# JavaScript

## What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive.

JavaScript contains a standard library of objects (Array, Date, Math etc.) and a core set of language elements (operators, control structures, statements). Core JavaScript can be extended for a variety of purposes.

- **Client-side JavaScript** extends the core language by supplying objects to control a browser and it's Document Object Model (DOM).

  For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.

- **Server-side JavaScript** extends the core language by supplying objects relevant to running JavaScript on a server.

  For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

JavaScript borrows most of its syntax from *Java, C, and C++*, but it has also been influenced by *Awk, Perl, and Python*. JavaScript is **case-sensitive** and uses the **Unicode character set**.

## JavaScript Integration >>

### 1. Internal Script

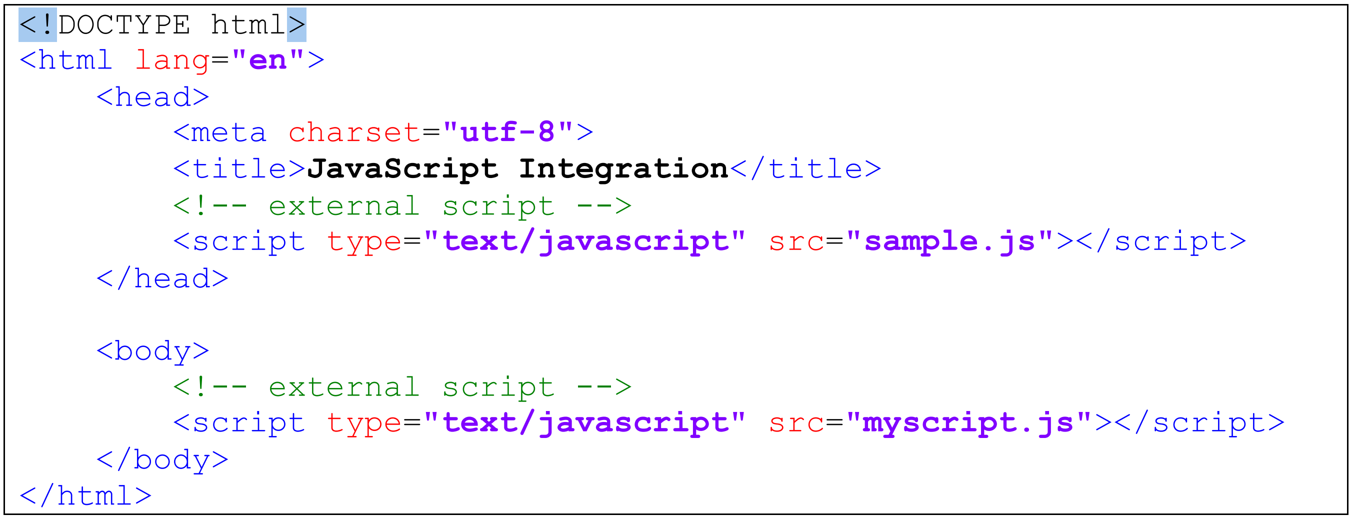
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>JavaScript Integration</title>

        <!-- internal embedded script -->
        <script type="text/javascript">
                console.log("within head section");
        </script>
    </head>

    <body>
        <!-- internal embedded script -->
        <script type="text/javascript">
            console.log("within body section");
        </script>
    </body>

