Funciones y Objetos

Laura Isabel López Naves

Funciones

- Esencialmente hay dos tipos de funciones: nombradas y anónimas:
- Las funciones nombradas se puede utilizar directamente, (ej: hello('Pepe')).
- Las funciones anónimas deben ser asignadas a una variable o utilizarlas como un callback (función que se pasa como argumento a otra función).
- Las funciones inmediatamente invocadas (IIFE) o funciones inmediatas se dejan de utilizar desde que existe el ámbito de bloque con ES6.

Nota: Con las funciones nombradas no se produce hoisting (se pueden invocar en cualquier momento). Con las funciones anónimas asignadas a una variable se produce hoisting (no se pueden invocar antes de su declaración).

```
// funcion nombrada:
function hello(nombre) {
    return `Hola ${nombre}`;
}
console.log(hello('Pepe')); // Hola Pepe
```

```
// funcion anónima asignada a una variable:
let hello = function (nombre) {
    return `Hola ${nombre}`;
}
console.log(hello('Pepe')); // Hola Pepe

// funcion que recibe una funcion como parámetro
function hola(foo) {
    foo();
};
// función anónima como parámetro:
hola(function () {
    console.log('Hola') // Hola
});

// funcion IIFE:
(function(){
    //...
}());
```

Funciones

- Aspectos a considerar en javascript:
  - Se pueden definir **funciones anidadas** (funciones dentro de otras funciones). Esto no existe en otros lenguajes como java/c#.
  - Además hay **funciones puras** que esencialmente son aquellas que no acceden al ámbito exterior (variable definida fuera de ellas) ni modifican los parámetros.
  - También existen la **funciones de orden superior** que son aquellas que reciben como parámetro o devuelven una función.

Funciones. Parámetros

- Se pueden utilizar los parámetros que se deseen en la firma y en la invocación sin ninguna validación.
- También se pueden leer los parámetros con el array “arguments” (se usa menos).
- Desde ES6 es posible especificar valores por defecto para los parámetros en la firma de la función.

```
// Parámetros:
function f1(nombre,edad) {
    console.log(`${nombre} - ${edad}`);
}
f1('Pepe');                     // Pepe - undefined
f1('Pepe',30,'Otro dato');      // Pepe - 30

// Parámetros utilizado "arguments" (se usa menos):
function f2(nombre,edad) {
    console.log(`${arguments[0]} - ${arguments[1]}`);
}
f2('Pepe');                     // Pepe - undefined
f2('Pepe',30,'Otro dato');      // Pepe - 30

// Parámetros con valores por defecto:
function f3(nombre='Paco',edad=20) {
    console.log(`${nombre} - ${edad}`);
}
f3('Pepe');                     // Pepe - 20
f3();                           // Paco - 20
```

Parámetros rest ...

Permiten agrupar múltiples elementos y condensarlos en uno solo donde se esperan diferentes parámetros. **Se utilizan en la declaración de funciones**.

```
function suma1(a, b) {
    return a + b;
}
console.log(suma1(10));          // NaN
console.log(suma1(10, 20));      // 30
console.log(suma1(10, 20, 30)); // 30

function suma2(...args) {
    let result = 0;
    for (const a of args) {
        result += a;
    }
    return result;
}
console.log(suma2());            // 0
console.log(suma2(10));          // 10
console.log(suma2(10, 20));      // 30
console.log(suma2(10, 20, 30)); // 60
```

Los parámetros “rest” siempre se ponen al final (después de otros posibles parámetros) y devuelven una array que se puede utilizar directamente.

```
function añadirApellido(nombre,...apellidos) {
    let result=nombre;
    for (const i in apellidos) {
        result+=" "+apellidos[i];
    }
    return result;
}
console.log(añadirApellido("Pedro")); // Pedro
console.log(añadirApellido("Luis","Ruiz")); // Luis Ruiz
console.log(añadirApellido("Ana","Gil","Sanz")); // Ana Gil Sanz
```

El operador de propagación o spread operator "…"

