
# Para Aplicaciones Web Modernas – Parte II

Fernando Saez
saezfernando@Gmail.com

# JavaScript Functions 4 Ways

```
// Function Declaration
function square(x) {
  return x * x;
}
// Function Expression
const square = function(x) {
  return x * x;
}
// Arrow Function Expression
const square = (x) => {
   return x * x;
}
// Concise Arrow Function Expression
const square = x => x * x;
```

# Contexto de Ejecución

El contexto de ejecución (EC) se define como el entorno en el que se ejecuta el código JavaScript. Por entorno, nos referimos al valor de this, variables, objetos y funciones a las que el código JavaScript tiene acceso en un momento determinado.

1. Contexto de ejecución global (GEC)

2. Contexto de ejecución funcional (FEC)

3. Eval: Contexto de ejecución dentro de la función eval.

Cada contexto de ejecución tiene 3 fases:

1. Fase de creación (Compilación o Revisión)
   - Crea el objeto this y establece el ámbito de las variables y funciones
2. Fase de ejecución

3. Fase de Finalización

# Contexto de Ejecución - Global

```
1  var x = 100
2  var y = 50
3  function getSum(n1, n2) {
4    var sum = n1 + n2
5    return sum
6  }
7  var sum1 = getSum(x, y)
8  var sum2 = getSum(10, 5)
```

**Creation Phase:**

**Line 1:** *x* variable is allocated memory and stores *"undefined"*

**Line 2:** *y* variable is allocated memory and stores *"undefined"*

**Line 3:** *getSum()* function is allocated memory and stores all the code

**Line 7:** *sum1* variable is allocated memory and stores *"undefined"*

**Line 8:** *sum2* variable is allocated memory and stores *"undefined"*

**Execution Phase:**

**Line 1:** Places the value of *100* into the *x* variable

**Line 2:** Places the value of *50* into the *y* variable

**Line 3:** Skips the function because there is nothing to execute

**Line 7:** Invokes the *getSum()* function and creates a new function execution context

# Contexto de Ejecución - Función

```
1  var x = 100
2  var y = 50
3  function getSum(n1, n2) {
4    var sum = n1 + n2
5    return sum
6  }
7  var sum1 = getSum(x, y)
8  var sum2 = getSum(10, 5)
```


Cont. Ejec. Global

**Function EC Creation Phase:**

**Line 3:** *n1 & n2* variables are allocated memory and stores *"undefined"*

**Line 4:** *sum* variable is allocated memory and stores *"undefined"*

**Function EC Execution Phase:**

**Line 3:** *n1 & n2* are assigned *100* and *50*

**Line 4:** Calculation is done and *150* is put into the *sum* variable

**Line 5:** return tells the function EC to return to the global EC with value of *sum* (150)


Fase de Finalización

**Line 7:** Returned *sum* value is put into the *sum1* variable

**Line 8:** Open another function execution context and do the same thing

# Entorno Léxico

El entorno léxico es el ámbito en el que se definen las variables y funciones.

Es importante porque determina dónde se pueden acceder las variables y funciones.

El cierre léxico se refiere a la capacidad de una función para acceder a variables en su entorno léxico externo, incluso después de que se haya completado la ejecución de la función.

```
function doSomething() {
  var age= 7;
  // Some more code
 }
```

Tipos de Entorno Léxico

1.-Entorno léxico global
2.-Entorno léxico local
3.- Entorno léxico de bloque

# This

En el Contexto de ejecución global (GEC)  esto se refiere al objeto global, que es el objeto Windows en el browser y Global en Node.

```
var occupation = "Frontend Developer";

function addOne(x) {
    console.log(x + 1)
}
```

```
window.occupation = "Frontend Developer";
window.addOne = (x) => {
console.log(x + 1)
};
```

En el caso del contexto de ejecución funcional (FEC), no se crea el objeto this. Más bien, obtiene acceso al entorno en el que está definido.

En los objetos, la palabra clave this no apunta al GEC, sino al objeto en sí mismo.

