JavaScript
functions, conditions & loops

Examples

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!
```

```html
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <script>
5              function sayHello() {
6              alert("Hello there!");
7              }
8          </script>
9
0      </head>
1
2  <body>
3      <p>Click the button and call the function</p>
4      <button onclick="sayHello()">Say Hello</button>
5
6  </body>
7  </html>
```

Return

- Functions often compute a return value ->return statement
- The **return** statement **ends** function execution **and specifies a value to be returned to the function caller.**

- What will alert show here?

```
function myFunction(p1, p2) {

  return(p1+p2);
  alert(p1+p2);
}

document.getElementById("demo").innerHTML = myFunction
    (4, 3);
```

Return

- return statement: stops a function from executing

```
<script>
function myFunction(p1, p2) {

    return(p1+p2);
    alert(p1+p2);
}

document.getElementById("demo").innerHTML = myFunction
    (4, 3);
```

Functions & Variables

- A variable declared inside a function -> only visible there!

- An outer variable -> can be accessed by a function!

- An outer variable -> can be modified by a function

- The outer variable is only used if there's no local one.

- If a same-named variable is **declared** inside the function -> it shadows the outer one, which is ignored.

Self-Invoking Functions

- **Function expressions** can be made "self-invoking".

- This means that they can be invoked automatically, without being called!

Syntax

```javascript
(function () {
  document.getElementById("demo").innerHTML = "Hello! I run by myself";
})();
```

Global variables & Functions

- Variables declared outside of **any** function-> global variables

- Global variables -> visible from any function (unless shadowed by local variables)

- Note: It's good practice to minimize the use of global variables. Modern code has **few or no** global variables.

Nested Functions

- JavaScript support nested functions

- When functions get particularly complex, it can be helpful to break them into smaller functions
  - clarify exactly what each section of the code is doing.
  - break up long sections into smaller more readable ones

- Creation of a nested function is simple: **declare the function as it is normally done, only within the scope of another function.**

Functions: Scope
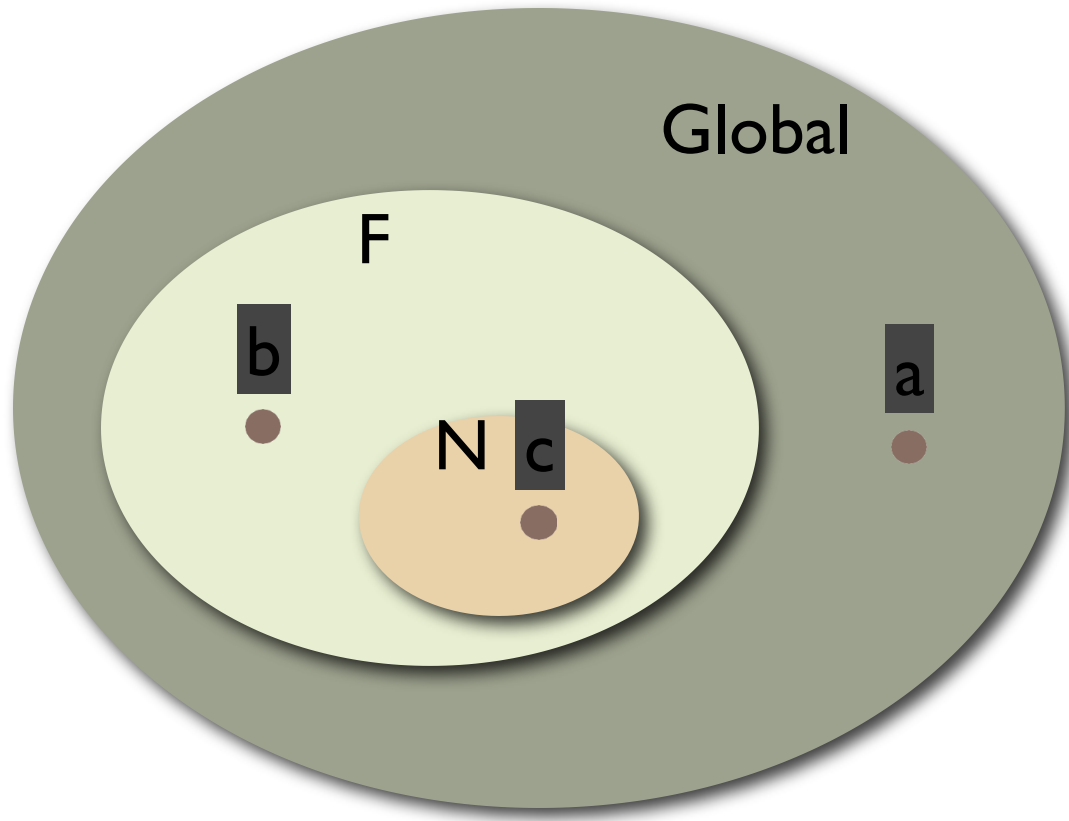

```
var a;
//..
function F() {
    var b;
    //..
    function N() {
        var c;
        //..
    }
}
```

Lexical scope

➤ In JavaScript **lexical scope** is the **definition area** of an **expression**.

➤ Item's lexical scope-> is the place it got created (its definition space)

➤ The place an item got invoked (or called) is not necessarily the item's lexical scope (Αλλού το δηλώνουμε αλλού το καλούμε)

➤ Functions create their environment (**scope**) when they are defined, not when they are executed.

What do we expect to see here?

