

JAVASCRIPT BASICO

▼ Variables:

Espacio en memoria donde yo guardo valores .Var es una palabra reservada del lenguaje

```
var curso = 'Aprendiendo JavaScript'
```

Undefined: indefinido

```
var minumero = 6  
var r
```

Paso de referencia

la referencia de r es undefined. Cuando no le digo que referencia debe de guardar siempre sera undifined

```
curso = r
```

```
> curso = r  
< undefined  
  
> r  
< undefined
```

Error de referencia

```
curso = bootcamp
```

```
> curso = bootcamp
Uncaught ReferenceError: bootcamp is not defined
    at <anonymous>:1:1
```

Al redeclarar una variable ocurre un undefined.

▼ Primitivas String y number Parte 2

Si no quiero redefinir el valor de una variable, usare una constante

```
const curso = 'Ingenieria de Software'
```

```
const curso = 'Ingenieria de Software'
var curso = 6

SyntaxError: Identifier 'curso' has already been declared. (2:4)

   1 | const curso = 'Ingenieria de Software'
>  2 | var curso = 6
     |     ^
     |
   3 |
   4 |
   5 |
```

Tipo de variable LET

```
let curso = 'Ingenieria de Software'
curso
curso = 6
curso

'Ingenieria de Software'
6
6
```

Puedo redefinir la variable con let

Existen 3 formas de declarar una variable:

- Let
- var

- const

Variables para almacenar números / operaciones con numeros

```
let numero1 = 1
let numero2 = 2
let total = numero1 + numero2
total
```

3

Concatenación

```
let string1 = 'hola'
let string2 = 'mundo'
string1 + string2
```

'holamundo'

String Literal

```
let string1 = 'hola'
let string2 = 'mundo'
let resultado = `${string1} | ${string2}`
resultado
```

'hola mundo'

JavaScript | MDN

JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo (just-in-time) con funciones de primera clase. Si bien es más conocido como un

 <https://developer.mozilla.org/es/docs/Web/JavaScript>



MDN Web Docs
moz://a

▼ Primitivas String y number Parte 3

Existen 7 tipos de variables en JS, y es bajamente tipado

```
var numero = 6
undefined
numero = 'palabra'
'palabra'
```

Operador Typeof

```
numero = 'palabra'    'string'
typeof numero
```

En JS no hay diferencia entre tipos de números, todos son numeros.

```
var numero = 6          'number'
typeof numero
var numero = 6.7        'number'
typeof numero
var numero = 585164651348648 'number'
typeof numero
```

▼ Object y null

```
let string = 'texto'
typeof string
let number1 = 1
typeof number1
```

'string'

'number'

Que otros valores pueden existir en JavaScript a parte de numeros, cadenas..?

Pueden haber objetos, cuya notación es con llaves

```
let object = {}
```

```
let objeto1 = {}
typeof objeto1
```

'object'

Los objetos pueden tener variables por dentro, a estas variables les llamamos propiedades.

```
let objeto1={propiedad1:'text1'}
```

Para acceder a las propiedades de un objeto podemos acceder con la notación de punto

```
let objeto1 = {propiedad1:'texto1'}
objeto1.propiedad1
```

'texto1'

propiedad1 tiene tipo, así que también puedo saberlo

```
let objeto1 = {propiedad1:'texto1'}
objeto1.propiedad1
typeof objeto1.propiedad1
```

'texto1'
'string'

Al objeto1 puedo agregarle números

```
let objeto1 = {propiedad1:'texto1' , prop2: 2}
objeto1.prop2
typeof objeto1.prop2
```

2
'number'

Yo puedo tener objetos dentro de un objeto

```
let objeto1 = {propiedad1:'texto1' ,
               prop2: 2,
               prop3: {}
}
objeto1.prop3
typeof objeto1.prop3
```

{}
'object'

Primitivas en JS

- Objetos
- Cadenas de texto
- Números

También hay otros tipos de primitivas

- **Valor null:** La característica más particular de null, es que typeof null retorna object. Es un bug

```
null
typeof null
```

null
'object'

