


Advanced JavaScript

Eng. Niveen Nasr El-Den
iTi

Day 3



*These are the
Golden Days of
JavaScript*



JavaScript is
Multi-paradigm
Programming Language.



JavaScript supports
programming
in **many** different **styles**.



Object-Oriented JavaScript

Object-Oriented JavaScript

- The main principle with **OOP** is the use of **Classes** to create objects, and that objects are implemented in a manner that allows them to adopt **Inheritance**, **Polymorphism**, and **Encapsulation**.
- In most other object-oriented languages you would instantiate an instance of a particular class, but that is not the case in JavaScript.
- Unlike most other object-oriented languages, JavaScript doesn't actually have a concept of **classes**. It looks and **behaves differently**.

Object-Oriented JavaScript

- JavaScript is a *class-free*, object-oriented language
- Although **ES6** introduces JavaScript **class** expressions and class declarations, to provide a much clearer syntax to create objects and deal with.
- In fact **classes** are **functions**
- **Custom Object** that you, as a JavaScript developer, create and use is the main actor in application.

Custom Object

- Objects that you, as a JavaScript developer, create and use.
- An object in JavaScript is a complex construct usually consisting of a constructor as well as zero or more methods and/or properties.
- Objects can be either **stand-alone** with their own set of properties & functions or they can **inherit** properties from other objects

Custom Object

- There are different ways to create an instance of an object class (**Functions** in JavaScript)
 - ▷ Basic Object Literal Pattern
 - ▷ Factory Function
 - ▷ Custom Object Constructor Function
 - ▷ ...

Literal Pattern Object Creation

```
var obj = { };  
  
obj.name = "banana"  
  
obj.click = function(){  
    alert( "you can eat" );  
}  
  
obj.details = {  
    mycolor: "yellow",  
    mycount: 12  
}  
  
//(obj instanceof Object) // true
```

Literal Pattern Object Creation

```
var obj = {  
  // Set the property names and values use key/value pairs  
  "name" : "banana", // name : "banana"  
  click : function(){  
    alert( "you can eat" );  
  },  
  //initialize entire object  
  details : {  
    mycolor: "yellow",  
    mycount:12  
  }  
};
```

Custom Object creation using basic object literal pattern

- We can create objects with a short syntax that defines an object inside curly braces. (**basic object literal pattern**)

```
var emp1 = { name:"Aly", age: 23};
```

```
var emp2 = { name: "Hassan", age: 32};
```

Custom Object creation using basic object literal pattern

- 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.
- For example, to add a property about the “Salary” for “Hassan”, the statement is:

```
var emp1 = { name: "Aly", age: 23};  
var emp2 = { name: "Hassan", age: 32};  
emp2.salary = 320;
```

- 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

- It is a way where **object** is created as a **return** of a function call assigned to a variable
- Used to create Multiple Objects with same interface
- No need to use “**new**” when calling a factory function

Creating New Instance from Custom Object using Factory 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 ();
```


Constructor Function

- A constructor function looks like any other JavaScript function, but its purpose is:
 - ▷ to define the initial structure of an object
 - ▷ to define its 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.
- MyConstructor
- myFunction

Creating New Instance from Custom Object using Constructor Method

- Constructor Function for Employee Object

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

- To create object instances using Constructor Function Method, invoke the function with the **new** keyword

```
var emp1 = new Employee ("Aly", 23);  
  
var emp2 = new Employee ("Hassan", 32);  
  
var emp3 = new Employee ();
```

Adding methods to Constructor Function (**Functional shared Pattern**)

- Functional shared pattern is used to save memory by adding methods to the constructor function:

```
function Employee(name, age){
```

```
    this.name = name;  
    this.age = age;
```

Property

```
    this.show = showAll;
```

Method

```
}
```

```
function showAll( ){
```

```
    alert("Employee " + this.name + " is " + this.age + " years  
    old.");
```