# This con Arrow Functions

Funciones flechas no tienen su propio this.

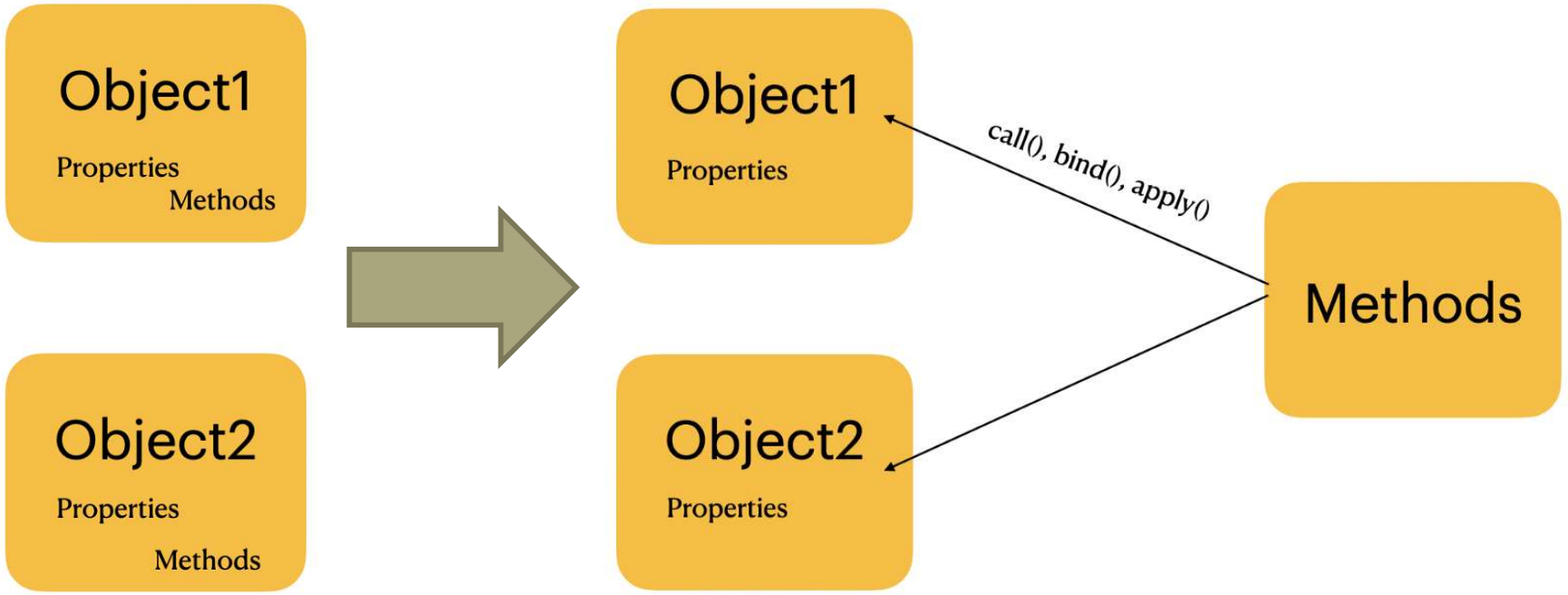
Cuando llamemos a this , esta referirá al ámbito padre.

**NO Usar funciones flechas para:**

1.- Crear métodos dentro de objetos

2.- Usarla como constructor

# Call, Apply y Bind




1

```
var obj = { num: 2 };

function add(a){
  return this.num + a;
}
```

2

```
add.call(obj, 3);
```

# Call, Apply y Bind

1

```
var obj = { num: 2 };

function add(a, b){
  return this.num + a + b;
}

console.log(add.call(obj, 3, 5));
```

2

```
console.log(add.apply(obj, [3, 5]));
```

3

```
const func = add.bind(obj, 3, 5);
func(); // Returns 10
```

# Closure

- Inner function tienen acceso a las variables locales de la outer function

Outer function

```
function iniciar() {
  var nombre = "Mozilla";  // La variable nombre es una variable local.
  function mostrarNombre() {    //mostrarNombre() es una función interna,
                                 // una clausura.
    alert(nombre);               // Usa una variable declarada en la
                                 //función externa.
  }
  mostrarNombre();
}
iniciar();
```

