



Introducción al Desarrollo Web

Ing. Marco Aedo López

Fundamentos del desarrollo Frontend

JavaScript

Tema 6

20. Funciones

- Es un bloque de código reutilizable que se ejecuta cuando es llamado
- Sirve para organizar, estructurar y reutilizar código, evitando su repetición
- Facilitan el mantenimiento del código
- También se les llama métodos, procedimientos, etc.

A yellow square with the letters 'JS' in black, representing JavaScript.

20. Funciones Declaradas

- Se define usando la palabra clave `function` seguida de un nombre, parámetros (opcional) y un bloque de código
- Pueden devolver valores con `return`
- Son elevadas (Hoisting): se puede llamar a la función antes de declararla

```
function nombreFuncion(parámetrosOpcionales) {  
    // cuerpo de la función  
    return resultado; // opcional  
}
```

```
function multiplicar(x, y) {  
    return x * y;  
}
```

```
console.log(multiplicar(4, 6)); // 24
```

20. Funciones Expresadas

- Se asigna a una variable o constante
- En lugar de declarar la función directamente con `function nombre() { }`, se escribe la función dentro de una asignación
- No hay hoisting: a diferencia de las declaradas, no se pueden llamar antes de definirlas

```
const nombre = function nombreFuncion(parámetrosOpcionales) {  
    // cuerpo de la función  
    return resultado; // opcional  
};
```

```
const nombre = function(parámetrosOpcionales) {  
    // cuerpo de la función  
    return resultado; // opcional  
};
```

20. Funciones Expresadas

// Función expresada con nombre

```
const sumar = function sumarNumeros(a, b) {  
    return a + b;  
};
```

// Función anónima

```
const sumar = function(a, b) {  
    return a + b;  
};
```

```
console.log(sumar(4, 6)); // 10
```

20. Funciones Flecha

- Forma más compacta y legible de escribir funciones
- Se definen usando el operador `=>` (flecha) en lugar de la palabra clave `function`

```
const nombreFuncion = (parámetrosOpcionales) => {  
  // cuerpo de la función  
  return resultado; // opcional  
};
```

```
const multiplicar = (a, b) => {  
  return a * b;  
};  
  
console.log(multiplicar(5,4));
```

```
const multiplicar = (a, b) => a * b;  
  
console.log(multiplicar(5,4));
```

// Función clásica (Function Declaration)

```
function multiplicar(a, b) {  
    return a * b;  
}
```

// Función expresada (Function Expression)

```
const multiplicar = function(a, b) {  
    return a * b;  
};
```

// Función flecha con bloque (Arrow function)

```
const multiplicar = (a, b) => {  
    return a * b;  
};
```

// Función flecha (Arrow Function)

```
const multiplicar = (a, b) => a * b;
```


20. Funciones Anidadas

- Función definida dentro de otra función
- La función interna sólo puede ser accedida desde la función externa, y tiene acceso a las variables de ésta

```
function saludo() {  
  let nombre = "Ana";  
  
  function mostrarNombre() { // función anidada  
    console.log("Hola " + nombre);  
  }  
  
  mostrarNombre(); // llamada a la función interna  
}  
  
saludo(); // Hola Ana
```

```
function saludo() {  
  let nombre = "Ana";  
  
  function mensaje() { // función anidada  
    return "Hola " + nombre; // retorna algo  
  }  
  
  return mensaje(); // devolvemos el resultado de l  
}  
  
console.log(saludo()); // Hola Ana
```

20. Funciones Anidadas

```
function calculadora(a, b) {  
  function sumar() {  
    return a + b;  
  }  
  
  console.log("Suma:", sumar());  
}  
  
calculadora(5, 3);
```

```
function calculadora(a, b) {  
  function sumar() {  
    return a + b;  
  }  
  function restar() {  
    return a - b;  
  }  
  function multiplicar() {  
    return a * b;  
  }  
  
  console.log("Suma:", sumar());  
  console.log("Resta:", restar());  
  console.log("Multiplicación:", multiplicar());  
}  
  
calculadora(5, 3);
```

20. Funciones Anidadas

```
function contador() {  
    let valor = 0; // variable de la función externa  
  
    function incrementar() { // función anidada  
        valor++;  
        console.log("Contador:", valor);  
    }  
  
    incrementar(); // se llama desde la función externa  
    incrementar();  
    incrementar();  
}  
  
contador();  
// Contador: 1  
// Contador: 2  
// Contador: 3
```