</html>
```

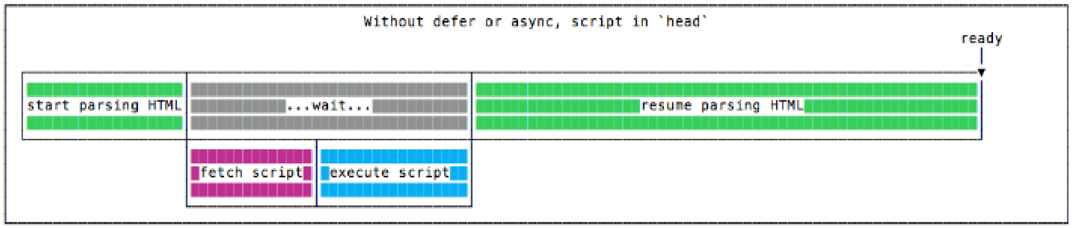
## 2. External Script



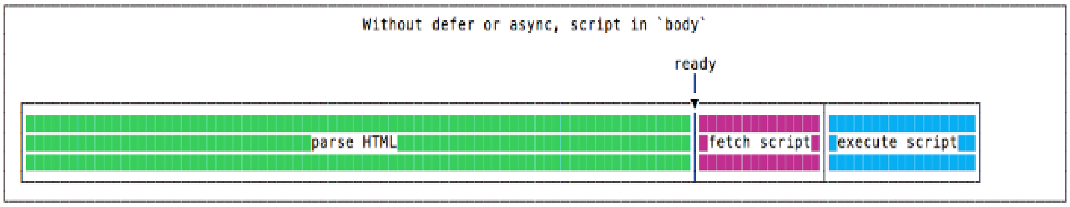
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>JavaScript Integration</title>
        <!-- external script -->
        <script type="text/javascript" src="sample.js"></script>
    </head>

    <body>
        <!-- external script -->
        <script type="text/javascript" src="myscript.js"></script>
    </body>
</html>
```

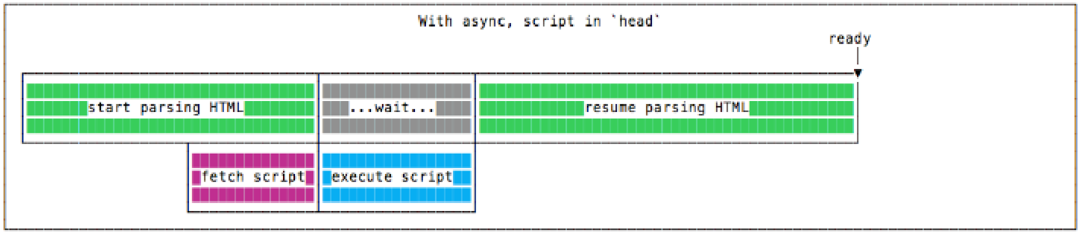
- No *defer* or *async* (<head> section)



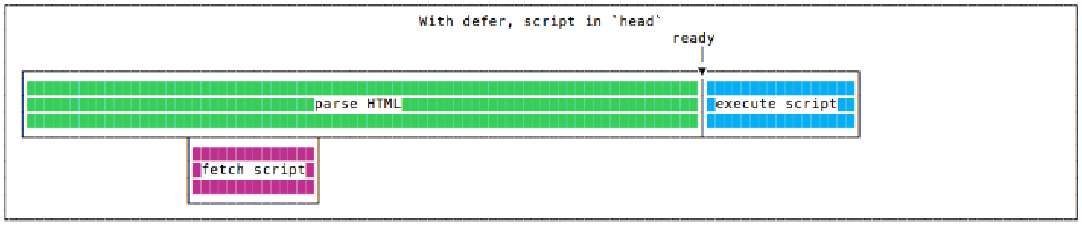
- No *defer* or *async* (at the end of <body> section, good practice)



- With *async* (<head> section)



- With *defer* (<head> section)



## Comments >>

```
<script type="text/javascript">
    // a one line comment

    /* this is a longer,
     * multi-line comment
     */

    /* You can't, however, /* nest comments */ SyntaxError */
</script>
```

## Variable declarations >>

- var x = 42 – declares both *function-scoped and globally-scoped* variable depending on the execution context, optionally initializing it to a value. [ Default value is *undefined* ]

  Duplicate variable declarations using *var* will not trigger an error, even in strict mode, and the variable will not lose its value, unless another assignment is performed. *var* declarations gets hoisted.

- let v=42 – declares a *block-scoped* local variable, optionally initializing it to a value.

  Variables declared by *let* have their scope in the block for which they are defined, as well as in any contained sub-blocks. Duplicate variable declarations using *let* raises a *SyntaxError*. *let* declarations does not get hoisted.

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

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

- const PI=3.14 – declares a *block-scoped read-only* named constant. You must specify its value in the same statement in which it is declared.

