# CLASS(E)

ı

Módulo 3. Funciones de primer orden

## Funciones de primera clase

## Funciones de primera clase

Llamamos funciones de primer orden a aquellas que pueden ser tratadas como otro tipo de datos, como si fueran objetos.

Esto nos permite pasar estas funciones como variables, como propiedades en un objeto, como argumentos en una función, etc.

Llamamos funciones de orden superior a aquellas que:

- A. Reciben una función como parámetro.
- B. Devuelven una función como resultado

## Funciones de orden superior

Llamamos funciones de orden superior a aquellas que:

- A. Reciben una función como parámetro.
- B. Devuelven una función como resultado

```
function executeTwice(func){ // función de primer orden
    func()
   func()
function printLorem(){
    console.log("lorem ipsum")
executeTwice(printLorem)
// lorem ipsum
// lorem ipsum
```

Algunas de las funciones que hemos visto hasta ahora son funciones de primer orden:

• map, filter, reduce, find...

Todas reciben una función por parámetro.

list.map(functionPassedByParameter)
list.filter(functionPassedByParameter)
list.reduce(functionPassedByParameter)
list.find(functionPassedByParameter)

Las funciones de primer orden son un mecanismo muy potente y Javascript está especialmente preparado para trabajar con ellas.

¡Javascript nos permite definir la función que pasamos por parámetro en la propia llamada!

```
executeTwice(function(){
    console.log("lorem ipsum")
})
// lorem ipsum
// lorem ipsum
```

## Ejercicio funciones

// SPAM! SPAM! SPAM!
repeat(() => console.log("SPAM!"), 3)

Implementa una función repeat que reciba una función y un número N y llame a la función N veces.

## Ejercicio funciones II

Implementa una función retry idéntica a repeat, pero que sólo repita en caso de producirse un error al llamar a la función.

## Ejercicio funciones III

 $map(x \Rightarrow x + 1, [1, 2, 3]) // [2, 3, 4]$ 

Implementa tu propia función map que reciba una función y una lista.

Debe devolver la lista resultante de aplicar la función a todos los elementos de la lista.

## Ejercicio funciones IV

filter(x => x >= 2, [1, 2, 3]) // [2, 3]

Implementa tu propia función filter.

## Ejercicio funciones V

reduce((acc, x) => acc + x, 0, [1, 2, 3]) // 6 reduce((acc, x) => acc + x, 10, [1, 2, 3]) // 16

Implementa tu propia función reduce que reciba una función, un valor inicial y una lista.

Debe devolver el resultado de acumular los valores de la lista mediante la función acumuladora y debe partir del estado inicial.

## Ejercicio funciones VI 🔥

Implementa map utilizando reduce.

No está permitido usar bucles.

## Ejercicio funciones VII 🔥

Implementa filter utilizando reduce.

No está permitido utilizar bucles.

## ¡map y filter son, en esencia, reduces!

## Devolver funciones

Hemos dicho que también consideramos funciones de primer orden aquellas que devuelven una función.

Un ejemplo es la función <mark>negate</mark>, que recibe una función que devuelve true/false e invierte su comportamiento.

```
function isEven(number){
    return number % 2 === 0
}

const isNotEven = negate(isEven)

[1, 2, 3, 4].filter(isNotEven) // [1, 3]
```

## Ejercicio funciones VIII

Implementa la función <mark>negate</mark>.

Para entender completamente cómo se comportan las funciones de primer orden es necesario comprender qué es una clausura.

```
function CrearNombreySaludo() {
    const name = 'Homer'
    function saludar() {
        console.log('Hola ' + name)
    }
}
saludar() // Cómo podemos acceder?
```

```
function CrearNombreySaludo() {
    const name = 'Homer'
    return function () {
        console.log('Hola ' + name)
    }
}

// Devolviendo la función
const saludar = CrearNombreYSaludo()
saludar()
```

```
function CrearNombreySaludo() {
    const name = 'Homer'
    return function () {
        console.log('Hola ' + name)
    }
}

const saludar = CrearNombreYSaludo()
saludar() // Que imprime esto?
```

```
function CrearNombreySaludo() {
    const name = 'Homer'
    return function () {
        console.log('Hola ' + name)
    }
}
let saludar = CrearNombreYSaludo()
saludar() // Hola Homer
```

```
function CrearNombreySaludo() {
    const name = 'Homer'
    return function () {
        console.log('Hola ' + name)
    }
}
let saludar = CrearNombreYSaludo()
saludar()
```

## Ejercicio clausuras

Crea una función que reciba un número y devuelva una función que sume ese número a un nuevo número que la función devuelta recibe por parámetro.

