

JS | Functions

Objetivos

- ✓ Nombrar, declarar e invocar correctamente las funciones de JavaScript.
- ✓ Usar parámetros y argumentos en sus funciones para mejorar la reutilización del código
- ✓ Comprender callback, funciones anónimas y arrow function
- ✓ Entender el alcance global y local
- ✓ Escribir funciones usando buenas prácticas
- ✓ Entender algunos de los conceptos básicos de la programación funcional

Funciones

Las funciones son uno de los bloques de construcción fundamentales en JavaScript.

Una función es un procedimiento de JavaScript: un conjunto de declaraciones que realiza una tarea o calcula un valor.

Declaración de funciones

La declaración de función es el proceso de **crear** una función, pero no ejecutarla.

```
// Function Declaration  
function sayHelloWorld() {  
    const whatToSay = 'Hello, World!';  
    console.log(whatToSay);  
}
```


Invocación de funciones

El proceso de **ejecución** (llamada) de la función se conoce como invocación de función.

```
// Function Invocation  
sayHelloWorld(); // => Hello, World!
```

Sintaxis de funciones

La sintaxis para declarar una función es:

```
function <name> ([<argument_list>]) {  
    <instructions>  
  
    [return <expression>;]  
}
```


RESUMIENDO

Declaración de funciones

Al declarar una función tenemos que asegurarnos de que existan:

- `function` keyword
- El nombre de la función
- parámetros (si los hay, si no solo `()`)
- cuerpo de la función, que es todo el código entre las llaves `{}`

Nombre de la función

- El nombre debe describir claramente lo que hace la función.
Como regla general, intentaremos usar verbos que describan acciones.
- En JavaScript, preferimos nombrar variables y funciones camelCase.
Ej: addTwoNumbers
- Los nombres de funciones siempre comienzan con una letra minúscula:
lowerCase -> SI
LowerCase -> NO

Parámetros de función y argumentos de función

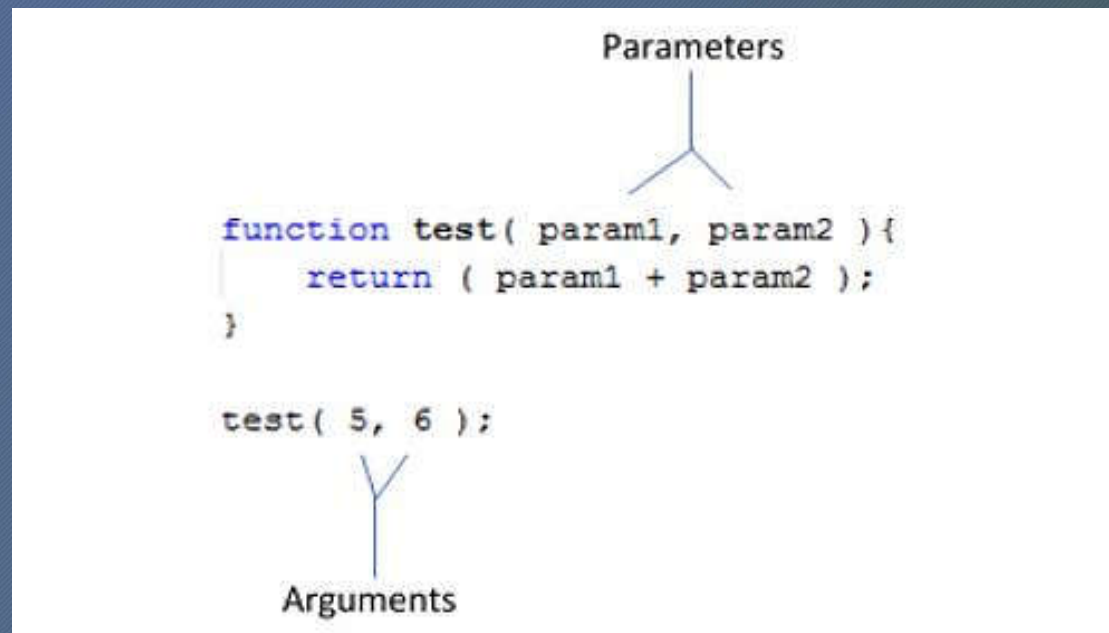
A veces podemos querer realizar varias cosas muy similares, pero no exactamente lo mismo cada vez. Aquí es cuando utilizamos parámetros.

```
function sayHello(name) {  
    console.log(`Hello ${name}!`);  
}  
  
sayHello('Mery');  
// "Hello Mery!"  
  
sayHello('John');  
// "Hello John!"  
  
sayHello('Lucy');  
// "Hello Lucy!"
```

Podemos tener tantos parámetros en una función como queramos.

