

TEMA 4 – FUNCIONES EN JAVASCRIPT.

Las **funciones** son bloques fundamental de código identificados con un nombre. Permiten agrupar y reutilizar código, y son esenciales para la programación modular, estructurada y funcional. Las instrucciones que se encuentran dentro de una función se ejecutan cada vez que se llama a la función y a las funciones se les llaman escribiendo el nombre seguida de paréntesis. Esta llamada se puede realizar desde cualquier parte del código y cada vez que sea necesario, lo que rompe completamente el procesamiento secuencial del programa.

1. Declaración de una función:

Hay muchas formas de definir una función (como se verá más adelante), pero al principio se utilizará lo que se conoce como **notación declarativa**.

Las funciones se declaran usando la palabra clave **function**, el nombre seguido de paréntesis y el código entre llaves. Para llamar a una función (**invocarla**) tenemos que declarar su nombre con un par de paréntesis al final.

```
11  <script>
12      function mostrarMensaje(){
13          return("Hola mundo");
14      }
15
16      alert(mostrarMensaje());
17  </script>
```

1.1. Valores de retorno:

Las funciones pueden tener o no tener un valor de retorno. Así que si **no** se especifica un **return** la función devuelve **undefined** por defecto. Por otro lado, **las funciones sólo retornan un valor**, por lo que si queremos retornar más de uno los podemos agrupar en arrays o objetos.

Además, como se puede ver en la función `mostrarMensaje()` de más arriba, la función no tiene argumentos. JavaScript tiene una característica notable que es que **no da error** si llamas a una **función con más argumentos de los que espera**. Simplemente, los ignora.

```

11 <script>
12     function mostrarMensaje(){
13         return("Hola mundo");
14     }
15
16     alert(mostrarMensaje("Extra"));
17 </script>

```



Como ya pasaba en Java, el orden de los argumentos es crucial. Los argumentos se asignan a los parámetros en el orden en que se pasan y se crea un objeto llamado **arguments**, el cual es un objeto similar a un array (no lo es como tal) que te permite acceder a los argumentos que se pasaron a esa función. Tiene propiedades como `length` y permite acceder a los elementos por su índice.

```

function miFuncion() {
    console.log(arguments);
    console.log(arguments[0]);
    console.log(arguments[1]);
    console.log(arguments.length);
}

miFuncion(10, 'hola', true);

```



2. Tipos de funciones:

Cuando hablamos de **funciones** en JavaScript estamos hablando de **objetos**. Esto significa que se les trata como cualquier objeto y **permite** asignar una función a una variable, pasarla como argumento de otras funciones (callbacks), devolverla desde una función e incluso agregarles propiedades y métodos.

Un concepto muy importante en la declaración de funciones en JavaScript es “**hoisting**” que significa que las declaraciones de variables y funciones son “movidas” a la parte superior de su contexto (ya sea global o dentro de una función) antes de que se ejecute el código.

- Las funciones pueden ser declaradas de forma **explícita o declarativa**, cargándose en tiempo de compilación y permitiendo su uso antes de la declaración. (hoisting)

```

mostrarMensaje();

function mostrarMensaje() {
    console.log("¡Hola mundo!");
}

```

¡Hola mundo!

prueba.html:28

- Las funciones pueden ser definidas mediante **expresiones**. Este tipo de funciones se evalúan en tiempo de ejecución y no soportan hoisting. La característica de estas funciones es que lo que se pretende hacer es relacionar la definición de una función con el identificador de una variable, por lo que el identificador de la función carece de utilzada, puesto que para acceder a la función hay que hacerlo usando el identificador de la variable.

```
const bienvenido = function sesionIniciada(){
  console.log("Bienvenido de nuevo");
}

bienvenido();
sesionIniciada();
```

✖ ▶ Uncaught ReferenceError: sesionIniciada is not defined prueba.html:97
at prueba.html:97:9

- Las expresiones de **función** pueden ser **anónimas** (sin nombre) o **funciones lambda**, por lo que no se les puede invocar a sí mismas, y no se pueden hacer recursivas. Estas funciones las puedes asignar directamente a una variable, pasarlas como argumento o usarlas en expresiones de funciones.

```
let restar = function (a, b) {
  return a - b;
};

console.log(restar(5, 3));
```