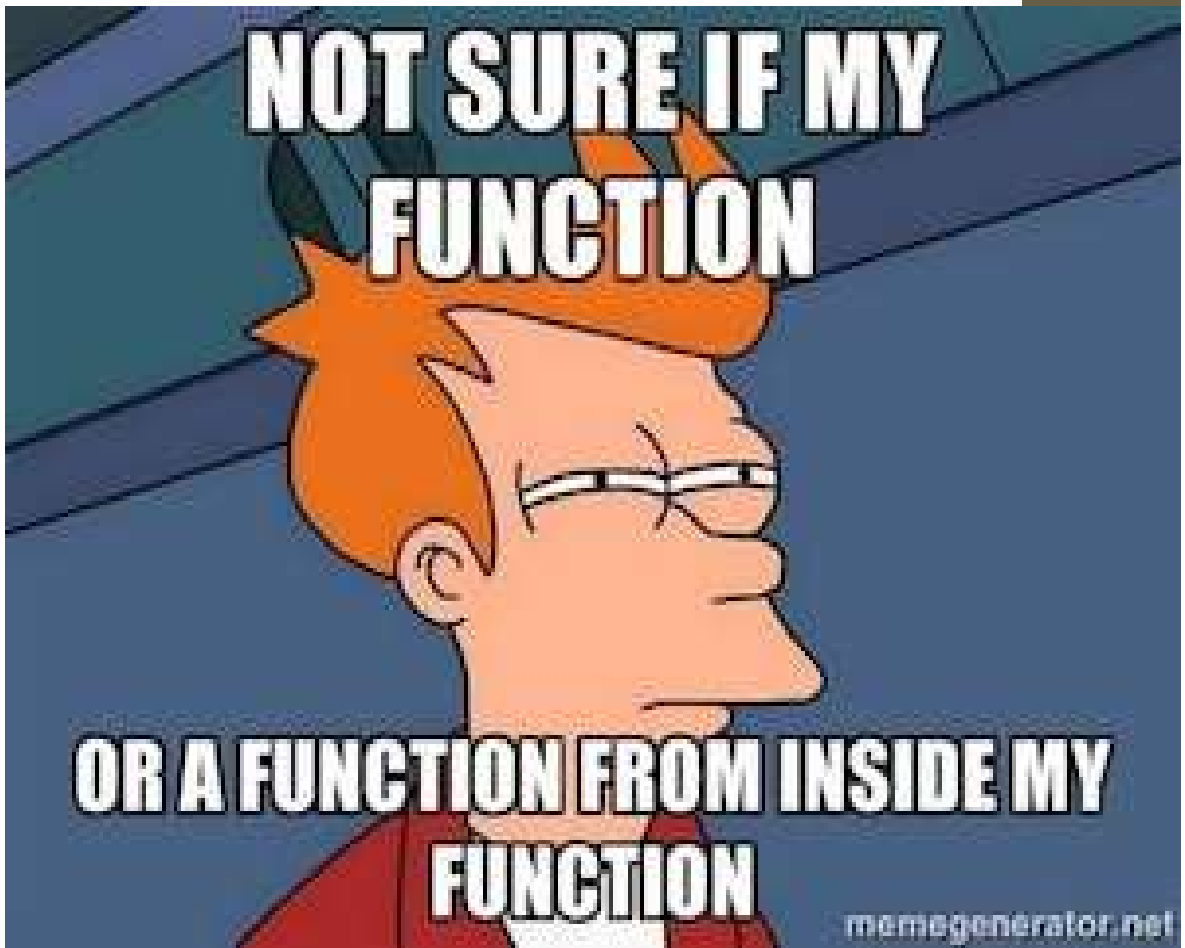
- Una closure es la combinación de una función y el lexical scoping en el cual la función se declaro.

