ES6 & JS

Object Oriented JavaScript

ITI – Assiut Branch
Eng. Hany Saad

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/ Function expression**)

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

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.

- 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);
```

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);};
```

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.

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

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.

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.)

- 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.)

- 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.)

- Adding methods to the constructor function:

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

        this.show = showAll;
}

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

Property

Method

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.
```

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 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

Closures

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).
- It is created when the inner function is somehow made available to any scope outside the outer function.
- If the inner function manages to survive beyond the life of the outer function; the variables and functions defined in the outer function will live longer than the outer function itself, since the inner function has access to the scope of the outer function.
- In short words:
  - a closure is the local variables for a function — kept alive after the function has returned

Closures (cont.)

- **Example:**

```
function sayHello2(name) {
    var text = 'Hello ' + name; // Local variable
    var sayAlert = function() { alert(text); }
    return sayAlert;    //returning reverence to the inner func.
}
var say2 = sayHello2('Bob');
//say2 holds a reference to the inner func. That access the outer func variables.
say2(); // alerts "Hello Bob"
```

- The above code has a closure because the anonymous function function() { alert(text); } is declared inside another function, sayHello2() in this example. In JavaScript, if you use the function keyword inside another function, you are creating a closure.
- In JavaScript, if you declare a function within another function, then the local variables can remain accessible after returning from the function you called. This is demonstrated above, because we call the function say2() after we have returned from sayHello2(). Notice that the code that we call access the variable text, which was a local variable of the function sayHello2().
- The anonymous function can reference text which holds the value 'Hello Bob' because the local variables of sayHello2() are kept in a closure.
- The magic is that in JavaScript a function reference also has a secret reference to the closure it was created in.

Closures (cont.)

- **Another Example (Problem):**

```
function closureTest(){
    var arr = [];
    for(var i = 0; i < 3; i ++) {
        arr.push(function(){
            alert(i);
        });
    }
    return arr;
}
var cFn = closureTest();
cFn[0](); //3
cFn[1](); //3
cFn[2](); //3
```

- Note that when you run the example, "3" is alerted three times! This is because there is only one closure for the local variables for closureTest.
- When the anonymous functions are called on the line cFn[0]; they all use the same single closure, and they use the current value for i and item within that one closure (where i has a value of 3 because the loop had completed, and item has a value of '3').

Closures (cont.)

- **Another Example (Solution):**

```
function closureTest(){
        var arr=[]
        var i;
        for(var i = 0; i < 3; i ++){
        arr.push((function(j){ return function(){
                                                alert(j);
                                        }
                                })(i)
                        );
        }
 return arr;
}
var cFn = closureTest();
cFn[0](); //0
cFn[1](); //1
cFn[2](); //2
```

ES6 new features

Variables – block scope with let

❑ Block variable declaration: let (New ES6 feature):

- There was no Block Scope before ES6, only function scope, let declaration introduced in ES6 allowing block scope
- Variables declared by let have as their scope the block in which they are defined, as well as in any contained sub-blocks .
- let variables are block-scoped. The scope of a variable declared with let is just the enclosing block, not the whole enclosing function.

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

Variables – block scope with let (Cont.)

❑ Block variable declaration: let (Cont.):

- Loops of the form for (let x...) create a fresh binding for x in each iteration, and the scope of the variable will be inside the for loop only.

```
function test(){
        ……
        for (let i = 0; i < messages.length; i++) {
                ... //let scope inside loop only, not whole function.
        }
}
```

- Global let variables are not properties on the global object. That is, you won't access them by writing window.variableName. Instead, they live in the scope of an invisible block that notionally encloses all JS code that runs in a web page.
- It's an error to try to use a let variable before its declaration is reached (as variables declared using let aren't hoisted).

```
function update() {
  document.write("your name:", t); // ReferenceError
  ...
  let t = "test";}
```

Variables - Constants

JavaScript Constants (new ES6 Feature):

- Variables declared with const are constant variables, youcan't assign to them, except at the point where they're declared.

