”

  - When function returns from another function it returns with all variables from its external scope
  - A new closure is created for each call to outside.
  - Closure wrap-up the entire environment with all variables from external scope

Closures

- It is created when the inner function is somehow made available to any scope outside the outer function.
- It provides a sort of security for the variables of the inner function, since they are not accessed by their outer function.
- Closures grant the inner function full access to all the variables and functions defined inside the outer function.
- The outer function does not have access to the variables and functions defined inside the inner function.

Closure is
created by the nested functions
to retain access to each of the
arguments.
So inner function have access
to all arguments of
outer scope

Problem

```
function closureTest(){
	var arr = [];

	for(var i = 0; i < 3; i ++) {
		arr.push(function(){
		console.log(i);
		});
	}

	return arr;
}

var cFn = closureTest();

cFn[0]();
cFn[1]();
cFn[2]();
```

Solution

```
function closureTest(){
    var arr = [];

    for(var i = 0; i < 3; i ++) {
        arr.push((function(j) {
                    return function(){console.log(j);}
                })(i)
        );
    }

    return arr;
}

var cFn = closureTest();

cFn[0]();
cFn[1]();
cFn[2]();
```

ES6+ : Solution using let

```
function closureTest(){
      var arr = [];

      for (let i = 0; i < 3; i++) {
        arr.push(function () {
            console.log(i);
        });
    }

      return arr;
}

var cFn = closureTest();

cFn[0]();
cFn[1]();
cFn[2]();
```

IIFE Pattern

- A common often extra ordinary used pattern
- 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

Assignments