

INVESTIGACIÓN JAVASCRIPT

Carla, David, Aleeza, Gaizka , Érika

TEMAS

A mind map with a central black circle labeled 'TEMAS'. Six lines radiate from this circle to six colored rounded rectangles: 'Sintaxis' (olive), 'Variables' (purple), 'Constantes' (blue), 'Tipos de datos y operadores' (green), 'Métodos predefinidos' (grey), and 'Funciones' (orange). The 'Funciones' node further branches into two more rounded rectangles: 'Nominativas y Anónimas' (purple) and 'Parámetros y argumentos' (blue). The background features large, abstract, wavy shapes in light purple and pink.

Sintaxis

Variables

Constantes

Tipos de datos y
operadores

Métodos predefinidos

Funciones

Nominativas y Anónimas

Parámetros y argumentos

Syntax

UBICACIÓN

01

DENTRO DE ETIQUETAS <SCRIPT> EN EL HTML

Este es el método más común y permite que el código se ejecute cuando el navegador encuentra la etiqueta.

```
<script>
  // Aquí se escribe el código JS
</script>
```

02

EN UN ARCHIVO EXTERNO

Se puede guardar en un archivo con extensión .js y luego incluirlo en el HTML utilizando la etiqueta <script>

```
<script src="js/main.js"></script>
```

03

EN LA CONSOLA DEL NAVEGADOR

Para pruebas rápidas o depuración, se puede escribir código directamente en la consola de desarrollo del navegador.

```
> console.log(
  '%cHola mundo!',
  'color: #f709bb; font-style:
  italic; text-decoration:
  underline; font-size: 1.5em;'
);
```

Hola mundo!

VM187:1

UNA SOLA LÍNEA

Se utilizan dos barras (//) para comentar una línea de código.

```
1 // Este es un comentario de una sola línea
```

VARIAS LÍNEAS

Se utilizan /* para iniciar el comentario y */ para cerrarlo.

```
1 /* Este es un comentario  
2    de varias líneas */
```

COMENTARIOS

ETIQUETAS

```
1 <script>  
2   console.log("Hola, mundo!");  
3 </script>
```

Se utiliza para incluir código JavaScript en un documento HTML. Puede estar en la sección <head> o al final de la sección <body> para asegurar que el DOM esté completamente cargado antes de ejecutar el script.

<SCRIPT>

<noscript>

JavaScript está deshabilitado en su navegador. Algunas funcionalidades pueden no estar disponibles.

</noscript>

Se utiliza para mostrar contenido alternativo en caso de que JavaScript esté deshabilitado en el navegador del usuario.

<NOSCRIPT>

MODO ESTRICTO

Se puede activar el modo estricto en JavaScript utilizando la declaración **"use strict"**. Esto ayuda a evitar errores comunes y a escribir un código más seguro y limpio.

```
"use strict";  
x = 3.14;
```

```
// Esto lanzará un error  
porque x no ha sido declarado
```

```
1 <script src="archivo.js" defer></script>
```

WEBPAGE

Una página web puede contener múltiples scripts, y es común que se estructuren de forma que se carguen de manera eficiente. Los scripts pueden ser asíncronos o diferidos utilizando los atributos async o defer en la etiqueta

Un archivo JavaScript (con extensión .js) puede contener funciones, variables y otros elementos de JavaScript. Se puede incluir en una página web para modularizar el código y mejorar la mantenibilidad.

```
1 // archivo.js  
2 function saludar(nombre) {  
3     console.log("Hola, " + nombre);  
4 }
```