```
const MAX_CAT_SIZE_KG = 3000;
MAX_CAT_SIZE_KG = 5000; // SyntaxError
MAX_CAT_SIZE_KG++; // SyntaxError
const theFairest; // SyntaxError, you can't declare const variable without assigning it a value
```

- A constant can be global or local to a function where it is declared.
- Constants also share a feature with variables declared using let in that they are block-scoped instead of function-scoped (and thus they are not hoisted)

Template Literals

- Template literals allow us to easily create templates in which we can embed different values to any spot we want.
- To do so we need to use the ${...} syntax everywhere where we want to insert the data that we can pass in from variables, arrays, or objects.

```
1  let customer = { title: 'Ms', firstname: 'Jane', surname: 'Doe', age: '
2
3  let template = `Dear ${customer.title} ${customer.firstname} ${customer.
4  Happy ${customer.age}th birthday!`;
5
6  console.log(template);
7  // Dear Ms Jane Doe! Happy 34th birthday!
```

Demo!

Arrow functions

- ECMAScript 6 facilitates how we write anonymous functions, as we can completely omit the function keyword.

- We only need to use the new syntax for arrow functions, named after the => arrow sign (fat arrow), that provides us with a great shortcut.

```
1  // 1. One parameter in ES6
2  let sum = (a, b) => a + b;
3
4    // in ES5
5    var sum = function(a, b) {
6      return a + b;
7    };
8
9  // 2. Without parameters in ES6
10 let randomNum = () => Math.random();
11
12   // in ES5
13   var randomNum = function() {
14     return Math.random();
15   };
16
17 // 3. Without return in ES6
18 let message = (name) => alert("Hi " + name + "!");
19
20   // in ES5
21   var message = function(yourName) {
22     alert("Hi " + yourName + "!");
23   };
```

Arrow functions

- Before arrow functions, every new function defined its own this value (a new object in the case of a constructor, undefined in strict mode function calls, the context object if the function is called as an "object method", etc.).
- An arrow function does not create its own this context, so this has its original meaning from the enclosing context.

Demo!

New spread Operator

- The new spread operator is marked with 3 dots (…), and we can use it to sign the place of multiple expected items.
- One of the most common use cases of the spread operator is:
  - inserting the elements of an array into another array.
  - We can also take leverage of the spread operator in function calls in which we want to pass in arguments from an array.

```
1  let myArray = [1, 2, 3];
2
3  let newArray = [...myArray, 4, 5, 6];
4
5  console.log(newArray);
6  // 1, 2, 3, 4, 5, 6
```

```
1  let myArray = [1, 2, 3];
2
3  function sum(a, b, c) {
4    return a + b + c;
5  }
6
7  console.log(sum(...myArray));
8  // 6
```

Default Values for Parameters & New Rest Parameters (Cont.)

- ES6 also introduces a new kind of parameter, the rest parameters.
- They look and work similarly to spread operators, They come handy if we don't know how many arguments will be passed in later in the function call.

```
1  function putInAlphabet(...args) {
2
3    let sorted = args.sort();
4    return sorted;
5
6  }
7
8  console.log( putInAlphabet("e","c","m","a","s","c","r","i","p","t") );
9  // a,c,c,e,i,m,p,r,s,t
```

Destructuring assignment

- The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects into distinct variables.

```
var a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

```
var x = [1, 2, 3, 4, 5];
var [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```

```
var a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
```

for..of

- The famous for..in loop whose first value is to iterate over the different keys of an object or an array.
  - When itarating over an array, index value is parsed to string : "0", "1", "2", etc.. This behaviour can lead to potential error when index is used in computation.
- The alternative .forEach() method oop allow a more secure iteration, but bring other downsides as:
  - Impossibility to halt the loop with the traditional break; and return; statements.
  - Array only dedicated method.
- ECMA consortium has so decided to proceed with establishment of a new enhanced version of the for..in loop. Thus was born the for..of loop which, from now on, will coexist with the previous one allowing to maintain the backward compatibility with former version of the standard.
  - for–of is not just for arrays. It also works on most array-like objects, like DOM NodeLists.
  - It also works on strings, treating the string as a sequence of Unicode characters

```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i); // "0", "1", "2",
}

for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

```
const str = 'sm00th';

for ( const chr of str ){
  console.log(chr); // 's', 'm', '0', '0', 't', 'h'
}
```

for..of (Cont.)