20. Funciones: Scope (Ámbito) y Closure

Scope

- Alcance o contexto en el que las variables, funciones y objetos son accesibles en un programa
- Define dónde puedes usar una variable y dónde no



20. Funciones: Scope (Ámbito) y Closure

- **Scope global:**
 - Las variables declaradas fuera de cualquier función o bloque pertenecen al ámbito global
 - Son accesibles desde cualquier parte del código

```
let x = 10; // variable global
function mostrar() {
    console.log(x); // puede acceder a x
}
mostrar(); // imprime 10
```

20. Funciones: Scope (Ámbito) y Closure

- **Scope local (de función):**
 - Las variables declaradas dentro de una función sólo existen dentro de ella
 - No pueden ser accedidas desde fuera

```
function ejemplo() {  
    let mensaje = "Hola";  
    console.log(mensaje); // accesible aquí  
}  
ejemplo();  
console.log(mensaje); // ❌ Error: mensaje no está definido
```

20. Funciones: Scope (Ámbito) y Closure

- **Scope de bloque (con let y const):**
 - Las variables declaradas dentro de un bloque { } con `let` o `const` sólo existen dentro de ese bloque

```
if (true) {  
  let y = 5;  
  const z = 7;  
}  
console.log(y); // ✗ Error  
console.log(z); // ✗ Error
```

20. Funciones: Scope (Ámbito) y Closure

Closure

- Cuando una función interna recuerda y accede a las variables del scope de su función externa en la que fue creada, aunque la función externa ya haya terminado de ejecutarse
- Las funciones mantienen una referencia al entorno léxico donde fueron definidas

```
function externa() {  
  let mensaje = "Hola";  
  
  function interna() {  
    console.log(mensaje);  
  }  
  
  return interna;  
}  
  
const funcion = externa();  
funcion(); // Imprime: Hola
```


20. Funciones: Scope (Ámbito) y Closure

```
function contador() {  
  let cuenta = 0; // variable privada  
  
  function incrementar() {  
    cuenta++;  
    return cuenta;  
  }  
  
  return incrementar;  
}  
  
const miContador = contador();  
  
console.log(miContador()); // 1  
console.log(miContador()); // 2  
console.log(miContador()); // 3
```

```

function crearContador(inicial) {
    let cuenta = inicial; // variable privada, sobre la que se creará un closure

    function incrementar() {
        cuenta++;
        return cuenta;
    }

    function decrementar() {
        cuenta--;
        return cuenta;
    }

    function resetear() {
        cuenta = inicial;
        return cuenta;
    }

    // Retornamos una función interna que recibe la acción deseada como argumento
    return function operacion(accion) {
        if (accion === "inc") return incrementar();
        if (accion === "dec") return decrementar();
        if (accion === "reset") return resetear();
        if (accion === "get") return cuenta;
        return undefined;
    };
}

// Uso
const contador = crearContador(5);
// El closure "cierra" sobre la variable cuenta, mantiene viva mientras la función retornada exista
console.log(contador.name); // "operacion"
console.log(contador("get")); // 5 //llama a la función interna correspondiente
console.log(contador("inc")); // 6
console.log(contador("inc")); // 7
console.log(contador("dec")); // 6
console.log(contador("reset")); // 5

```

20. Funciones: Parámetros por defecto y operador rest

- Los **parámetros por defecto** (default parameters) permiten que una función asigne un valor predeterminado a un parámetro si no se le proporciona un valor al invocar la función
- Esto evita tener que verificar manualmente si un argumento es undefined dentro de la función

```
function saludar(nombre = "Invitado") {  
  console.log("Hola "+nombre+"!");  
}
```

```
saludar();           // Hola, Invitado!  
saludar("Juancito"); // Hola, Juancito!
```

20. Funciones: Parámetros por defecto y operador rest

```
function calcularTotal(precio, cantidad = 1, impuesto = 0.18) {  
  const subtotal = precio * cantidad;  
  const total = subtotal * (1 + impuesto);  
  return total;  
}
```

// Ejemplos de uso

```
console.log(calcularTotal(100));           // 100 * 1 * 1.18 = 118  
console.log(calcularTotal(50, 3));         // 50 * 3 * 1.18 = 177  
console.log(calcularTotal(200, 2, 0.1));   // 200 * 2 * 1.1 = 440  
console.log(calcularTotal(150, undefined, 0.2)); // 150 * 1 * 1.2 = 180
```

