Funciones 

En JavaScript, las funciones son bloques de código reutilizables que pueden realizar tareas específicas y ser invocadas (llamadas) en cualquier parte del programa. Las funciones son fundamentales para dividir el código en piezas más pequeñas y más manejables, lo que facilita su mantenimiento y reutilización.

Una función en JavaScript puede recibir cero o más parámetros (valores que se pasan a la función para su procesamiento) y puede devolver un valor como resultado. Una función se define utilizando la palabra clave function, seguida por un nombre que identifica la función y una lista de parámetros entre paréntesis. El código que compone la función se encuentra entre llaves {}.

La sintaxis básica para definir una función en JavaScript:

```
function nombreFuncion(parametro1, parametro2, ...) {
  Código de la función
  Puede realizar acciones y cálculos
  Puede utilizar los parámetros recibidos
  Puede devolver un valor con la palabra clave "return"
}
```

Otro ejemplo sumando 2 números:

```
function sumar(num1, num2) {
  let resultado = num1 + num2;
  return resultado;
}
Llamamos a la función y almacenamos el resultado en una variable
let resultadoSuma = sumar(5, 3);

console.log(resultadoSuma); // Imprimirá 8
```

Importante:

con la introducción de ECMAScript 6 (también conocido como ES6 o ECMAScript 2015), se agregaron nuevas formas de definir funciones en JavaScript, además de otras características avanzadas que facilitan aún más la escritura de código.

Por lo tanto, las nuevas formas de definir una función son las siguientes:

Funciones de flecha (Arrow functions):
Las funciones de flecha son una forma más concisa de definir funciones en ES6. Tienen una sintaxis más corta y clara.

Sintaxis:

```
const nombreFuncion = (parametro1, parametro2) => {
  Código de la función
};
```

Otro ejemplo sumando 2 números:

```
const sumar = (num1, num2) => {
  return num1 + num2;
};
```

Funciones de flecha con retorno implícito:
Si la función de flecha solo tiene una expresión, puedes omitir las llaves y la palabra clave return, y la función devolverá automáticamente el resultado de la expresión.

Sintaxis:

```
const nombreFuncion = (parametro1, parametro2) => expresion;
```

Otro ejemplo:

```
const sumar = (num1, num2) => num1 + num2;
```

Funciones de un solo parámetro:
Si una función de flecha tiene un solo parámetro, puedes omitir los paréntesis alrededor del parámetro.

Sintaxis:

```
const nombreFuncion = parametro => {
  Código de la función
};
```

Otro ejemplo:

```
const cuadrado = numero => numero * numero;
```

Funciones con parámetros predeterminados:
ES6 permite definir valores predeterminados para los parámetros de una función, lo que significa que si no se proporciona un valor al llamar la función, tomará el valor predeterminado.

Ejemplo:

```
const nombreFuncion = (parametro1 = valorPredeterminado1) => {
  Código de la función
};
```

Otro ejemplo:

```
const saludar = (nombre = "Invitado") => {
  console.log(`Hola, ${nombre}!`);
};
```

¿Qué es return?

En JavaScript, return es una palabra clave que se utiliza dentro de una función para especificar qué valor debe ser devuelto cuando se llama a esa función. Cuando una función alcanza una instrucción return, se detiene su ejecución y el valor especificado en return se convierte en el resultado de la función.

La sintaxis básica de return es la siguiente:

```
function nombreFuncion(parametro1, parametro2, ...) {
  Código de la función
  return valor; Valor que se devuelve al llamar a la función
}
```

Aquí, valor puede ser cualquier expresión o variable que desees devolver como resultado de la función. Puede ser un número, una cadena de texto, un objeto, un arreglo o incluso otro tipo de dato.

Además, si una función no contiene una instrucción return, se considera que devuelve undefined de manera implícita.

Buenas prácticas para nombrar funciones

Nombrar funciones de manera adecuada es esencial para tener un código limpio, legible y fácil de mantener.

A continuación veremos algunas buenas prácticas:

Usar nombres descriptivos:
Elige nombres que sean descriptivos y reflejen claramente lo que hace la función. Un nombre bien elegido debería indicar la acción o el propósito de la función sin necesidad de mirar el código interno.

Ejemplo:

```
 Mal nombre de función
function fn() {
  Código de la función
}

Buen nombre de función
function calcularPromedio(listaNumeros) {
Código de la función
}
```

Usar verbos para funciones que realizan acciones:
Prefiere usar verbos en el nombre de las funciones que realizan acciones o modifican datos. Esto ayuda a entender rápidamente qué hará la función.