## Variable Hoisting >>

- In JavaScript, you can refer to a variable declared (using *var*) later, without getting an exception. This concept is known as **hoisting**; variables in JavaScript are in a sense *"hoisted" or lifted to the top* of the function or statement. However, variables that are hoisted return a value of *undefined*

```
function do_something() {
  console.log(bar); // undefined
  var bar = 111;
  console.log(bar); // 111
}
```

```
function do_something() {
  var bar;
  console.log(bar); // undefined
  bar = 111;
  console.log(bar); // 111
}
```

- Declaring variables using *let* and *const* doesn’t hoist the variable to the top of the block.

```
function not_hoisted() {
  console.log(foo); // ReferenceError
  let foo = 111;
  console.log(foo); // 111
}
```

## Control flow and Loop statements >>

1. **if … else** statement:

```
var book = "maths";
if( book == "history" ){
   document.write("<b>History Book</b>");
}
else if( book == "maths" ){
   document.write("<b>Maths Book</b>");
}
else if( book == "economics" ){
   document.write("<b>Economics Book</b>");
}
else{
   document.write("<b>Unknown Book</b>");
}
```

**Falsy values:** false, undefined, null, 0, NaN, “” (empty string)

2. **switch** statement:

```
var grade = 'A';
document.write("Entering switch block<br />");

switch (grade)
{
   case 'A':
        document.write("Good job<br />");
        break;
   case 'F':
        document.write("Failed<br />");
        break;
   default:
        document.write("Unknown grade<br />");
}

document.write("Exiting switch block");
```

### 3. throw statement:

```
throw 'Error2';    // String type
throw 42;          // Number type
throw true;        // Boolean type
throw {toString: function() { return "I'm an object!"; } };
```

### 4. try … catch statement:

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch(e) {
    console.log(1);
    return true;    // this return statement is suspended
                    // until finally block has completed
    console.log(2); // not reachable
  } finally {
    console.log(3);
    return false;   // overwrites the previous "return"
    console.log(4); // not reachable
  }
  // "return false" is executed now
  console.log(5);   // not reachable
}
console.log(f()); // 0, 1, 3, false
```

### 5. for statement:

```
var count;
document.write("Starting Loop" + "<br/>");

for(count = 0; count < 10; count++){
   document.write("Current Count : " + count );
   document.write("<br />");
}

document.write("Loop stopped!");
```

### 6. do … while statement:

```
var i = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

### 7. while statement:

```
var n = 0;
var x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

### 8. break statement:

```
var x = 1;
document.write("Entering the loop<br/>");

while (x < 20)
{
    if (x == 5){
       break; // breaks out of loop completely
    }
    x = x + 1;
    document.write( x + "<br />");
}
document.write("Exiting the loop!<br /> ");
```

### 9. continue statement:

```
var i = 0;
var n = 0;
while (i < 5) {
  i++;
  if (i == 3) {
    continue;
  }
  n += i;
  console.log(n);
}
//1,3,7,12
```

### 10. for … in statement:

```
var arr=['item0','item1','item2'];

for(let ind in arr){
    console.log(ind+" => "+arr[ind]); ///iterates over property names
}
```

## Functions >>

### 1. Function Declaration / Statement

```
function fnname(param1, param2, param3, ... ){
    // processing
    return return_value;
}
```

- **primitive** parameters(**ex.** numbers) are passed to functions by **value**.
- **non-primitive** parameters (**ex.** array, objects) are passed by **reference**.

### 2. Function Expressions

This type of function can be anonymous and can’t be hoisted. This type is convenient when passing a function as an argument to another function.

```
function multiplier_factory(multiplier){
    var fnexpr = function (value){
        return multiplier*value;
    };
    return fnexpr;
}

var _3multiplier = multiplier_factory(3);
console.log(_3multiplier);
console.log(_3multiplier(8));
```

### 3. Immediately Invokable Function Expression (IIFE) – runs as soon as it is defined

```
var param=100;
(function (p1){
    var privatevar='abcd';
    console.log(p1); // 100
    console.log(privatevar); // abcd
})(param);

console.log(privatevar); //ReferenceError-can't access this variable
```

### 4. Arrow function

```
var a = ['Hydrogen','Helium','Lithium','Beryllium'];

var a2 = a.map(function(s) { return s.length; });
console.log(a2); // logs [8, 6, 7, 9]

var a3 = a.map(s => s.length);
console.log(a3); // logs [8, 6, 7, 9]
```

- **Predefined Functions**

| |
|---|
| eval()<br>- evaluates JS code represented as a string |
| encodeURI()<br>- encodes a URI by replacing each instance of certain characters by 1, 2, 3 or 4 escape sequences.<br><br>decodeURI()<br>- decodes a URI previously created by encodeURI() |

## Function Hoisting >>

For functions, only the **function declaration** gets hoisted to the top and not the **function expression**.

