CODERHOUSE



Unidad 3: Funciones y parámetros

# **Introducción a las Funciones**

**¿Qué es una función?**

**Introducción**

Las **funciones** son fundamentales en cualquier lenguaje de programación, y JavaScript no es la excepción. Permiten agrupar código en bloques que realizan tareas específicas, lo cual facilita la organización, el mantenimiento y la reutilización del código.

**Estructura Básica de una Función**

En JavaScript, una función se define con la palabra clave function, seguida de un nombre, paréntesis () que pueden contener parámetros, y finalmente un bloque de código encerrado entre llaves {}. Aquí tienes un ejemplo sencillo:

| function saludar() {  console.log("¡Hola, mundo!");  } |
| --- |

**Invocación de Funciones**

Para ejecutar la función saludar que definimos, la "llamamos" o "invocamos" usando su nombre seguido de paréntesis:

| saludar(); // Muestra: ¡Hola, mundo! |
| --- |

### 

### **Parámetros y Argumentos**

Las funciones pueden recibir datos externos mediante parámetros. Los parámetros actúan como variables locales dentro de la función. Aquí un ejemplo con parámetros:

| function sumar(a, b)//parametros  {  return a + b;  }  let resultado = sumar(5, 3); //argumentos  console.log(resultado); // Muestra: 8 |
| --- |

**Beneficios de Usar Funciones**

1. **Reutilización de Código:** Una vez definida, una función puede ser invocada múltiples veces desde distintas partes del programa.
2. **Modularidad:** Divide el programa en pequeños bloques, facilitando la lectura y mantenimiento.
3. **Abstracción:** Permite a los desarrolladores pensar en problemas a un nivel más alto sin preocuparse por los detalles de implementación.

**Buenas Prácticas**

* **Nombres descriptivos:** Elige nombres que describan claramente lo que la función realiza.
* **Funciones cortas:** Idealmente, una función debe realizar una sola tarea. Esto las hace más fáciles de testear y mantener.
* **Evitar efectos secundarios:** Una función idealmente no debería modificar variables externas o el estado del programa.

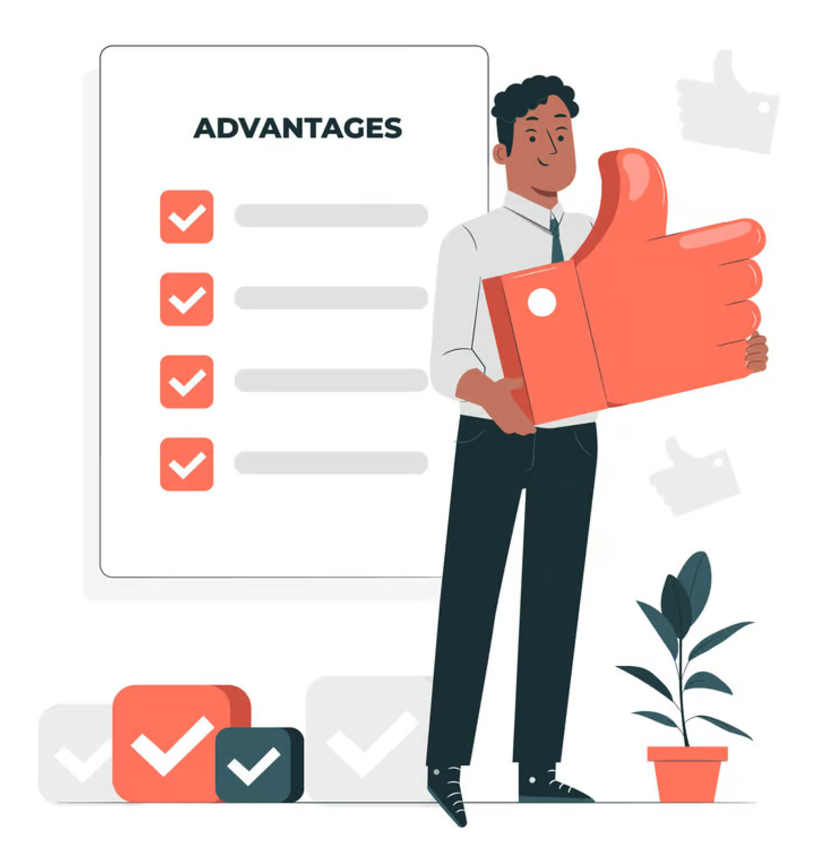
**Conclusión**

Las funciones son herramientas poderosas que te ayudan a escribir código más limpio, legible y reutilizable. A medida que avances, encontrarás patrones y técnicas más avanzadas que te permitirán aprovechar todo el potencial de las funciones en JavaScript.

