



JS Notes"loosely typed language"

Where ?!

- we could write js code in-
- internal in the HTML file with the tag `<script>window.alert("ERROR")</script>`
- external in an external file and link it `<script src=""></script>`
- or in tag in button `<button onclick="window.alert('Welcome')"></button>`

Output

- JavaScript can "display" data in different ways:
- Writing into an HTML element, using `innerHTML` .

EX-

```
var username="mahmoud"
document.getElementById('demo').innerHTML=username
```

- Writing into the HTML output using `document.write()` .

Ex-

```
var username="mahmoud"  
document.write(username)
```

- Writing into an alert box, using `window.alert()`.

EX-

```
var username="mahmoud"  
window.alert(username)
```

- Writing into the browser console, using `console.log()`

EX-

```
var username="mahmoud"  
console.log(username)
```

Data Types

- **JavaScript has 8 Datatypes**

1. String
2. Number
3. BigInt
4. Boolean
5. Undefined
6. Null
7. Symbol
8. Object

- **The Object Datatype**

The object data type can contain:

1. An object
2. An array
3. A date

Input

- There are many ways to take input-
- using `window.prompt()`

EX-

```
var username=window.prompt("Enter Your name")
document.getElementById("demo").innerHTML="Welcome "+username.toUpperCase()
```

semicolon

- The best way is to put semicolon in every line to avoid bugs in case of archive the file

For Loop

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

Implicit - explicit conversion

[JavaScript Type Conversions \(with Examples\) \(programiz.com\)](#)

Ex-

```
var num1 = Number(window.prompt("enter num1: "))
var num2 = Number(window.prompt("Enter num2: "))
var result = num1 + num2;
console.log(result)
```

- number ("50") \Rightarrow 50
- "ahmed" \Rightarrow NaN
- "" \Rightarrow 0
- " " \Rightarrow 0
- true \Rightarrow 1
- false \Rightarrow 0
- NaN \Rightarrow NaN
- null \Rightarrow 0
- undefined \Rightarrow undefined

for more examples see the link above

Arithmetic operation

+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (<u>ES2016</u>)
/	Division
%	Modulus (Remainder)
++	Increment
--	Decrement

EX-

```
let x = 5;
let y = 2;
```

```
let z = x + y;  
console.log(z);
```

Comparison operator

==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true
		x !== 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

- so we can say that two equal make a comparison in value not type ,but three equals makes the opposite

Logical operator

&&	and	(x < 10 && y > 1) is true
	or	(x == 5 y == 5) is false
!	not	!(x == y) is true

Assignment Operator

=	x = y	x = y
---	-------	-------

+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

If condition

EX-

```
var greeting="";
var time=window.prompt();
if (time < 10) {
    greeting = "Good morning";
}
else if (time < 20) {
    greeting = "Good day";
}
else {
    greeting = "Good evening";
}
console.log(greeting);
```

Switch

EX-

```
let x = "0";
var text="";
switch (x) {
    case 0:
        text = "Off";
        break;
    case 1:
        text = "On";
        break;
    default:
        text = "No value found";
}
```

```
}  
console.log(text);
```

Function

Void function

EX-

```
function getsalary(usersalary,profit,tex)  
{  
    var result = usersalary*profit/tex;  
    console.log(result);  
}  
getsalary(2000,200,20);
```

Return

```
function toCelsius(fahrenheit) {  
    return (5/9) * (fahrenheit-32);  
}  
  
let value = toCelsius(77);  
console.log(value);
```

Hoisting

JavaScript Declarations are Hoisted In JavaScript, a variable can be declared after it has been used.

