

# JAVASCRIPT INTERMEDIO

## Booleans y condicionales

### Booleano

El objeto `Boolean` es un objeto contenedor para un valor booleano.

<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>
<code>!""</code>	<code>true</code>
<code>!'false'</code>	<code>false</code>
<code>!!'false'</code>	<code>true</code>
<code>true + true</code>	<code>2</code>
<code>!!1</code>	<code>true</code>
<code>!0</code>	<code>true</code>

### Ejemplo:

```
const persona = {
  nombre: 'Billie',
  apellido: 'Eilish',
  edad: 17
}
const persona2 = {
  nombre: 'Aurora',
  apellido: 'Aksnes',
  edad: 21
}
if(persona.edad < 18){
  console.log('no puede tomar cerveza')
}else{
  console.log('si puede tomar cerveza')
}

if(persona2.edad < 18){
  console.log('no puede tomar cerveza')
}else{
  console.log('si puede tomar cerveza')
}
```

Veremos como retorna un valor que sale de una expresion

```
const persona2 = {  
  nombre: 'Aurora',  
  apellido: 'Aksnes',  
  edad: 21  
}  
  
persona2.edad >= 18    true
```

```
const persona2 = {  
  nombre: 'Aurora',  
  apellido: 'Aksnes',  
  edad: 21  
}  
  
persona2.edad <= 18    false
```

```
const persona = {  
  nombre: 'Billie',  
  apellido: 'Eilish',  
  edad: 17  
}  
  
const persona2 = {  
  nombre: 'Jose',  
  apellido: 'Banguera',  
  edad: 18  
}  
  
if (persona2.edad > 18) {  
  console.log('toma cerveza')  
} else if (persona.nombre === 'Billie') {  
  console.log('canta')  
} else {  
  console.log('no cantan y no toman cerveza')  
}
```

El triple igual en JS además de verificar el valor, verifica el tipo.

```

    nombre: 'María',
    apellido: 'Eillish',
    edad: 17
  }

  const persona2 = {
    nombre: 'Jose',
    apellido: 'Banguera',
    edad: 19
  }

  if(persona.apellido === 'sh' && persona2.nombre === 'Jose') {
    console.log('Hola Eillish y Jose')
  } else {
    console.log('hola extraños')
  }

```

## Loops: for, while y forEach

### Loops:

Formas de repetir tareas

#### Array Push

El método `push()` añade uno o más elementos al final de un array y devuelve la nueva longitud del array.

```

let miArray = []
miArray.push()

```

0

```

let miArray = []
miArray.push(1)

```

1

<code>let miArray = []</code>	
<code>miArray.push(1)</code>	1
<code>miArray.push(2)</code>	2
<code>miArray.push(5)</code>	3
<code>miArray</code>	[ 1, 2, 5 ]

La tarea anterior de añadir elementos al array se hace muy repetitiva, para ello utilizaremos loops.

```
let miArray =[];
for(let i=0; i<10; i=i+1){
  miArray.push(i);
}

miArray
```

```
> let miArray =[];
  for(let i=0; i<10; i=i+1){
    miArray.push(i);
  }

  miArray
< ▼ (10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 1
  0: 0
  1: 1
  2: 2
  3: 3
  4: 4
  5: 5
  6: 6
  7: 7
  8: 8
  9: 9
  length: 10
  ► [[Prototype]]: Array(0)
```

## While

```
let miArray2= []
let control = 0;
while(control< 10){
  miArray2.push(control);
```

```
control = control+1
}
miArray2
```

```
let miArray2= []
let control = 0;
while(control< 10){
  miArray2.push(control);
  control = control+1
}
miArray2
```

10  
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

Otro Ejemplo de uso de while

```
let miArray2= []
let control = 0;
while(control< 100){
  miArray2.push(control);
  if(control%5===0){
    control = control + 2;
  }else{
    control = control+1
  }
}
miArray2
```

100  
[ 0, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 22, 23, 24, 25, 27, 28, 29, 30, 32, 33, 34, 35, 37, 38, 39, 40, 42, 43, 44, 45, 47, 48, 49, 50, 52, 53, 54, 55, 57, 58, 59, 60, 62, 63, 64, 65, 67, 68, 69, 70, 72, 73, 74, 75, 77, 78, 79, 80, 82, 83, 84, 85, 87, 88, 89, 90, 92, 93, 94, 95, 97, 98, 99 ]

typeof Array

typeof Array

'function'

```
typeof miArray2 | 'object'
```

## ForEach

El método `forEach()` ejecuta la función indicada una vez por cada elemento del array.

