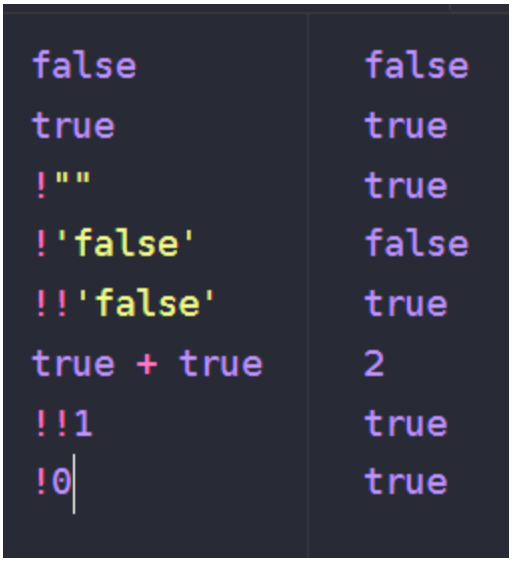
JAVASCRIPT INTERMEDIO

Booleans y condicionales

Booleano

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

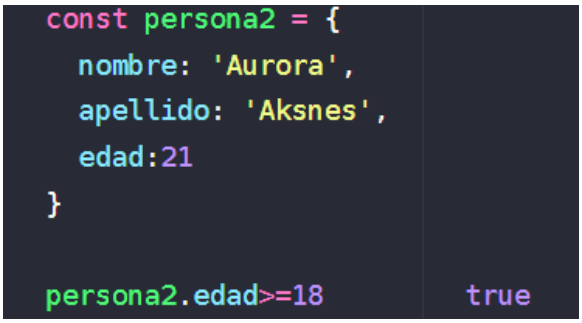


Ejemplo:

```
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')
 }
```

```
'no puede tomar cerveza'
'si puede tomar cerveza'
```

Veremos como retorna un valor que sale de una expresion



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

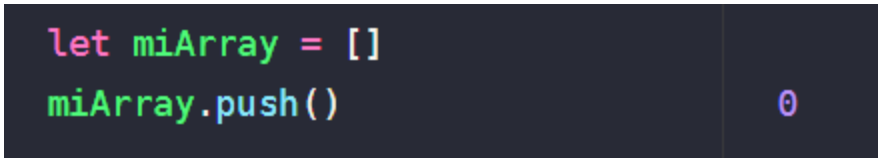
Loops: for, while y forEach

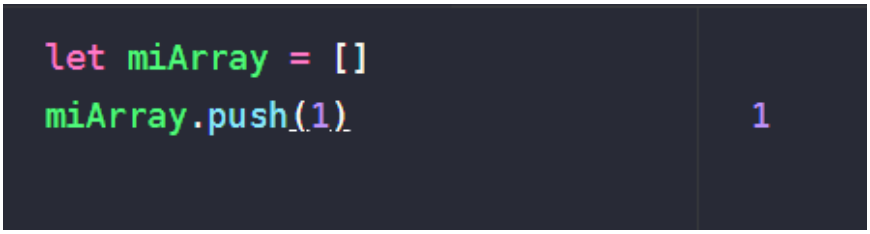
Loops:

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.

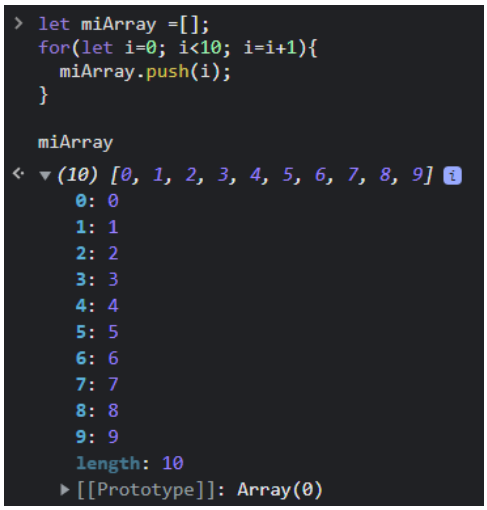




```
let miArray = []                    
miArray.push(1)                     1
miArray.push(2)                     2
miArray.push(5)                     3
miArray                             [ 1, 2, 5 ]
```