EX-

```
x=10  
console.log(x);  
var x;
```

Interview Qeustion

```
function foo(){
  return bar();
  function bar(){return 3;}
  var bar =function(){return 8};
}
alert(foo())
```

in this funciton the output will be 3 because the hoisting work only on declarion funciton

```
function foo(){
  function bar(){return 3;}
  return bar();
  function bar(){return 8;}
}
alert(foo())
```

in this function the output will be 8 because the funcion here is declarion type

Scope

JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block

EX-


```
{  
  let x = 2;  
  console.log(x)  
}
```

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

EX-

```
// code here can NOT use carName  
  
function myFunction() {  
  let carName = "Volvo";  
  // code here CAN use carName  
  console.log(carName)  
}  
  
// code here can NOT use carName
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**

EX-

```
let carName = "Volvo";  
// code here can use carName  
  
function myFunction() {  
  // code here can also use carName  
  console.log(carName)  
}
```

Self invoked function

the meaning of self invoked function is the function that doesn't need to call it or invoke it in main (in other way a function call itself)

EX-

```
(function(){
    console.log("Hello World!")
}
)();
```

- The self invoked function must have semicolon “;” at the end of it

Object

EX-

```
var posts = {
  title:"this is a title",
  description:"this is a description",
  comments:{
    title:"this is a title",
    like:40,
    Reply:"replies"
  }
}
console.log(posts.title)
console.log(posts.comments.like)
```

- We can also add function to the object

EX-

```
var posts = {
  title:"this is a title",
  description:"this is a description",
  comments:{
    title:"this is a title",
    like:40,
    Reply:"replies"
  },
  share:function(){
    return "share";
  }
}

console.log(posts.title)
console.log(posts.comments.like)
```

```
var share=posts.share()
console.log(share)
```

Array

EX-

```
var car=['volvo','volks vagen','BMW']
for(var i=0;i<car.length;i++){
  console.log(car[i]);
}
```

- we can also make an array of objects so we can store more than one type of data

Ex-

```
var user={
  name:"ahmed",
  age:20,
  email:"m@gmail.com"
}
var users = [user,user,user,user]
console.log(users)
```

Array Method

Push() : insert at the end of array

Ex-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");
console.log(fruits)
```

unshift () : insert at the start of the array

Ex-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Kiwi");
console.log(fruits)
```

pop() : delete the last element of the array

Ex-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
console.log(fruits)
```

shift() : delete the first element of the array

Ex-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();
console.log(fruits)
```

Splice() : can be used to add new items to an array:

Ex-

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(fruits)
```

- The first parameter (2) defines the position **where** new elements should be **added** (spliced in).
- The second parameter (0) defines **how many** elements should be **removed**.
- The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

DOM “document object model”

Methods

getElementById/ InnerHTML

```
document.getElementById("demo").innerHTML = "Hello World!";
```

Document

Method	Description
<code>document.getElementById(id)</code>	Find an element by element id
<code>document.getElementsByTagName(name)</code>	Find elements by tag name
<code>document.getElementsByClassName(name)</code>	Find elements by class name
<code>document.getElementsByName('test')</code>	for input and it's equalaitve
<code>document.querySelectorAll('.demo h3')</code>	<i>all selectors</i>
<code>document.querySelector('.demo h3')</code>	only first element that match that value

Style

- there are 2 ways to change the HTML elemnents

```
//this is the first way but it has a disadvantage it doesn't change the bootstrap element
function changecolor(){
    var color=document.getElementById('color')
    color.style.backgroundColor="white"
}
```

```
// the second way make you write a css code with all it's attributes in JS code
function changecolor(){
    var color=document.getElementById('color')
    color.style.cssText=`
background-color: white !important;
color: black;
`
}
```

Adding ,Delete, replace elements

```
function changecolor(){
    var color=document.getElementById('color')
    color.classList.add('color')
}
```

```
function changecolor(){
    var color=document.getElementById('color')
    color.classList.remove('color')
}
```

```
function changecolor(){
    var color=document.getElementById('color')
    color.classList.replace('bg-info','color')
}
```

Add EventHandler

```
var btn=document.getElementById('btn')
btn.addEventListener('click',function changecolor(){
    alert("done")
})
```

it can also take more than `click`

-- `mousemove` -> if and only if the mouse moves on the button the function will be executed