```
// function declaration gets hoisted
foo(); // "bar"
function foo() {
    console.log('bar');
}
```

```
// function expression won't be hoisted
baz(); // TypeError: baz is not a function

var baz = function() {
    console.log('bar2');
};
```

## Function Scope >>

- Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- A function can access all variables and functions defined inside the scope in which it is **defined**.

```
// The following variables are defined in the global scope
var num1 = 20, num2 = 3, name = 'Chamahk';

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}
multiply(); // Returns 60

// A nested function example
function getScore() {
  var num1 = 2, num2 = 3;

  function add() {
    return name + ' scored ' + (num1 + num2);
  }

  return add();
}
console.log(getScore()); // Returns "Chamahk scored 5"
```

## Nested Function >>

- You can nest a function within another function. The nested (inner) function is private to its containing (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.

  - A closure is an expression that can have free variables together with an environment that binds those variables. We can say that inner functions contains the scope of the outer function.

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // logs 6 (1 + 2 + 3)
```

here,
- B forms a closure including A (i.e. B can access A's arguments and variables)
- C forms a closure including B.
- Because B's closure includes A, C's closure includes A, C can access both B and A's arguments and variables.

## argument Object >>

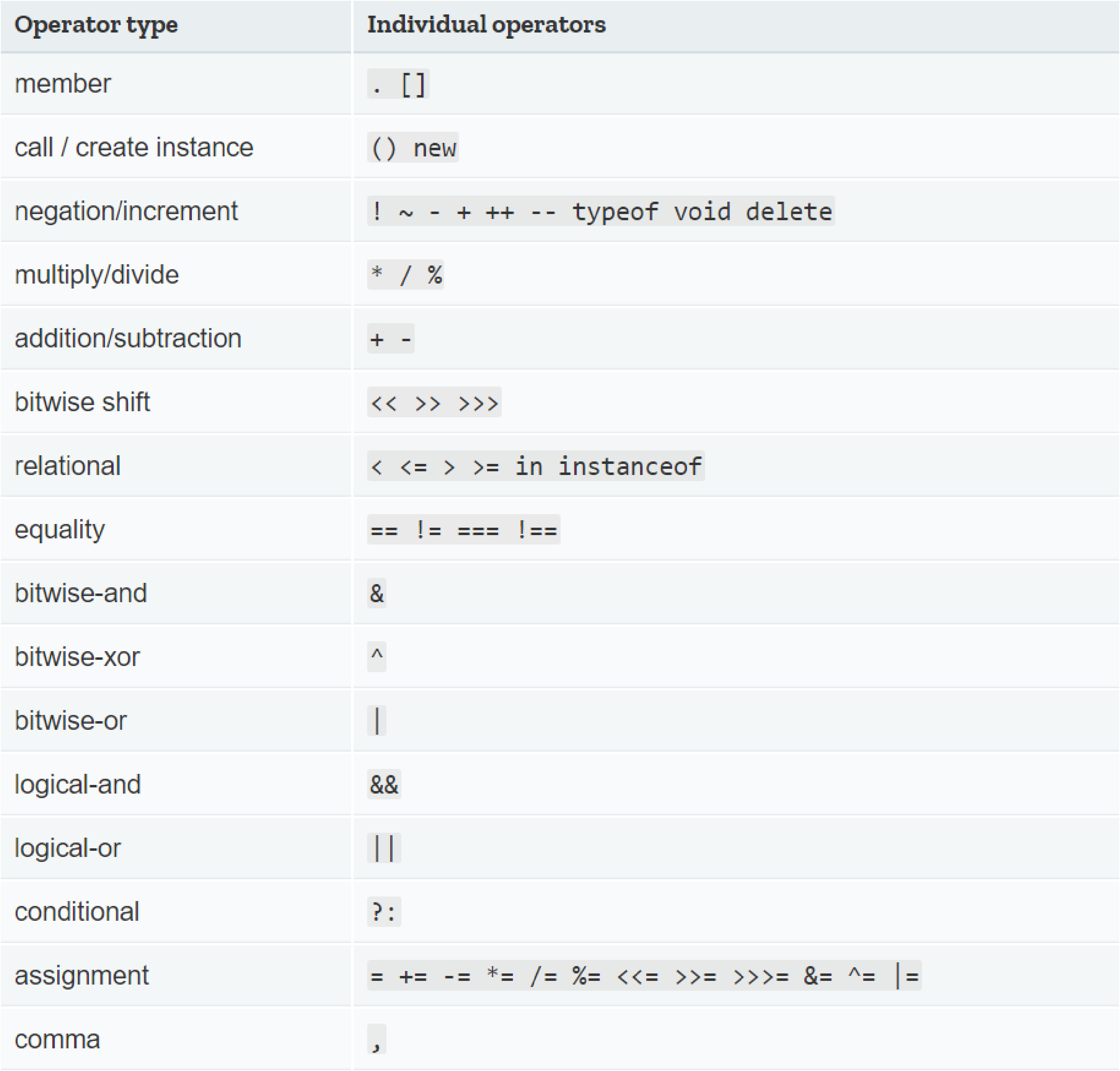
- The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it by accessing an array named *arguments*

```
function myConcat(separator) {
   var result = ''; // initialize list
   var i;
   // iterate through arguments
   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}
// returns "red, orange, blue, "
console.log(myConcat(', ', 'red', 'orange', 'blue'));
```

