

Javascript

PART 4

Javascript Objects

- Object Definition

- You define (and create) a JavaScript object with an object literal:

```
var person = {  
  firstName: "John",  
  lastName : "Doe",  
  age      : 50,  
  eyeColor : "blue",  
  fullName : function() {  
    return this.firstName + " " + this.lastName;  
  }  
};
```

- You can access object properties in two ways:

objectName.propertyName
objectName["propertyName"]

- You access an object method with the following syntax:

objectName.methodName()

```
document.getElementById("demo").innerHTML =  
person.fullName() + " is " + person.age + " years old.";
```

Javascript Core Objects

JavaScript provides a number of standard objects, with properties and methods to perform various manipulations. Core objects are independent of the client browser

Object	Description
Array	The Array object as the name suggest, is used to create arrays. It has many methods to add, delete or extract elements from an array and sort them.
Boolean	The Boolean object is used to create Boolean values. Elements with two states: true and false.
Date	The Date object is used to create dates and time. It also offers methods to manipulate them.
Function	The Function object allows you to define custom functions.
Math	The Math object allows you to manipulate mathematical functions, such as trigonometric functions.
Number	The Number object allows you to perform basic operations on numbers..
RegExp	The RegExp object allows you to create regular expressions.
String	The String object provides a variety of methods for manipulating strings.

Javascript Arrays

JavaScript arrays are used to store multiple values in a single variable.

There are two ways for creating an array:

- Using an array literal:

```
var cars = ["Saab", "Volvo", "BMW"];
```

- With the array constructor:

```
var cars = new Array("Saab", "Volvo", "BMW");
```

The two examples above do exactly the same. There is no need to use `new Array()`. For simplicity, readability and execution speed, use the first one (the array literal method).

Javascript Arrays

It is possible to create an empty array and then add the elements:

- Using an array literal:

```
var fruits=[];  
fruits[0]='orange';  
fruits[2]='apple';  
  
alert (fruits[2]);
```

- Or with the array constructor:

```
var fruits=new Array();  
fruits[0]='orange';  
fruits[2]='apple';  
  
alert (fruits[1]); //Undefined
```

More about Javascript Arrays

Defining the size of an array? No use

```
var fruits=new Array(3);
fruits[0]='orange';
fruits[1]='mandarine';
fruits[2]='apple';
fruits[3]='pear';
fruits[4]='strawberry';

alert (fruits[4]); //strawberry
```

Multidimensional arrays:

```
var board=new Array();
board[0]=new Array('Pepe', 'Juan', 'Belen');
board[1]=new Array('Tom', 'Mary', 'Ann');

for (i=0;i<board.length;i++) {
    for (j=0;j<board[i].length;j++) {
        document.write(board[i][j]+';');
    }
}
```

And be careful...

```
var fruits=new Array(3);
alert(fruits[0]); //undefined

var fruits=new Array('3');
alert(fruits[0]); //3
```

```
var fruits=new Array();
fruits[0] ='orange';
fruits[1] ='mandarine';
fruits[2] ='apple';

alert(fruits[2]); // apple
fruits.length=2;
alert(fruits[2]); // undefined
```

```
var points = new Array(40, 100); // Creates an array with two elements (40 and 100)
var points = new Array(40);      // Creates an array with 40 undefined elements !!!!!
```

Javascript Arrays

- Access the Elements of an Array:

```
var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars[0];
```

- Access the full array:

```
var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars;
```

- Array Elements Can Be Objects

- Because of this, you can have variables of different types in the same Array.

```
myArray[0] = Date.now;  
myArray[1] = myFunction;  
myArray[2] = myCars;
```

Javascript Arrays

- Adding array elements:
 - The easiest way to add a new element to an array is using the push method:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Lemon");           // adds a new element (Lemon) to fruits
```

- New element can also be added to an array using the length property:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Lemon"; // adds a new element (Lemon) to fruits
```


Javascript Arrays

- Looping array elements:

```
var fruits, text, fLen, i;

fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
    text += "<li>" + fruits[i] + "</li>";
}
```