ARCHIVO .JS

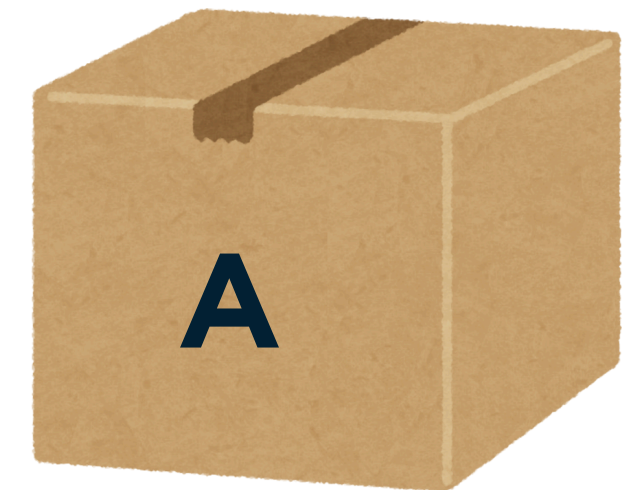
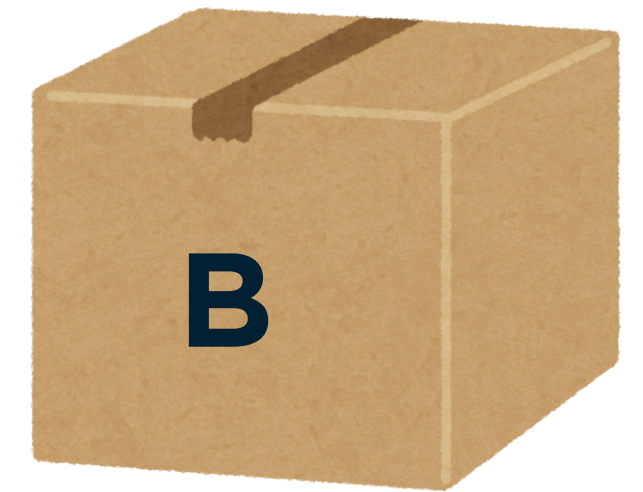
Variables

VARIABLES

Una **variable** es una palabra que **representa** algo que **cambia** o experimenta algún tipo de cambio.

En programación, una variable es como un “**almacén** o caja con un **nombre**” en la cual podemos guardar objetos (números, texto, etc).

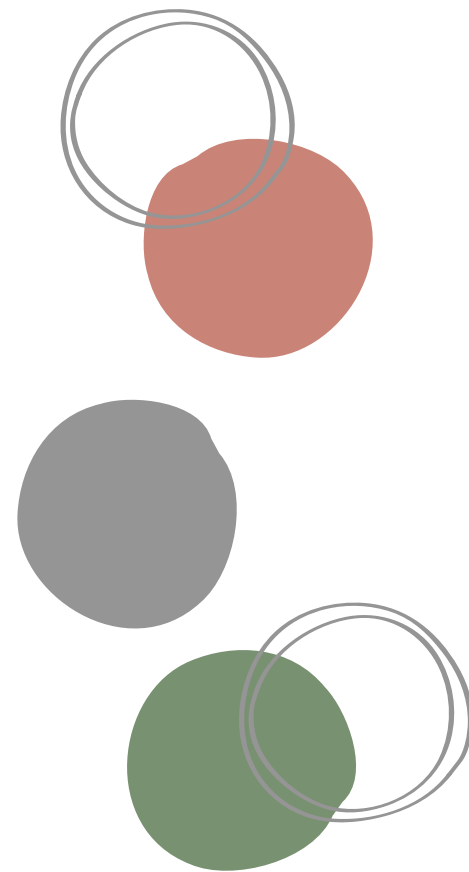
Las variables poseen un nombre llamado: **identificador**. Es como ponerles un nombre a la caja con alguna **etiqueta**.



VARIABLES

PALABRAS CLAVE

- **let-** es la forma moderna de declaración de una variable. Alcance de bloque.
- **var-** es la declaración de variable de la vieja escuela. Alcance funcional o global.
- **const-** es como let, pero una vez asignado, el valor de la variable no podrá modificarse.



ELEMENTOS

- **Nombre:** es el identificador único que se utiliza para referirse a la variable.
- **Tipo de dato:** es el tipo de información que se almacena en la variable, como números enteros, decimales etc.
- **Valor:** es el contenido actual almacenado en la variable.

TIPOS

(DATOS PRIMITIVOS)

STRING

Contiene secuencias de caracteres, como texto.

```
let userName = "Carla Siles";
```

BOOLEAN

Representa valores verdaderos (true) o falsos (false)

```
let isStudent = true;  
let isTeacher = false;
```

NUMBER

Almacena valores numéricos, ya sean enteros o decimales

```
let age = 29;  
let height = 1.60;
```

SYMBOL

Identificador único utilizado para propiedades de objetos

```
const uniqueId = Symbol('uniqueId');  
const hiddenKey = Symbol('hiddenKey');  
let user = {  
  name: "Carla",  
  age: 29,  
  [uniqueId]: "79111111",  
  [hiddenKey]: "hiddenData" };
```

UNDEFINED//NULL

Undefined: Cuando se ha declarado, pero sin asignar valor.