- Undefined:

```
undefined  
typeof undefined
```

'undefined'

- Booleanos:

```
let variable1 = undefined  
let variable2 = true  
let variable3 = false  
let variable4  
  
typeof variable1  
typeof variable2  
typeof variable3  
typeof variable4
```

'undefined'
'boolean'
'boolean'
'undefined'

Cuando una variable no esta definida su variable es *undefined* por defecto.

```
typeof typeof variable4
```

'string'

- Symbol():

```
typeof Symbol()
```

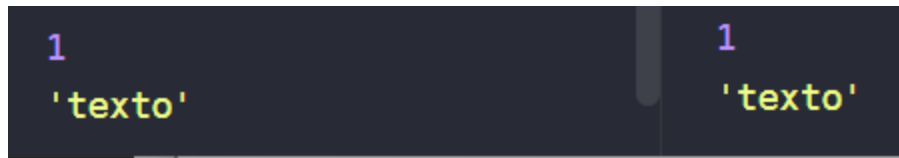
'symbol'

Expresiones

Puede ser mas fácil de identificar cuando estoy declarando una variable y de una vez la estoy inicializando.

```
let var  
const nombreVariable = 1
```

Una **expresión** es el lado derecho del igual, es algo que el lenguaje me entiende pero no puedo ejecutar operaciones con esa línea de Código.



```
1  
'texto'
```

▼ JSON y el objeto window

¿Que es un JSON?

Notación de objeto estándar para casi todas las aplicaciones.

JavaScript Object Notation

Los objetos son un tipo de valor muy importante dentro de los lenguajes de programación.


```
let persona = {  
  nombre: 'Monique M',  
  apellido : 'Matamoros',  
  email: 'tiff@gmail.com'  
}  
  
persona.nombre      'Monique M'  
persona.apellido    'Matamoros'  
persona.email       'tiff@gmail.com'
```

Hay dos objetos muy importantes que el motor de JS que vive dentro de un navegador tiene

1. **Window**

window tiene un montón de propiedades que son necesarias para el desarrollador en el lado del navegador.

Ejemplo: Document, History, navigator y muchos mas.

```

> window
< Window {0: Window, 1: Window, 2: Window, 3: Window, 4: global, window: Window, self:
  Window, document: document, name: '', location: Location, ...} ⓘ
  ▶ 0: Window {window: Window, self: Window, document: document, name: '', location: Lo
  ▶ 1: Window {window: Window, self: Window, document: document, name: '', location: Lo
  ▶ 2: Window {window: Window, self: Window, location: Location, closed: false, frames:
  ▶ 3: Window {window: Window, self: Window, document: document, name: '', location: Lo
  ▶ 4: global {window: global, self: global, location: {...}, closed: false, frames: glob
  ▶ $: f (e,t)
  ▶ AWIN: {Tracking: {...}, sProtocol: 'https://', iScriptCount: 0, tldDomains: Array(267
  DEBUG: true
  ▶ DataLayer: {events: {...}, shared: {...}, rmuid: 'f6a62c2a-dc71-428f-a8fb-2fd843108fec'
  ▶ GetParams: f e(t)
    GoogleAnalyticsObject: "ga"
  ▶ ImpactRadiusEvent: f Event(a,v,c)
  ▶ Intercom: (...t)=> {...}
  ▶ KI: {l8i: {...}, t8: f, yq: f, z3: f, gn: f, ...}
  ▶ OneTrust: {AllowAll: f, BlockGoogleAnalytics: f, Close: f, getCSS: f, GetDomainData
    OneTrustStub: null
    OnetrustActiveGroups: ",C0003,C0005,C0004,C0001,C0002,"
  ▶ Optanon: {AllowAll: f, BlockGoogleAnalytics: f, Close: f, getCSS: f, GetDomainData:
    OptanonActiveGroups: ",C0003,C0005,C0004,C0001,C0002,"
  ▶ OptanonWrapper: f OptanonWrapper()
  ▶ PR: {PR_ATTRIB_NAME: 'atn', PR_ATTRIB_VALUE: 'atv', createSimpleLexer: f, registerL
    PR_SHOULD_USE_CONTINUATION: true
  ▶ PX: {Events: {...}, ClientUid: '8d852770-91fd-11ec-a6c6-217b5c3788ce', setChallenge:
  ▶ PXZHH9f9x0: {Events: {...}, ClientUid: '8d852770-91fd-11ec-a6c6-217b5c3788ce', setCh
  ▶ PluginDetect: {version: '0.9.1', name: 'PluginDetect', openTag: '<', addPlugin: f,
  ▶ Pusher: f e(t,r)
  ▶ SENTRY_RELEASE: {id: '452f38a58c018a65cbf57b39bd758c47085f82b0'}
  ...
  : Console What's New x