```javascript
function func1(){
    var v=1;
    return func2();
}

function func2(){
    return v;
}

function func3(){
    var v=2;
    return(
        function(){
            return v;
        }
    )();              //self Invoking function

}

    console.log("one : "+func3());
    console.log("two : "+ func1());
</script>
```

Lexical scoping

scope is determined by the **placement** of variables & functions in the code

it remains the **same throughout the execution of the program**

Scopes in JS 1/2

- **Lexical scope:** scope of variable or function based on **where it is defined** in the source code

- **Global Scope**: variable defined **outside of any functions** or **blocks**, has global scope.
  - ➤This means that it is accessible from everywhere in the program

- **Local Scope**: variable defined within a function or block, it has a local scope.
  - ➤This means it can only be accessible inside the context of that function or block.

Scopes in JS 2/2

- **Nested Scope:** function defined inside another function, has access to variables defined in the parent function

- **Block Scope**: ES6 introduced the **let & const** keywords
  - ➢These keywords allow variables to have block scope.
  - ➢That variables defined within a block of code can only be accessed within that block.

function arguments

- Function arguments : parameters passed to functions
- They are listed inside the parentheses () in function definition
- They act as **local variables**

```
<script>
function myFunction(p1, p2) {

    alert(p1+p2);
}
myFunction(2,3);
```

Anonymous function

Anonymous functions:

- are defined without using a name

- must be assigned to a variable.

- the variable is then used as function

```
<p id="demo"></p>

<script>
var x = function (a, b) {return a * b};
document.getElementById("demo").innerHTML = x(4, 3);

//The function above is actually anonymous function (
a function without a name).

//Functions stored in variables do not need function
names as they are always called using the variable
name!
```

Note that normal functions are run before any other code, meaning they do not have to be declared before the usage of them.

Anonymous functions are created at run time.

Interesting source: https://dev.to/chris_bertrand/coding-concepts---anonymous-methods-a9o

More function properties:

- JavaScript functions can be used as **values and in expressions**!

```
function myFunction(a, b) {
  return a * b;
}
var x = myFunction(4, 3);
var x = myFunction(4, 3) * 2;
```

Arrow functions

arrow functions

➢alternative to traditional functions
➢**can't be** used in all cases

Arrow functions Syntax

- One param

    param => expression

    x => x+10

- Multiple params require ()

    (param1, paramN) => expression

    (x, y, z) => x + y+ z + 10;

- Multiline statements need { }
  & return:

    (handwritten) r (param1, param2)

    param => {

      let x = 2;

      let y= 3:

      return x*y + param;

    }

Arrow functions Syntax

- no params

  var x = 2;
  var y= 3:
  () => x + y + 100;

- Default parameters :

- (x=2, y=3, c) => x + y + c*100

Arrow functions

Difference in syntax example:

```
// Traditional Function
function nameme (y){
  return y + 100;
}


// Arrow Function
Let nameme = y => y + 100;
```

```
<script>
var result1,result2;

  // Traditional Anonymous Function
  result1 = function (a){
    return a + 100;
  }

  // Arrow Function
  result2 = a => a + 100;

  console.log(result1(10)+" "+result2(20)); // 110 120

</script>

</body>
```

We return only one thing, no other commands, so syntax can be that simple

Conditions

if statement

- if statement -> the most common type of condition

- if statement runs only if the condition enclosed in parentheses () is *truthy*

- extend if statement with an **else** statement -> adds another block to run when the if conditional doesn't pass

- **else if** statement, which adds another condition with its own block

truthy value

- In JavaScript, a *truthy* value is a value that is considered **true** when encountered in a **Boolean context**

- Truthy= not falsy

JS falsy values

Some **falsy values**:

- false

- 0 and -0

- ""“: Empty string

- Null: absence of any value

- undefined

- NaN: Not a Number

What will outcome be?

```javascript
var outcome ;
if (false) {
    outcome = "if block";
} else if (true) {
    outcome = "first else if block";
} else if (true) {
    outcome = "second else if block";
} else {
    outcome = "else block";
}
console.log(outcome);
```

What will outcome be?

```javascript
var result1 ={
  one:"one",
  two:"two"
}
var result2 ={
  one:"one",
  two:"two"
}

if (result1===result2){
  console.log("=== is true")
}else if(result1==result2){
  console.log("== is true")
}else{
  console.log("we are not equal!")
}
```

Object Comparison

- In the code above, we have **two separate objects, result1 and result** both of which have the **same** <u>properties and property values</u>.

- result1 & result2 are two **distinct objects in memory**, even though they have the same property names and values

In JavaScript, <mark>objects are **compared by reference, not by their content**</mark>. When you use the equality operators (== and ===) to compare objects, you are comparing whether the two **objects reference the same location in memory,** not whether they have the same properties and values!

What will console.logs show below?

```
var object1={ a:"val-a"};
var object2= object1;

console.log(object1);
console.log(object2);


object2.a="changed";


console.log(object1);
console.log(object1);
```

What will console.logs show below?