Null: Indica que una variable no tiene valor o está vacía.

```
//Undefined//  
let name;
```

```
let selectedProduct = null;
```

BIGINT

Para manejar números enteros grandes

```
let PlanB = BigInt(20000000000000000);
```

TIPOS (DATOS REFERENCIA)

FUNCTION

Un bloque de código reutilizable que puede ser llamado.

```
let greet = function (name) {  
  return `Hola, ${name}!`;};
```

OBJECT

Estructura de datos que puede contener propiedades y métodos.

```
let myCv = {  
  profession: "full-Stack Developer",  
  year: "2025"};
```

ARRAY/DATE

Array: Almacena una lista ordenada de elementos.

Date: Trabaja con fechas y horas, permitiendo obtener, manipular y formatear valores de tiempo.

```
let languages = ["HTML", "CSS", "JavaScript"];
```

```
let today = new Date();
```

SET

Almacena valores únicos, eliminando duplicados automáticamente.

```
let mySet = new Set();  
// Agregar valores al Set  
mySet.add(1);  
mySet.add(2);  
mySet.add(3);
```

MAP

Sirve para almacenar pares clave-valor únicos, permitiendo cualquier tipo de dato como clave.

```
let myMap = new Map();  
// Agregamos elementos al Map  
myMap.set("userName", "Carla Siles");  
myMap.set("age", 29);  
myMap.set("profession", "Full-stack developer");
```

REGEXP

Sirve para buscar y manipular patrones dentro de cadenas de texto, como validación de formatos o encontrar coincidencias.

```
let regex = /hello/i;  
// Usamos la expresión regular  
let text = "Hello, world!";  
// Buscamos coincidencias  
let result = text.match(regex);
```



Constantes

CONSTANTES

Las variables constantes presentan un **block scope** tal y como las variables `var` y `let`, la diferencia es que una constante, no puede cambiar a través de la reasignación.

Por tanto, **no se puede redeclarar, ni reasignar.**

En el caso de que la asignación a la constante sea un objeto, el objeto sí que puede ser alterado.

Para declarar una variable constante (inmutable se usa **Const** en vez de `let`).

Las variables declaradas utilizando `const` se llaman constantes y no pueden ser alteradas.

```
//const//  
const myBirthday = "12-08-1995"
```

Tipos de Operadores



Tipos de Operadores

Estos son los diversos operadores que admite JavaScript:

Operadores Aritméticos

Operadores de Asignación

Operadores de cadena

Operadores de comparación

Operadores lógicos

Operadores Bitwise

Operadores especiales





OPERADORES ARITMÉTICOS

+ **(Suma)**: Suma dos valores.

- **(Resta)**: Resta el segundo valor del primero.


***** **(Multiplicación)**: Multiplica dos valores.

/ **(División)**: Divide el primer valor entre el segundo.

% **(Módulo)**: Devuelve el residuo de la división entre dos valores.

++ **(Incremento)**: Aumenta el valor de la variable en 1.



-- **(Decremento)**: Disminuye el valor de la variable en 1.



```
let a = 10, b = 5;

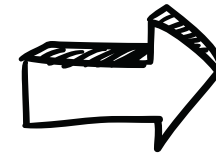
console.log(a + b); // 15 (Suma)
console.log(a - b); // 5 (Resta)
console.log(a * b); // 50 (Multiplicación)
console.log(a / b); // 2 (División)
console.log(a % b); // 0 (Módulo, resto de 10 dividido entre 5)

let c = 5;
console.log(c++); // 5, luego c será 6 (Incremento después de la operación)
console.log(--c); // 5, primero decrementa c a 5 y luego muestra 5 (Decremento antes de la operación)
```



OPERADORES DE CADENA

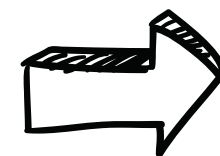
- **+** Concatenación.
- **+=** Concatenar y Asignar



```
let saludo = 'Hola';  
let nombre = 'Juan';  
let mensaje = saludo + ' ' + nombre + '!'; // Resultado: 'Hola Juan!'  
  
saludo += ', bienvenido'; // saludo = saludo + ', bienvenido';  
console.log(saludo); // Resultado: 'Hola, bienvenido'
```