## Default Parameters >>

```
function multiply(a, b = 1) {
  return a * b;
}
console.log(multiply(5)); // 5
```

## Operators >>



| Operator type | Individual operators |
|---|---|
| member | `. []` |
| call / create instance | `() new` |
| negation/increment | `! ~ - + ++ -- typeof void delete` |
| multiply/divide | `* / %` |
| addition/subtraction | `+ -` |
| bitwise shift | `<< >> >>>` |
| relational | `< <= > >= in instanceof` |
| equality | `== != === !==` |
| bitwise-and | `&` |
| bitwise-xor | `^` |
| bitwise-or | `\|` |
| logical-and | `&&` |
| logical-or | `\|\|` |
| conditional | `?:` |
| assignment | `= += -= *= /= %= <<= >>= >>>= &= ^= \|=` |
| comma | `,` |

## Date >>

```
var today = new Date();
var birthday = new Date('December 17, 1995 03:24:00');
var birthday = new Date('1995-12-17T03:24:00');
var birthday = new Date(1995,11,17);
var birthday = new Date(1995,11,17,3,24,0);
```

**Local time:**

getSeconds(), getMinutes(), getHours(), getDate(), getMonth(), getFullYear()

**UTC:**

getUTCSeconds(),getUTCMinutes(), getUTCHours(), getUTCDate(), getUTCMonth(), getUTCFullYear()

## Number >>

### literals

dec: 42, 0888(strict)
oct: 0777(non-strict), 0o777
bin: 0b0101, 0B1101
hex: 0x12A, 0XAF9
exp: 1E3, 2e6

```
let int = 42;
let oct = 077;
let hex = 0xFF;
let bin = 0b1011;
let exp = 2e4;

console.log(int+" "+oct+" "+hex+" "+bin+" "+exp);
// output: 42  63  255  11  20000
```

### Number Object

constructor:
const a = new Number(‘123’);

properties:
MAX_VALUE
MIN_VALUE
NaN
NEGATIVE_INFINITY
POSITIVE_INFINITY

methods:
parseFloat(str)
parseInt(str)
isInteger()
isNaN()

```
const mystr = '123.456';
let mynum = undefined/10;
console.log(Number.parseFloat(mystr));
console.log(Number.parseInt(mystr));
console.log(Number.isNaN(mynum));
```

### Math Object

properties:
PI
E

methods:
random()
abs(x)
ceil(x), floor(x), round(x), trunc(x)
sqrt(x), pow(x, y), exp(x)
log(x), log10(x), log2(x)
sin(x), cos(x), tan(x)
max(x, y, z, … ), min(x, y, z, … )

```
// degree to radian conversion
function degToRad(deg){
    return deg*(Math.PI/180);
}
console.log(degToRad(30));

//generating random numbers with range [min, max]
function randomGenerator(min, max){
    return Math.floor(Math.random()*(max-min+1))
+min;
}
console.log(randomGenerator(3,8));
```

## String >>

JavaScript's *String* type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values (UTF-16 code units). Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on.