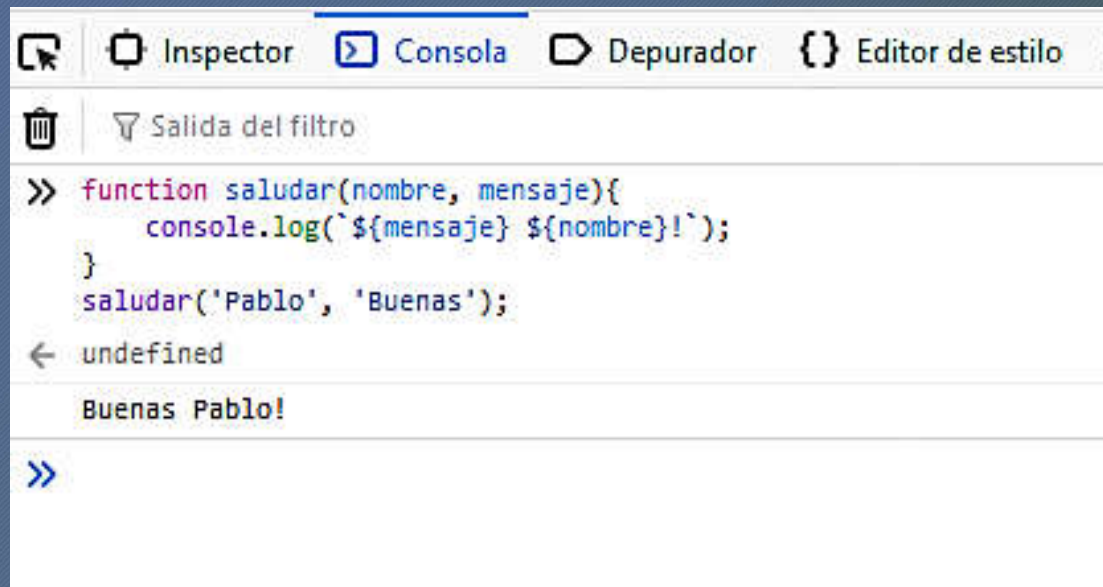
Parámetro vs. argumento

- Parámetro: el nombre de la variable que forma parte de la declaración de la función.
- Argumento: el valor pasado a la función en el momento de su invocación.



Al invocar una función siempre obtendremos un valor de retorno.

Si no se especifica la keyword return, el valor será 'undefined'.



```
>> function saludar(nombre, mensaje){  
    console.log(`${mensaje} ${nombre}!`);  
}  
saludar('Pablo', 'Buenas');  
← undefined  
Buenas Pablo!  
>>
```

The screenshot shows a web browser's developer console with the 'Consola' tab selected. The console displays the execution of a JavaScript function named 'saludar'. The function takes two arguments, 'nombre' and 'mensaje', and logs a string formatted as `\${mensaje} \${nombre}!`. The function is called with 'Pablo' and 'Buenas'. The console shows the log output 'Buenas Pablo!' and the return value 'undefined', indicating that the function did not have a 'return' statement.

JavaScript Callback Functions

Una función **callback** es una función que se pasa a otra función como parámetro y se ejecutará después de que otra función termine de ejecutarse, y de ahí proviene el nombre: "callback".

JavaScript Callback Functions

```
function eatDinner(){  
    console.log("Eating the dinner");  
}  
  
function eatDessert(){  
    console.log("Eating the dessert");  
}  
  
eatDinner(); // <== Eating the dinner  
eatDessert(); // <== Eating the dessert
```

¿por qué pasaríamos una función como parámetro a la otra función?