# Ejercicio 1

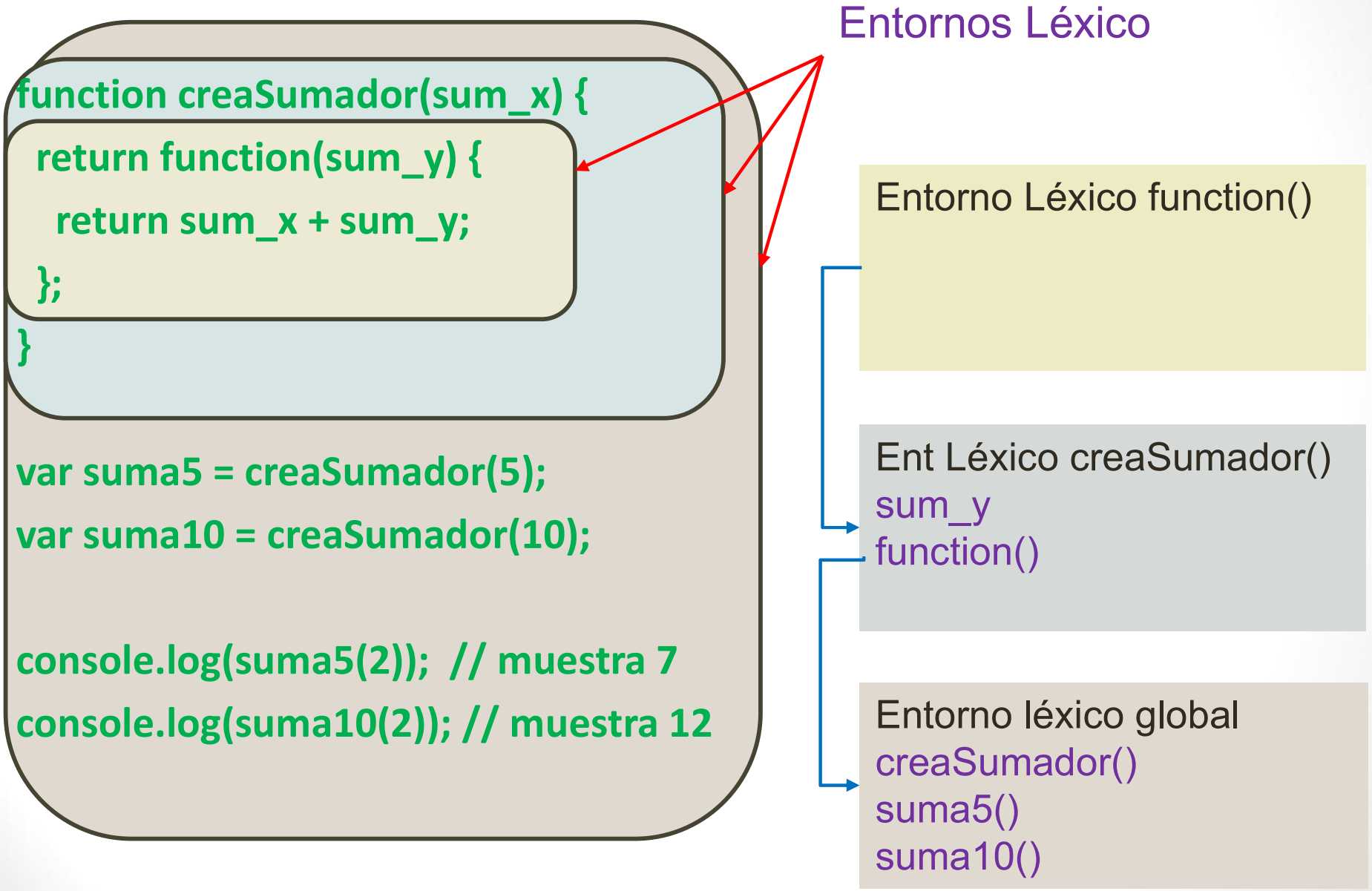
```
function foo(){
  let value=10;
  return () => {console.log(value++);}
}

var bar1 = foo();
var bar2 = foo();

bar1(); // ??
bar1(); // ??
bar2(); // ??
bar2(); // ??
```