### 1. Creating Strings

```
// creating strings
const str1 = 'string literal';
const str2 = "string literal";
const str3 = new String('string object');
// JavaScript automatically converts the string literal to a
temporary String object, calls the method, then discards the
temporary String object.

console.log(str1); // literal
console.log(str2); // literal
console.log(str3); // object
```

### 2. String Object property – length

```
// String Object property - length
const hello = 'Hello, World!';
const helloLength = hello.length; // length: 13
hello[0] = 'L'; // This has no effect, because strings are immutable
console.log(hello[0]); // This returns "H"
```

### 3. String Object methods

| Method Name | Return Value | Sample Code |
|---|---|---|
| toLowerCase()<br>toUpperCase() | - lowercase string<br>- uppercase string | `const mystr1 = 'AbCd';`<br>`console.log(mystr1.toLowerCase());`<br>`// abcd`<br>`console.log(mystr1.toUpperCase());`<br>`// ABCD` |
| indexOf(substr [,fromIndex])<br>lastIndexOf(substr [,fromIndex]) | - index of the first occurrence or -1 if not found<br>- index of the first occurrence from backward or -1 if not found | `// indexOf(substr[, fromIndex]) and lastIndexOf(substr[, fromIndex])`<br>`const mystr2 = 'abcdxyzabcd';`<br>`console.log(mystr2.indexOf('abcd'));`<br>`// 0`<br>`console.log(mystr2.lastIndexOf('abcd'));`<br>`// 7` |
| split(separator) | - array of substrings | `// split(separator)`<br>`const str = 'The quick brown fox jumps over the lazy dog.';`<br>`let splitarr = str.split(' ');`<br>`console.log(splitarr);` |

| | | |
|---|---|---|
| slice(startInd [,uptoInd])<br>substring(startIn [,uptoInd])<br>substr(startInd [,length]) | - new string<br>- new string<br>- new string | // slice(startIndex[, uptoIndex]) , substring(startIndex[, uptoIndex]) and substr(startIndex[, length])<br>const mystr3 = 'abcdefghij';<br>console.log(mystr3.slice(1,3)); // bc<br>console.log(mystr3.substring(1,3)); // bc<br>console.log(mystr3.substr(1,2)); // bc |
| + operator<br>concat(str2, str3, … ) | - concatenate two or more strings | const mystr4 = "Hello";<br>const mystr5 ="world";<br>console.log(mystr4+" "+mystr5);<br>console.log(mystr4.concat(" ", mystr5)); |
| Other Methods:<br>▪ charAt(), charCodeAt()<br>▪ startsWith(), endsWith(), includes()<br>▪ trim() | | |

## 4. Embedded expressions:

```
const five = 5;
const ten = 10;
console.log('Fifteen is ' + (five + ten) + ' and not ' + (2 * five +
ten) + '.');
// "Fifteen is 15 and not 20."

console.log(`Fifteen is ${five + ten} and not ${2 * five + ten}.`);
// use backtick template literal
// "Fifteen is 15 and not 20."
```

## Regular Expression >>

| | |
|---|---|
| Declarations | /* format: /patter/flag */<br>/* flag: i = case-insensitive, g = global etc. */<br>var re1 = /^[A-Za-z][A-Za-z_#-]*$/;<br>var re2 = /^[A-Z]{2,5}$/i;<br>// using RegExp class<br>var re3 = new RegExp('[A-Za-z]{1,5}');<br>var re4 = new RegExp('[A-Za-z]{1,5}','i'); |
| Regular Expression Syntax | https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions |
| RegExp Object methods:<br>exec(), test() | var re2 = /^[A-Z]{2,5}$/i;<br>var str = "abcd";<br>if(re2.test(str)){<br>    console.log("match found");<br>} else{<br>    console.log("no match found");<br>} |
| String Object methods | match(), matchAll(), search(), replace(), split() |

## Array >>

- An array is an ordered list of values that you refer to with **a name and an index**. Example, *emp[0]*
- JavaScript does not have an explicit array data type. However, you can use the predefined **Array object** and its methods to work with arrays in your applications.