- In a nutshel for..of comes to:
  - Address for..in loop gaps
  - Allow a simplified iteration over iterable objects (Array, String, Maps, Sets, Generators,NodeList, arguments)
  - Unlike .foreach()Allow using break, continue, return.

Sets

- The Set object lets you store unique values of any type, whether primitive values or object references.
- Syntax: **var mySet=new Set([iterable]);**
  - If an iterable object is passed, all of its elements will be added to the new Set. If null is passed instead of iterable, it is treated as not passing iterable at all.
- More details:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Demo!

Maps

- The Map object is a simple key/value map. Any value (both objects and primitive values) may be used as either a key or a value.
- Syntax: **var new Map([iterable]);**
  - Iterable is an Array or other iterable object whose elements are key-value pairs (2-element Arrays). Each key-value pair is added to the new Map. null is treated as undefined.
- Maps Vs. Objects:
  - Map instances are only useful for collections, and you should consider adapting your code where you have previously used objects for such.
  - Objects shall be used as records, with fields and methods.
  - If you're still not sure which one to use
- More details:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

```
1  var myMap = new Map();
2  myMap.set(NaN, 'not a number');
3
4  myMap.get(NaN); // "not a number"
```

Demo!

Generators

- Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.
- Calling a generator function does not execute its body immediately; an iterator object for the function is returned instead.
  - When the iterator's next() method is called, the generator function's body is executed until the first yield expression , which specifies the value to be returned from the iterator or, with yield*, delegates to another generator function.
  - The next() method returns an object with a value property containing the yielded value and a done property which indicates whether the generator has yielded its last value as a boolean.

- The function* declaration (function keyword followed by an asterisk) defines a generator function, which returns a Generator object.

Demo!

Classes

- ES6 introduces JavaScript classes that are built upon the existing prototype-based inheritance.
- The new syntax makes it more straightforward to create objects, take leverage of inheritance, and reuse code.
- Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

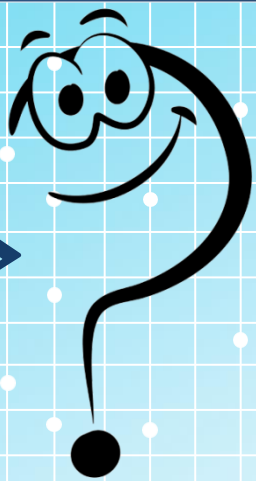
```
1  class Polygon {
2    constructor(height, width) { //class constructor
3      this.name = 'Polygon';
4      this.height = height;
5      this.width = width;
6    }
7
8    sayName() { //class method
9      console.log('Hi, I am a', this.name + '.');
10   }
11 }
12
13 let myPolygon = new Polygon(5, 6);
14
15 console.log(myPolygon.sayName());
16 // Hi, I am a Polygon.
```

Demo!

Resources

- **Online Resources:**
  - http://www.hongkiat.com/blog/ecmascript-6/
  - https://webapplog.com/es6/
  - http://exploringjs.com/es6/ch_overviews.html
  - https://developer.mozilla.org
  - https://developers.google.com/web/fundamentals/getting-started/primers/promises
  - https://leanpub.com/understandinges6/read/
- **Books:**
  - *Understanding ECMAScript 6* by Nicolas Zakas book
  - ES6 Cheatsheet (FREE PDF)
  - *Exploring ES6* by Dr. Axel Rauschmayer
- **Resources:**
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

<SCRIPT >



</SCRIPT>

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