- El **spread operator** "…" permite que un iterable (array, string, etc.) sea expandido en situaciones donde se esperan múltiples argumentos, por ejemplo en llamadas a funciones o múltiples elementos (arrays literales).

```
let values = [1, 2, 3];
console.log(values); // [ 1, 2, 3 ]
let copy = [...values];
console.log(copy)    // [ 1, 2, 3 ]
let morevalues = [4, 5];
values.push(...morevalues);
console.log(values); // [ 1, 2, 3, 4, 5 ]
let other = [0, ...morevalues];
console.log(other);  // [ 0, 4, 5 ]
```

```
function suma(a, b, c) {
    return a + b + c;
}
let array1=[1,2,3];
console.log(suma(...array1)); // 6
let array2=[1,2,3,4];
console.log(suma(...array2)); // 6

let array3=[1,2,3,4];
function add(...args){
    let rest=0;
    for (const a of args) {
        rest+=a;
    }
    return rest;
}
console.log(add(...array3)); // 10
```

Nota: En el caso de función "add" utilizamos el parámetro "rest" y luego en la invocación el "spread operator.

El operador de propagación o spread operator "…"

- Aunque menos utilizado, a partir de ES2018 se permite el operador spread en objetos:

```
// Operador spread con objetos:
const persona = { nombre: 'Pepe', edad: 27 };
const cliente = { ventas: 120, ...persona };
console.log(cliente);
const copiaCliente = { ...cliente };
console.log(copiaCliente);
```

```
{ ventas: 120, nombre: 'Pepe', edad: 27 }
{ ventas: 120, nombre: 'Pepe', edad: 27 }
```

- Se permite utilizar el operador para crear otros objetos (como el ejemplo) pero no en las llamadas a funciones.

Funciones flecha o arrow functions

- La funciones flecha permiten escribir funciones de forma concisa.
- La sintaxis es:

  (parámetros) => código

  o

  (parámetros) => {código}

- Si solo recibe un parámetro, no es necesario escribir paréntesis.
- Si solo tiene una línea, no es necesario escribir “return”.

```
let saludo1 = function () {
    return "Hola";
}
let saludo2 = () => "Hola";
console.log(saludo1());  // Hola
console.log(saludo2());  // Hola

let saludo3 = function (nombre) {
    return `Hola ${nombre}`;
}
let saludo4 = nombre => `Hola ${nombre}`;
console.log(saludo3("Pepe")); // Hola Pepe
console.log(saludo4("Pepe")); // Hola Pepe

let saludo5 = function(nombre,apellido) {
    return `Hola ${nombre}, ${apellido}`;
}
let saludo6 = (nombre,apellido) => {
    return `Hola ${nombre}, ${apellido}`;
}
console.log(saludo5("Pepe","Ruiz")); // Hola Pepe, Ruiz
console.log(saludo6("Pepe","Ruiz")); // Hola Pepe, Ruiz
```

Funciones que devuelven iteradores

- Devuelven un objeto “iterador” a partir de un array.
- Sobre los iteradores solo se puede utilizar el método next(), for..of o el operador spread. No permiten índices.
- Cuando se ha iterado por todos los elementos, el puntero no vuelve al inicio.
- Ejemplos (funciones que devuelven iteradores): keys(), values(), entries()

```
let t1=['a','b','c'];
let k=t1.keys();
for(const e of k)
    console.log(e); // 0 1 2
for(const e of k)
    console.log(e); // <- nada
```

Nota: el segundo for..of no visualiza nada porque el iterador ya ha llegado al final.

```
let t2=['x','y','z'];
k=t2.keys();
console.log(...k); // 0 1 2
let v=t2.values();
console.log(...v); // x y z
let e=t2.entries();
console.log(...e); // [ 0, 'x' ] [ 1, 'y' ] [ 2, 'z' ]
```

Array.from() es una función que permite obtener un array a partir de un iterador. ->

```
let t3=['x','y','z'];
let r=Array.from(t3.entries());
console.log(r[1]); // [ 1, 'y' ]
```