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
```



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'
```

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:FEGEGREGRGRE
```

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

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

function convertirMayusculas(texto){
  console.log(texto.toUpperCase());
}
 miArray.forEach(convertirMayusculas)
```

```
'HOLA'
'BUEN DIA'
'ADIOS'
```

```
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
```

```
'HOLA'
0
[ 'hola', 'buen día', 'adiós' ]
'BUEN DÍA'
1
[ 'hola', 'buen día', 'adiós' ]
'ADIÓS'
2
[ 'hola', 'buen día', 'adiós' ]
```

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;
});
```

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

```
[ 2, 10, 20, 30 ]
```

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 ]
```

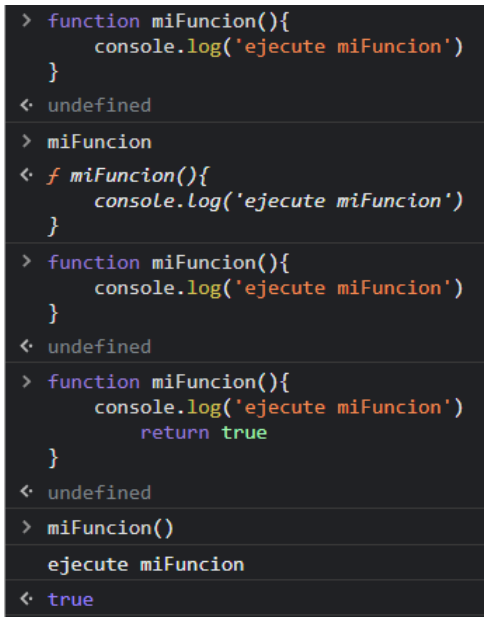
Arrow functions (funciones de flecha)

```
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'
```



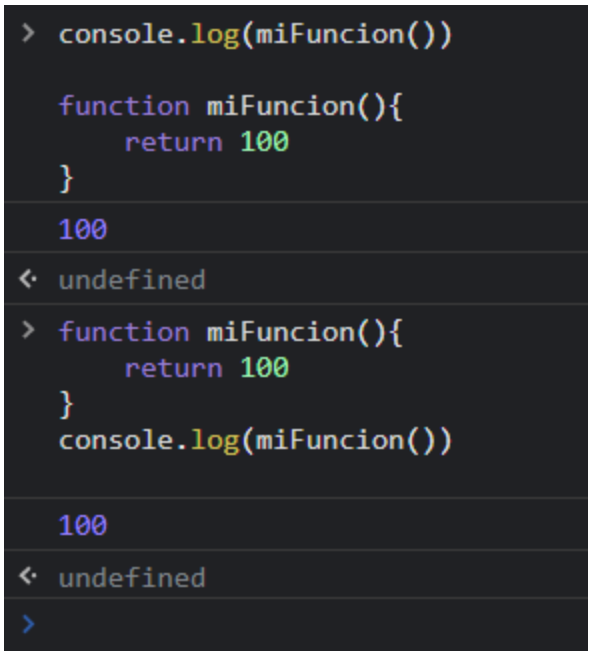
La forma anterior es la ejecución de una función. A continuación voy a convertir las funciones en arrow functions

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

Las `Arrow Functions` tienen otra particularidad:

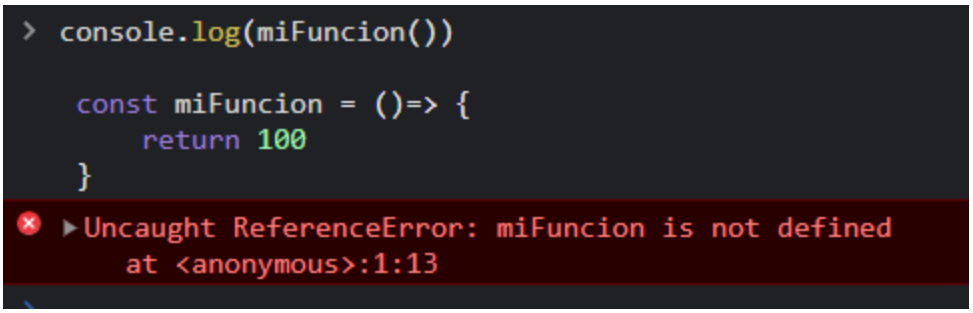
```
console.log(miFuncion())

function miFuncion(){
  return 100
}
```