El `forEach` no devuelve nada, si queremos ver la salida debemos poner un `console log`

**JavaScript Demo: `Array.forEach()` a continuacion:**

```
const array1 = ['a', 'b', 'c'];

array1.forEach(element => console.log(element));

// expected output: "a"
// expected output: "b"
// expected output: "c"
```

```
let miArray = ['hola' , 'buen dia', 'adios'];

function convertirMayusculas(texto){
  return texto.toUpperCase();
}
convertirMayusculas('fegegregegre');

// expected output:FEGEGREGREGRE
```

El `toUpperCase()` método devuelve el valor convertido en mayúsculas de la cadena que realiza la llamada.

<pre>let miArray = ['hola' , 'buen dia', 'adios'];  function convertirMayusculas(texto){   console.log(texto.toUpperCase()); }  miArray.forEach(convertirMayusculas)</pre>	<pre>'HOLA' 'BUEN DIA' 'ADIOS'</pre>
--	--------------------------------------

<pre>let miArray = ['hola', 'buen día', 'adiós'];  function convertirMayusculas(actual, indice, arrayCompleto) {   const respuesta = actual.toUpperCase()   console.log(respuesta)   console.log(indice)   console.log(arrayCompleto) }  let miVar = miArray.forEach(convertirMayusculas) miVar</pre>	<pre>'HOLA' 0 [ 'hola', 'buen día', 'adiós' ] 'BUEN DÍA' 1 [ 'hola', 'buen día', 'adiós' ] 'ADIÓS' 2 [ 'hola', 'buen día', 'adiós' ]</pre>
---	--

## Array.map

El método `map()` crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

```
var numbers = [1, 5, 10, 15];
var doubles = numbers.map(function(x) {
  return x * 2;
});
```

<pre>var numbers = [1, 5, 10, 15]; var doubles = numbers.map(function(x) {   return x * 2; }); console.log(doubles)</pre>	<pre>[ 2, 10, 20, 30 ]</pre>
---	------------------------------

son parecidos al `forEach` pero el `forEach` no entrega ningun retorno

---

## Array.filter

Me devuelve un array con los elementos cuya condicion del retorno de la funcion que le pase al filter sea verdadera

```
let miArray = [1,2,3,4,5,6,7,8,9]

const soloPares = (numeroActual) => numeroActual %2 ===0

let resultado = miArray.filter(soloPares)|
resultado                                     [ 2, 4, 6, 8 ]
```

---

## Arrow functions (funciones de flecha)

```
function miFuncion2(){
  console.log('ejecute miFuncion2')
}
```

```
function miFuncion(){
  console.log('1 ejecuta miFuncion')
  console.log('2 ejecute miFuncion')
  miFuncion2()
}
```

```
function miFuncion(){
  console.log('ejecute miFuncion')
}

miFuncion()
```

'ejecute miFuncion'



```

> function miFuncion(){
  console.log('ejecute miFuncion')
}
< undefined
> miFuncion
< f miFuncion(){
  console.log('ejecute miFuncion')
}
> function miFuncion(){
  console.log('ejecute miFuncion')
}
< undefined
> function miFuncion(){
  console.log('ejecute miFuncion')
  return true
}
< undefined
> miFuncion()
ejecute miFuncion
< true

```

La forma anterior es la ejecución de una función. A continuación voy a convertir las funciones en arrow functions

```

const miFuncion2 = ()=>{
  console.log('ejecuta miFuncion2')
}

```

Las **Arrow Functions** tienen otra particularidad:

```

console.log(miFuncion())

function miFuncion(){
  return 100
}