```

```

> let persona = {
  nombre: 'Monique M',
  apellido : 'Matamoros',
  email: 'tiff@gmail.com'
}
< undefined
> persona
< {nombre: 'Monique M', apellido: 'Matamoros', email: 'tiff@gmail.com'} ⓘ
  apellido: "Matamoros"
  email: "tiff@gmail.com"
  nombre: "Monique M"
  ▶ [[Prototype]]: Object
>

```

Propiedad width

```
> window.width  
◀ 1536
```

Para ver cuanto ancho mide la parte visual donde se renderiza la pagina.

Su propio ancho

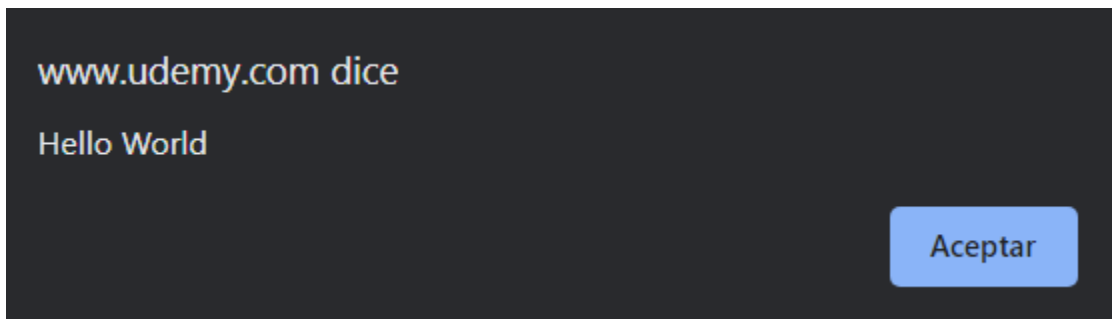
```
> window.innerWidth  
◀ 767
```

window es el valor por defecto en donde esta parado la consola de JS cuando la abrimos.

▼ Window, alert, document

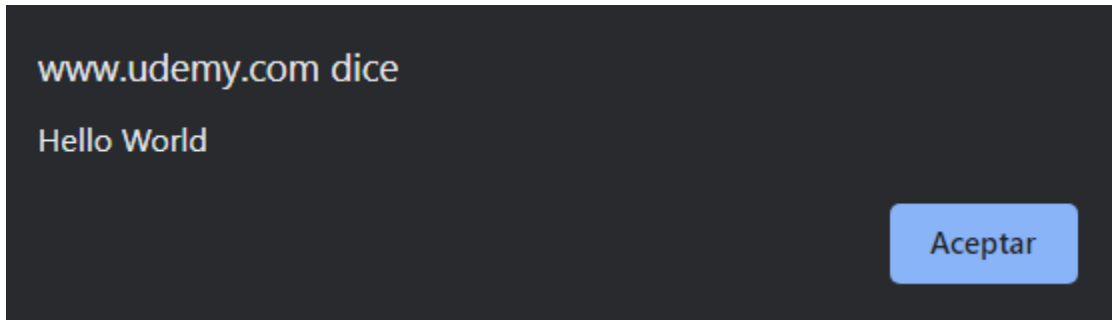
Hay una propiedad que es muy usada y sirve para lanzar un mensaje, es la propiedad alert. Hare mi hello world con alert

```
alert('Hello Word')
```