### 1. Creating an Array

```
// declarations
let arr = new Array('item0','item1','item2');
let arr1 = Array('item0','item1','item2');
let arr2 = ['item0','item1','item2'];

// an array with non-zero length, but without any items
let len=10;
let arr3=new Array(len);
let arr4=Array(len);
let arr5=[];
arr5.length=len;
```

### 2. Accessing Array elements

```
// accessing array elements
let myarray = ['item0', 'item1', 'item2'];
console.log(arr[0]);
console.log(arr[1]);
console.log(arr[2]);
```

### 3. Populating an Array

```
// populating an array
let myarray1 = []
myarray1[0] = 'one';
myarray1[1] = 'two';
myarray1[2] = 'three';
```

### 4. Array Object property - length

```
// array object property - length
let myarray2 = [];
myarray2[30] = 'thirty';
console.log(myarray2.length); // 31
```

## 5. Iterating over Arrays:

```
// Iterating over Arrays

let myarray3 = ['item0', 'item1', 'item2', 'item3'];
myarray3[10] = 'item10';

// way 1
for(let ind=0;ind<myarray3.length;ind++){
    console.log(ind+" => "+myarray3[ind]);
}

// way 2 - it is not advisable to iterate through JavaScript arrays
using for...in loops, because normal elements and all enumerable
properties will be listed.
for(let ind in myarray3){
    console.log(ind+" => "+myarray3[ind]);
}
```

## 6. Array Object methods

| Method Name | Return Value | Sample Code |
|---|---|---|
| concat(e1, e2, … ) | new array | `let myArray = new Array('1', '2', '3');`<br>`myArray = myArray.concat('a', 'b', 'c');`<br>`console.log(myArray);`<br>`// myArray is now ["1", "2", "3", "a", "b", "c"]` |
| join(delimeter = ’,’) | string | `let myArray1 = new Array('Wind', 'Rain', 'Fire');`<br>`let list = myArray1.join(' - ');`<br>`console.log(list);`<br>`// list is "Wind - Rain - Fire"` |
| push(e1, e2, … )<br><br>pop() | - length of the array<br><br>- last element that is popped | `let myArray2 = new Array('1', '2');`<br>`myArray2.push('3', '4');`<br>`console.log(myArray2);`<br>`// myArray2 is now ["1", "2", "3", "4"]`<br><br>`let last = myArray2.pop();`<br>`console.log(myArray2);`<br>`console.log(last);`<br>`// myArray2 is now ["1","2","3"],last="4"` |
| unshift(e1, e2, … )<br><br>shift() | - length of the array<br><br>- first element that is removed | `let myArray3 = new Array('1', '2', '3');`<br>`myArray3.unshift('4', '5');`<br>`console.log(myArray3);`<br>`// myArray3 becomes ["4","5","1","2","3"]`<br><br>`let first = myArray3.shift();`<br>`console.log(myArray3);`<br>`console.log(first);`<br>`// myArray3 is now ["5","1","2","3"], first is "4"` |

| | | |
|---|---|---|
| slice(start_ind, upto_ind)<br>splice(ind, count_to_remove, addelm1, addelm2, … …) | - new subarray<br>- returns the removed items array | // slice(start_index, upto_index)<br>let myArray4 = new Array('a','b','c','d','e');<br>newArray = myArray4.slice(1, 4);<br>console.log(newArray);<br>// starts at index 1 and extracts all elements until index 3, returning [ "b", "c", "d"]<br><br>// splice(index, count_to_remove, addelm1, addelm2, ...)<br>let myArray5 = new Array('1','2','3','4','5');<br>remArray = myArray5.splice(1,3,'a','b','c','d');<br>console.log(remArray);<br>console.log(myArray5);<br>// remArray is now ["2", "3", "4"]<br>// myArray5 is now ["1","a","b","c","d","5"] |
| reverse()<br>sort() / sort(sortFn) | - reference to the array<br>- reference to the array | let myArray6 = new Array('1', '2', '3');<br>myArray6.reverse();<br>console.log(myArray6);<br>// transposes the array so that myArray6 = ["3", "2", "1"]<br><br>let myArray7 = new Array('Wind', 'Rain', 'Fire');<br>myArray7.sort();<br>console.log(myArray7);<br>// sorts the array so that myArray7 = ["Fire", "Rain", "Wind"] |
| indexOf(searchelm [, fromIndex])<br>lastIndexOf( searchelm [,fromIndex]) | - index of the first match<br>- index of the first match from backward | let myArray9 = ['a', 'b', 'a', 'b', 'a'];<br>console.log(myArray9.indexOf('b'));<br>// output: 1<br><br>console.log(myArray9.lastIndexOf('b'));<br>// output: 3 |