## 

## 

## **Ventajas de las funciones**



### **Reducción de Código Repetido**

Una de las principales ventajas de las funciones en JavaScript es la reducción de código repetido. Siguiendo el principio **DRY** ("Don't Repeat Yourself"), las funciones permiten escribir una pieza de código una sola vez y reutilizarla en diferentes partes del programa. Esto no solo ahorra tiempo en la escritura de código, sino que también facilita su mantenimiento y actualización.

* **Ejemplo práctico:**

| // Sin usar funciones (código repetitivo)  console.log("Hola, Ana");  console.log("Hola, Juan");  console.log("Hola, María");  // Usando funciones  function saludar(nombre) {  console.log(`Hola, ${nombre}`);  }  saludar("Ana");  saludar("Juan");  saludar("María"); |
| --- |

### **Simplicidad y Claridad**

Aplicando el principio **KISS** ("Keep It Simple, Stupid"), las funciones ayudan a mantener la simplicidad del código. Al encapsular operaciones complejas dentro de funciones, el código principal se vuelve más sencillo y claro. En lugar de repetir las mismas líneas de código, una llamada a una función puede reemplazar múltiples instancias de código repetitivo, haciendo que el programa sea más legible y fácil de entender.

* **Ejemplo práctico:**

| // Sin encapsular operaciones complejas  let base = 10;  let altura = 5;  let areaTriangulo = (base \* altura) / 2;  console.log(`El área del triángulo es ${areaTriangulo}`);  // Encapsulando la lógica en una función  function calcularAreaTriangulo(base, altura) {  return (base \* altura) / 2;  }  console.log(`El área del triángulo es ${calcularAreaTriangulo(10, 5)}`); |
| --- |

### **Reutilización de Código**

Las funciones también promueven la reutilización de código. Una vez que una función está escrita, puede ser utilizada en múltiples lugares del programa, o incluso en diferentes proyectos, sin necesidad de reescribir la función. Esto es particularmente útil en proyectos grandes donde las mismas tareas se realizan repetidamente.

* **Ejemplo práctico:**

| // Función para calcular el precio final con descuento  function calcularPrecioFinal(precio, descuento) {  return precio - (precio \* descuento / 100);  }  // Reutilizando la función en diferentes contextos  let precioCamisa = calcularPrecioFinal(50, 10);  let precioPantalon = calcularPrecioFinal(80, 15);  console.log(`Precio final de la camisa: ${precioCamisa}`);  console.log(`Precio final del pantalón: ${precioPantalon}`); |
| --- |

### **Modularidad**

La modularidad es otra ventaja significativa de usar funciones. Al dividir el código en módulos más pequeños y manejables, es más fácil testear, depurar y colaborar en proyectos. Cada función puede ser diseñada para realizar una tarea específica, lo que simplifica la detección de errores y la optimización del código.

* **Ejemplo práctico:**

| // Función para calcular el descuento  function calcularDescuento(precio, porcentajeDescuento) {  return precio - (precio \* porcentajeDescuento / 100);  }  // Función para calcular el precio final incluyendo impuestos  function calcularPrecioFinal(precio, descuento, impuesto) {  let precioConDescuento = calcularDescuento(precio, descuento);  return precioConDescuento + (precioConDescuento \* impuesto / 100);  }  // Invocación  let total = calcularPrecioFinal(100, 10, 21);  console.log(total); // Muestra: 98.9 |
| --- |

Esta función calcularPrecioFinal ilustra cómo un bloque de código simple puede ser reutilizado para calcular el precio final donde sea necesario en el programa, manteniendo el código organizado y evitando la repetición.

**Conclusión**

Utilizar funciones es fundamental para escribir código limpio, organizado y eficiente. Permite a los desarrolladores construir aplicaciones más robustas y mantenibles, facilitando la gestión de proyectos más complejos y su escalabilidad.