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



Las Arrow Functions no se pueden acceder antes de su definicion

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

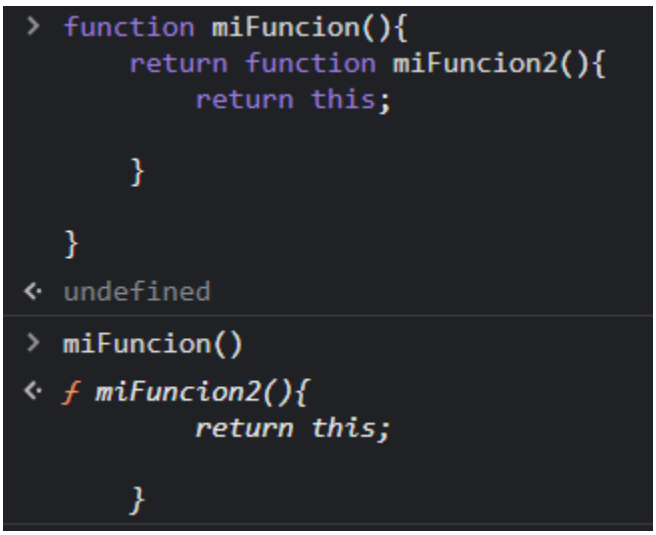


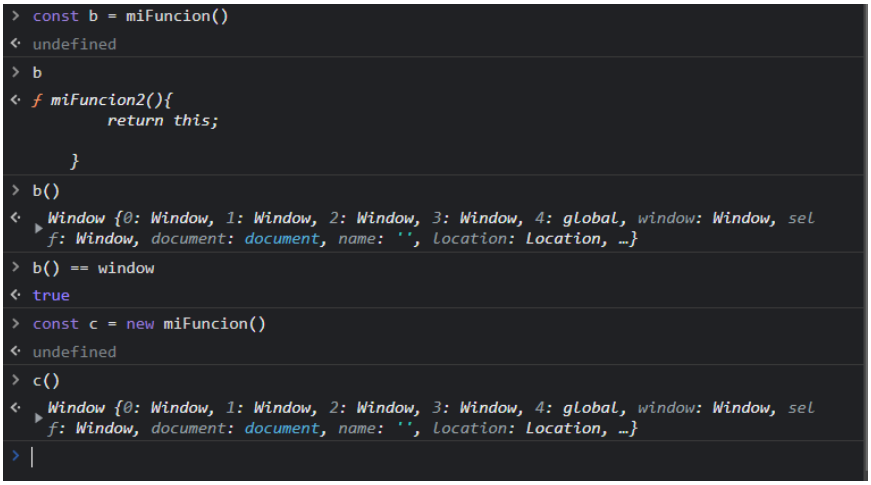
Scope

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.



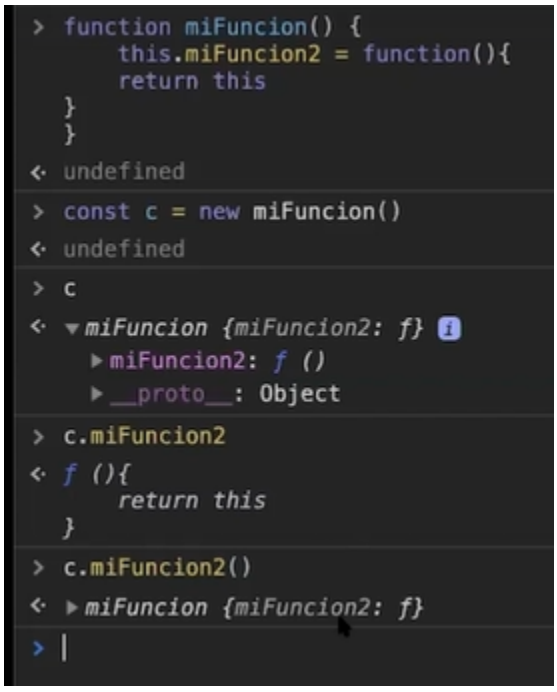


Ahora veremos el ejemplo donde el `scope` es `window` cuando usamos una Arrow Function