```javascript
var object1={ a:"val-a"};
var object2= object1;

console.log(object1);
console.log(object2);


object2.a="changed";


console.log(object1);
console.log(object1);
```

▸ {a: 'val-a'}

▸ {a: 'val-a'}

▸ {a: 'changed'}

▸ {a: 'changed'}

When we **pass** an object (or array) it is possible to modify the contents of that object.

Here a reference to *object1* is assigned to *object2*.
Think of it like the same object is accessible by two names.

- let name = 'Marina'; let name2 = name;

- **Objects** in JavaScript **are passed by reference.**
- When more than one variable is set to store either an object or array those variables **will point to the same allocated space** in the memory.

- const animals = ['Cat', 'Dog', 'Horse', 'Snake'];
- let animals2 = animals
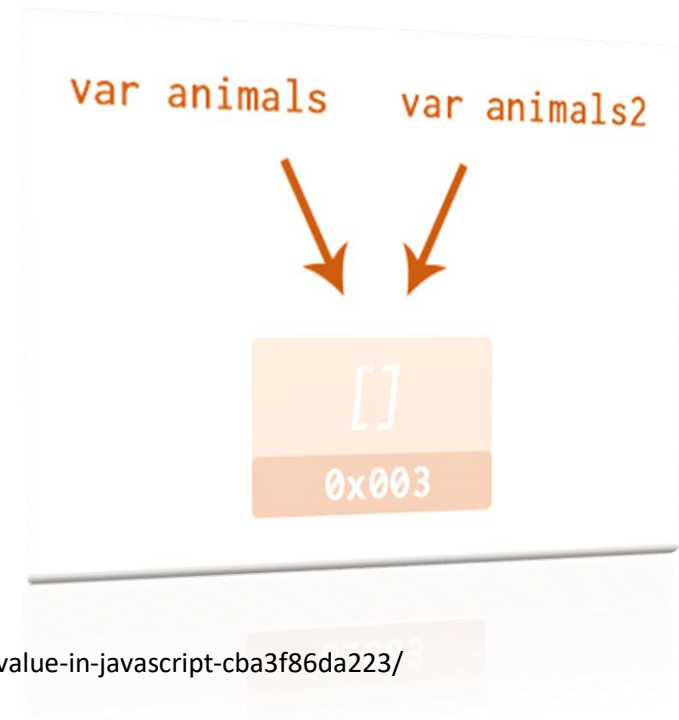
Logical Operators

- Logical operators are widely used in conditions

- && and
- || or
- ! not

Operator precedence

- The sequence that logical operators are executed is the following:

- !   && ||

What console log will display below

```
const a = true , b = false , c = false;
var d =  a && b || a && c;
console.log(d);


var z =  !b && a || a && c;
console.log(z);
```

What console log will display below

```javascript
//logical operator presedence
const a = true , b = false , c = false;
var d =  a && b || a && c;
//a && b => false
//a && c => false
//false
console.Log(d);


var z =  !b && a || a && c;

//!b && a true
//a && c false
//true

console.Log(z);
```

What console log will display below

```
const a = true , b = false , c = false;
```

```
d =  a && ! b || a ;
```

Switch Statement

- **switch** statement -> is used to perform **different actions** based on **different conditions**

Switch syntax

```
switch(expression) {
  case x:
    // code block
   break;
  case y:
      // code block
   break;
  default:
      // code block
}
```

What the above means

- switch(expression) -> is evaluated once

- **value of the expression ->**is compared with the values of each **case**
  - ➢strict comparison (===)->same type and value

- If there is a match -> associated block of code is executed

- If there is no match -> default code block is executed

What the above means

- When JS reaches the **break** keyword -> it breaks out of the switch block

- SOS: If you omit the break statement, the **next case** will be executed even if the evaluation does not match the case.

Lets see an example!

What would you like to eat:

```
Oranges
```

Try it

Oranges are $0.59 a pound.

Apples are $0.32 a pound.

Is there anything else you'd like?

>

no break

```javascript
var expr = document.getElementById('eat').value;

switch (expr) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
  case 'Apples':
    console.log('Apples are $0.32 a pound.');
    break;
  case 'Bananas':
    console.log('Bananas are $0.48 a pound.');
    break;
  case 'Cherries':
    console.log('Cherries are $3.00 a pound.');
    break;
  case 'Mangoes':
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
    break;
  default:
    console.log('Sorry, we are out of ' + expr + '.');
}