En la mayoría de situaciones, una función va a tardar más en completarse que otra. Por eso deberíamos usar *callbacks*. Por ejemplo en esta situación:

```
function eatDinner() {
  setTimeout(function() {
    console.log("Eating the dinner ");
  }, 1000)
}

function eatDessert() {
  console.log("Eating the dessert ");
}

eatDinner();
eatDessert();

// the console:
// Eating the dessert
// Eating the dinner
```


Los callbacks son la forma de asegurarse de que una parte del código no se ejecute antes de que otro código no haya terminado de ejecutarse.

```
function eatDinner(callback) {
    setTimeout(function() {
        console.log("Eating the dinner ");
    }, 1000);
    callback();
}

function eatDessert() {
    console.log("Eating the dessert ");
}

eatDinner(eatDessert);

// Eating the dinner
// Eating the dessert
```

PRÁCTICA

¿Qué se imprime en la consola y por qué?

```
function functionOne(x) {  
    console.log(x);  
}  
  
function functionTwo(num, func) {  
    func(num);  
}  
  
functionTwo(2, functionOne);
```


Returning values

Las funciones también tienen una propiedad interesante; retornan un valor.

No es obligatorio devolver explícitamente un valor en una función, pero se recomienda que lo haga cuando tenga sentido.

More than one return statement

Una función puede tener más de una declaración de retorno.

```
function calculateTotalPrice(price, taxPercent){  
  if(typeof price !== 'number' || typeof taxPercent !== 'number'){  
    return `You have to pass number values!`;  
  }  
  const theTaxPart = price * taxPercent / 100;  
  return `${price + theTaxPart} €`;  
}  
  
calculateTotalPrice(5, 7); // <= 5.35 €
```


Return another function

También podemos devolver otra función

```
function anotherFunction(text) {  
    console.log(`Hello ${text}!`);  
}  
  
function oneFunction(name) {  
    return anotherFunction(name);  
}  
  
oneFunction("Lluis");  
  
// Prints "Hello Lluis!"
```

Funciones anónimas

Es una función que se declaró sin ningún nombre para referirse a ella.

Generalmente no es accesible después de su creación inicial, a menos que la asignemos a una variable:

```
const anon = function() {  
    console.log("An0nym0us Funct10n");  
}  
  
anon();  
// An0nym0us Funct10n
```


Scope

El scope puede definirse como el alcance que una variable tendrá en tu código.

Existen dos tipos:

- *scope global*
- *scope local*

Scope local

Una variable declarada dentro de la función tiene un alcance para esa función, lo que significa que no es posible acceder a ella fuera de la función.

```
function sayHello() {  
  let firstName = "Ana"; // <== local variable  
  console.log(`Hello ${firstName}!`);  
}  
  
sayHello(); // <== Hello Ana!  
console.log(firstName); // <== ReferenceError: firstName is not defined
```


Scope global

Una variable declarada fuera de una función, se convierte en *global*.

Una variable global tiene alcance global: todos los scripts y funciones en una página web pueden acceder a ella.

```
const firstName = "Ana"; // <== global variable

function sayHello() {
    console.log(`Hello ${firstName}!`);
}

sayHello(); // <== Hello Ana!
console.log(firstName); // <== Ana
```

Scope global

Es posible *modificar* las variables definidas en el alcance global

```
var firstName = "Ana"; // <== global variable
function sayHello() {
    firstName = "Martina";
    console.log(`Hello ${firstName}!`);
}


sayHello();
// 'Hello Martina!'

console.log(`Hello ${firstName}!`);
// 'Hello Martina!'
```

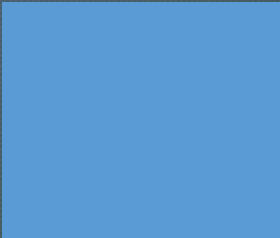

Scope global

También es posible declarar las mismas variables en ambos ámbitos.

La
declaramos



```
var firstName = "Ana"; // <== global variable
function sayHello() {
  let firstName = "Martina";
  console.log(`Hello ${firstName}!`);
}
sayHello();
// 'Hello Martina!'
console.log(`Hello ${firstName}!`);
// 'Hello Ana!'
```



Las variables globales son accesibles desde cualquier función a menos que haya variables locales (variables creadas localmente con el mismo nombre).

function declaration vs. function expression

```
function someName(someParameters) {  
  // some code here  
}
```

Podemos llamar a una función antes de que se inicialice.

```
let someName = function() {  
  // some code here  
}
```

No podemos llamar a una función antes de que se inicialice.

Arrow functions

Una de las actualizaciones de ES6 fue la introducción de una sintaxis más simple para crear funciones: **arrow functions**.

```
// function expression:  
let greeting = function(name) {  
    console.log(`Hello, ${name}`);  
}  
  
// arrow function:  
let greeting = name => {  
    return name;  
}
```


Arrow functions

En caso de que no se pasen parámetros, los **paréntesis vacíos** son **obligatorios**:

```
let greeting = () => console.log("Hello there!");  
  
greeting();  
  
// 'Hello there!'
```

Writing Good Functions

Las funciones son uno de los pilares de la programación. Las funciones nos ayudan a mantener nuestro código limpio y bien organizado.