# Clousure:
# Retornar funciones-parametrizadas



# Closure:
# Un ejemplo práctico: Crear contexto privado y optimizar (1)

```
var meses = [   "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
"Julio",  "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"];

function getMes (n) {
        if (n < 1 || n>12) throw new RangeError("Rango incorrecto");
        return meses[n - 1];
}

console.log(getMes(3)); // Marzo
```

# Closure:
# Un ejemplo práctico: Crear contexto privado y optimizar (2)

```
function getMes (n) {
  var meses = [
    "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
    "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"];

  if (n < 1 || n>12) throw new RangeError("Rango incorrecto");
  return meses[n - 1];
}

console.log(getMes(9)); // Septiembre
```

# Closure:
# Un ejemplo práctico: Crear contexto privado y optimizer (3)

```
var getMes = (function () {

var meses = [ "Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio",
   "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre"];

  return function inner(n) {
   if (n < 1 || n>12) throw new RangeError("Rango incorrecto");
   return meses[n - 1];
 };
})();

console.log(getMes(12)); // Diciembre
```

# IIFE Immediately-invoked Function Expression

```
(function () {
    statements
})();
```

1

```
(function () {
    var nombre = "Barry";
})();
Console.log(nombre) // throws "Uncaught
ReferenceError: aName is not defined“
```

2

```
var resultado = (function () {
    var saludo = “Hola";
    return saludo;
})();
Console.log(result); // "Barry"
```

# POO - Clases

```
class Documento {
  constructor(titulo, autor, esPublicado) {
    this.titulo = titulo;
    this.autor = autor;
    this.esPublicado = esPublicado;    }
  publicar(){
    this.esPublicado = true;
  }
}

class Libro extends Documento{
  constructor(titulo, autor, topico){
    super(titulo, autor, true);
    this.topico = topico;
  }
}
```

# Clases

```
class MyObject {
    constructor(param1, param2) {
    let atributo1 = param1; // atributo privado
    this.atributo2 = param2; // atributo público
    this.metodo1 = function(…) { // método
                                        público.
      // cuerpo del método
   }
     let metodo2 = function(…) { // método
                                        privado
      // cuerpo del método
  }
}
var obj = new MyObject(x,y); //crea instancia
Obj.atributo1 //Error
```

# Métodos estáticos

```
class Punto {
  constructor ( x , y ){
    this.x = x;
    this.y = y;
  }
  static distancia ( a , b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.sqrt ( dx * dx + dy * dy );
  }
}
const p1 = new Punto(5, 5);
const p2 = new Punto(10, 10);
console.log (Punto.distancia(p1, p2)); //
 7.071067811865475
```

# Módulos (Exportar e Importar)

```
// exportar un modulo en el archive lib/greetings.js
module "utils" {
    export function greeting(name){
        console.log("Hi! " + name);
    }
}
```

---

```
// importa la function greeting desde el modulo utils
import { greeting } from "utils";
var app = {
     welcome: function(){
          greeting("Mike");
} }
```

# Módulos

```
// ES6 // lib/math.js
export function mult(a, b){
  return a*b;
}
export const PI = 3.141593;
export default function(a, b){
  return a + b;
}
//Podemos exportar todo lo que necesitemos en una única
//línea al final del archivo
//export { mult, PI}
```

```
Import defaultmember {mult, PI} from "./math.js"
```

# Módulos

```
// existen varias formas de importar un módulo
import defaultMember from "module-name";
import * as name from "module-name";
import { member } from "module-name";
import { member as alias } from "module-name";
import { member1 , member2 } from "module-name";
import { member1 , member2 as alias2 , [...] } from
"module-name";
import defaultMember, { member [ , [...] ] } from "module-
name";
import defaultMember, * as name from "module-name";
import "module-name";
```

# Módulos (Importar en el navegador)

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

1) Los módulos solo se ejecutan una vez, incluso si se les ha hecho referencia en varias etiquetas <script>.