20. Funciones: Parámetros por defecto y operador rest

- El **operador rest** (...) permite capturar un número indefinido de argumentos de una función en un solo arreglo
- Es muy útil cuando no sabemos cuántos argumentos recibirá la función

```
function ejemplo(a, b, ...resto) {  
  console.log(a);      // primer parámetro  
  console.log(b);      // segundo parámetro  
  console.log(resto);  // todos los demás  
}
```

```
ejemplo(1, 2, 3, 4, 5); // 1, 2, [3, 4, 5]
```

```
function sumar(...numeros) {  
  let total = 0;  
  for (let i = 0; i < numeros.length; i++) {  
    total += numeros[i];  
  }  
  return total;  
}
```

// Ejemplos de uso

```
console.log(sumar(10, 20, 30)); // 60  
console.log(sumar(5, 15));      // 20  
console.log(sumar(7));          // 7
```

20. Funciones: Parámetros por defecto y operador rest

```
function sumar(precioInicial, ...otrosPrecios) {  
    let total = precioInicial;  
    for (let p of otrosPrecios) {  
        total += p;  
    }  
    return total;  
}
```

// Ejemplos de uso

```
console.log(sumar(100, 50, 200, 150)); // 500  
console.log(sumar(10, 20, 30));         // 60  
console.log(sumar(5));                   // 5
```

20. Funciones de orden superior y Callbacks

- Funciones de **orden superior** son funciones que operan sobre otras funciones, permitiendo un estilo de programación más flexible y funcional
- Cumple al menos una de estas dos condiciones:
 - Recibe una o más funciones como argumentos
 - Devuelve otra función como resultado
- Permite modularidad y reutilización del código

A yellow square logo with the letters 'JS' in a bold, black, sans-serif font.

20. Funciones de orden superior y Callbacks

```
function aplicarOperacion(a, b, operacion) {  
    return operacion(a, b);  
}
```

```
function sumar(x, y) { return x + y; }  
function multiplicar(x, y) { return x * y; }
```

```
console.log(aplicarOperacion(5, 3, sumar));          // 8  
console.log(aplicarOperacion(5, 3, multiplicar));    // 15
```


20. Funciones de orden superior y Callbacks

```
function crearMultiplicador(factor) {  
  return function(numero) {  
    return numero * factor;  
  }  
}  
  
const duplicar = crearMultiplicador(2);  
const triplicar = crearMultiplicador(3);  
  
console.log(duplicar(5)); // 10  
console.log(triplicar(5)); // 15
```

20. Funciones de orden superior y Callbacks

- Un **callback** es simplemente una función que se pasa como argumento a otra función y que se ejecuta dentro de esa función
- Los callbacks se usan para controlar el flujo de ejecución, especialmente en operaciones asíncronas o cuando queremos ejecutar código después de que ocurra algo

```
function saludar(nombre, callback) {  
    console.log("Hola "+nombre+"!");  
    callback();  
}
```

```
function despedirse() {  
    console.log("Adiós!");  
}
```

```
saludar("Juancito", despedirse);
```

20. Funciones de orden superior y Callbacks

```
function operar(a, b, fn) { // fn es el callback
  return fn(a, b);
}

function sumar(x, y) {      // sumar es solo una función normal
  return x + y;
}

console.log(operar(3, 4, sumar)); // 7
```

```
function saludar(nombre, callback) {
  console.log("Hola "+nombre+"!");
  callback();
}

saludar("Ana", function() { // funcion anónima
  console.log("Nos vemos pronto!");
});
```

20. Funciones de orden superior y Callbacks

```
setTimeout(function() {  
    console.log("Esto se ejecuta después de 2 segundos");  
}, 2000);
```

```
function procesarPedido(nombreProducto, callback) {  
    console.log(`Procesando pedido de ${nombreProducto}...`);  
  
    // Simulamos el tiempo de preparación del pedido  
    setTimeout(function() {  
        const mensaje = `Pedido de ${nombreProducto} listo para entregar`;  
        callback(mensaje); // ejecutamos el callback al finalizar  
    }, 3000); // 3 segundos  
}  
  
// Uso de la función  
procesarPedido("Pizza", function(resultado) {  
    console.log(resultado);  
    console.log("Notificando al cliente que su pedido está listo...");  
});
```

21. Estructuras de Datos

- Una manera organizada de almacenar y manejar datos para poder accederlos, modificarlos y procesarlos de manera eficiente
- Permiten agrupar y relacionar múltiples valores
 - Array (arreglo / lista)
 - Set
 - Map
 - Objeto (Object)