console.log("Is there anything else you'd like?");
}
```

Keep in mind:

- If multiple cases matches a case value ->first case is selected

- If no matching cases are found -> default label is executed

- If no default label is found-> code runs beyond switch

Loops

Loops

- A loop is -> **block of code** that allows you to repeat a section of code a certain number of times

Pros

- repeat lines of code without retyping them

- save time/trouble/errors of repeatedly typing the same lines of code

- change one or more variable values in each time loop

- set the number of iterations dynamically

Conditional Loops

- Conditional loops are often defined by *where* **the condition** is written-> **in reference to** the executable block of code:

  - ➤ Pre-test loops: condition located *before* the executable block
    - ➤ As a result, there is a possibility that a pre-test loop *may never execute*. (ie while loop)

  - ➤ Post-test loops: condition located *after* the executable block
    - ➤ A pre-test loop's executable block *will always execute at least once (ie* do ... while loop)

For Loop

- 3 parts : initialization, condition, iteration

- **loop initialization** -> we initialize counter to starting value

  ➢ initialization statement is executed **before** the loop begins

- **test statement** -> tests if the given condition is true or not

  ➢ If condition is true then code given inside the loop will be executed otherwise loop will come out

- **iteration statement** -> where we increase /decrease the counter
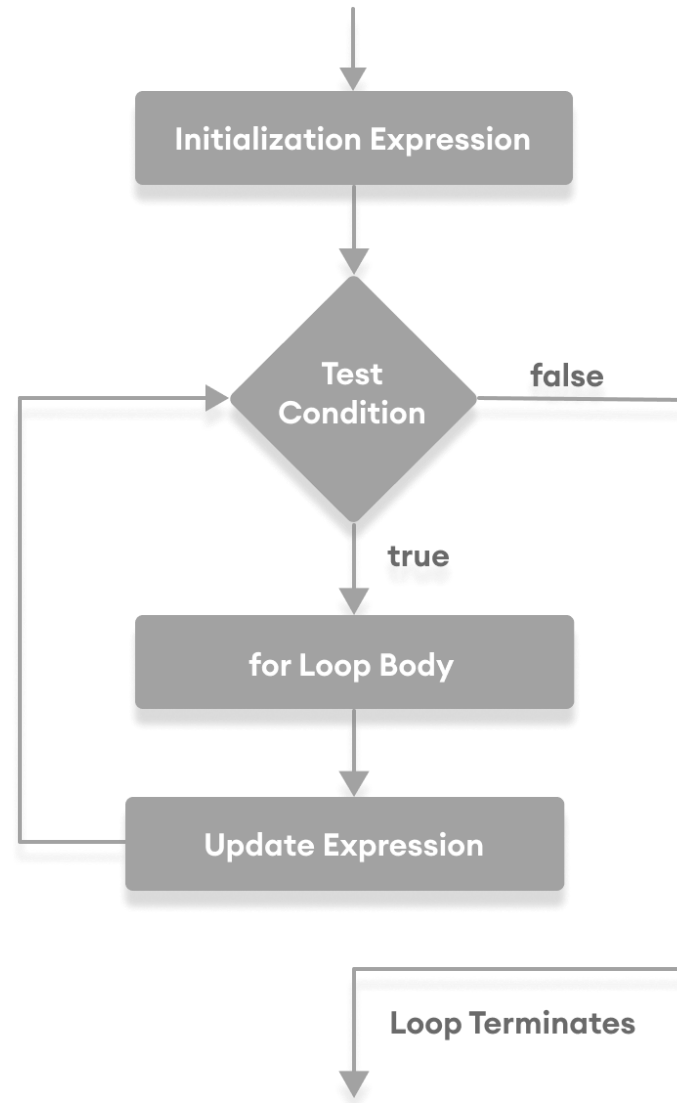
For loop

for (varname=1;varname<11;varname+=1)

initialization

As long as
varname value
is less than 11
run

Determines the rate at
which the variable is
changed and whether it
gets larger or smaller

For loop Example

JavaScript 1rs For Loop

Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10

```html
<p id="demo"></p>

<script>
var num = "";

for (var i = 0; i <=10; i+=1) {

    num +=  "Number is " + i + "<br>"; // num = num + "Number is " + i + "<br>";
    console.log(num);
}

document.getElementById("demo").innerHTML = num ;
```

More about for loop

Iteration can begin anywhere -> it can end anywhere

```
for ( j = −10; j <= 10; j = j + 1) { . . . }
for ( j = 2.5; j <= 6; j = j + 1) { . . . }
```

- **test statement** -> any expression **resulting in a Boolean value**

- It must involve the **iteration variable**

- **Iteration statement** allows us to specify how big or small the change in the iteration variable

- The amount of change is known as **step or step size**:
        i=i+1   i+=2   i+=10

Beware of Infinite Loops

- It is possible to create infinite loops that never terminate!

- If the **test statement** is based on values that don't change in the loop, the loop will never end !

```
var i=0;
for ( j = 1 ; j <= 3; i = i + 1) {
    document.write("I will run forever yooo  <br>");
}
```

Nested Loops

- Nested Loop ->Loop in a Loop

- Programming languages allow loops to nest

- NOTE: Inner and outer loops must use different iteration variables or else they will **interfere with each other**

What will we see with the code below?

```javascript
function myFunction() {

var n = 5;
var stars = "";

for(var i=0; i<10; i++){
        stars = "";
    for(var j=i; j<n; j++){
        stars = stars + " *";// *   // **
        document.write( "i: "+i+",j: "+j+" = "+stars+"<br>");
    }
    console.log(i);
  }
}
```

example

```
i: 0,j: 0 = *
i: 0,j: 1 = * *
i: 0,j: 2 = * * *
i: 0,j: 3 = * * * *
i: 0,j: 4 = * * * * *
i: 1,j: 1 = *
i: 1,j: 2 = * *
i: 1,j: 3 = * * *
i: 1,j: 4 = * * * *
i: 2,j: 2 = *
i: 2,j: 3 = * *
i: 2,j: 4 = * * *
i: 3,j: 3 = *
i: 3,j: 4 = * *
i: 4,j: 4 = *
```

```javascript
function myFunction() {

var n = 5;
var stars = "";

for(var i=0; i<10; i++){//i=0 // i=1;//i=5
    stars = "";
    for(var j=i; j<n; j++){//j=i ,j=0 , 0<5 //  j=1; 1<5 ;    // j=1 //j=5
        stars = stars + " *";// *  // **
        document.write( "i: "+i+",j: "+j+" = "+stars+"<br>");
    }
    console.log(i);
}
}
```
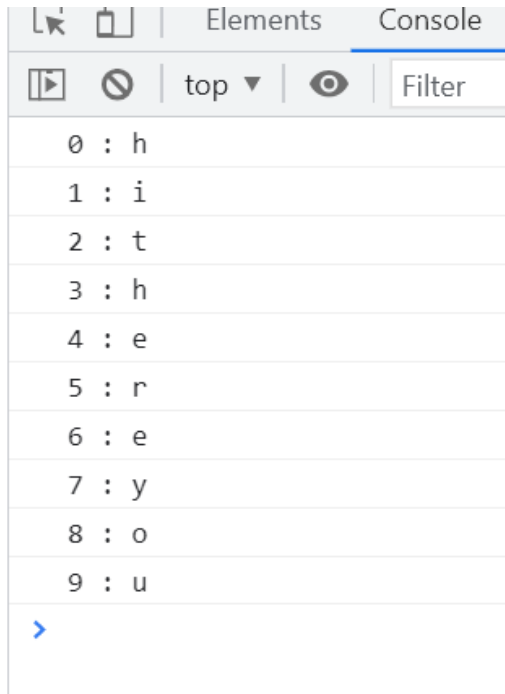
for/in loop

- for/in statement -> loops through properties of an **object (or array)**
- block of code inside the loop will be executed **once for each property**

- **Syntax:**

for (var in object) {

  code block to be executed

}

Lets see an example