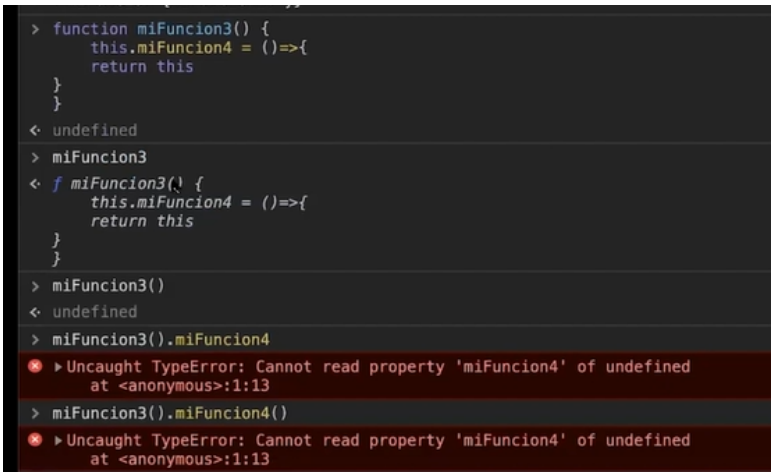
Comportamientos de arrow functions comparados con las funciones normales

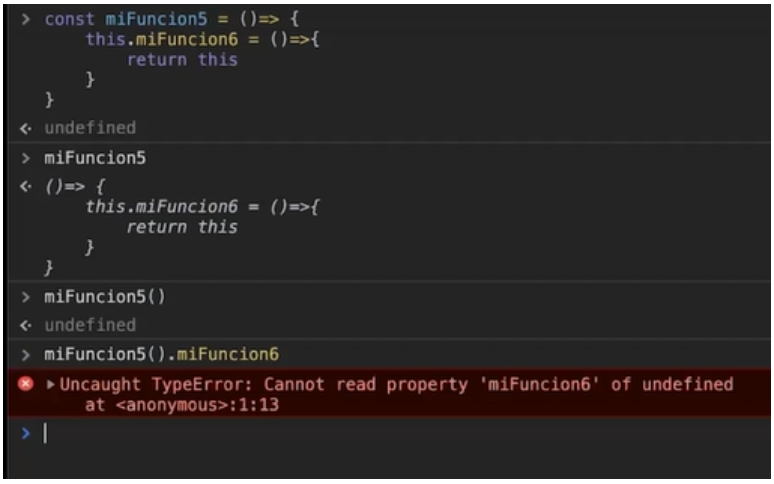
Comparación con ejemplos:

Función



Arrow Function





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: ƒ}
> objeto.metodo
← ƒ metodo() {
          return this;
      }
```

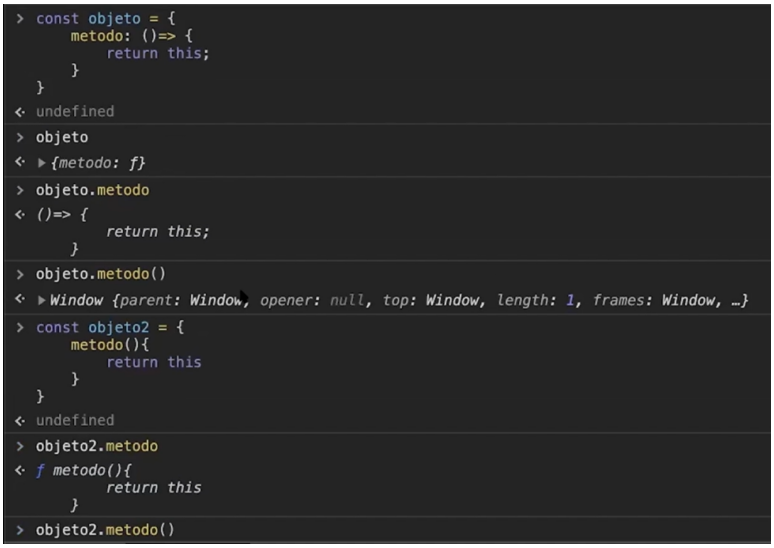
En este caso el this que retorna es el método

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

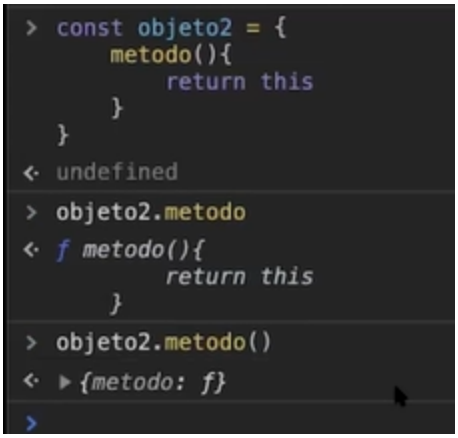
Arrow Function