```
}
```

Adding methods to Constructor Function (**Functional Class Pattern**)

- Adding methods to the constructor function using **Function Literal**:

```
function Employee(name, age) {
```

```
    this.name = name;  
    this.age = age;
```

Property

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

Function
Literal

```
}
```

Instance Object Creation

```
// Class using constructor function
function User( name ) {
    this.name = name;
    this.display = function(){return this.name;}
}

// Instance object of user
var me = new User( "My Name" );

// Test
alert( me.name );
alert( me.display());
alert( me.constructor == User); // true
alert( me.constructor == Object); // false
```

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
```

Creating New Instance from Custom Object via Constructor **Overloading**

- Assign a *default value* to a Property:

```
function Employee (id='idx' /*ES6*/),name, age,salary=2000/*ES6*/){  
  this.name = typeof name ==="undefined"? "Nour": name;  
  this.age = age || 0; //ES5  
  this.salary = salary;  
  this.id = id;  
}
```

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

```
var emp1 = new Employee( );  
emp1.name = "Aly";  
emp1.age = 23;
```

Overloading

- A common feature in other object-oriented languages is the ability to “overload” functions.
- Overloading occurs when more than one method within the same class have the same method name but different in parameters (different numbers and/or types of passed arguments) to perform different behaviors
- Overloading can be fulfilled via
 - function arguments property using default parameters & any conditional statement.
 - Creating function that calls the meant function with proper requirement

Constructor Function, new & this

- When function invoked with **new**, functions return an object known as **this**.
- “**new**” before any function call **turns** it into **constructor** function
- JavaScript uses “**this**” keyword to refer to the current object.
- “**this**” is confusing sometimes, when it doesn't return the expected object.
- You have a chance of modifying **this** before it is returned

“new” Operator

- When using “new”
 - ▷ A brand new empty object is created
 - ▷ That object get linked to another object
 - ▷ It gets bound as “this” keyword as a purpose for function call
 - ▷ If the function doesn’t return any thing, it will return “this”.
- Note:
 - ▷ Primitive datatypes pass by value while Objects and Arrays pass by reference;
 - When any change happens in obj1 it is reflected in obj2
 - ▷ using new can over come this problem

Example: “this” and Closure

```
function Employee(nm, age){  
  this.eName = nm;  
  this.age = age;  
  this.show = function () {  
  
    setTimeout(function (){  
      alert("Employee " + this.eName + " is " + this.age + " years");  
    },5000);  
  }  
}
```

local-ntp says

Employee undefined is undefined years

OK

```
var me = new Employee("Nour",5);  
me.show()
```

?

“this” keyword & Binding

- Every function while executing, has a reference to its execution context called “this”.
- “this” is an identifier that gets the value of object bound to it, it behaves like normal parameters.
- “this” binding is dependent on its “call site” (where the function get executed)

“this” keyword

- “**this**” is dynamic since it looks for things at runtime, based upon how you call things
- 4 rules for binding “**this**” (in terms of order precedence) depending on call site
 - ▷ Default Binding
 - ▷ Implicit Binding
 - ▷ Explicit Binding
 - ▷ new keyword

priority ↑
low
high

Hard Binding

Default & Implicit Binding

- Default Binding

- It is applied on a standalone functions & IIFEs
 - Function defined in Global Scope
- Depends on Strict Mode of code running inside a function
 - Its value is **undefined** in strict mode,
 - To be applied globally should be called as `window.fn`
 - otherwise its value is **Global** Object

- Implicit Binding

- An object is calling the function
 - Object on the left of the `(.)` function call

Example

```
function myFun(){  
    console.log(this.val)  
}  
var val = "myVal";  
  
var myObj1 = {val : "obj1Val", myFun: myFun};  
  
var myObj2 = {val : "obj2Val", myFun : myFun};  
  
myFun(); //myVal  
myObj1.myFun(); //obj1Val  
myObj2.myFun(); //obj2Val
```

Explicit Binding

- It's a hard binding
- When function is called, it predict its object
- If you want to set a specific object other than the calling object make hard binding using Function Object methods.
 - ▷ `bind()`
 - ▷ `apply()`
 - ▷ `call()`

Example!