Array.prototype methods: forEach()

The `forEach()` method executes a provided function once for each array element

(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

```
const array1 = ['a', 'b', 'c'];
array1.forEach(element => console.log(element));

const copyArray1 = [];
array1.forEach(item => copyArray1.push(item));
console.log("Copied array: "+copyArray1); //a,b,c

copyArray1.forEach((item, index) => {
  console.log(item);
  if (index === 2) {
    copyArray1.shift(); //Deletes de first element from the array
  });
});
console.log("Shifted array: "+copyArray1); //b,c
```

Array.prototype methods: Map()











The map() method creates a new array populated with the results of calling a provided function on every element in the calling
(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

```
const array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
console.log(map1); // [2, 8, 18, 32]












const array2 = ['1', '4', '9', '16'];
const map2 = array2.map(Number);
console.log(map2); // [1, 4, 9, 16]

const map3 = array1.map((num, index) => {
  if (index < 2) return num;
  else return 0;});
console.log(map3); // [1, 4, 0, 0]
```

Javascript Arrays: Methods

concat() 	Returns a new array comprised of this array joined with other array(s) and/or value(s).	join() 	Joins all elements of an array into a string.
every() 	Returns true if every element in this array satisfies the provided testing function.	lastIndexOf() 	Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
filter() 	Creates a new array with all of the elements of this array for which the provided filtering function returns true.	map() 	Creates a new array with the results of calling a provided function on every element in this array.
forEach() 	Calls a function for each element in the array.	pop() 	Removes the last element from an array and returns that element.
indexOf() 	Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.	push() 	Adds one or more elements to the end of an array and returns the new length of the array.

Javascript Arrays: Methods

reduce() 	Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.	toSource() 	Represents the source code of an object
reduceRight() 	Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.	sort() 	Sorts the elements of an array
reverse() 	Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.	splice() 	Adds and/or removes elements from an array.
shift() 	Removes the first element from an array and returns that element.	toString() 	Returns a string representing the array and its elements.
slice() 	Extracts a section of an array and returns a new array.	unshift() 	Adds one or more elements to the front of an array and returns the new length of the array.
some() 	Returns true if at least one element in this array satisfies the provided testing function.	Examples: https://www.w3schools.com/js/js_array_methods.asp	

Javascript Arrays

- Difference between arrays and objects:
 - In JavaScript, objects use named indexes.
 - Arrays are a special kind of objects, with numbered indexes.
 - If you use a named index, JavaScript will redefine the array to a standard object.
 - After that, all array methods and properties will produce incorrect results.

```
var person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
var x = person.length;  
var y = person[0];
```

```
var person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
var x = person.length;           // person.length will return 0  
var y = person[0];              // person[0] will return undefined
```

Javascript String objects: Properties & methods

Property	Description
length	Returns the length of the string.

Method	Description
charAt(position)	Returns the character at the specified position (in Number).
charCodeAt(position)	Returns a number indicating the Unicode value of the character at the given position (in Number).
concat([string,...])	Joins specified string literal values (specify multiple strings separated by comma) and returns a new string.
indexOf(SearchString, Position)	Returns the index of first occurrence of specified String starting from specified number index. Returns -1 if not found.

Javascript String objects: Methods

Method	Description
<code>lastIndexOf(SearchString, Position)</code>	Returns the last occurrence index of specified SearchString, starting from specified position. Returns -1 if not found.
<code>localeCompare(string,position)</code>	Compares two strings in the current locale.
<code>match(RegExp)</code>	Search a string for a match using specified regular expression. Returns a matching array.
<code>replace(searchValue, replaceValue)</code>	Search specified string value and replace with specified replace Value string and return new string. Regular expression can also be used as searchValue.
<code>search(RegExp)</code>	Search for a match based on specified regular expression.
<code>slice(startNumber, endNumber)</code>	Extracts a section of a string based on specified starting and ending index and returns a new string.
<code>split(separatorString, limitNumber)</code>	Splits a String into an array of strings by separating the string into substrings based on specified separator. Regular expression can also be used as separator.