```
0 : h
1 : i
2 : t
3 : h
4 : e
5 : r
6 : e
7 : y
8 : o
9 : u
```

```html
<p id="demo"></p>

<script>
const string = 'hithereyou';

// using for...in loop
for (let i in string) {

    //i is the index of elements in the array
    console.log(i+" : "+string[i]);
}
</script>
```

Lets see an example

Jack 24000

Paul 34000

Monica 55000

```html
<script>
var salaries= {
    Jack : 24000,
    Paul : 34000,
    Monica : 55000
}

// using for...in
for ( let i in salaries) {

    //The object key name is assigned to variable i.
    console.log( i+" "+salaries[i]);
}
```

for/of Statement

- for/of statement -> loops through the values of an iterable object

```
var cars = ['BMW', 'Volvo', 'Mini'];
var car;

for (car of cars) {
  document.write(car + "<br >");
}
</script>
```

for..of vs. for..in

- Both for..of and for..in statements iterate over lists
- the values iterated on are different though:

- for..in returns a **list of keys** on the object being iterated

- whereas for..of returns a **list of values** of the object being iterated

Examples

```javascript
var cars = ['BMW', 'Volvo', 'Mini'];
var car;

for (car of cars) {
  document.write(car + "<br >"); // BMW  Volvo  Mini
}

for (car in cars) {
  document.write(car + "<br >");//0 1 2
}
</script>
```

```
7   <script>
8   document.getElementById("demo").innerHTML = santaDecision([
9       {
10          name: "Angelina Jolie",
11          status: "good"
12      },
13      {
14          name: "Severus Snape",
15          status: "good"
16      },
17      {
18          name: "Voldermort",
19          status: "bad"
20      },
21      {
22          name: "Malfoy",
23          status: "bad"
24      },
25      {
26          name: "Professor",
27          status: "good"
28      }
29  ]);
30
31  function santaDecision(arr){
32
33  }
34
```

Angelina Jolie Gets a present
Severus Snape Gets a present
Voldermort does not a present
Malfoy does not a present
Professor Gets a present

While loop

- Allows repeatable code until a given condition is met

- Pre-Test Loop (Loop may NEVER execute)

```
var x = 1 ;

while ( x < 10 ) {
    document.write ( x ) ; //123456789
    x = x + 1 ;
}
```

For vs while

- When the **exact number of iterations** is known you may use the 'for' loop

- When the number of iterations **depend upon a condition being met** you may use the 'while' loop

do/while Loop

- Allows repeatable code **while a given condition** is true.

- Must have a way of terminating the structure from within the loop!

- Note that: it is Post-Test Loop (Loop always executes **at least once**)

Example

0: Loop
1: Loop
2: Loop
3: Loop
4: Loop

```javascript
function myFunction() {
  var text = "";
  var i = 0;

  do {
    text += i+ ": Loop <br>";
    i++;
  }
  while (i < 5);

  document.getElementById("demo").innerHTML = text;
}
</script>
```

Arrays more…

Syntaxes for creating an empty array:

var arrname = new Array();
var arrname = [];

Note: There is no need to use new Array()! For simplicity, readability and execution speed, use the [] syntax (put initial elements in the brackets.)

join method

- **join method** of an Array -> returns a **string** elements of an array, separated by the string supplied in the function's argument
- Syntax

  array.join(separator)

- If an separator is not specified, elements are separated with a comma

```
var fruits = ["Banana", "Orange", "Apple"];
document.getElementById("demo").innerHTML = fruits.join(" and ");
```

Banana and Orange and Apple

Array.sort

- Sorting data
- Putting data in a particular order, such as ascending or descending
- Array object in JavaScript has a built-in method sort()
- no arguments-> the method uses string comparisons to determine the sorting order of the Array elements
- reverse() method reverses the elements in an array= sort an array in descending order: first sort then reserve

Array.sort

- What will we see below?

```
var nums = [1, 7, 32, 100];
nums.sort()
alert(nums);
```

Array.sort

- What will we see below?

```javascript
var nums = [1, 7, 32, 100];
nums.sort()
alert(nums);
```

127.0.0.1:5500 says

1,100,32,7

OK

Array.sort

- sort() function sorts values as strings -> However, if numbers are sorted as strings, "31" is bigger than "100", because "3" is bigger than "1".

- You can fix this by providing a compare function

Array.map()

- **map() method ->** creates **a new array** by performing a **function** on each array element

- map() method -> **only** executes the function for array elements with values

- map() method -> **does not** change the original array

```
var array1 = [2, 4, 6, 12, 20];
//num (or another name) is required
//num is a value of the current element
var array2 = array1.map( num => num*2);
```

array2 : 4,8,12,24,40

Using Array.length

- The following code averages the values stored in an array

```
<script>
var grades=[10,9,5,7,8,3];
var sumGrades=0;
for(var i=0; i<grades.length; i++)
    {
        sumGrades += grades[i]
    }
        avgScore = sumGrades/grades.length;
document.getElementById("demo").innerHTML = avgScore;
```

Note that…

- When iterating over all the elements of an Array, use a for…in statement to ensure that you manipulate **only the existing elements** of the Array

- **for…in** statement skips any undefined elements in the array

Lets see some examples

- What will the result be below?