# Vamos a ver un caso más complicado

```
function counter() {
  return () => {
    let i = 0;
    return i++;
  };
}
```

```
// c1 contiene una función
const c1 = counter();
```

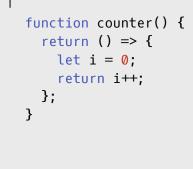
```
function counter() {
Clausuras
                                                     return () => {
                                                      let i = 0;
                                                      return i++;
const c1 = counter();
                                                    };
console.log(c1()); // ?
```

```
function counter() {
Clausuras
                                                     return () => {
                                                      let i = 0;
                                                      return i++;
const c1 = counter();
                                                    };
console.log(c1()); // 0
```

```
function counter() {
Clausuras
                                                    return () => {
                                                      let i = 0;
                                                      return i++;
const c1 = counter();
                                                    };
console.log(c1()); // 0
console.log(c1()); // ?
```

```
Clausuras
```

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 0
console.log(c1()); // 0
```



```
const c1 = counter();
```

## C1

```
() => {
  let i = 0;
  return i++;
};
```

```
const c1 = counter();
c1();
```

## CT

```
() => {
  let i = 0;
  return i++;
};
```

```
const c1 = counter();
c1();
```

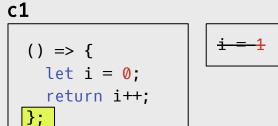
```
i = 0
let i = 0;
return i++;
};
```

```
const c1 = counter();
c1(); // 0
```

```
c1

() => {
   let i = 0;
   return i++;
};
```

```
const c1 = counter();
c1(); // 0
```



```
const c1 = counter();
c1(); // 0
c1();
```

```
i = 0

let i = 0;
  return i++;
};
```

```
const c1 = counter();
c1(); // 0
c1(); // 0
```

```
c1

() => {
   let i = 0;
   return i++;
};
```

```
function counter() {
    let i = 0;    // Ahora declaramos fuera
    return () => i++;
}
```

```
function counter() {
  let i = 0;
  return () => i++;
}
```

```
const c1 = counter();
console.log(c1()); // ?
```

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // ?
```

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 1
console.log(c1()); // 2
console.log(c1()); // 3
```

```
const c1 = counter();
let i = 10;
console.log(c1()); // ???
console.log(i); // ???
```

```
const c1 = counter();
let i = 10;
console.log(c1()); // 0
console.log(i); // 10
```

# Una clausura es el scope en el momento de creación de una función

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // ???
```

```
function counter() {
    let i = 0;
    return () => i++;
}
```

# Volvemos a las funciones de primer orden

#### Devolver funciones

La capacidad de devolver nuevas funciones nos permite modificar el comportamiento de las funciones de entrada.

#### Algunos ejemplos:

- Cambiar la forma o el número de los parámetros que espera esa función.
- Cambiar la frecuencia con la que se llama esa función.
- Controlar que la función solo se ejecute si se cumplen determinadas condiciones.

#### Cachear funciones

Por ejemplo, podemos implementar una función memoize que cachee los resultados si la función ya ha sido llamada con los mismos parámetros.

```
function sum(a, b){
    console.log("Calculando...")
    return a + b
}

const memoizedSum = memoize(sum)

memoizedSum(2, 2) // Calculando... 4
memoizedSum(2, 2) // 4
memoizedSum(3, 3) // Calculando... 6
```

# Memoize es una gran herramienta para optimizar operaciones pesadas

# Ejercicio funciones IX

Implementa la función memoize para funciones de un sólo parámetro.

# Ejercicio funciones X 🔥 🔥



Implementa la función <mark>memoize</mark> para funciones de cualquier número de parámetros.

```
function writeInDB(databaseURL, key, value){
Partial
                                   // guarda un dato clave-valor
                               writeInDB("132.48.82.251:27017", "user-1", "Homer Simpson")
Podemos evitar repetición
                               writeInDB("132.48.82.251:27017", "user-2", "Philip J. Fry")
                               writeInDB("132.48.82.251:27017", "user-3", "Peter Griffin")
de código utilizando
funciones de primer orden.
Dada la función writeInDB...
```

```
const write = partial(writeInDb, "132.48.82.251:27017")
Partial
                               // equivalente a llamar writeInDb con 3 parámetros
                               write("user-1", "Homer Simpson")
                               write("user-2", "Philip J. Fry")
Podemos evitar repetición
                               write("user-3", "Peter Griffin")
de código utilizando
funciones de primer orden.
Dada la función writeInDB...
```

#### Ejercicio funciones XI

```
reduce((acc, x) => acc + x, 0, [1, 2, 3]) // 6
reduce((acc, x) => acc + x, 10, [1, 2, 3]) // 16
```

Implementa la función partial para un sólo parámetro.

# Veremos más ejemplos en el módulo de asincronía