Javascript String objects: Methods

Method	Description
<code>substr(start, length)</code>	Returns the characters in a string from specified starting position through the specified number of characters (length).
<code>substring(start, end)</code>	Returns the characters in a string between start and end indexes.
<code>toLocaleLowerCase()</code>	Converts a string to lower case according to current locale.
<code>toLocaleUpperCase()</code>	Converts a string to upper case according to current locale.
<code>toLowerCase()</code>	Returns lower case string value.
<code>toString()</code>	Returns the value of String object.
<code>toUpperCase()</code>	Returns upper case string value.
<code>valueOf()</code>	Returns the primitive value of the specified string object.

Examples: https://www.w3schools.com/js/js_string_methods.asp

Javascript Math objects: Properties

JavaScript provides 8 mathematical constants that can be accessed with the Math object:

```
Math.E          // returns Euler's number
Math.PI         // returns PI
Math.SQRT2      // returns the square root of 2
Math.SQRT1_2    // returns the square root of 1/2
Math.LN2        // returns the natural logarithm of 2
Math.LN10       // returns the natural logarithm of 10
Math.LOG2E      // returns base 2 logarithm of E
Math.LOG10E     // returns base 10 logarithm of E
```

Javascript Math objects: Methods

Method	Description
<code>abs(x)</code>	Returns the absolute value of x
<code>acos(x)</code>	Returns the arccosine of x, in radians
<code>asin(x)</code>	Returns the arcsine of x, in radians
<code>atan(x)</code>	Returns the arctangent of x as a numeric value between $-\pi/2$ and $\pi/2$ radians
<code>atan2(y, x)</code>	Returns the arctangent of the quotient of its arguments
<code>ceil(x)</code>	Returns the value of x rounded up to its nearest integer
<code>cos(x)</code>	Returns the cosine of x (x is in radians)
<code>exp(x)</code>	Returns the value of E^x
<code>floor(x)</code>	Returns the value of x rounded down to its nearest integer

Javascript Math objects: Methods

Method	Description
log(x)	Returns the natural logarithm (base E) of x
max(x, y, z, ..., n)	Returns the number with the highest value
min(x, y, z, ..., n)	Returns the number with the lowest value
pow(x, y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Returns the value of x rounded to its nearest integer
sin(x)	Returns the sine of x (x is in radians)
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of an angle

Javascript Date objects

- A date consists of a year, a month, a day, an hour, a minute, a second, and milliseconds.
- Date objects are created with the **new Date()** constructor.
- There are **4 ways** of initiating a date:

```
new Date() //Creates a new date object with the current date and time
new Date(dateString) //Creates a new date object from the specified date and time
new Date(milliseconds) //Creates a new date object as zero time plus the number
                        //Zero time is 01 January 1970 00:00:00 UTC
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

- Examples:

```
var d1 = new Date(); //Sun Apr 02 2017 10:46:03 GMT+0200
var d2 = new Date("October 13, 2014 11:13:00"); //Mon Oct 13 2014 11:13:00 GMT+0200
var d3 = new Date(86400000); //Fri Jan 02 1970 01:00:00 GMT+0100
var d4 = new Date(99, 5, 24, 11, 33, 30, 0); //Thu Jun 24 1999 11:33:30 GMT+0200
```

Javascript Date objects: Format

- There are generally 4 types of JavaScript date input formats:

Type	Example
ISO Date	"2015-03-25" (The International Standard)
Short Date	"03/25/2015"
Long Date	"Mar 25 2015" or "25 Mar 2015"
Full Date	"Wednesday March 25 2015"

- Some other options:

```
var d1 = new Date("2015-03"); //Sun Mar 01 2015 01:00:00 GMT+0100
var d2 = new Date("2015"); //Thu Jan 01 2015 01:00:00 GMT+0100
var d3 = new Date("2015-03-25T12:00:00Z"); //Wed Mar 25 2015 13:00:00 GMT+0100
var d4 = new Date("JANUARY, 25, 2015"); //Sun Jan 25 2015 00:00:00 GMT+0100
var d = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (W. Europe Standard Time)"
//Wed Mar 25 2015 09:56:24 GMT+0100
```

Javascript Date objects: Methods

Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday as a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)
<code>getMilliseconds()</code>	Get the milliseconds (0-999)
<code>getMinutes()</code>	Get the minutes (0-59)
<code>getMonth()</code>	Get the month (0-11)
<code>getSeconds()</code>	Get the seconds (0-59)
<code>getTime()</code>	Get the time (milliseconds since January 1, 1970)

Javascript Date objects: Methods

Method	Description
<code>setDate()</code>	Set the day as a number (1-31)
<code>setFullYear()</code>	Set the year (optionally month and day)
<code>setHours()</code>	Set the hour (0-23)
<code>setMilliseconds()</code>	Set the milliseconds (0-999)
<code>setMinutes()</code>	Set the minutes (0-59)
<code>setMonth()</code>	Set the month (0-11)
<code>setSeconds()</code>	Set the seconds (0-59)
<code>setTime()</code>	Set the time (milliseconds since January 1, 1970)

Examples: https://www.w3schools.com/js/js_date_methods.asp

Javascript Functions

- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when "something" invokes it (calls it).
- Purpose: Define the code once, and use it many times.
- Definition:

```
function name(parameter1, parameter2, parameter3) {  
    code to be executed  
}
```

- Function **parameters** are the **names** listed in the function definition.
- Function **arguments** are the real **values** received by the function when it is invoked.

```
var x = myFunction(4, 3);
```

```
function myFunction(a, b) {  
    return a * b;  
}
```

- Function declarations and function variables are always moved ('hoisted') to the top of their JavaScript scope by the JavaScript interpreter

Javascript Functions

- Some other ways of declaring a function:
 - Function operator: It is not hoisted to the top of their JavaScript scope by the JavaScript interpreter

```
//anonymous function expression
var a = function() { return 3};
alert(a); //function() { return 3}
alert(a()); //3
```

```
//named function expression
var a = function bar() {return 3;};
alert(a); //function bar() {return 3}
alert(a()); //3
alert(a.bar()); //Error
```

```
//self invoking function expression
(function a()
    {alert(3);})(); //3
alert(a()); //Error
```

- New operator (Not recommended):

```
var myFunction = new Function("a", "b", "return a * b");
document.getElementById("demo").innerHTML = myFunction(4, 3);
```

Javascript Functions

- Dynamic declaration of a Function:
 - Using the window object from BOM:

```
001  var nombreFuncion = 'cuadrado';
002  var argumentoFuncion = 'x';
003  var codigoFuncion = 'return x * x;';
004  window[nombreFuncion] = new Function (argumentoFuncion, codigoFuncion);
005  alert(window[nombreFuncion](3));
```

Javascript Functions

- Invoking a function: Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Instead of using a variable to store the return value of a function:

```
var x = toCelsius(77);  
var text = "The temperature is " + x + " Celsius";
```

You can use the function directly, as a variable value:

```
var text = "The temperature is " + toCelsius(77) + " Celsius";
```

- Functions are objects: They have both **properties** and **methods**.

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

```
function myFunction(a, b) {  
    return a * b;  
}
```

```
var txt = myFunction.toString();
```

Javascript Functions

- If a function is called with **missing arguments** (less than declared), the missing values are set to: **undefined**

```
function myFunction(x, y) {  
    if (y === undefined) {  
        y = 0;  
    }  
    return x * y;  
}  
document.getElementById("demo").innerHTML = myFunction(4);
```

- A function can be also declared with less arguments that it is going to be invoked:

```
function suma() {  
    var i;  
    var resultado = 0;  
    for (i=0;i<arguments.length;i++) {  
        resultado += arguments[i];  
    }  
    return resultado;  
}  
alert(suma(3,5,7,9));
```

Javascript Functions

- Arguments are passed by value, objects are passed by reference
 - Changes to arguments are not visible (reflected) outside the function.
 - Changes to object properties are visible (reflected) outside the function.