OPERADORES DE COMPARACIÓN

- **==** Igualdad (Valor)
- **===** Igualdad estricta (Valor y tipo)
- **!=** Desigualdad (Valor)
- **!==** Desigualdad Estricta
- **>** Mayor que
- **<** Menor que
- **>=** Mayor igual
- **<=** Menor igual



```
let a = 5;  
let b = '5';  
  
console.log(a == b); // true (compara solo el valor)  
console.log(a === b); // false (compara valor y tipo)  
  
console.log(a != 10); // true (5 no es igual a 10)  
console.log(a !== '5'); // true (diferente en tipo)  
  
console.log(a > 3); // true  
console.log(a < 3); // false  
console.log(a >= 5); // true  
console.log(a <= 4); // false
```

OPERADORES LOGICOS

- `||` .
- `??` Nullish Coalescing
- `!` NOT (no lógico)
- `&&` **AND** (y lógico)

```
let edad = 20;
let tieneCarnet = true;

// AND: Ambas condiciones deben ser verdaderas
if (edad >= 18 && tieneCarnet) {
  console.log("Puede conducir");
} else {
  console.log("No puede conducir");
}

// OR: Una condición es suficiente
if (edad >= 18 || tieneCarnet) {
  console.log("Al menos cumple una condición");
}

// NOT: Invierte el valor
let permiso = false;
if (!permiso) {
  console.log("No tiene permiso");
}

// Nullish Coalescing: Asigna valor por defecto
let nombre = null;
console.log(nombre ?? "Sin nombre"); // "Sin nombre"
```

OPERADORES BITWISE

- `&` AND
- `|` OR
- `^` XOR
- `~` NOT
- `<<` Desplazamiento a la izquierda
- `>>` Desplazamiento a la derecha

```
let a = 5; // 0101 en binario
let b = 3; // 0011 en binario

console.log(a & b); // 1 (AND)
console.log(a | b); // 7 (OR)
console.log(a ^ b); // 6 (XOR)
console.log(~a); // -6 (NOT)
console.log(a << 1); // 10 (Desplazamiento a la izquierda)
console.log(a >> 1); // 2 (Desplazamiento a la derecha)
console.log(a >>> 1); // 2 (Desplazamiento a la derecha sin signo)
```

OPERADORES ESPECIALES

- **Ternario:** Simplifica condicionales
- **Typeof:** Devuelve el tipo de una variable
- **instanceof:** Verifica si un objeto es una instancia de una clase
- **delete:** Elimina una propiedad de un objeto
- **in :** Verifica si una propiedad existe en un objeto

```
// Operador Ternario
let edad = 20;
let permiso = (edad >= 18) ? 'Puede votar' : 'No puede votar';
console.log(permiso); // "Puede votar"

// typeof - Identificar el tipo de datos
let nombre = "Carlos";
console.log(typeof nombre); // "string"
console.log(typeof edad); // "number"

// instanceof - Verificar instancia de un objeto
let fecha = new Date();
console.log(fecha instanceof Date); // true

// delete - Eliminar propiedad de un objeto
let persona = {
  nombre: 'Ana',
  edad: 30,
  ciudad: 'Madrid'
};
delete persona.ciudad;
console.log(persona); // { nombre: 'Ana', edad: 30 }

// in - Comprobar si una propiedad existe en un objeto
console.log('edad' in persona); // true
console.log('ciudad' in persona); // false
```

Métodos Predefinidos

MÉTODO PREDEFINIDOS

En JavaScript, existen varios **métodos predefinidos** que facilitan **tareas comunes**, como **mostrar mensajes**, **interactuar con el usuario** y **trabajar** con el entorno del **navegador**. A continuación te explico los métodos básicos más utilizados.

ALERT()

Muestra mensajes simples

CONSOLE.LOG()

Muestra mensajes para depuración en la consola

CONFIRM()

Pide confirmación del usuario

PROMPT()

Pide datos al usuario

ALERT()

- Muestra un cuadrado de mensaje emergente con un texto.
- Es útil mostrar información rápida al usuario.

```
// Alert Function
function showAlert() {
  alert("¡Hola, este es un mensaje de alerta!");
}
```

CONSOLE.LOG()

- Escribe mensajes en la consola del navegador (herramienta de desarrolladores).
- Útil para depuración o verificar valores sin molestos cuadros emergentes.

```
>> console.log("mensaje a la consola")
mensaje a la consola
```

CONFIRM()

Muestra un cuadro con dos opciones:

- Aceptar
- cancelar.