-- `dblclick` → double click to execute

OR

```
var btn=document.getElementById('btn')
btn.addEventListener('click',changecolor)
function changecolor(){
    alert("done")
}
```

Ajax “Asynchronous JavaScript And XML”

AJAX is a developer's dream, because you can:

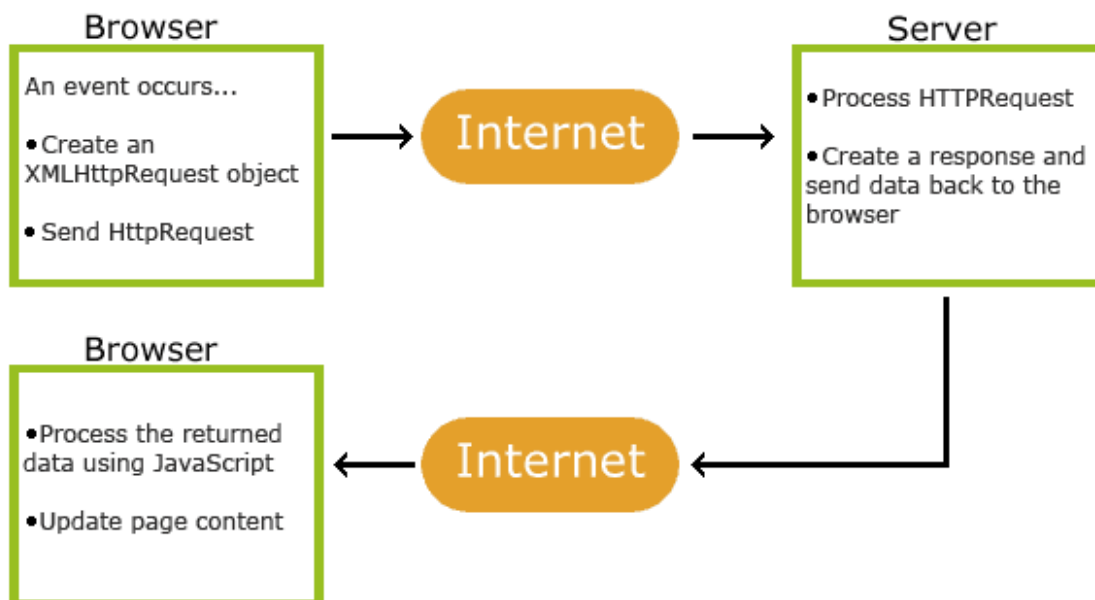
- Read data from a web server - after the page has loaded
- Update a web page without reloading the page
- Send data to a web server - in the background

AJAX just uses a combination of:

- A browser built-in `XMLHttpRequest` object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

How AJAX Works

How AJAX Works



- An event occurs in a web page (the page is loaded, a button is clicked)
- An XMLHttpRequest object is created by JavaScript
- The XMLHttpRequest object sends a request to a web server
- The server processes the request

- The server sends a response back to the web page
- The response is read by JavaScript
- Proper action (like page update) is performed by JavaScript

Web API

The fun Fact about api that it's a backend-developer task to make API but we must understand how to link it to our project

EX-

```
var httprequest=new XMLHttpRequest()
httprequest.open('GET','https://forkify-api.herokuapp.com/api/search?q=pizza')
httprequest.send()
var allposts=[]
httprequest.addEventListener('readystatechange',function(){
  if(httprequest.readyState==4){
    allposts=JSON.parse(httprequest.response).recipes
    // console.log(JSON.parse(httprequest.response));
    display()
  }
})
```

- First we need to declare a variable and assign to it `XMLHttpRequest()`
- second `open()` method and give two parameters to it `'GET'` and the link of api
- `send()` method to sends the request to the server.
- finally the `readystate` — Holds the status of the XMLHttpRequest.