2) las características del módulo se importan al alcance de un solo script — no están disponibles en el alcance global.

3) Import y export están disponibles solo para módulos, no podemos utilizarlos en scripts standars

4) Agregan algunos mecanismos de seguridad extra

# Iterables

Un objeto es **iterable** si define cómo se itera.

Tipos integrados iterables: Array, Map, Set y TypedArray.

for...of.

Ver Iteradores y generadores
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_Generators
Ver TypedArray
https://developer.mozilla.org/es/docs/Web/JavaScript/Vectores_tipados

# for .. of

```
//ES5
var numbers = [1,2,3,4,5];
numbers.forEach(function(value) {
    console.log(value);
}); //1, 2, 3, 4, 5
```

---

```
//ES6
var numbers = [1,2,3,4,5];
for(let item of numbers){
    console.log(item);
}; //1, 2, 3, 4, 5
```

# Diferencias entre for .. in y for .. of

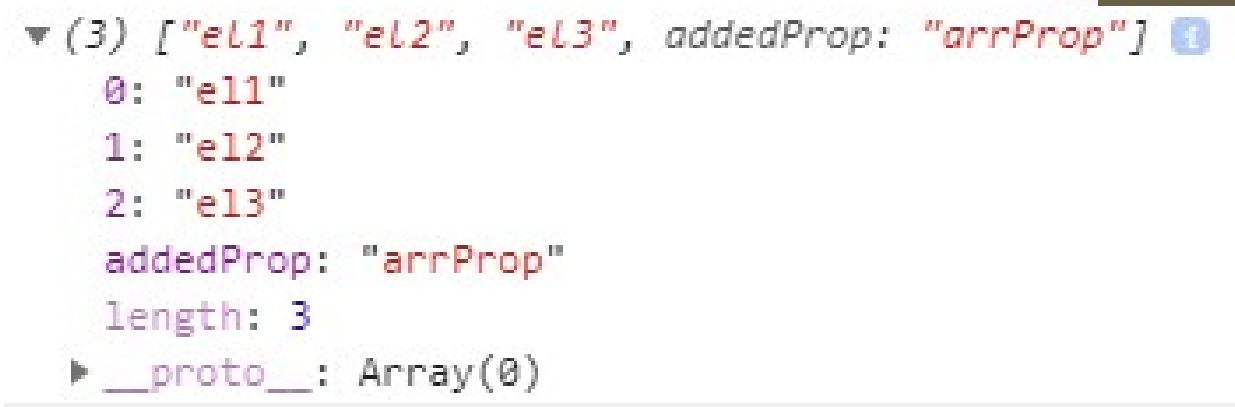
for..in itera sobre todas las propiedades enumerables de un objeto.

for..of itera sobre los valores de un objeto iterable. (arrays, strings, map, set)

```
let arr = ['el1', 'el2', 'el3'];
arr.addedProp = 'arrProp';

for (let elKey in arr) {
  console.log(elKey); // 0, 1, 2, addedProp
}

for (let elValue of arr) {
  console.log(elValue) // el1, el2, el3
}
```



# Map

```
//ES6
let map = new Map();
map.set('foo', 123);
let user = {userId: 1}; //object
map.set(user, 'Alex');
map.get('foo'); //123
map.get(user); //Alex
map.size; //2
map.has('foo'); //true
map.delete('foo'); //true
map.has('foo'); //false
```

size

clear()
forEach()
get()
has()
keys()
set()
values()

# Map

Un objeto Map puede iterar sobre sus elementos en orden de inserción.