- Reutilizar código
- Separación de preocupaciones
- Principio de responsabilidad única

Refactorización de código

Es una técnica en el desarrollo de software mediante la cual cambiamos la forma en que se estructura el código, manteniendo la misma funcionalidad.

Es una buena práctica ya que nos ayudará a hacerlo mejor, más modular y más fácil de mantener.

Ejercicio: Miremos este ejemplo

```
function avg(array) {  
  if (array.length === 0) { return; }  
  
  let sum = 0;  
  
  for (let i=0; i < array.length; i++) {  
    sum += array[i];  
  }  
  return sum/array.length;  
}
```

Podemos mejorar aún más esto aislando uno de esos cálculos en una función separada. Necesitamos desglosar el código para que haga lo mismo, pero sea más fácil de entender.

Refactorizando el ejemplo:

```
function sum(array) {  
  let sum=0;  
  for (let i=0; i < array.length; i++) {  
    sum += array[i];  
  }  
  return sum;  
}  
  
function avg(array) {  
  if (array.length === 0) { return; }  
  return sum(array) / array.length;  
}
```

BONUS: Recursividad

Algunas definiciones de funciones se implementan de manera que usan su propia definición. Es como el inicio de la función.

Por ejemplo, una función factorial se define como: factorial de n es n veces el factorial de $n - 1$, excepto el factorial de 0, que es 1. Podemos escribir esto como:

```
function factorial(number) {  
    if (number === 0) { return 1; }  
    return number * factorial(number - 1);  
}  
  
factorial(4);  
  
//24
```


Programación Funcional

Es un paradigma de programación basado en los siguientes principios:

- usa funciones puras
- evita la mutación de datos
- evita los efectos secundarios

La programación Funcional tiene las siguientes características:

-First-Class Function:

```
var add = function (a, b) {  
    return a + b;  
};  
  
add(2, 3); // => 5
```

-High-Order Functions:

```
var add = function (a) {  
    return function (b) {  
        return a + b;  
    };  
};  
  
var add2 = add(2);  
  
add2(3);  
// => 5  
  
add(4)(5);  
// => 9
```


RESUMIENDO

Programación funcional prefiere:

- funciones puras sobre efectos secundarios
- inmutabilidad sobre datos mutantes
- código reutilizable basado en high-order functions
- enfoque declarativo sobre imperativo

Imperative

```
let arr = [1, 2, 3, 4, 5],  
    arr2 = [];  
  
for (var i=0; i<arr.length; i++) {  
    arr2[i] = arr[i]*2;  
}  
  
console.log(arr2);
```

Declarative

```
let arr = [1, 2, 3, 4, 5];  
  
arr2 = arr.map(function(v, i) {  
    return v * 2;  
});  
  
console.log(arr2);
```


Ejercicios de clase:

1) Cree una función llamada `fullName` usando function declaration syntax.

La función debe tomar 2 argumentos, `firstName` y `lastName` y devolver un nuevo string que representa el nombre completo.

```
// Your code here
```

```
console.log( fullName('Bob', 'Smith') ); // expected: Bob Smith
```

2) Cree la misma función como function expression, pero esta vez llámela fullNameExpr.

```
// Your code here

console.log( fullNameExpr('Sarah', 'O\'Connor') );
// expected: Sarah O'Connor
```

3) Cree lo mismo pero como arrow function, y esta vez llámela fullNameArrow.

```
// Your code here

console.log( fullNameArrow('John', 'Wick') ); // expected: John Wick
```


Bonus

Cree la misma función pero de forma concisa (concise arrow function).

```
// Your code here

console.log( fullNameArrow('Mike', 'Smith') );
// expected: Mike Smith
```

Resumiendo. Hemos aprendiendo...

- ✓ a declarar y llamar a funciones en JavaScript
- ✓ qué son y cómo usar parámetros y argumentos
- ✓ cómo usar el scope local y global
- ✓ las reglas de redacción de código limpio y refactorización, reutilización de código y división de responsabilidades.
- ✓ recursividad
- ✓ los conceptos básicos de la programación funcional.