Devuelve true si el usuario hace click en aceptar y false si elige cancelar.

```
// Confirm Function
function getConfirm() {
  const userConfirmed = confirm("¿Estas seguro de continuar?");
  if (userConfirmed) {
    alert("Continuando...");
  } else {
    alert("Operacion cancelada");
  }
}
```

PROMPT()

- Muestra un cuadro de entrada para que el usuario escriba un valor.
- Devuelve el valor que el usuario escribió como texto.

```
// Prompt Function
function getPrompt() {
  const userInput = prompt("¿Cual es tu nombre?", "Aleeza");
  if (userInput) {
    alert("Hola, " + userInput + "!");
  } else {
    alert("Hola, ${nombre}");
  }
}
```

DOCUMENT.WRITE()

- Este método escribe directamente en el documento HTML.
- Es útil para contenido dinámico básico, pero no es recomendable usarlo después de que la página haya cargado por completo, ya que puede sobrescribir todo el contenido del documento.

```
//Document Write Function  
function showDocumentwrite(){  
    document.write("<h1>¡Hola, mundo!</h1>");  
}
```

¡Hola, mundo!



Funciones

FUNCIONES

¿QUÉ ES?

Una **función** es un **bloque de código** que realiza una tarea específica y se puede **reutilizar**.

DECLARACIÓN:

Definir **su nombre**, los **parámetros** (si los hay) y el **conjunto** de **instrucciones** que ejecutará.

```
function nombre(parametros) {  
  # Código que realiza la tarea  
}
```

EJECUCIÓN

«Llamar» a la función:
nombre ()

EJEMPLO

```
function saludar() {  
  console.log("¡Hola, mundo!");  
}  
  
// Llamamos a la función  
saludar(); // Salida: ¡Hola, mundo!
```

PARA USAR FUNCIONES
HAY QUE HACER
2 COSAS

En este ejemplo hemos declarado la función y además, hemos ejecutado la función (en la última línea) llamándola por su nombre y seguida de ambos paréntesis, que nos indican que es una función. Se nos mostrará en la consola Javascript el mensaje de saludo.

FUNCIONES NOMINATIVAS

¿QUÉ SON?

Se denominan nominativas porque **tienen un nombre** al definirlas.

Así podremos **usarlas** durante **todo el código**

```
function saludar() {  
  return "Hola";  
}
```

CARACTERÍSTICAS

Tienen un **nombre** explícito

Se pueden **declarar** en **cualquier parte** del programa

Compatibles con el **hoisting**

Son **reutilizables**

DECLARACIÓN

Definir **su nombre**, los **parámetros** (si los hay) y el **conjunto de instrucciones** que ejecutará.

```
function nombre(params) {  
  # Código que realiza la tarea  
}
```

USO EN MÉTODOS DE OBJETOS

Las funciones nominativas también pueden ser **métodos** de un **objeto**

```
const objeto = {  
  saludar: function() {  
    console.log("¡Hola!");  
  }  
};  
  
// Llamar a la función  
objeto.saludar(); // Salida: ¡Hola!
```

En este caso, el objeto tiene un método llamado saludar dentro del propio objeto

USO EN CLASES

Las funciones nominativas también aparecen como **métodos** dentro de las clases.

```
class Persona {  
  saludar() {  
    console.log("¡Hola!");  
  }  
}  
  
// Crear una instancia de la clase  
const persona = new Persona();  
  
// Llamar al método  
persona.saludar(); // Salida: ¡Hola!
```

La clase Persona tiene una función definida (método de clase) llamada saludar.

¿CUÁNDO USARLAS?

Funciones reutilizables:

Cuando necesitas llamar una misma función en varios lugares del código.

Funciones complejas:

Cuando es importante tener un nombre significativo que describa lo que hace la función.

Depuración:

Si esperas posibles errores en la lógica, el nombre ayudará a rastrear el problema.

Recursión:

Cuando una función se llama a sí misma dentro de la definición de la función.