# **Creación y Uso de Funciones**

## **Declarar una función**

Las funciones te ayudan a organizar tu código para que sea más claro, reutilizable y fácil de mantener. Son herramientas personalizadas que puedes crear para realizar tareas específicas dentro de tu proyecto. Desde algo tan simple como mostrar un mensaje hasta cálculos complejos o manejo de datos, las funciones te ayudan a llevar tu código al siguiente nivel.

### **Paso 1: Definir la función**

Para definir una función, empieza con la palabra clave function, seguida por el nombre de la función, paréntesis (), y un bloque de código encerrado entre llaves {}.

| function mostrarLibros() {  console.log("Bienvenido a la biblioteca. ¡Estos son los libros disponibles!");  } |
| --- |

### **Paso 2: Llamar a la función**

Una vez definida la función, puedes ejecutarla llamándola por su nombre seguido de paréntesis.

| mostrarLibros(); // Muestra: Bienvenido a la biblioteca. ¡Estos son los libros disponibles! |
| --- |

### **Paso 3: Añadir parámetros a la función**

Las funciones pueden recibir datos a través de parámetros, que actúan como variables dentro de la función. Los parámetros se definen dentro de los paréntesis en el momento de declarar la función.

| function buscarLibro(titulo) {  console.log(`Buscando el libro: ${titulo}...`);  }  buscarLibro("El Principito"); // Muestra: Buscando el libro: El Principito... |
| --- |

### **Paso 4: Utilizar el retorno de la función**

Para que una función envíe un valor de vuelta al punto donde fue llamada, utiliza la palabra clave return. El valor retornado puede ser guardado en una variable, usado en una expresión o simplemente mostrado.

| function calcularMulta(diasRetraso) {  const multaPorDia = 0.50; // 50 centavos por día de retraso  return diasRetraso \* multaPorDia;  }  let multa = calcularMulta(5);  console.log(`La multa total es: $${multa}`); // Muestra: La multa total es: $2.5 |
| --- |

### **Consejos adicionales**

* **Nombres claros:** Los nombres de las funciones deben describir su propósito de manera concisa y específica. Por ejemplo, agregarLibro() es mucho más claro que func1(), ya que comunica exactamente lo que la función hace. Esto mejora la legibilidad y el mantenimiento del código, especialmente en proyectos colaborativos.
* **Funciones pequeñas y especializadas:** Cada función debe tener una responsabilidad única. Esto hace que tu código sea más fácil de leer y mantener.
* **Evita efectos secundarios:** Idealmente, una función no debería modificar variables globales o alterar el estado del programa de maneras inesperadas.

### **Ejemplo Práctico**

| function agregarLibro(titulo, autor) {  return { titulo, autor };  }  function mostrarDetalleLibro(libro) {  console.log(`Título: ${libro.titulo}, Autor: ${libro.autor}`);  }  // Agregar un libro  let libro1 = agregarLibro("1984", "George Orwell");  // Mostrar detalles del libro  mostrarDetalleLibro(libro1);  // Muestra: Título: 1984, Autor: George Orwell |
| --- |

Esta guía te ayudará a entender cómo implementar y utilizar funciones en JavaScript, facilitando la estructura y modularidad de tu código.

## **Parámetros y argumentos**

### **¿Qué son los Parámetros?**

Un parámetro es una variable en la declaración de la función. Actúa como un placeholder que se rellena con un valor o una referencia cuando la función es llamada (o invocada). Los parámetros permiten que las funciones sean más flexibles y adaptativas a diferentes situaciones.

### **Declaración de una Función con Parámetros**

Para declarar una función con parámetros en JavaScript, simplemente añade nombres de variables entre los paréntesis de la función. Estos nombres serán usados dentro de la función para referirse a los valores pasados durante la llamada de la función.

| function agregarLibro(titulo, autor) {  console.log(`Libro agregado: "${titulo}" por ${autor}.`);  } |
| --- |

En este ejemplo, titulo y autor son parámetros de la función agregarLibro.