```

La segunda parte del Código en la imagen vendría siendo la primera parte, después de que JS lo compila.

```

> console.log(miFuncion())

function miFuncion(){
  return 100
}

100
< undefined
> function miFuncion(){
  return 100
}
console.log(miFuncion())

100
< undefined
>

```

Las Arrow Functions no se pueden acceder antes de su definicion

```

console.log(miFuncion())
const miFuncion = () =>{
  return 100
}

```

```

> console.log(miFuncion())

const miFuncion = ()=> {
  return 100
}

✖ ▶ Uncaught ReferenceError: miFuncion is not defined
  at <anonymous>:1:13

```

## Scope

El scope de una variable hace referencia al lugar donde esta va a vivir , o podrá ser accesible.

Podríamos decir también que scope es el alcance que determina la accesibilidad de las variables en cada parte de nuestro código.

El objeto window es un ejemplo de scope global.

```
> function miFuncion(){
    return function miFuncion2(){
        return this;
    }
}
< undefined
> miFuncion()
< f miFuncion2(){
    return this;
}
```

```
> const b = miFuncion()
< undefined
> b
< f miFuncion2(){
    return this;
}
> b()
< Window {0: Window, 1: Window, 2: Window, 3: Window, 4: global, window: Window, sel
  f: Window, document: document, name: '', location: Location, ...}
> b() == window
< true
> const c = new miFuncion()
< undefined
> c()
< Window {0: Window, 1: Window, 2: Window, 3: Window, 4: global, window: Window, sel
  f: Window, document: document, name: '', location: Location, ...}
> |
```

Ahora veremos el ejemplo donde el `scope` es `window` cuando usamos una Arrow Function

```

> function miFuncion() {
  return ()=> {
    return this;
  }
}
< undefined
> miFuncion
< f miFuncion() {
  return ()=> {
    return this;
  }
}
> miFuncion()
< ()=> {
  return this;
}
> miFuncion()()
< ▶Window {parent: Window, opener: null, top: Window, length: 1, frames: Window, ...}
> |

```

## Comportamientos de arrow functions comparados con las funciones normales

Comparación con ejemplos:

Función

```

> function miFuncion() {
  this.miFuncion2 = function(){
    return this
  }
}
< undefined
> const c = new miFuncion()
< undefined
> c
< ▼miFuncion {miFuncion2: f} ⓘ
  ▶ miFuncion2: f ()
  ▶ __proto__: Object
> c.miFuncion2
< f (){
  return this
}
> c.miFuncion2()
< ▶miFuncion {miFuncion2: f}
> |

```

Arrow Function

```

> function miFuncion3() {
  this.miFuncion4 = ()=>{
    return this
  }
}
< undefined
> miFuncion3
< f miFuncion3() {
  this.miFuncion4 = ()=>{
    return this
  }
}
> miFuncion3()
< undefined
> miFuncion3().miFuncion4
✖ ▶ Uncaught TypeError: Cannot read property 'miFuncion4' of undefined
  at <anonymous>:1:13
> miFuncion3().miFuncion4()
✖ ▶ Uncaught TypeError: Cannot read property 'miFuncion4' of undefined
  at <anonymous>:1:13

```

```

> const d = new miFuncion3()
< undefined
> d
< ▶ miFuncion3 {miFuncion4: f}
[Violation] 'message' handler took 170ms
> d.miFuncion4
< ()=>{
  return this
}
> |

```

```

> const miFuncion5 = ()=> {
  this.miFuncion6 = ()=>{
    return this
  }
}
< undefined
> miFuncion5
< ()=> {
  this.miFuncion6 = ()=>{
    return this
  }
}
> miFuncion5()
< undefined
> miFuncion5().miFuncion6
✖ ▶ Uncaught TypeError: Cannot read property 'miFuncion6' of undefined
  at <anonymous>:1:13
> |

```

Crear un objeto a partir de la función 5

```

> const e = new miFuncion5()
✖ ▶ Uncaught TypeError: miFuncion5 is not a constructor
  at <anonymous>:1:11
>

```

```

> const objeto = {
  metodo() {
    return this;
  }
}
< undefined
> objeto
< ▶ {metodo: f}
> objeto.metodo
< f metodo() {
  return this;
}