```
<script>
var grades=[10,,5,7,8,3];
var sumGrades=0;
for(var i=0; i<grades.length; i++)
    {
        sumGrades += grades[i];
    }
        avgScore = sumGrades/grades.length;
document.getElementById("demo").innerHTML = avgScore;
```

Lets see some examples: for with undefined

- In the array below we have an undefined element. What will the result be?

```html
<script>
var grades=[10,,5,7,8,3];
var sumGrades=0;
for(var i=0; i<grades.length; i++)
    {
        sumGrades += grades[i];
    }
        avgScore = sumGrades/grades.length;
document.getElementById("demo").innerHTML = avgScore;
```

$\rightarrow$ NaN

Lets see some examples:

- How could we fix the above issue?

Lets see some examples:

- How could we fix the above issue?

```javascript
var grades=[10,5, ,7,8,3];
var sumGrades=0;
for(var i=0; i<grades.length; i++){
    if(!isNaN(grades[i]))
        sumGrades += grades[i];
}
    avgScore = sumGrades/grades.length;
document.getElementById("demo").innerHTML = avgScore;
```

```javascript
var grades=[10,,5,7,8,3];
var sumGrades=0;

for(var i in grades){
    sumGrades += grades[i];
}

    avgScore = sumGrades/grades.length;
document.getElementById("demo").innerHTML = avgScore;
```

Sol. 1   Sol. 2

forEach() Method

Arrays forEach() method -> **calls a function once** for each element in an array

- forEach doesn't return anything, (map creates another array)
- Note: the function is not executed for array elements without values.

What will console log print?

```
<script>
var sum = 0;
var sum1 = 0;

var numbers = [65, 44, 12, 4];

const array1= numbers.forEach(item => sum+=item);

console.log(array1);

const array2=numbers.map(item => sum1+=item);

console.log(array2);

console.log(numbers);
```

Lets see an example…

```javascript
var sum = 0;
var sum1 = 0;

var numbers = [65, 44, 12, 4];
// forEach() method does not create a new array based on the given array.
const array1= numbers.forEach(item => sum+=item);
// forEach() method returns "undefined".
console.log(array1);

//map() method creates an entirely new array
const array2=numbers.map(item => sum1+=item);
//map() method returns the newly created array
console.log(array2);

console.log(numbers);
```

```
undefined
▶ (4) [65, 109, 121, 125]
▶ (4) [65, 44, 12, 4]
>
```

$array.\text{forEach}(function(currentValue, index, arr))$

```
<script>
var sum = 0;
var numbers = [5, 10, 1, 3];
numbers.forEach(myFunction);

console.log(sum);//19

function myFunction(item) {
  sum += item;
}
</script>
```

function()	Required. A function to run for each array element.
currentValue	Required. The value of the current element.
index	Optional. The index of the current element.
arr	Optional. The array of the current element.

Source: https://www.w3schools.com/

Array.filter()

- array filter() method -> creates a **new array** with array elements that **pass a condition**