Un bucle for..of devolverá un array de [clave, valor] en cada iteración.

```
map = new Map([['user1','Alex'], ['user2', 'Vicky'], ['user3', 'Enrique']]);
```

Diferencias entre objetos y mapas

# Set

Los sets son conjuntos de elementos no repetidos, que pueden ser tanto objetos, como valores primitivos.

```
let set = new Set();
set.add('foo');
set.add({bar:’baz’});
set.size  //2
for(let item of set){
   console.log(item);
}
//"foo"
//{bar:’baz’}
```

size

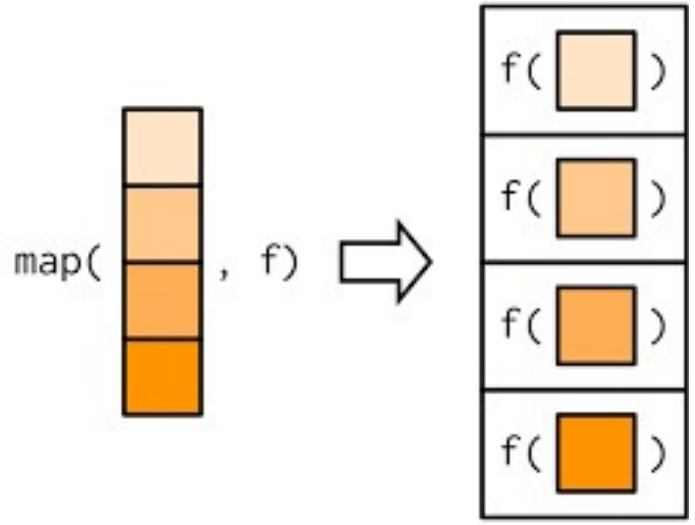
clear()
forEach()
get()
has()
keys()
add()
values()

# Un poco de programación funcional

Las técnicas funcionales pueden ayudarte a escribir código más declarativo.

- map
- filter
- reduce
- find
- forEach

# Un poco de programación funcional




## map

Cuando llamas a map() en un array, este ejecuta una función en cada elemento dentro de él, retornando un nuevo array con los valores que la función retorna.

```
var myArray = [10, 20, 30];
var newArray = myArray.map(number => number + 1);
console.log(newArray);
// [11, 21, 31]
```

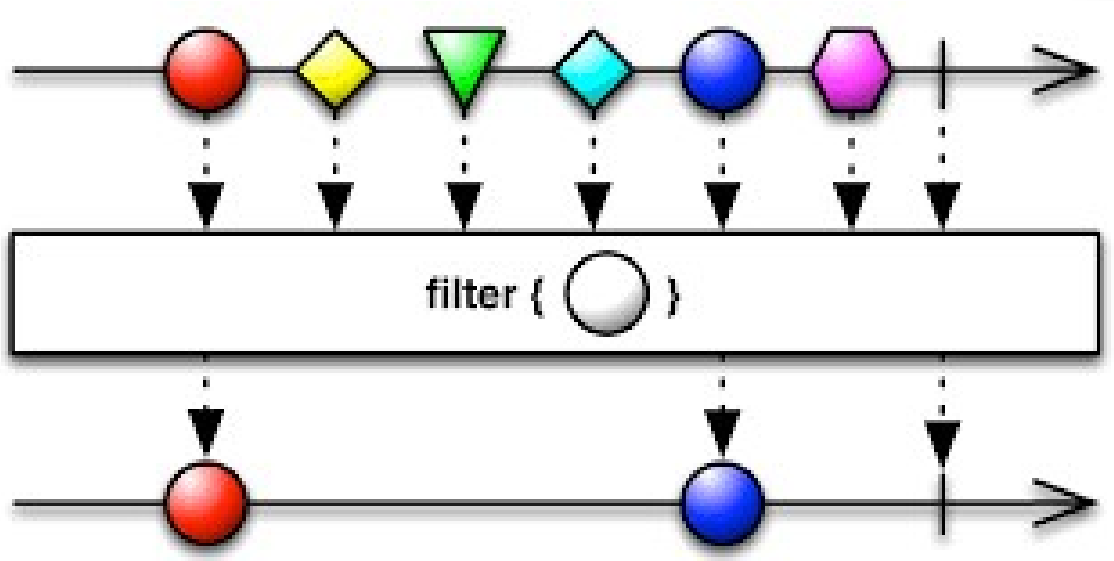
# Map() - Ejemplo

```
const reg = /\d{3}/g;
const str = "Java323Scr995ip4894545t";
const nuevaStr = str.replace(reg, "");
console.log(nuevaStr);

const arr = [
  "fer555nan123do",
  "hola534 789que ta983l",
  "c532om453o estas234!!",
];
arr.map((item) => item.replace(reg, "")).forEach((item) =>
console.log(item));
```