0: request not initialized

1: server connection established

2: request received

3: processing request

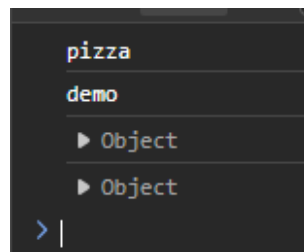
4: request finished and response is ready

JS Async


```
getsalad()  
getpasta()  
getpizza()  
demo()
```

Callback

- if we take a look into this picture we will find that the logical sorting is (salad-pasta-pizza-demo)
- but in other hand the real execution will be



```
pizza  
demo  
▶ Object  
▶ Object  
> |
```

- so if we want to deal with that matter we will use the callback

EX-

```
function myDisplayer(some) {  
    document.getElementById("demo").innerHTML = some;  
    console.log("Function 1")  
}  
  
function myCalculator(num1, num2, myCallback) {  
    let sum = num1 + num2;  
    myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);
```

Promise

EX-

```
function demo1(){  
    return new Promise(function (callback) {  
        var httpRequest=new XMLHttpRequest()  
        // ...  
    });  
}
```

```

httprequest.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza')
httprequest.send()
var allposts=[]
httprequest.addEventListener('readystatechange', function(){
  if(httprequest.readyState==4){
    allposts=JSON.parse(httprequest.response).recipes
    console.log(JSON.parse(httprequest.response));
    callback()
  }
})
})
}

function demo2(){
  return new Promise(function(callback){
    var httprequest=new XMLHttpRequest()
    httprequest.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza')
    httprequest.send()
    var allposts=[]
    httprequest.addEventListener('readystatechange', function(){
      if(httprequest.readyState==4){
        allposts=JSON.parse(httprequest.response).recipes
        console.log(JSON.parse(httprequest.response));
        callback()
      }
    })
  })
}

function demo3(){
  return new Promise(function(){
    console.log("demo3");
  })
}

demo1().then(function(){demo2()})
demo2().then(function(){
  demo3()
})

```

Async-await

- “The function has to return a promise first”

EX-

```

function demo1(){
  return new Promise(function (callback) {
    var httprequest=new XMLHttpRequest()
    httprequest.open('GET', 'https://forkify-api.herokuapp.com/api/search?q=pizza')

```

```

httprequest.send()
var allposts=[]
console.log("=====demo1=====");
httprequest.addEventListener('readystatechange',function(){
    if(httprequest.readyState==4){
        allposts=JSON.parse(httprequest.response).recipes
        console.log(JSON.parse(httprequest.response));
        callback()
    }
})
})
}

function demo2(){
    return new Promise(function(callback){
        var httprequest=new XMLHttpRequest()
        httprequest.open('GET','https://forkify-api.herokuapp.com/api/search?q=pizza')
        httprequest.send()
        var allposts=[]
        console.log("=====demo2=====");
        httprequest.addEventListener('readystatechange',function(){
            if(httprequest.readyState==4){
                allposts=JSON.parse(httprequest.response).recipes
                console.log(JSON.parse(httprequest.response));
                callback()
            }
        })
    })
}

function demo3(){
    return new Promise(function(){
        console.log("=====demo3=====");
    })
}

async function test(){
    await demo1()
    await demo2()
    await demo3()
}
test()

```

The output will be (demo1-demo2-demo3)

Fetch

- The Fetch API interface allows web browser to make HTTP requests to web servers.

😊 No need for XMLHttpRequest anymore.

EX-

```
async function test(){
  var data = await fetch('https://forkify-api.herokuapp.com/api/search?q=pizza')
  var result= await data.json()
  console.log(result);
}
test()
```

“use strict”

- used to make the language more strict about data types and other stuff

EX-

```
"use strict"
console.log(x)
let x=10 // in this code it will print an error
```

TRY.....Catch “Exception handling”

- used to handle errors and bugs in JS
- EX-

```
"use strict"
try{
  console.log(x);
  let x=3.14
}
catch(error){
  console.log("ERORR");
}
console.log("done");
```

Default parameters

- used to initialize the variables in the function in case there are no parameters have added while calling it

EX-

```
function calcsalary(name="system user",age = 0,salary=2000){
  console.log(`your name is ${name} and your age is ${age} and your salary is ${salary+44}`);
}
calcsalary()
```

Destructuring

- it's used to access a nested object by a more professional way

EX-

```
let user={
  name:"ahmed",
  age:30,
  email:"a@gmail.com",
  x:{
    y:{
      z:{
        name1:"name1",
        name2:"name2",
        name3:"name3",
      }
    }
  }
}
let{name1,name2,name3}=user.x.y.z
console.log(name1);
console.log(name2);
console.log(name3);
```

Arrow Function

- The handling of `this` is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of `this`.