```
var myarray = [45, 4, 9, 16, 25];
var condition = myarray.filter(num => num < 10); //4 9
```

DOM

What is the DOM

The DOM defines a standard for accessing documents:

> ➢ "The W3C **Document Object Model** (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

- The Document Object Model (DOM) is a programming interface for web documents.

- The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

HTML DOM

- When a web page is loaded -> the browser **creates** a **Document Object Model** of the page, DOM for short

- HTML DOM -> allows JavaScript to access and change all elements of an HTML document

- So, DOM **gives access** to all the elements on a web page -> Using JavaScript we are able to create, modify, remove elements in the page dynamically.

Introduction

- The HTML DOM model is constructed as
  a tree of Objects:

- **Element Node** – contains an HTML tag

- **Text Node** – contains text

  ➢ **Text Nodes** are contained in
     Element Nodes
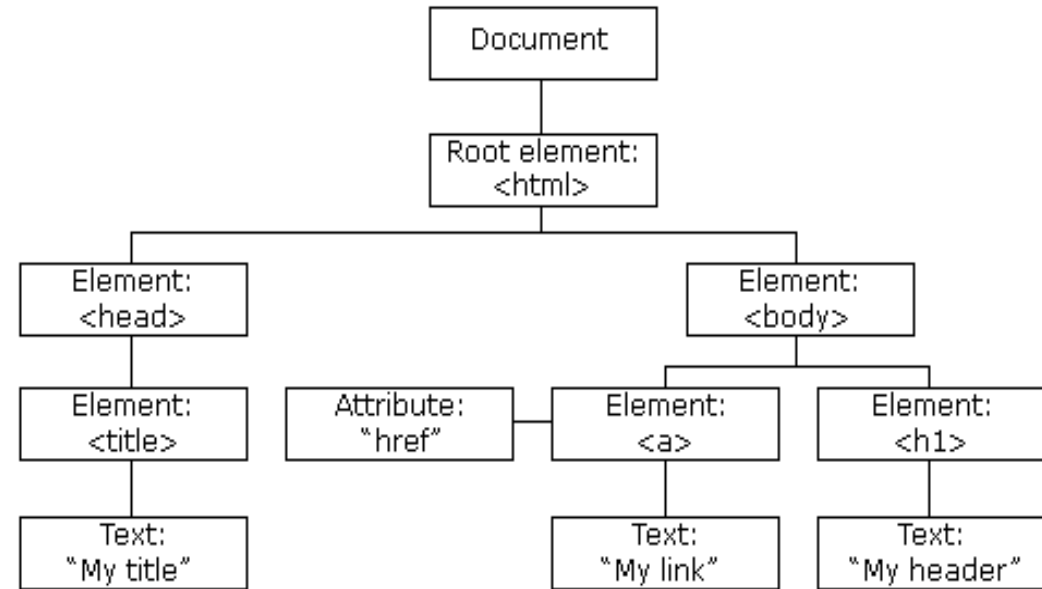
Note that:
The Document Object Model (DOM) is not part of the JavaScript programming language; it is a Web API used to construct webpages.
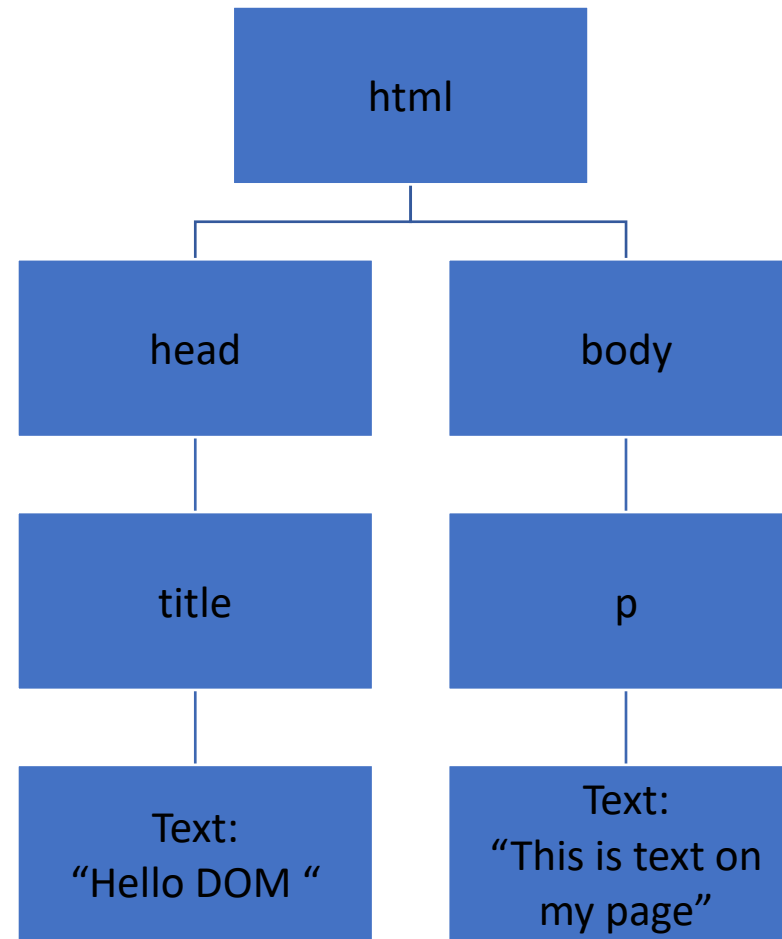
The Document Object Model was designed to be independent of any specific programming language.

Even while most web developers will only utilize the DOM via JavaScript, DOM implementations can be created for any language.

Nodes Organise the Page

```
<html>
    <head>
        <title>Hello DOM</title>
    </head>
    <body>
        <p>This is text on my page</p>
    </body>
</html>
```

DOM Nodes and Trees

- **nodes** in a document -> make up the page's DOM tree
  - ➤ This tree describes the **relationships among elements**

- **nodes** are related to each other -> through child-parent relationships

- a **node** -> may have multiple children

- a **node** -> has **only one** parent

- **nodes** with the same **parent** node-> named **siblings**

- **root node ->** has no parent

childNodes vs children

```
<body><!-- This is a comment node! -->
<p>Click the button get info about the body element's child nodes.</p>
<button onclick="myFunction()">Try it</button>
<p><strong>Note:</strong> Whitespace inside elements is considered as text, and text
is considered as nodes. Comments are also considered as nodes.</p>

<p id="demo"></p>

<script>
function myFunction() {
// var t= document.getElementsByTagName(button).childNodes;
  var c = document.body.childNodes;
// var c = document.body.children;
  var txt = "";
  var i;
  for (i = 0; i < c.length; i++) {
    txt = txt + c[i].nodeName + "<br>";
  }

  document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

Click the button get info about the body element's child nodes.

[Try it]

Note: Whitespace inside elements is considered as text, and text is considered as nodes. Comments are also considered as nodes.

#comment
#text
P
#text
BUTTON
#text
P
#text
P
#text
SCRIPT
#text
#comment
#text
SCRIPT
#text

.children -> is a property of an Element
        Elements have .children-> these children are all of type Element

.childNodes -> is a property of Node(text is also a node)-> contains any node

Tip: if you **do not** want to loop over Text or Comment nodes use .children

childNodes vs children

- childNodes:
    - returns a Nodelist of child nodes.
    - Nodelist items are objects ->they can be accessed using index numbers
    - The first childNode starts at index 0.

- children
    - returns the child elements of an element as objects.

childNodes vs children

- **children** work upon elements

- **childNodes** on nodes including **non-element** nodes like text and comment nodes.

- The text inside elements forms text nodes, labelled as #text.

- A text node contains only a string ->is always a leaf of the tree.

- Spaces and newlines are totally valid characters, like letters and digits. They form text nodes and become a part of the DOM.