```
> const objeto = {
      metodo: ()=> {
          return this;
      }
  }
← undefined
> objeto
← ▶ {metodo: ƒ}
> 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.



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



Hay que tener cuidado con esto

> 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.

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.

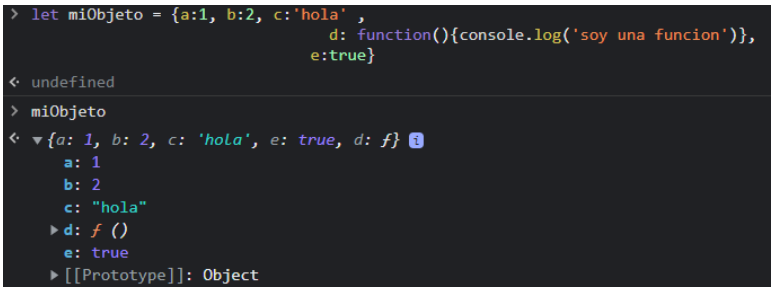
Ahora hasta el momento vamos por ECMAScript 2022.

D**estructuración:**

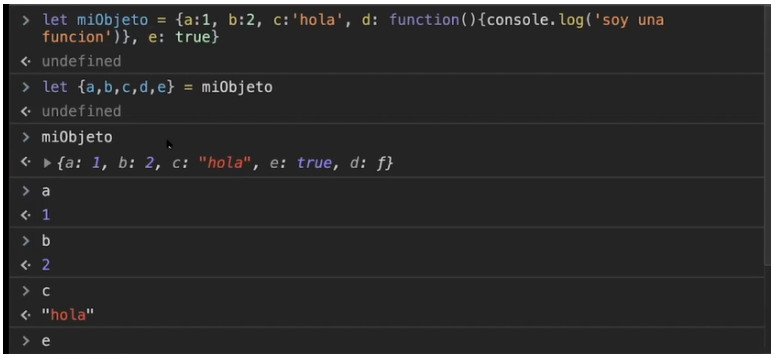
Es algo para trabajar con objetos y Arrays de una manera mas simple, mas elegante. La **destructuración** puede hacerse en Arrays tanto como en objetos.

Ejemplo:

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



La destructuracion es acceder o desaclopar ese objeto envariables mas 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:



Destructuracion de un Array.

Podemos hacer lo anterior con un Array tambien

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

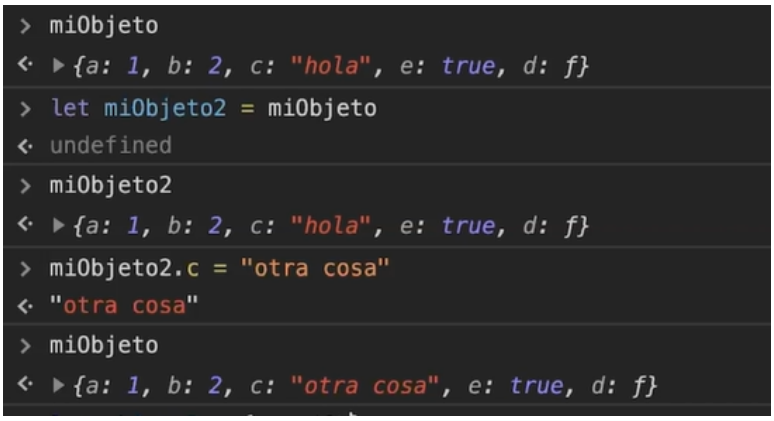
```
> let miArray = [1,  2, 'hola' , ()=> console.log('Soy Funcion'), true]
< undefined
> miArray
< ▶ (5) [1, 2, 'hola', ƒ, true]
> let [numeroUno, numeroDos , hola, unaFuncion, buleano] = miArray
< undefined
> numeroUno
< 1
> numeroDos
< 2
> hola
< 'hola'
> buleano
< true
```