In regular functions the `this` keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the `this` keyword *always* represents the object that defined the arrow function.

Let us take a look at two examples to understand the difference.

Both examples call a method twice, first when the page loads, and once again when the user clicks a button.

The first example uses a regular function, and the second example uses an arrow function.

The result shows that the first example returns two different objects (window and button), and the second example returns the window object twice, because the window object is the "owner" of the function

EX-

```
let user={
  name:"ahmed",
  age:30,
  email:"a@gmail.com",
  foo:function(){
    let demo={()=>{
      console.log(this);
    }}
    demo()
  }
}
user.foo()
```

Spread operator

EX-

```
function sum(x,y,a,b,c){
  console.log(x+y+a+b+c);
}
var arr=[1,2,3,4,5]
sum(...arr)
```

OOP (object oriented programming)

There are 2 types of OOP :-

- Proto type-baese
- Class -based

Proto type-based

EX-

```
let Person=function(Pname,Page,Pgender){
  this.name=Pname
  this.age=Page
  this.gender=Pgender
  this.demo=function(){
    console.log("Demo");
  }
}

let obj = new Person('ahmed',24,'male')
obj.demo()
console.log(obj);
```

Class Based

```
class Person{
  constructor(Pname,Page,Pgender){
    this.name=Pname
    this.age=Page
    this.gender=Pgender
  }
  display(){
    console.log(this.name+" "+this.age+" "+this.gender);
  }
}

let obj = new Person('ali',20,'male')
obj.display()
```

- Note - The performance in Proto type-Based is better than Class - based ,but the popular way is Class - based

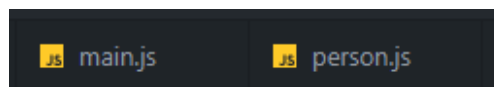
Inheritance

```
class Person{
  constructor(Pname,Page,Pgender){
    this.name=Pname
    this.age=Page
    this.gender=Pgender
  }
  display(){
    console.log(this.name+" "+this.age+" "+this.gender);
  }
}
class Eng extends Person{
  constructor(Pname,Page,Pgender,deperment){
    super(Pname,Page,Pgender)
    this.deperment=deperment
  }
}

let obj = new Person('ali',20,'male')
let obj2 = new Eng('mahmoud',20,'male','cs')
console.log(obj2);
obj.display()
```

JS module

- it used to make a several files of Js work together



if we take a look at this photo we will notice that we have more than one file of js
to do something like that we have to add a property in HTML “type=module”

```
<script type="module" src="JS/main.js"></script>
```

- second,in Js files we have to do this:-

EX-

```
//main file
import Person from'./person.js'
class Eng extends Person{
  constructor(Pname,Page,Pgender,deperment){
```



```

        super(Pname, Page, Pgender)
        this.deperment=deperment
    }
}

let obj = new Person('ali',20,'male')
let obj2 = new Eng('mahmoud',20,'male','cs')
console.log(obj2);
obj.display()

```

```

//in export or (external) file
export default class Person{
    constructor(Pname,Page,Pgender){
        this.name=Pname
        this.age=Page
        this.gender=Pgender
    }
    display(){
        console.log(this.name+" "+this.age+" "+this.gender);
    }
}

```

Regex

```

[abc]-> for digits and it means a or b or c
(ahmed|mahmoud) -> for words and it means ahmed or mahmoud
[^abc] -> means not in another way all the letters are allowed except a,b,c
^ for the start of string
$ for the end of string
? zero or one
* zero or more
+ one or more

```