- •node.childNodes
- •node.firstChild
- •node.lastChild
- •node.parentNode
- •node.nextSibling
- •node.previousSibling

DOM allows Js to create dynamic HTML

JavaScript can

➢Change/ remove /add HTML elements in the page

➢Change/ remove /add HTML attributes in the page

➢change all CSS styles in the page

➢react to all existing HTML events in a webpage

terminology…

- HTML DOM **methods** are -> actions you can perform on HTML Elements (ie. add or delete)

- HTML DOM **properties** are -> values of HTML Elements that you can set or change (like changing content of an HTML element)

terminology…

Example:

- document.getElementById("demo").innerHTML = "Hi there!";

- getElementById-> **method**

- innerHTML   -> **property** (used for getting/replacing content of HTML elements)

Finding HTML Elements

- getElementById() -> allows you to find and work with elements based on their individual id

- getElementsByTagName() -> allows you to find and work with groups of elements based on their tag name(This method returns an array)

- document.getElementsByClassName(name)-> allows you to find and work with elements based on their class name

getElementsByTagName()

Example:

- var c= document.getElementsByTagName("button");
- var c = c[0].childNodes;

- Note: it is an array, so if we want first button occurrence: index 0

```html
<!DOCTYPE html>
<html>
<body>
<!-- This is a comment node! -->
<p>Click the button get info about the body element's child nodes.</p>
<button onclick="myFunction()">Try it</button>
<button onclick="myFunction()"><!-- This is a comment node! -->
  <a>lalalal</a>
  <!--<p>.hiiiiiiiiiiii</p> -->
</button>
<p id="demo"></p>

<script>
function myFunction() {
 var c= document.getElementsByTagName("button");
 c = c[1].children;

  var txt = "";
  var i;
  for (i = 0; i < c.length; i++) {
    txt = txt + c[i].nodeName + "<br>";
  }

  document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

getElementById

```
<label for="name">Name 1:</label><br>
<input type="text" id="name" name="name"><br>


<label for="surname">Surname:</label><br>
<input type="text" id="surname" name="surname"><br>


<button onclick="myFunction()">Register</button>



<p id="result">


<script>


function myFunction() {
  var name = document.getElementById('name').value;
  var surname = document.getElementById('surname').value;
  document.getElementById("result").innerHTML = name +"   "+surname;
}
```

Methods: Changing /Adding /Deleting Elements

element.setAttribute*(attribute, value)*	Change the **attribute** value of element
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new, old*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

Attribute Nodes

- Lets look at:
  - getAttribute()
  - setAttribute()

get/set Attribute Method

- getAttribute() method -> returns value of the attribute with the specified name of an element

- setAttribute() method -> **adds** specified attribute to an element & **gives** it the specified value
  - ➢In case the specified attribute already exists -> only the value is set/changed.

- Syntax:
  - ➢element.setAttribute(attributename, attributevalue)
  - ➢ie. attribs.setAttribute("class","democlass1");

- Note that: the removeAttribute() method removes an attribute from an element.

Task

- we have a some p elements in our html and a button.

- On click of the button and -> get all p elements and for each one of them, we get the class attribute.

- Then we must store the class attribute in an array.

- In case a p element does not have a class attribute, set the class attribute to "democlass1

- Present the above result with "," in another p element

```html
<!DOCTYPE html>
<html>
<head>
<style>
.democlass1 {
  color: ■red;
}
.democlass2 {
  color: ■blue;
}
.democlass3 {
  color: ■yellow;
}
.democlass4 {
  color: ■green;
}
</style>
</head>
<body>


<p>Click the button to display the value of the class attribute of the p elements.</p>


<p class="democlass4"> This class is Red</p>
<p class="democlass2"> This class is Blue</p>
<p class="democlass3"> This class is Yellow</p>

<button id="btn" onclick="dispAttribs()">Try it</button>

<p id="demo"></p>


<script>

function dispAttribs() {
    var messg=[], attribs = document.getElementsByTagName("p");

    for (var i = 0; i < attribs.length; i++) {

        if (attribs[i].getAttribute("class")!=null){
            messg.push(attribs[i].getAttribute("class"));
        }else{
            attribs[i].setAttribute("class","democlass1");
        }
        console.log(messg);
        }
    document.getElementById("demo").innerHTML = messg.join(",");
  document.getElementById("btn").remove();


}

</script>


</body>
</html>
```

Question

- How can we use DOM methods to add a p element with some text in our webpage?

Question

There are five steps:

1. Create new Element
2. Create new Text
3. Append the new Text to the new Element
4. Find an existing Element
5. Append the new Element to the existing Element

1. Create New Element Node

create a new <p> element so that we can attach some text to it

Code:

var newNode = document.createElement("p");

2. Create a Text Node

Next, create a text node:

Code:

var myText = "This is new text to be added to the page dynamically.";

var newText = document.createTextNode(myText);

3. Attach the New Text Node to the New Element

To put the text into the page, we have to attach the text node to the new HTML element:

newNode.appendChild(newText);

4.Find an Existing Element

The new element with our text node attached to it is still *floating around* in a Javascript world

- So we must now **find** an existing element so that we can attach it!:

```
<p id="thisLocation">New text will appear below here</p>
```

```
var docElement= document.getElementById(location);
```

5. Append the New Element to the Existing Element

To insert our text into the page, we now have to append the new element to the existing element

docElement.appendChild(newNode);

To be continued...

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

String to Number: https://dev.to/sanchithasr/7-ways-to-convert-a-string-to-number-in-javascript-4l

https://careerkarma.com/blog/javascript-queryselector-vs-getelementbyid/