```

En este caso el this que retorna es el método

```

> objeto.metodo()
< ▼ {metodo: f}
  ▼ metodo: f metodo()
    arguments: (...)
    caller: (...)
    length: 0
    name: "metodo"
    ▶ __proto__: f ()
      [[FunctionLocation]]: VM3396:2
      ▶ [[Scopes]]: Scopes[2]
      ▶ __proto__: Object
> const objeto = {
  metodo() {
    return this;
  }
}

```

Arrow Function

```

> const objeto = {
  metodo: () => {
    return this;
  }
}
< undefined
> objeto
< ▶ {metodo: f}
> objeto.metodo
< () => {
  return this;
}
> objeto.metodo()
< ▶ Window {parent: Window, opener: null, top: Window, length: 1, frames: Window, ...}
>

```

En JavaScript podemos crear las funciones dentro de los objetos sin necesidad de poner la palabra Function.

```
> const objeto = {
  metodo: () => {
    return this;
  }
}
< undefined
> objeto
< {metodo: f}
> objeto.metodo
< () => {
  return this;
}
> objeto.metodo()
< Window {parent: Window, opener: null, top: Window, length: 1, frames: Window, ...}
> const objeto2 = {
  metodo(){
    return this
  }
}
< undefined
> objeto2.metodo
< f metodo(){
  return this
}
> objeto2.metodo()
```

Hay diferencias de una función dentro de un objeto con respecto a una Arrow Function dentro de un objeto.

```
> const objeto2 = {
  metodo(){
    return this
  }
}
< undefined
> objeto2.metodo
< f metodo(){
  return this
}
> objeto2.metodo()
< {metodo: f}
```

Hay que tener cuidado con esto

```

> objeto2.metodo()
< ▶ {metodo: f}
> objeto.metodo()
< ▶ Window {parent: Window, opener: null, top: Window, length: 1, frames: Window, ...}
>

```

Cuando utilizo Arrow Functions y cuando utilizo Funciones normales dependiendo de donde estén declaradas y de a donde las este usando esas Arrows Functions y esas funciones cambian de scope. Hay que tener eso presente.

```

> const miFuncion = () => {
  return function miFuncion2(){
    return this
  }
}
< undefined
> miFuncion
< () => {
  return function miFuncion2(){
    return this
  }
}
> miFuncion()
< f miFuncion2(){
  return this
}
> miFuncion()()
< ▶ Window {parent: Window, opener: null, top: Window, length: 1, frames: Window, ...}
>

```

## Destructuring (destructuración)

Cualidad que se le agrego a JS en ECMAScript v6

### ECMAScript v6

Es el estándar que sigue JavaScript desde Junio de 2015. Hasta ese momento la versión de JS que estábamos usando en nuestros navegadores y Node.js, era la v5.

Ahora hasta el momento vamos por ECMAScript 2022.

### Destructuración:

Es algo para trabajar con objetos y Arrays de una manera mas simple, mas elegante. La **destructuración** puede hacerse en Arrays tanto como en objetos.



Ejemplo:

```
let miObjeto = {a:1, b:2, c:'hola' ,  
               d: function(){console.log('soy una funcion')},  
               e:true}
```

```
> let miObjeto = {a:1, b:2, c:'hola' ,  
                  d: function(){console.log('soy una funcion')},  
                  e:true}  
← undefined  
> miObjeto  
← {a: 1, b: 2, c: 'hola', e: true, d: f() }  
  a: 1  
  b: 2  
  c: "hola"  
  d: f()  
  e: true  
  [[Prototype]]: Object
```

La destructuración es acceder o desacoplar ese objeto en variables más simples. Si necesito trabajar con esas propiedades en un futuro, o con algunas partes de ese objeto pues simplemente lo destructuro. De la siguiente forma:

```
> let miObjeto = {a:1, b:2, c:'hola', d: function(){console.log('soy una  
  funcion')}, e: true}  
← undefined  
> let {a,b,c,d,e} = miObjeto  
← undefined  
> miObjeto  
← {a: 1, b: 2, c: "hola", e: true, d: f()}  
> a  
← 1  
> b  
← 2  
> c  
← "hola"  
> e
```