22. Estructuras de Datos - Array

- Arreglo o lista, es una estructura de datos que permite almacenar una colección ordenada de elementos
- Cada elemento tiene un índice numérico, empezando desde 0, y puede ser de cualquier tipo de dato, incluso otros arrays u objetos
- Son dinámicos!!!

```
// ARRAYS
```

```
let numeros = [10, 20, 30, 40];  
let frutas = ["manzana", "uva", "plátano"];  
let varios = ["hola", 5, true, 3.14];
```

```
// Arrays vacios
```

```
let array1=[];  
let array2=new Array();
```

22. Estructuras de Datos - Array

- Características:
 - Ordenado: cada elemento tiene un índice numérico
 - Heterogéneo: puede contener distintos tipos de datos
 - Dinámico: puede cambiar de tamaño (agregar o eliminar elementos)
 - Objeto: un array es un objeto con propiedades y métodos especiales

```
// Array literal
```

```
let numeros = [10, 20, 30];  
console.log(numeros);
```

```
▶ (3) [10, 20, 30]
```

```
// Array con distintos tipos de datos
```

```
let mixto = [1, "Hola", true];  
console.log(mixto);
```

```
▶ (3) [1, 'Hola', true]
```

22. Estructuras de Datos - Array

```
let nums = new Array(1, 2, 3, 4, 5);  
let colores = new Array("rojo", "verde", "azul");
```

```
let texto = "hola";  
let letras = Array.from(texto);
```

// cuidado



```
let num=[5];  
let num1=new Array(5);
```

▸ [5]

▸ (5) [empty × 5]

22. Estructuras de Datos - Array

- Declarar con let y const
 - Ambas son válidas
 - let: cuando se planea reasignar el arreglo
 - const: cuando el arreglo no cambiará de referencia (pero si se puede modificar los elementos)

```
const nums = [1, 2, 3];  
nums[0] = 99;      //  permitido  
nums = [10, 20];   //  error
```

22. Estructuras de Datos - Array

- Acceso a elementos

```
// Array literal  
let numeros = [10, 20, 30];  
console.log(numeros);  
console.log(numeros[0]); // 10
```

- length: devuelve el número de elementos del array

```
console.log(numeros.length); // 3
```

22. Estructuras de Datos - Array

Métodos más comunes

- Agregar elementos

```
// Array literal  
let numeros = [10, 20, 30];  
console.log(numeros);
```

```
numeros.push(40); // al final  
console.log(numeros);
```

```
numeros.unshift(5); // al inicio  
console.log(numeros);
```

- Eliminar elementos

```
numeros.pop(); // del final  
console.log(numeros);
```

```
numeros.shift(); // del inicio  
console.log(numeros);
```

22. Estructuras de Datos - Array

- Recorrer un arreglo

```
const misNumeros = [10, 20, 30, 40, 50];
```

```
for (let i = 0; i < misNumeros.length; i++) {  
    console.log(misNumeros[i]);  
}
```

```
for (const n of misNumeros) {  
    console.log(n);  
}
```

```
for (const i in misNumeros) {  
    console.log(i, misNumeros[i]);  
}
```

22. Estructuras de Datos - Array

- Recorrer un arreglo

```
function mostrarNumero(n) {  
    console.log(n);  
}
```

```
misNumeros.forEach(mostrarNumero);
```

```
//con función anónima
```

```
misNumeros.forEach(function(n) {  
    console.log(n);  
});
```

```
//con función flecha
```

```
misNumeros.forEach(n => console.log(n));
```

22. Estructuras de Datos – Array Multidimensional

- Son arrays que contienen arrays

```
let matriz = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];
```

```
console.log(matriz[0][0]); // 1  
console.log(matriz[1][2]); // 6  
console.log(matriz[2][1]); // 8
```

```
let matriz = [];
```

```
matriz[0] = [1, 2, 3];  
matriz[1] = [4, 5, 6];  
matriz[2] = [7, 8, 9];
```

```
console.log(matriz);
```

22. Estructuras de Datos – Array Multidimensional

```
let filas = 2;
let columnas = 3;
let matrizBi = [];

for (let i = 0; i < filas; i++) {
  matrizBi[i] = [];
  for (let j = 0; j < columnas; j++) {
    matrizBi[i][j] = i + j; // ejemplo: valor = suma de índices
  }
}

console.log(matrizBi); // [[0,1,2],[1,2,3]]
console.log(matrizBi.length);
console.log(matrizBi[0].length);
```