Nos entrega una caja de texto con el mensaje que le pasamos, y esa es a su vez algo de window

```
window.alert('Hello World')
```

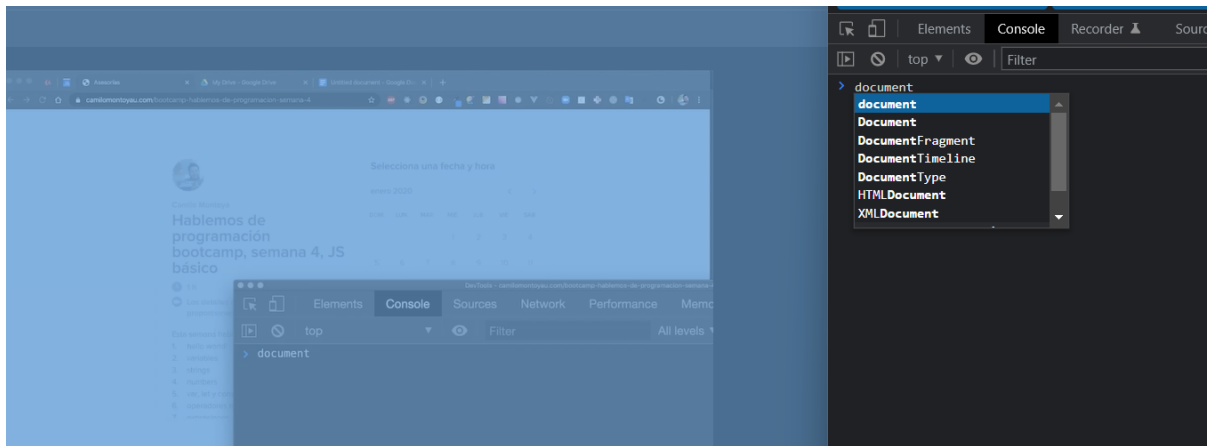


SCOPE

Ámbito, contexto. Va a ser super importante porque queremos saber el SCOPE, el ámbito o el contexto en donde estamos parados en el momento.

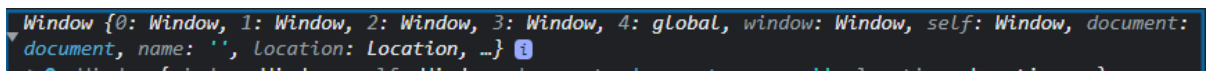
Objeto Document

Representación de todo el HTML que tiene la pagina dentro de un objeto de JS.
Document normalmente solo lo vamos a encontrar en navegadores.



Document es una API es un recurso que los navegadores le agregan al JS.

Document esta dentro de window



```
> window.document === document  
< true
```

▼ Nesting (anidamiento) y scope (alcance)

Document:

Tiene una particularidad y es que tiene mas objetos dentro de si. Esa particularidad se llama anidamiento, en ingles **NESTING**.

Nesting: no solo tiene que ver con objetos, si no tambien con bloques de código.