HOISTING EN FUNCIONES NOMINATIVAS

El **hoisting** es una característica que permite que las **funciones** declaradas nominativamente puedan **ser llamadas antes** de su **declaración** en el código

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

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

Aunque la función `multiplicar` está **declarada después de la llamada**, el motor de JavaScript **eleva (hoist)** su definición **al inicio** del contexto, permitiendo que sea **accesible antes** de su aparición.

FUNCIONES ANÓNIMAS

¿QUÉ SON?

Las **funciones anónimas** (o funciones **lambda**) son un tipo de funciones que se declaran **sin definir** un nombre de función

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

CARACTERÍSTICAS

No tienen **nombre** propio

Se **asignan** a **variables** o **constantes**

Muy útiles en **callbacks** como **argumentos** de otras **funciones**

Compatibles con **funciones flecha**

¿PARA QUÉ SIRVEN?

Cuando necesites crear una función y **ejecutarla inmediatamente** y **desecharla**

Pueden ser usadas en **cualquier lugar** donde se necesite una función.

Usar una función anónima en **un evento** (onclick, map...)

CASOS DE USO

ASIGNACIÓN A UNA VARIABLE

Permiten **guardar** una **función** en una **variable** para ser utilizada más tarde.

```
const saludo = function(nombre) {  
  return `Hola, ${nombre}!`;  
};  
console.log(saludo("Mundo")); // Salida: Hola, Mundo!
```

FUNCIÓN ANÓNIMA EN UN EVENTO

Pasamos una **función sin nombre** como argumento a un **manejador de eventos**. Esta función define qué sucede cuando el evento ocurre, y es útil si el comportamiento no necesita reutilizarse

```
document.querySelector('button').addEventListener('click', function () {  
  alert('¡Botón clickeado!');  
});
```

USO COMO CALLBACK

Son muy comunes en métodos como `setTimeout`, `setInterval`, y en funciones de orden superior como `forEach`.

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

FUNCIÓN AUTOEJECUTABLE IIFE

(Immediately Invoked Function Expression)

Las funciones anónimas también pueden **ejecutarse inmediatamente** después de ser declaradas.

```
(function() {  
  console.log("Esta función se ejecuta inmediatamente");  
})();
```

PARÁMETROS AUTOEJECUTABLE

Podemos **utilizar parámetros** en dichas funciones autoejecutables. Se pasan **al final** de la función entre **paréntesis**.

```
(function (name) {  
  console.log(`¡Hola, ${name}!`);  
})("Goku");
```

ALMACENAJE DE RETURN

Si la función devuelve algún valor con **return**, lo que se **almacena** en la **variable** es el **valor** que **devuelve** la función autoejecutada:

```
const value = (function (name) {  
  return `¡Hola, ${name}!`;  
})("Goku");
```

CLOSURES

Las **funciones anónimas** suelen **usarse en closures** para encapsular valores.

En Javascript, una clausura o cierre se define como **una función que «encierra» variables** en su propio ámbito (y que **continúan existiendo** aún habiendo terminado de ejecutar la función).

CUANDO SE ELIMINA LA REFERENCIA AL CLOSURE

```
function contador() {  
  let cuenta = 0; // Variable cerrada  
  return function() {  
    cuenta++;  
    return cuenta;  
  };  
}  
  
let incrementar = contador(); // Crear un closure  
console.log(incrementar()); // 1  
console.log(incrementar()); // 2  
  
incrementar = null; // Eliminar la referencia
```

En este caso, la función anónima devuelta por contador **recuerda** el valor de la **variable** *cuenta*, incluso **después** de que la función contador haya terminado de **ejecutarse**.

Si no hay ninguna referencia activa a una función **closure** o **al objeto** que **la contiene**, el **garbage collector** (gestor de memoria de JavaScript) liberará la memoria asociada.

DIFERENCIA CON FUNCIONES NOMINATIVAS:

Característica	Función Anónima	Función Nominativa
Nombre	No tiene	Tiene un identificador único
Asignación a variables	Se asigna directamente	Opcional (puede usarse sin asignación)
Reutilización	Necesita asignarse a una variable	Se puede llamar directamente por su nombre
Depuración	Más difícil debido a la falta de nombre	Más fácil porque el nombre aparece en el stack trace

FUNCIONES FLECHA

La función flecha tiene una **sintaxis más corta** en comparación con las de función clásicas:

No tiene su propio *this*, *arguments*, *super* o *new.target*.