### Destructuración de un Array.

Podemos hacer lo anterior con un Array también

```
let miArray = [1, 2, 'hola' , ()=> console.log('Soy Funcion'), true]
```

```

> let miArray = [1, 2, 'hola', ()=> console.log('Soy Funcion'), true]
< undefined
> miArray
< ▶ (5) [1, 2, 'hola', f, true]
> let [numeroUno, numeroDos, hola, unaFuncion, buleano] = miArray
< undefined
> numeroUno
< 1
> numeroDos
< 2
> hola
< 'hola'
> buleano
< true
>

```

Si yo quiero excluir un elemento del objeto puedo hacer lo siguiente

```

> let {d} = miObjeto
< undefined
> d
< f (){console.log('soy una funcion')}
> let {d, ...otros} = miObjeto
< undefined
> otros
< ▶ {a: 1, b: 2, c: 'hola', e: true}

```

```

> miArray
< ▶ (5) [1, 2, 'hola', f, true]
> let [numeroUno, ... elResto] = miArray
< undefined
> numeroUno
< 1
> elResto
< ▶ (4) [2, 'hola', f, true]
> let [uno, dos, ... restoDePropiedades] = miArray
< undefined
> uno
< 1
> dos
< 2
> restoDePropiedades
< ▶ (3) ['hola', f, true]

```

## Mutacion de un Objeto

```
> miObjeto
< ▶ {a: 1, b: 2, c: "hola", e: true, d: f}
> let miObjeto2 = miObjeto
< undefined
> miObjeto2
< ▶ {a: 1, b: 2, c: "hola", e: true, d: f}
> miObjeto2.c = "otra cosa"
< "otra cosa"
> miObjeto
< ▶ {a: 1, b: 2, c: "otra cosa", e: true, d: f}
```

# Destructuring (destructuración) clonación profunda y superficial

## Lodash

Una moderna biblioteca de utilidades de JavaScript que ofrece modularidad, rendimiento y extras

Haciendo una copia profunda de mi objeto.

```
const lodash = require('lodash')

let miObjeto = {a:1, b:2, c:'hola', d: function() {
  console.log('soy una funcion')}, e: true, f: {f1:'soy f1', f2:
  'soy f2', f3: ()=>{}}}

miObjeto

let miObjeto2 = lodash.cloneDeep(miObjeto)
miObjeto2

miObjeto.f.f1

miObjeto2.f.f1 = "he cambiado mi valor"

miObjeto.f.f1
```

```

  {
    a: 1,
    b: 2,
    c: 'hola',
    d: [Function: d],
    e: true,
    f: { f1: 'soy f1', f2: 'soy f2', f3: [Function: f3] }
  }
  {
    a: 1,
    b: 2,
    c: 'hola',
    d: [Function: d],
    e: true,
    f: { f1: 'soy f1', f2: 'soy f2', f3: [Function: f3] }
  }
  'soy f1'
  'he cambiado mi valor'
  'soy f1'
```

```
let miObjeto = {a:1, b:2, c:'hola', d: function() {console.log('soy una funcion')}, e: true, f: {f1:'soy f1', f2:'soy f2', f3: ()=>{}}}

miObjeto

let miObjeto2 = JSON.parse(JSON.stringify(miObjeto))
miObjeto2.f.f1 = "he cambiado"

miObjeto.f.f1 //no ha mutado
```

```
{
  a: 1,
  b: 2,
  c: 'hola',
  d: [Function: d],
  e: true,
  f: { f1: 'soy f1', f2: 'soy f2', f3: [Function: f3] }
}
'he cambiado'
'soy f1'
```