Funciones de repetición que iteran en arrays

- Pueden tener una función como parámetro (típicamente una función flecha).
- Utilizan la “api fluida” y encadenamiento (Fluent API and Chaining).
  - **foreach()** : permite iterar por cada uno de los elementos de un array.
  - **map()** : devuelve un nuevo array resultado de transformar el array inicial.
  - **reduce()** : devuelve un valor resultado de una operación sobre los elementos del array
  - **reduceRight()** : igual que reduce() pero itera de derecha a izquierda.
  - **filter()** : obtiene un nuevo array filtrando el array inicial.
  - **every()** : devuelve true si todos los elementos del array cumplen una condición.
  - **some()** : devuelve true si alguno de los elementos del array cumple una condición.
  - **find()** : devuelve un valor buscado en un array o undefined en caso contrario.
  - **findIndex()** : devuelve el índice de un valor buscado en un array o -1 si no lo encuentra.

```
let a = ['a', 'b', 'c'];
a.forEach(function (e) {  // a
    console.log(e)        // b
})                        // c
```

```
let b = [1, 2, 3];
console.log(b.map(e => e + 1)); // [2,3,4]
```

Funciones de repetición. Ejemplos.

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```
let a = ['a', 'b'];
a.forEach((e)=>console.log(e));
a.forEach((e, i) => console.log(`Valor:${e}, índice:${i}`));
a.forEach((e, i, a) => console.log(`Valor:${e}, índice:${i}, array:${a}`));
```

```
a
b
Valor:a, índice:0
Valor:b, índice:1
Valor:a, índice:0, array:a,b
Valor:b, índice:1, array:a,b
```

```
// map : obtiene un arraytransformado
const b = ['a', 'b'];
console.log(b.map(e => e + 'x'));                    // [ 'ax', 'bx' ]
console.log(b.map((e, i) => e + i));                 // [ 'a0', 'b1' ]
console.log(b.map((e, i, m) => e + i + m.length)); // [ 'a02', 'b12' ]
```

```
// reduce : obtiene un valor resultado de un calculo
const c = [10, 20, 30];
console.log(c.reduce(ac => ac));                          // 10
console.log(c.reduce((ac, n) => ac + n));                 // 60
console.log(c.reduce((ac, n, i) => ac + n + i));          // 63
console.log(c.reduce((ac, n, i, a) => ac + n + i + a[2])); // 123
console.log(c.reduce((ac, n) => ac + n, 100));            // 160
```

```
let value = arr.reduce(function(previousValue, item, index, array) {
  // ...
}, [initial]);
```

Reduce() : La función se aplic a todos los elementos de la matriz uno tras otro y “acarrea" su resultado a la siguiente llamada.
Parámetros:
-previousValue: resultado de la última iteración, se usa como acumulador.
- item: valor del elemento actual.
- index: valor del índice actual.
- array: matriz actual.
-initial: valor inicial del acumulador (por defecto inicialmente “previus value” es 0).

Funciones de repetición. Ejemplos.

```
let a = [1, 2, 3];
console.log(a.filter(x => x > 1));      // [ 2, 3]
console.log(a.every(x => x != 5));      // true
console.log(a.some(x => x % 10 === 0)); // false
console.log(a.find(x => x === 3));      // 3
console.log(a.findIndex(x => x === 3)); // 2
```

```
let frase = "no luce la luna sin traérmela en sueños";

let res1 = frase.split(" ")
    .filter(x => x[0] === 'l')
    .map(x => x.toUpperCase());
console.log(res1);  // [ 'LUCE', 'LA', 'LUNA' ]
```

```
const res2 = frase.split(' ')
  .map(x => x.length)
  .filter(x => x < 3)
  .reduce((ac, x) => ac + x);
console.log(res2);     // 6
```

Ejemplo de encadenamiento.

-En el primer ejemplo se filtran las palabras que empiezan por "l" y se copian en mayúsculas en una nueva matriz resultante.
-En el segundo caso se obtiene la suma de la longitud de las palabras que tengan menos de tres caracteres.