2

prueba.html:35

Por el contrario, si invertimos el orden, el resultado sería:

✖ ▶ Uncaught ReferenceError: Cannot access 'restar' before initialization prueba.html:31
at prueba.html:31:21

- Las **funciones callback**, son funciones anónimas que pueden pasarse como parámetros a otra función. Su estrategia es definir una función que, al ser anónima, se vincula a una variable y luego se utiliza esa variable como parámetro de otra función, de manera que esta última pueda ejecutar el contenido de la primera.

```
const saludo = function () {  
  return "Bienvenido de nuevo,";  
}  
  
const usuario = function (callback) {  
  console.log(callback() + "Javier");  
}  
  
usuario(saludo);
```

Bienvenido de nuevo,Javier

[prueba.html:138](#)

- Las **funciones flecha** representan una simplificación sintáctica de las funciones anónimas tradicionales. Son una forma más moderna y concisa de escribir funciones. Estas son sus principales características:

- **Sintaxis concisa:** No es necesario usar la palabra clave function, return, ni las llaves si la función solo tiene una expresión.

```
const suma = (a, b) => a + b;
```

- **Constantes por defecto:** Se recomienda declarar las funciones flecha utilizando const en lugar de let, ya que no puede cambiar su referencia.

```
//función flecha  
const restar1 = (a, b) => a - b;  
//función tradicional  
const restar2 = function (a, b) {  
  return a - b;  
};
```

- **No son hoisted:** No son elevadas como las funciones tradicionales. Esto significa que no pueden ser invocadas antes de su declaración en el código.
- **Uso de {} y return:** Si la función flecha tiene más de una línea de código o más de una instrucción, es necesario utilizar llaves {} y la palabra clave return.

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

- **Las funciones flecha no tienen su propio this.** Esto significa que si intentas usar una función flecha como método en un objeto, no funcionará como se espera, porque this no se refiere al objeto.

```
const coche = {
  marca: "Toyota",
  modelo: "Corolla",
  mostrarInfo: () => {
    console.log(`Este coche es un ${this.marca} ${this.modelo}`);
  }
};

coche.mostrarInfo(); //Este coche es un undefined undefined
```

Por lo que algo negativo de estas funciones es que a veces pueden ser demasiado cortas y ser confundidas con asignaciones y no pueden ser usadas como métodos o constructores por no tener el contexto (this). Una pequeña comparativa sería:

```
const persona = {
  nombre: 'Pepe',
  apellido: 'García',

  // Función regular
  consulta: function () {
    return `${this.nombre} ${this.apellido}`;
  },

  // Función flecha
  consultar: () => {
    // En este contexto, `this` no se refiere al objeto persona
    return `${this.nombre} ${this.apellido}`;
  }
};

console.log(persona.consulta()); // Salida: Pepe García
console.log(persona.consultar()); // Salida: undefined undefined
```

3. Ámbito de las funciones y variables:

Las funciones y las variables deben estar dentro del ámbito en el que se llaman. Esto son algunos de los conceptos clave para sacar el mayor provecho a la utilidades de JavaScript:

- **Hoisting:** Es el comportamiento por el cual las declaraciones de las funciones y variables son elevadas al inicio de su contexto de ejecución.
 - Funciona con las funciones declaradas (con nombre).

```
console.log(multiplicar(2, 3)); // 6

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

- **No hoisting:** Se refiere a la incapacidad de llamar a funciones definidas como expresiones de función antes de su declaración en el código.

- Funciones definidas como expresiones (flecha o anónimas).

```
console.log(dividir(10, 2)); // Error

let dividir = function (a, b) {
  return a / b;
};
```

✖ ▶ Uncaught ReferenceError: Cannot access 'dividir' before initialization
at prueba.html:95:21

- **Ámbito global:** Es el ámbito más amplio, donde las variables y funciones pueden ser accedidas desde cualquier parte del código. Todo lo que se declare en el ámbito global está disponible en todo el código. Las variables y funciones son “visibles”.