# Un poco de programación funcional

filter()




```
var myArray = [10, 20, 30, 40];
var filteredValues = myArray.filter(number => number > 20);
filteredValues
// [30, 40]
```

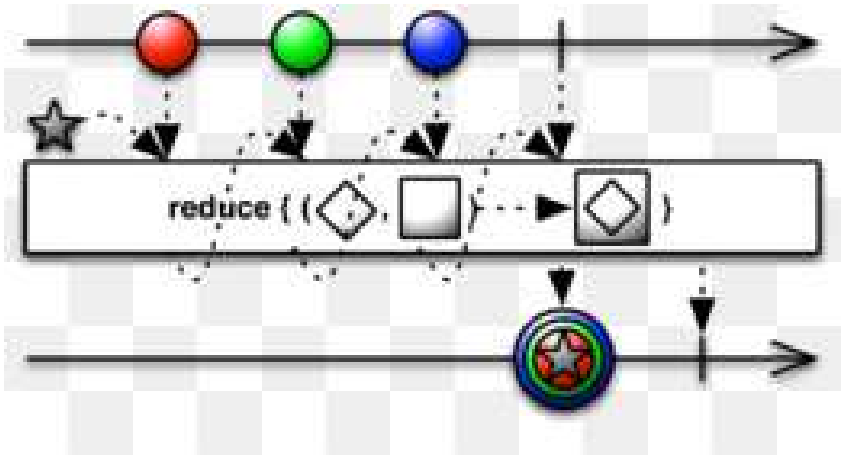
# filter() - Ejemplo

```
let usuarios = [
  {id: 1, name: “Jose“, isAdmin:true},
  {id: 2, name: “Ana“, isAdmin:false},
  {id: 3, name: “Juan“, isAdmin:true}
];

// Retorna array con los 2 primeros usuarios
let usuarios1y2 = usuarios.filter(item => item.id < 3);
alert(usuarios1y2.length); // 2
//Retorna los usuarios administradores
Let usuariosAdmin = usuarios.filter(item=> item.isAdmin)
alert(usuariosAdmin.length); // 2
```

# Un poco de programación funcional

reduce()




```
var myArray = [10, 20, 30];
var total = myArray.reduce((accumulator,actual) => {
  return accumulator + actual;
});
total; // 60
```

# reduce() - Ejemplo

```
let usuarios = [
  { name: "Jose", job: "Data Analyst", country: "AR" },
  { name: "juan", job: "Developer", country: "US" },
  { name: "Ana", job: "Developer", country: "US" },
  { name: "Karen", job: "Software Eng", country: "CA" },
  { name: "Jonas", job: "QA", country: "CA" },
  { name: "Ale", job: "Designer", country: "AR" },
];
let usuariosAgrupadosPorPais = usuarios.reduce((acumuladorGrupo,
usuario) => {
  let newkey = usuario["country"];
  if (!acumuladorGrupo[newkey]) acumuladorGrupo[newkey] = [];
  acumuladorGrupo[newkey].push(usuario);
  return acumuladorGrupo;
}, []);
```

# Un poco de programación funcional

reduce()

```
var myArray = [10, 20, 30];
var objectCreatedFromArray =
myArray.reduce((accumulator, number, index, array) => {
    accumulator[`number${index}`] = number;
    return accumulator;
}, {});
objectCreatedFromArray;
// {number0: 10, number1: 20, number2: 30}
```

# find()

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.find(element => element > 10);
console.log(found);
// output: 12

const foundIndex = array1.findIndex(element => element > 10);
console.log(found);
// output: 1
```

# forEach()

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

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

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


¿ALGUNA PREGUNTA?
makeameme.org