Anidamiento es un nivel de profundidad que tenemos de un contexto mas grande hacia uno mas pequeño.

```

> document
<  ▼ #document
    <!DOCTYPE html>
    <html lang="es-es">
    ▶ <head>...</head>
    ▶ <body id="es" class="udemy">...</body> flex
    </html>

> console.log(document)
  ▼ #document
    <!DOCTYPE html>
    <html lang="es-es">
    ▶ <head>...</head>
    ▶ <body id="es" class="udemy">...</body> flex
    </html>

<  undefined

> console.dir(document)
  ▼ #document 1
    ▶ close: f close()
    ▶ createElement: f ()
    ▶ evaluate: f ()
    ▶ getElementById: f ()
    ▶ getElementsByName: f ()
    ▶ getElementsByTagName: f ()
    ▶ location: Location {ancestorOrigins: DOMStringList, href: 'https://www.udemy.com/course/fullstack-js-en-espanol/learn/lecture/18426666#questions/13412798'}
    ▶ open: f open()
    ▶ querySelector: f ()
    ▶ querySelectorAll: f ()
    ▶ vdata1645327872569: 120
    ▶ write: f write()
    ▶ writeln: f writeln()
    ▶ __sentry_instrumentation_handlers__: {click: {...}}
    ▶ _reactListenersID742597280569536: 0
    ▶ URL: "https://www.udemy.com/course/fullstack-js-en-espanol/learn/lecture/18426666#questions/13412798"
    ▶ activeElement: video#playerId__22835968--32_html5_api.vjs-tech
    ▶ adoptedStyleSheets: []
    ▶ alinkColor: ""
    ▶ all: HTMLAllCollection(1212) [html, head, title, script.optanon-category-C0002-C0004, script.optanon-category-C0005-C0006, ...]
    ▶ anchors: HTMLCollection []

```

```

{
  let i = 'Hola'
  let u = 'otra cosa'
  {
    const r = 3
    var j = 18
    {
      u = 5
    }
  }
}

```

5

Diferencias entre let , var y const

- Una de las diferencias es el bloque de código en donde estas existen

```

var z = 'soy la primera var'
var j = 'y'
{
  let i = 'Hola'
  let u = 'otra cosa'
  {
    const r = 3
    var j = 18
    {
      u = 5
    }
  }
}

```

var nos permite hacer una doble declaración, no es como const y let.

Ejemplo de USOS

Si quiero ver el valor de x, seria posible mostrarlo porque forma parte del bloque principal, pero si quiero ver el valor de i no puedo.

```
var z = 'soy la primera var'
var x = 'y'
{
  let i = 'Hola'
  let u = 'otra cosa'
  {
    const r = 3
    var j = 18
    {
      u = 5
    }
  }
}
x
i
```

ReferenceError: i is not defined

5
'y'

Sin embargo si cambio la el tipo de variable de i , de let a var si se podria

```
var z = 'soy la primera var'
var x = 'y'
{
  var i = 'Hola'
  let u = 'otra cosa'
  {
    const r = 3
    var j = 18
    {
      u = 5
    }
  }
}
x
i
```

5
'y'
'Hola'

si la cambio por const, no me dejaría.

```
var z = 'soy la primera var'
var x = 'y'
{
  const i = 'Hola'
  let u = 'otra cosa'
  {
    const r = 3
    var j = 18
    {
      u = 5
    }
  }
}
x
i
```

ReferenceError: i is not defined

5
'y'

la variable **i** no esta al alcance del **SCOPE** de afuera. Por que esta en el interior.

Const y let : se mantienen garbage collector, limpia. Ya que la variable no se usara mas fuera del alcance que tiene.

SCOPE = alcance

var se convierte en una variable global.

Lo ideal es usar const y let.

▼ Funciones

Estructura:

```
function nombre_funcion(param1, param2){
  const resultado = param1+ param2;
  return resultado;
}
```

```
nombre_funcion(1,2) //Call the function
```

```
function nombre_funcion(param1, param2)
{
    const resultado = param1+ param2;
    return resultado;
}

nombre_funcion(1,2)
```

3

```
function nombre_funcion(param1, param2){
    console.log(arguments)
    const resultado = param1+ param2;
    return resultado;
}

nombre_funcion(1,2)

nombre_funcion(5,5)

{
  '0': 1,
  '1': 2,
  length: 2,
  callee: f nombre_funcion(),
  __proto__: {+}
}

{
  '0': 5,
  '1': 5,
  length: 2,
  callee: f nombre_funcion(),
  __proto__: {+}
}

3
10
```