### **Llamada de la Función con Argumentos**

Los argumentos son los valores específicos que se pasan a los parámetros de una función cuando esta es invocada. Estos valores determinan cómo se ejecutará la función. Usando el ejemplo de la biblioteca, aquí tienes cómo llamar a la función calcularMulta con argumentos:

| function calcularMulta(diasRetraso) {  const multaPorDia = 0.50; // 50 centavos por día de retraso  return diasRetraso \* multaPorDia;  }  // Llamada a la función con un argumento  let resultado = calcularMulta(7);  console.log(`La multa por 7 días de retraso es: $${resultado}`);  // Muestra: La multa por 7 días de retraso es: $3.5 |
| --- |

En este ejemplo, el valor 7 es el argumento que se pasa al parámetro diasRetraso, y la función utiliza este valor para calcular el resultado.

### **Función con Múltiples Parámetros**

Las funciones pueden tener varios parámetros. Cada parámetro en la declaración de la función corresponde a un argumento específico en la misma secuencia cuando la función es llamada.

| function prestarLibro(titulo, usuario) {  console.log(`El libro "${titulo}" ha sido prestado a ${usuario}.`);  }  prestarLibro("1984", "Ana");  // Muestra: El libro "1984" ha sido prestado a Ana. |
| --- |

### **Parámetros y Escalabilidad del Código**

Utilizar parámetros hace que tu código sea más escalable y fácil de mantener. Por ejemplo, si necesitas realizar una operación similar en diferentes partes de tu aplicación, puedes escribir una función con parámetros y reutilizarla en lugar de repetir el mismo código.

### **Uso Avanzado: Parámetros Predeterminados**

JavaScript permite definir valores predeterminados para los parámetros, que se usarán en caso de que no se pase un argumento:

| function devolverLibro(titulo, diasRetraso = 0) {  const multa = diasRetraso \* 0.50;  const mensaje = diasRetraso > 0  ? `Devuelto con ${diasRetraso} días de retraso. Multa: $${multa}`  : "Devuelto a tiempo. No hay multa.";  console.log(`Libro "${titulo}": ${mensaje}`);  }  devolverLibro("El Principito");  // Muestra: Libro "El Principito": Devuelto a tiempo. No hay multa.  devolverLibro("El Principito", 3);  // Muestra: Libro "El Principito": Devuelto con 3 días de retraso. Multa: $1.5. |
| --- |

### **Conclusión**

Los parámetros son herramientas esenciales en JavaScript para crear funciones flexibles y reutilizables. Facilitan la adaptación del comportamiento de la función a diferentes datos de entrada, aumentando así la generalidad y la potencia de tu código. A medida que avances, descubrirás patrones más complejos y poderosos que involucran parámetros para optimizar aún más tus aplicaciones.

# **Scope**

## **Introducción al Scope**

En JavaScript, el **scope** o ámbito de una variable define la parte del código donde dicha variable está disponible para ser utilizada. Comprender el scope es esencial para gestionar la asignación y visibilidad de las variables, evitando errores y comportamientos inesperados en el programa.

### **Scope Global**

Cuando una variable se declara fuera de cualquier función o bloque, tiene un **scope global**, lo que significa que es accesible desde cualquier parte del código después de su declaración.

**Ejemplo:**

| let color = 'azul'; // Variable global  function mostrarColor() {  console.log(color); // Accede a la variable global  }  mostrarColor(); // Imprime: azul |
| --- |

**Ventajas:**

* Ofrecen accesibilidad universal en el código.

**Desventajas:**

* Pueden causar conflictos en proyectos grandes al ser modificadas desde cualquier parte.

### **Scope Local**

El scope local restringe el acceso a una variable al bloque o función donde se declara. Existen dos tipos principales:

**Scope de Función:** Variables declaradas dentro de una función no son accesibles fuera de ella.  
  
 **Ejemplo:**

| function establecerColor() {  let color = 'rojo'; // Variable local a la función  }  // console.log(color); // Error: color no está definido |
| --- |

**Scope de Bloque:** Introducido con let y const en ES6, permite limitar el scope a un bloque específico, como en ciclos o condicionales.