```
var cantidades = new Array(3,5,7,9);
function suma(sumandos) {
    var i;
    var resultado = 0;
    while(sumandos.length>0) {
        resultado += sumandos.shift();
    }
    return resultado;
}
alert(suma (cantidades));
alert(cantidades.length);
```

Javascript Functions

- Invoking a function with a Function constructor:
 - If a function invocation is preceded with the **new** keyword, it is a constructor invocation.
 - It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

```
function myFunction(arg1, arg2) {  
    this.firstName = arg1;  
    this.lastName  = arg2;  
}  
  
var x = new myFunction("John", "Doe")  
document.getElementById("demo").innerHTML = x.firstName;
```

Javascript Functions

- Invoking a Function with a Function Method:
 - **call()** and **apply()** are predefined JavaScript function methods.
 - The only difference is that call() takes the function arguments separately, and apply() takes the function arguments in an array.

```
function myFunction(a, b) {  
    return a * b;  
}  
myObject = myFunction.call(myObject, 10, 2);    // Will return 20  
  
function myFunction(a, b) {  
    return a * b;  
}  
myArray = [10, 2];  
myObject = myFunction.apply(myObject, myArray); // Will also return 20
```


Javascript Functions

- Invoking a function as an argument of other function:

```
function sumar(a,b){  
    return a+b;  
}  
function restar(a,b){  
    return a-b;  
}  
function mostrar(operacion,operando1,operando2){  
    return(operacion(operando1,operando2));  
}  
alert(mostrar(restar,4,5));  
alert(cantidades.length);
```

Javascript Functions

- Function recursivity:

```
001 function factorial(n){  
002     if(n==1){  
003         return 1;  
004     }else{  
005         return n * factorial(n-1);  
006     }  
007     alert(factorial(5));
```

```
function greaterThan(numbers, position)  
{  
    if (position < 0) return aux;  
    else {  
        if (numbers [position] > aux)  
            { aux=numbers[position];}  
        return greaterThan (numbers, position-1);  
    }  
}  
aux = -Infinity;  
vector= new Array (8,2,3,-4,5,6,5,4,3);  
alert(greaterThan(vector, vector.length-1));
```

- Be careful with the capacity of browser memory resources

Javascript Function closures

- Global variables can be made local (private) with **closures**.
- A closure is an inner function that has access to the outer (enclosing) function's variables:

Example:

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

Problem:

The counter should only be changed by the add() function.

The problem is, that any script on the page can change the counter, without calling add().

```
var counter = 0;
```

```
function add() {  
    counter += 1;  
}
```

```
add();  
add();  
add();
```

```
// the counter is now equal to 3
```

Javascript Function closures

Example (Cont.):

We could declare the counter inside the function, so nobody will be able to change it without calling add():

Problem:

Every time I call the add() function, the counter is set to 1.

```
function add() {  
    var counter = 0;  
    counter += 1;  
}
```

```
add();  
add();  
add();
```

```
// the counter should now be 3, but it does not work !
```

Javascript Function closures

Example (Cont.):

We could use nested functions: Nested functions have access to the scope "above" them.

In this example, the inner function `plus()` has access to the `counter` variable in the parent function:

Problem:

It is not possible to call the `plus()` function from the outside.

We also need to find a way to execute `counter = 0` only once.

```
function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();  
    return counter;  
}
```

Javascript closures

Example (Cont.):

Solution: Javascript closures

- *The variable add is assigned the return value of a self-invoking function.*
- *The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.*
- *This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.*
- *This is called a JavaScript closure. It makes it possible for a function to have "private" variables.*
- *The counter is protected by the scope of the anonymous function, and can only be changed using the add function.*

```
var add = (function () {  
    var counter = 0;  
    return function () {return counter += 1;}  
})();  
  
add();  
add();  
add();  
  
// the counter is now 3
```