Using call() and apply()

```
var myObj={  
  name:"myObj Object",  
  myFunc:function(){  
    alert(this.name)  
  },  
  myFuncArgs:function(x,y){  
    alert(this.name+" " + x + " "+y)  
  }  
};  
  
var obj1={name:"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
```

Using bind()

```
var myObj={  
  name:"myObj Object",  
  myFunc:function(){  
    alert(this.name)  
  },  
  myFuncArgs:function(x,y){  
    alert(this.name+" " + x +" "+y)  
  }  
};  
  
var obj1={name:"obj1 Object"};
```

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

```
myObj.myFuncArgs.bind(obj1)(5,6); //obj1 Object 5 6
```

```
myObj.myFuncArgs.bind(obj1,5)(6); //obj1 Object 5 6
```

```
myObj.myFuncArgs.bind(obj1,5,6)(); //obj1 Object 5 6
```

"this" and Closure

```
function Employee(name, age){  
  this.eName = name;  
  this.age = age;  
  this.show = function () {  
    var that=this; //_that|_self|_this  
    setTimeout(function (){  
      alert("Employee " + that.eName + " is " + that.age + " years");  
    },5000);  
  }  
}
```

local-ntp says

Employee Nour is 5 years

OK

```
var me = new Employee("Nour",5);  
me.show()
```



Reminder: “Scope” Basics

- A scope is the lifespan of a variable
- Programming languages have block and function scope
- In **ES5**; only functions have scope. Blocks (if, while, for, switch) do not have scope.
- **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 global object (**window object**).

Reminder: “Scope” Basics

- 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
- When variables are *not declared using the **var** keyword*, they are declared **globally**.

Reminder: “Scope” Basics

```
var myVar = “Hello”; // myVar is a global variable
```

```
// create function to modify its own myVar variable
```

```
function test (){
```

```
    var myVar = “Bye”; //Local variable to test()
```

```
    //this is called shadowing
```

```
}
```

```
test();
```

```
alert( myVar); // Global myVar still equals “Hello”
```

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

Privileged Method

- 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:
 - They can access private methods and values within the object
 - They are also publicly accessible

Privileged Method

```
var User = function (name, age) {  
    var year = ((new Date().getFullYear() )- age);  
  
    //year is a 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
```


Private Methods

- Private methods are functions that are only accessible to methods inside the object and cannot be accessed by external code.

```
var User = function (name) {  
    this.name = name;  
      
    function welcome () {  
        alert( "Welcome back, " + this.name + ".");  
    }  
    welcome();  
}
```

Inner
Function =
Nested
Function

Private
Method

// Create a new User

```
var me = new User( "Aly" ); // alerts: "Welcome back, Aly."
```

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

"this", Closure and Private Method

```
function Employee(name, age, yr){  
    this.eName = name;  
    this.age = age;  
    var yrbrn=yr;  
  
    function welcoming() {  
        alert("welcome " + this.eName + " you were born in " + yrbrn );  
    }  
  
    this.welExec=function(){  
  
        return val();  
    }  
  
    //welcoming.call(this);  
    var val=welcoming.bind(this) //val();  
}
```

Hard binding

no matter what is the invocation context.

Make a function that calls
internally and manually
an **explicit binding**

and

force to do the same instruction
no matter where and
how you invoke that function

Class Properties & Methods

- Class Properties and methods are similar to static properties and methods in other object oriented languages.
- This can be created by adding either property or method to a constructor function object.
- This is possible because functions in JavaScript are plain objects that can have properties and methods of their own.

Class Properties & Methods

```
function Employee(name, age){  
    this.name = name;  
    this.age = age;  
}  
  
Employee.count=0;  
  
Employee.getCount=function(){  
    return Employee.count  
}
```

Example!



Assignments