```
107 let a = 1;
108
109 function global() {
110   console.log(a);
111 }
112
113 global(); // 1
114 console.log(a); // 1
```

1	prueba.html:103
1	prueba.html:107

- **Ámbito Local o de función:** Es el ámbito creado dentro de una función. Las variables y funciones declaradas dentro de una función son accesibles solo dentro de esa función, y no pueden ser accedidas desde fuera.

```
function local() {
  let a = 2;
  console.log(a); // 2
}


local();
console.log(a); // Error: a is not defined
```

✖ ▶ Uncaught ReferenceError: a is not defined
at global (prueba.html:103:25)
at prueba.html:106:9

- **Ámbito de bloque:** Introducido en ES6, se refiere al ámbito creado por estructuras de control como if, for, o while. Las variables declaradas con let o const dentro de un bloque solo son accesibles dentro de ese bloque.

```
for (let i = 0; i < 10; i++) {
  console.log(i); // Imprime números del 0 al 9
}

console.log(i); // Error: i is not defined
```

0	prueba.html:118
1	prueba.html:118
2	prueba.html:118
3	prueba.html:118
4	prueba.html:118
5	prueba.html:118
6	prueba.html:118
7	prueba.html:118
8	prueba.html:118
9	prueba.html:118
<div>  ▶ Uncaught ReferenceError: i is not defined at prueba.html:121:21 </div>	

Algo que aprendisteis el año pasado y que debéis recordar es que los parámetros que reciben los argumentos se pasan por “valor” o por “referencia”. Esto influye en cómo se comportan los datos dentro de la función cuando los modificamos.

- **Paso por valor:** Cuando pasas un valor primitivo a una función (Number, String, boolean, null, undefined), se pasa una copia del valor. Es decir, cualquier cambio que se haga a ese parámetro dentro de la función no afecta al valor original fuera de la función.

```
let num1 = 7;
let num2 = 8;

function menor(primer, segundo) {
  let elmenor = primer;
  if (segundo < primer) {
    elmenor = segundo;
  }
  return elmenor;
}

console.log(menor(num1, num2));
```

7

prueba.html:156

Cuando la ejecución del programa llama a la última línea y se invoca a la función, automáticamente el valor de número1 (7) se copia en la variable primer, y el valor de número 2 (8) se copia en la variable segundo. Así la función hace sus cálculos y devuelve el valor 7. La clave de este proceso reside en la expresión “se copia”, porque si en el interior de

la función se modifica el valor de primero o el de segundo, las variables globales num1 y num2 seguirán teniendo sus valores iniciales.

```
let num1 = 7;
let num2 = 8;

function menor(primer, segundo) {
  num1 = 10;
  num2 = 4;
  let elmenor = primero;
  if (segundo < primero) {
    elmenor = segundo;
  }
  return elmenor;
}

console.log(menor(num1, num2));
console.log(num1);
console.log(num2);
```

7	prueba.html:158
10	prueba.html:159
4	prueba.html:160

- **Paso por referencia:** Cuando pasas un objeto o array a una función, se pasa la referencia al objeto en la memoria, no una copia. Eso significa que si modificas el objeto o array dentro de la función, los cambios afectan al objeto original fuera de la función.

Para establecer valor por referencia la mayoría de los lenguajes de programación tienen su propia sintaxis. Por ejemplo en Java se presenta como menor(&primero, &segundo). En JavaScript, en cambio, no **existe esa posibilidad de calificar un parámetro cualquiera por referencia**, sino que existen ciertas variables que cuando hacen referencia a un objeto, su propio identificador es ya una referencia.

```
let vector = [6, 2, 9, 5, 3];

function menor(elarray) {
  let elmenor = elarray[0];
  let posicion = 0;
  for (let i = 0; i < elarray.length; i++) {
    if (elarray[i] < elmenor) {
      elmenor = elarray[i];
      posicion = i;
    }
    elarray[posicion] = -1;
  }
  return elmenor;
}

console.log(menor(vector));
console.log(vector);
console.log(vector);
```

► (5) [-1, 2, 9, 5, 3]	prueba.html:176
► (5) [-1, 2, 9, 5, 3]	prueba.html:177