<pre> function sumar(numeros, r) { console.log(arguments) const resultado = numeros.num1 + numeros.num2; return resultado; } const r = sumar({num2:2,num1:1}) const z = sumar({num1:5, num2:6}) const y = sumar({num1:r, num2:z}) const a = sumar({num1:z, num2:z}) a </pre>	<pre> [Arguments] { '0': { num2: 2, num1: 1 } } [Arguments] { '0': { num1: 5, num2: 6 } } [Arguments] { '0': { num1: 3, num2: 11 } } [Arguments] { '0': { num1: 11, num2: 11 } } 22 </pre>
---	--

Para JS es indiferente si declaramos una variable mas en los parametros de la funcion y no lo usamos.

▼ Métodos e iteradores

Métodos:

Son funciones que pertenecen a objetos.

Ejemplo: Metodo para obtener los nombres de una persona.

```

const persona = {
  nombre: 'Monique',
  apellido: 'Matamoros',
  id: 5,
  nombreCompleto : function(){
    return `${this.nombre} ${this.apellido}`
  }
}

//llamado de la funcion
persona.nombreCompleto()

```

<pre>const persona = { nombre: 'Monique', apellido: 'Matamoros', id: 5, nombreCompleto : function(){ return `\${this.nombre} \${this.apellido}` } } //llamado de la funcion persona.nombreCompleto()</pre>	<pre>'Monique Matamoros'</pre>
---	--------------------------------

Arrays

```
const miArray = [1,2,3, 'hola', 28, {a:1}]

miArray
```

<pre>const miArray = [1,2,3, 'hola', 28, {a:1}] miArray</pre>	<pre>[1, 2, 3, 'hola', 28, { a: 1 }]</pre>
--	--

El array es de tipo object, pero este object es iterable

<pre>typeof miArray</pre>	<pre>'object'</pre>
---------------------------	---------------------

Crear una variable iterador donde guardare el array

```
var iterador = miArray[Symbol.iterator]();
```

<pre>var iterador = miArray[Symbol.iterator]();</pre>	
<pre>iterador.next().value</pre>	1
<pre>iterador.next().value</pre>	2
<pre>iterador.next().value</pre>	3
<pre>iterador.next().value</pre>	'hola'
<pre>iterador.next().value</pre>	28
<pre>iterador.next().value</pre>	{ a: 1 }

para saber si podemos iterar algo, debemos de verificar si tenemos el symbol iterator.

```
▼ Symbol(Symbol.iterator): f values()  
  length: 0  
  name: "values"  
  arguments: (...)  
  caller: (...)  
  ► [[Prototype]]: f ()  
  ► [[Scopes]]: Scopes[0]  
  Symbol(Symbol.toStringTag): "Set"  
  ► get size: f size()  
  ► [[Prototype]]: Object
```

El iterador en JS ya viene implícito y eso es porque el motor que corre JS, está hecho en C++. Desde C++ se está creando ese iterador.

▼ Arrays, iterables y Array.length

Dentro de las propiedades de un Array está **Length = Longitud**

```
> miArray
< ▼ (6) [1, 2, 3, 'hola', 28, {...}] ⓘ
  0: 1
  1: 2
  2: 3
  3: "hola"
  4: 28
  ▶ 5: {a: 1}
    length: 6
  ▶ [[Prototype]]: Array(0)
```

Los Arrays ya tienen métodos implícitos, uno de ellos es push

```
miArray.push(10) //Agregar al array un elemento nuevo que le de
//El retorno del array push es el length
```

```
> miArray.push(10)
< 7
```

Array.prototype.length

La propiedad `length` de un objeto que es una instancia de tipo Array establece o devuelve la cantidad de elementos en esa matriz. El valor es un entero sin signo de 32 bits que siempre es numéricamente mayor que el índice más alto en la matriz.

```
const clothing = ['shoes', 'shirts', 'socks', 'sweaters'];

console.log(clothing.length);
// expected output: 4
```