## Prototype (prototipo)

Los prototipos son un enlace , una propiedad de los objetos en JavaScript

```
function MiObjeto(nombre, apellido){
  this.nombre= nombre
  this.apellido = apellido
  this.getNombreCompleto = function(){
    return `${this.nombre} ${this.apellido}`
  }
}
```

```
> function MiObjeto(nombre, apellido){
  this.nombre= nombre
  this.apellido = apellido
  this.getNombreCompleto = function(){
    return `${this.nombre} ${this.apellido}`
  }
}
< undefined
> let objeto1 = new MiObjeto('Monique' , 'Matamoros')
< undefined
> objeto1
< ▶ MiObjeto {nombre: 'Monique', apellido: 'Matamoros', getNombreCompleto: f}
> MiObjeto
< f MiObjeto(nombre, apellido){
  this.nombre= nombre
  this.apellido = apellido
  this.getNombreCompleto = function(){
    return `${this.nombre} ${this.apellido}`
  }
}
```

## Comparación entre objeto y JSON

```
> let objeto1 = new MiObjeto('Monique', 'Matamoros')
< undefined
> objeto1
< ▶ MiObjeto {nombre: 'Monique', apellido: 'Matamoros', getNombreCompleto: f}
> MiObjeto
< f MiObjeto(nombre, apellido){
  this.nombre= nombre
  this.apellido = apellido
  this.getNombreCompleto = function(){
    return `${this.nombre} ${this.apellido}`
  }
}
> let objetoJson = {nombre: 'Monique', apellido: 'Montoya', getNombreCompleto(){}}
< undefined
> objetoJson
< ▶ {nombre: 'Monique', apellido: 'Montoya', getNombreCompleto: f}
>
```

## Prototype

```
> objeto1
< ▼ MiObjeto {nombre: 'Monique', apellido: 'Matamoros', getNombreCompleto: f} ⓘ
  apellido: "Matamoros"
  ▶ getNombreCompleto: f ()
    nombre: "Monique"
  ➔ ▶ [[Prototype]]: Object
    ▶ constructor: f MiObjeto(nombre, apellido)
      arguments: null
      caller: null
      length: 2
      name: "MiObjeto"
      ▶ prototype: {constructor: f}
        [[FunctionLocation]]: VM2523:1
      ▶ [[Prototype]]: f ()
      ▶ [[Scopes]]: Scopes[2]
      ▶ [[Prototype]]: Object
```

```
> function MiObjeto() {
  this.getNombreCompleto = function(){
    return `${this.nombre} ${this.apellido}`
  }
  this.setNombre = function(nombre) {
    this.nombre = nombre
  }
  this.setApellido = function(apellido) {
    this.apellido = apellido
  }
}
```

Podemos Settear los parametros de un objeto, quedan el el prototipo.

```
▶ getNombreCompleto: f ()
▶ setNombre: f (nombre)
▶ setApellido: f (apellido)
▼ __proto__:
  nombre: "vacio"
  apellido: "vacio"
  ▼ constructor: f MiObjeto()
    length: 0
    name: "MiObjeto"
    arguments: null
    caller: null
  ▼ prototype:
    nombre: "vacio"
    apellido: "vacio"
    ▶ constructor: f MiObjeto()
    ▶ __proto__: Object
  ▶ prototype . f ()
```

```
> MiObjeto.prototype.apellido = 'vacio'
< "vacio"
> MiObjeto.prototype.nombre = 'vacio'
< "vacio"
> objeto1
< ▶ MiObjeto {}
> objeto1.getNombreCompleto()
< "vacio vacio"
> objeto1.nombre
< "vacio"
>
```

```
> objeto1.setApellido('Camilo')
< undefined
> objeto1.setNombre('Montoya')
< undefined
> objeto1.getNombreCompleto()
< "Montoya Camilo"
>
```

Los prototipos son un mecanismo mediante el cual los objetos en JavaScript heredan características entre sí.

## **¿Un lenguaje basado en prototipos?**

JavaScript es a menudo descrito como un **lenguaje basado en prototipos** - para proporcionar mecanismos de herencia, los objetos pueden tener un **objeto prototipo**, el cual actúa como un objeto plantilla que hereda métodos y propiedades.

Un objeto prototipo del objeto puede tener a su vez otro objeto prototipo, el cual hereda métodos y propiedades, y así sucesivamente. Esto es conocido con frecuencia como la **cadena de prototipos**, y explica por qué objetos diferentes pueden tener disponibles propiedades y métodos definidos en otros objetos.