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

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

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

Mutacion de un Objeto

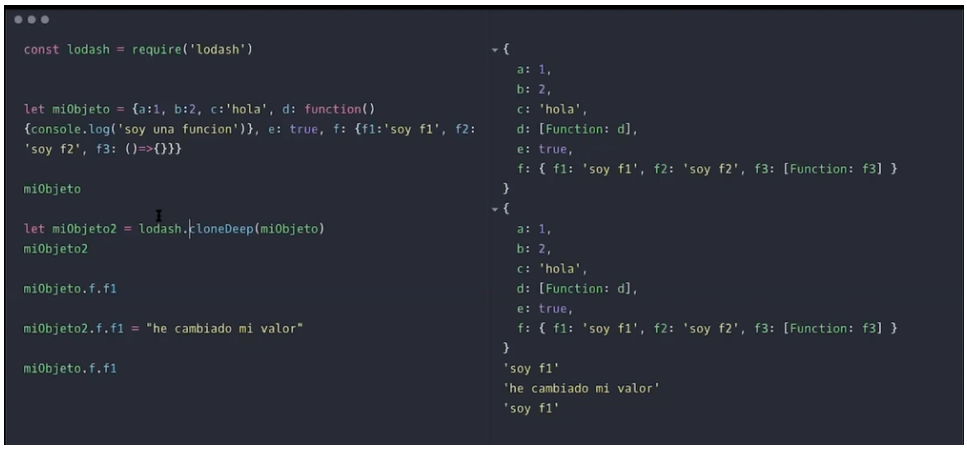


Destructuring (destructuración) clonación profunda y superficial

Lodash

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

Haciendo una copia profunda de mi objeto.



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

Comparación entre objeto y JSON

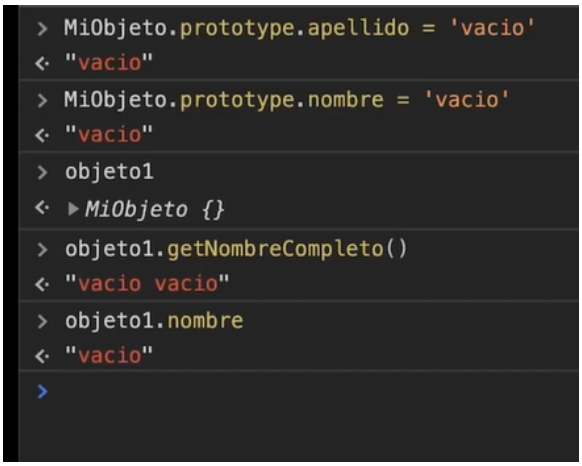
```
> let objeto1 = new MiObjeto('Monique' , 'Matamoros')
< undefined
> objeto1
< ▶ MiObjeto {nombre: 'Monique', apellido: 'Matamoros', getNombreCompleto: ƒ}
> MiObjeto
< ƒ MiObjeto(nombre, apellido){
      this.nombre= nombre
      this.apellido = apellido
      this.getNombreCompleto = function(){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

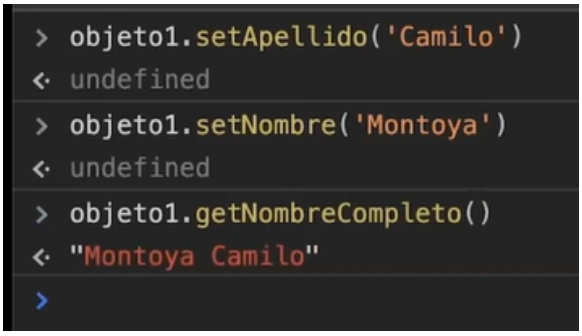
Prototype

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





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.