## 7. Manipulating Array elements

| | |
|---|---|
| forEach(callbackfn)<br>- executes *callbackfn* on every array items and console.log the output | let a = [1, 2, 3]<br>a.forEach(function(element) { console.log(element+10); })<br>// logs each item in turn 11, 12, 13 |
| map(callbackfn)<br>- executes *callbackfn* on every array items and returns a new array containing the return values | // map(callbackfn)<br>let a1 = ['a', 'b', 'c']<br>let a2 = a1.map(function(item) { return item.toUpperCase(); })<br>console.log(a2) // logs ['A', 'B', 'C'] |

Other Methods:

- filter(callbackfn) – returns a new array containing items for which callbackfn returnd true.
- every(callbackfn) – returns true if callbackfn returns true for all elements of the array.
- some(callbackfn) – returns true of callbackfn returns true for at least one element of the array.
- reduce(callbackfn) – returns the final value returned by callbackfn.

## Objects >>

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method.

### 1. Creating new Object

- Method 1 – Object Initializer

```
var obj = { property_1:   value_1, //property_# may be an identifier
            2:            value_2, // or a number...
            // ...,
            'property n': value_n  // or a string
          };
```

Example :

```
var obj_literal={
    key1: 'named key',
    1: 'integer key',
    "key2": 'string key'
};

for(let prop in obj_literal){
    console.log(obj_literal[prop]);
}
```

- Method 2 – Object Declaration

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;

for(let prop in myCar){
    console.log(myCar[prop]);
}
```

- Method 3 – Function Constructor

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
var mycar = new Car('Eagle', 'Talon TSi', 1993);

for(let prop in mycar){
    console.log(mycar[prop]);
}
```

## 2. Accessing Object properties:

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

```
console.log(myCar.model);

var k='make';
console.log(myCar[k]);

console.log(myCar['year']);
```

## 3. Defining methods

```
var obj1 = {
    key1: function (p1=1, p2=1){
        return p1*p2;
    },
    key2: 100
};

console.log(obj1.key1(10,5));
// ------------------------------
var obj2 = new Object();
obj2.key1 = function (p1=1, p2=1){
    return p1*p2;
};
obj2.key2 = 100;

console.log(obj2.key1(10,5));
// ------------------------------
function ObjectFn(k1, k2){
    this.key1 = k1;
    this.key2 = k2;
}
function method(p1=1, p2=1){
    return p1*p2;
}
var obj3 = new ObjectFn(method, 100);
console.log(obj3.key1(10, 5));
```

## JavaScript Promises >>

A *Promise* is an object representing the eventual completion or failure of an asynchronous operation.

```
// problem
function f1(){
  var x = "before timeout";
  console.log(x);
  // this asynchronous function will break the
serial
  setTimeout(
    function (){
      x = "after 2 seconds";
      console.log(x);
    }
    ,
    2000
  );

}
function f2(){
  console.log("a line of function f2");
}
f1();
f2();
```

**Output:**

```
line in f1 before timeout
a line of function f2
after 2 seconds
```

### Solution:

```
// promise
function f1(){
    return new Promise(
        function(resolve, reject){
   //asynchronous operation(function body) here
            var x = "before timeout";
            console.log(x);

            setTimeout(
              function (){
                x = "after 2 seconds";
                console.log(x);

                resolve();
              }
              ,
              2000
        );
    });
}
function f2(){
  console.log("a line of function f2");
}
var prom=f1();
prom.then(f2);
```

**Output:**

```
line in f1 before timeout
after 2 seconds
a line of function f2
```

## References:

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
2. https://www.w3schools.com/js/