**Ejemplo:**

| if (true) {  let color = 'verde'; // Variable de scope de bloque  console.log(color); // Imprime: verde  }  // console.log(color); // Error: color no está definido |
| --- |

### **Comparativa entre Variables Globales y Locales**

| **Aspecto** | **Globales** | **Locales** |
| --- | --- | --- |
| **Definición** | Declaradas fuera de funciones o bloques. | Declaradas dentro de funciones o bloques. |
| **Accesibilidad** | Disponibles en todo el programa. | Disponibles solo dentro de su ámbito. |
| **Impacto en el código** | Mayor riesgo de conflictos y contaminación. | Más seguras y fáciles de rastrear. |
| **Gestión de memoria** | Permanecen mientras el programa se ejecuta. | Se eliminan al finalizar su bloque o función. |

### **Importancia del Scope en JavaScript**

1. **Seguridad:** Minimiza errores al restringir la modificación no intencionada de variables.
2. **Mantenibilidad:** Facilita el seguimiento de variables, especialmente en proyectos grandes.
3. **Eficiencia:** Las variables locales se destruyen al finalizar su bloque, liberando memoria.

**Conclusión**

Entender y aplicar correctamente el scope en JavaScript es crucial para escribir código robusto y fácil de mantener. Mientras que las variables globales ofrecen accesibilidad universal, las variables locales proporcionan mayor protección y claridad. Adoptar buenas prácticas, como limitar el uso de variables globales y aprovechar los scopes de función y bloque, conduce a aplicaciones más limpias, seguras y eficientes.

# **Funciones Anónimas y Funciones Flecha**

## **¿Qué son las Funciones Anónimas?**

Las funciones anónimas son funciones sin nombre que pueden ser asignadas a variables, pasadas como argumentos o ejecutadas inmediatamente. Son útiles para encapsular lógica que no necesita reutilizarse o referenciarse directamente.

**Ejemplo:**

| const suma = function(a, b) {  return a + b;  };  console.log(suma(5, 3)); // Salida: 8 |
| --- |

**Usos comunes:**

* **Callbacks:** Manejo de eventos o lógica asincrónica.

| document.getElementById("miBoton").addEventListener("click", function() {  alert("¡Botón pulsado!");  }); |
| --- |

* **IIFE (Immediately Invoked Function Expressions):** Ejecución inmediata para crear un alcance privado.

| (function() {  let mensaje = "Ejecutada al instante";  console.log(mensaje);  })(); |
| --- |

* **Métodos de arrays:**

| let dobles = [1, 2, 3].map(function(num) {  return num \* 2;  }); |
| --- |

### **¿Qué son las Funciones Flecha?**

Introducidas en ES6, las funciones flecha ofrecen una sintaxis más concisa para escribir funciones anónimas. No tienen su propio this, lo que las hace ideales para callbacks y programación funcional.

**Ejemplo:**

| const suma = (a, b) => a + b;  console.log(suma(5, 3)); // Salida: 8 |
| --- |

**Usos comunes:**

* **Callbacks**

| let cuadrados = [1, 2, 3].map(n => n \* n); |
| --- |

* **Programación funcional:** Por su brevedad y claridad.

### **Comparación: Funciones Anónimas vs. Funciones Flecha**

| **Característica** | **Funciones Anónimas** | **Funciones Flecha** |
| --- | --- | --- |
| **Sintaxis** | Más extensa | Más corta y legible |
| **Contexto this** | Propio contexto | Heredado del entorno exterior |
| **Uso de arguments** | Disponible | No disponible, usa parámetros explícitos |
| **Uso como métodos** | Adecuadas | No recomendadas |
| **Uso como constructores** | Posible | No permitido |

### **Ventajas y Desventajas**

**Funciones Anónimas**

**Ventajas:**

* Control completo sobre el contexto de this.
* Acceso al objeto arguments.

**Desventajas:**

* Sintaxis más extensa.
* Menos intuitivas para manejar this en callbacks.

**Funciones Flecha**

**Ventajas:**

* Sintaxis concisa y moderna.
* Heredan el contexto de this.

**Desventajas:**

* No adecuadas para métodos que requieren su propio this.
* No pueden ser usadas como constructores ni acceder a arguments.

**Recomendaciones**

* **Funciones Anónimas:** Útiles cuando necesitas un control detallado de this o acceso a arguments.
* **Funciones Flecha:** Ideales para callbacks y métodos funcionales en arrays o programación moderna.