23. Estructuras de Datos - Set

- Representa una **colección de valores únicos**, es decir, no permite elementos duplicados
- No mantiene índices
- Cada valor puede aparecer una sola vez
- Es iterable y heterogéneo

```
// Crear un Set vacío
```

```
const losNumeros = new Set();
```

```
// Crear un Set con elementos
```

```
const letras = new Set(["a", "b", "c", "a"]);
```

```
console.log(letras); // Set { 'a', 'b', 'c' }
```

```
let mix = new Set([1, "uno", true]);
```

```
console.log(mix);
```


Método / Propiedad	Descripción	Ejemplo
add(valor)	Agrega un valor al Set. Si ya existe, no lo duplica	const numeros = new Set(); numeros.add(1); numeros.add(2);
has(valor)	Verifica si un valor existe en el Set	numeros.has(1); // true numeros.has(3); // false
delete(valor)	Elimina un valor del Set si existe	numeros.delete(2);
clear()	Elimina todos los elementos del Set	numeros.clear();
size	Devuelve la cantidad de elementos en el Set	let cantidad=letras.size);

```
let numeros = new Set();

numeros.add(1);
numeros.add(2);
numeros.add(2); // ignorado, ya existe
numeros.add(3);

console.log(numeros.size); // 3
console.log(numeros.has(2)); // true
numeros.delete(1);
console.log(numeros); // Set {2, 3}
```

```
let frutas = new Set(["manzana", "pera", "uva"]);

for (let fruta of frutas) {
  console.log(fruta);
}
```

23. Estructuras de Datos - Set

```
let lista = [1,2,3,3,4,4,5];  
let sinDuplicados = [...new Set(lista)];  
console.log(sinDuplicados); // [1,2,3,4,5]
```

```
let arr = Array.from(frutas); //frutas es Set  
console.log(arr);
```

23. Estructuras de Datos - Set

```
// Creamos dos conjuntos con algunos elementos repetidos
let num1 = new Set([1, 2, 3, 3, 4, 5]);
let num2 = new Set([4, 5, 6, 7, 8]);

// Unión: elementos que están en num1 o en num2
let union = new Set([...num1, ...num2]);

// Intersección: elementos que están en num1 y también en num2
let interseccion = new Set([...num1].filter(x => num2.has(x)));

// Diferencia: elementos que están en A pero no en num2
let diferencia = new Set([...num1].filter(x => !num2.has(x)));

// Mostramos los resultados
console.log("num1:", [...num1]);
console.log("num2:", [...num2]);
console.log("Unión:", [...union]);
console.log("Intersección:", [...interseccion]);
console.log("Diferencia num1-num2:", [...diferencia]);
```

Ejercicios

1. Crea una función llamada promedio que reciba tres números y devuelva su promedio. 4 versiones: declarada, expresada, flecha con bloque y flecha simplificada
2. Crea una función flecha llamada evaluarNumero que reciba un número y devuelva un mensaje indicando si es positivo, negativo o cero
3. Crea una función operar que reciba dos números y una función como parámetro (la operación que se aplicará, crear para las 6 operaciones aritméticas conocidas)
4. Crea una función calcularTotal que reciba dos precios y una función callback para mostrar el resultado
5. Crea una función llamada mostrarResultado que use otra función interna para calcular el doble de un número
6. Crea una función que reciba un arreglo de números y devuelva la suma de los positivos
7. Crea una función que reciba un arreglo y devuelva un nuevo arreglo con el menor y el mayor número, en ese orden
8. Crea una función que devuelva un nuevo arreglo con los elementos en orden inverso
9. Crea una función que reciba un arreglo y un número, y devuelva cuántos elementos son múltiplos de ese número
10. Dado un arreglo de números, crea una función que devuelva un nuevo arreglo con los números pares elevados al cuadrado

11. Crea una función que reciba un array y devuelva cuántos valores únicos contiene
12. Crea una función que reciba dos arreglos heterogéneos y que devuelva un arreglo con la unión, intersección y diferencia
13. Crea una función soloUnaVez(arr) que reciba un arreglo de números y retorne un nuevo arreglo con los elementos que aparecen exactamente una vez en el original
14. Crea una función diferenciaSimetrica(a, b) que devuelva un arreglo con los elementos que están en uno solo de los arreglos (no en ambos)
15. Crea una función booleana esSubconjunto(a, b) que determine si todos los elementos de a están contenidos en b (detector de subconjunto)