Siempre son **anónimas**.

```
() => expression;
```

```
const saludar = () => '¡Hola!';  
console.log(saludar()); // ¡Hola!
```

FUNCIONES MÁS CORTAS

```
(param1, paramN) => expression;
```

```
const saludarConNombre = (nombre) => `¡Hola, ${nombre}!`;   
console.log(saludarConNombre('Data')); // ¡Hola, Data!
```

Parámetros y argumentos

PARÁMETROS EN FUNCIONES

¿QUÉ SON?

- Los **parámetros** son variables definidas en la declaración de la función.
- **Actúan como marcadores de posición** para los valores que la función recibirá cuando sea invocada.
- Los **nombres** de los parámetros solo tienen **significado dentro** de la **función**, y sirven para operar con los datos que se pasen a la función.

¿QUÉ SIGNIFICA?

ACTUAR COMO MARCADOR DE POSICIÓN

Significa que **no contienen un valor específico** en el momento en que defines la función, sino que simplemente **indican dónde se colocarán** los valores una vez que la función sea llamada.

Si intentas **acceder** a ese parámetro (variable) por **fuera** de la función **dará un error!**

PARÁMETROS ASIGNADOS

Conocidos como **parámetros por defecto**, son aquellos que se les asigna un **valor predeterminado** directamente **en la definición** de una función.

Esto significa que si al llamar a la función no se proporciona un valor para ese parámetro, se utilizará el valor por defecto.

```
function saludar(nombre = 'Miguel Angel') {  
  console.log('Hola ' + nombre);  
}
```

Esta función recibe un parámetro llamado "**nombre**", con un valor **predeterminado**. Este valor se asignará **en caso** que al invocar a la función **no le pasemos nada**.

`saludar();`

Eso **produciría** la salida por consola "**Hola Miguel Angel**".

Pero, aunque indiquemos un valor predeterminado, podemos seguir **invocando a la función** enviando un **valor diferente** como parámetro.

`saludar('DesarrolloWeb.com');`

La **regla** más importante es la del **orden de los parámetros** en las llamadas a las funciones, que debe realizarse **tal como está definido** en la cabecera de la función. Esto es obvio, pero afecta directamente a los parámetros con valores por defecto en las funciones.



PARÁMETROS REST

Nos permite representar un número **indefinido** de **argumentos** como un **array**.

```
function nombreFuncion(...rest) {  
    // 'rest' es un array que contiene  
    // el resto de los argumentos  
}
```


PUNTOS CLAVE:

Flexibilidad: Permite crear funciones más versátiles.

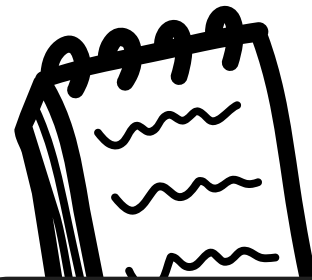
Único: Solo puede haber un parámetro rest por función.

Posición: Debe ser el último parámetro.

Array: Los argumentos adicionales se almacenan en un array.



PARÁMETROS ...REST



EJEMPLO

```
/* Parámetros rest */  
function listarFrutas(...frutas) {  
    console.log("Frutas recibidas:");  
    frutas.forEach((fruta) => console.log(fruta));  
}
```

Este código utiliza el **operador rest (...)** para permitir que la función ***listarFrutas*** reciba un **número variable de argumentos** y los procese **como un array**.

RETORNO DE VALORES

Cuando una **instrucción de retorno se llama** en una función, se **detiene** la ejecución de esta.

Si se **especifica un valor** dado, este se **devuelve** a quien llama a la función.

Si se omite la expresión, se devuelve ***undefined*** en su lugar.

```
return;  
return true;  
return false;  
return x;  
return x + y / 3;
```

Todas las siguientes sentencias de retorno rompen la ejecución de la función:

return valor

return debe usarse como **última instrucción** y entre la palabra y el valor a devolver no puede haber punto y coma.

La instrucción de retorno **return** se ve **afectada** por la **inserción automática de punto y coma** (ASI). No se permite el terminador de línea entre la palabra clave de retorno y la expresión.

```
return;  
a + b;
```

se transforma por ASI en:
`return; a + b`



¡GRACIAS!