Ejemplo:

```
 Mal nombre de función
function datosUsuario() {
  Código de la función
}

Buen nombre de función
No olvidar el uso de ingles
function obtenerNombreUsuario() {
Código de la función
}
```

Usar nombres en camelCase:
Es una convención en JavaScript utilizar el estilo camelCase para nombrar funciones. Esto significa que la primera palabra comienza con minúscula, y las palabras subsiguientes comienzan con mayúscula y se escriben juntas, sin espacios ni guiones.

Ejemplo:

```
 Mal nombre de función
function NombreUsuario() {
  Código de la función
}

Buen nombre de función
No olvidar el uso de ingles
function obtenerNombreUsuario() {
Código de la función
}
```

Evitar abreviaciones confusas:
A menos que una abreviatura sea ampliamente reconocida y entendida, evita utilizarlas en los nombres de funciones. Los nombres descriptivos son más claros y ayudan a entender el propósito de la función.

Ejemplo:

```
 Mal nombre de función
function calcSuma() {
  Código de la función
}

Buen nombre de función
No olvidar el uso de ingles
function calcularSuma() {
Código de la función
}
```

¿Qué son las funciones anónimas?

Las funciones anónimas son funciones en JavaScript que no tienen un nombre identificador. En lugar de tener un nombre, se definen directamente como expresiones dentro de una instrucción o asignadas a variables. Estas funciones son muy útiles cuando necesitas crear una función temporal o cuando deseas pasar una función como argumento a otra función.

La sintaxis de una función anónima es la siguiente:

```
Función anónima definida como expresión
es posible asignar una función anónima a una variable,
lo que se conoce como Expresión de Función
const miFuncionAnonima = function(parametro1, parametro2) {
  Código de la función
}

Función anónima asignada a una variable
const otraFuncionAnonima = (parametro1, parametro2) => {
Código de la función
}
```

Otro tipo de función anónima son las funciones IIFE

las IIFE O funciones autoejecutables se utilizan en patrones de diseño como la Función Autoejecutable (Immediately Invoked Function Expression - IIFE), donde la función se declara y se ejecuta inmediatamente.

La sintaxis de una función IIFE es la siguiente:

```
IIFE (Función Autoejecutable)
(function() {
  console.log("Esta es una IIFE");
})();
```

Funciones como argumentos

Funciones como argumentos significa que una función se puede tratar como cualquier otro valor, incluidos los argumentos que se pueden pasar a otras funciones. Esto permite que las funciones se utilicen como argumentos de otras funciones, lo que es una característica poderosa que se conoce como "funciones como argumentos" o "funciones de orden superior" (higher-order functions).

Pasar una función como argumento a otra función es útil cuando deseas que la función receptora realice alguna operación específica en función del comportamiento proporcionado por la función argumento. Esto es especialmente útil cuando necesitas lograr un código más flexible y reutilizable.

Ejemplo:

```
function ejecutarOperacion(operacion, num1, num2) {
  return operacion(num1, num2);
}

function suma(a, b) {
  return a + b;
}

function resta(a, b) {
  return a - b;
}
Imprime 8, porque 5 + 3 = 8
console.log(ejecutarOperacion(suma, 5, 3));
Imprime 2, porque 5 - 3 = 2
console.log(ejecutarOperacion(resta, 5, 3));
```

Que es el hoisting

El hoisting es un comportamiento en JavaScript donde las declaraciones de variables y funciones se mueven (se "elevan") al principio del ámbito en el que están definidas durante la fase de compilación, antes de que se ejecute el código. Esto significa que las variables y funciones pueden ser utilizadas antes de ser declaradas en el código, lo cual puede ser sorprendente para algunos desarrolladores.

Algunos ejemplos para entender cómo funciona el hoisting:

```
console.log(miVariable);Imprimirá "undefined"
var miVariable = 10;
console.log(miVariable);Imprimirá 10
```

En este ejemplo, aunque se utiliza console.log para imprimir el valor de miVariable antes de su declaración, no se produce un error. Esto se debe al hoisting. Durante la fase de compilación, la declaración de miVariable se mueve al principio del ámbito, lo que significa que la variable existe pero aún no se le ha asignado un valor (por lo que su valor es undefined). Luego, en la fase de ejecución, se asigna el valor 10 a miVariable, y el segundo console.log imprime el valor actualizado.

Ejemplo 2: Hoisting de funciones

```
saludar();Imprimirá "¡Hola!"

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

En este ejemplo, la función saludar se llama antes de su declaración. Esto es posible debido al hoisting de funciones. Durante la fase de compilación, la declaración de la función se mueve al principio del ámbito, lo que permite que la función se llame antes de ser definida en el código.

Sin embargo, es esencial tener en cuenta que solo las declaraciones de funciones se elevan, no las funciones definidas utilizando expresiones de función (como las funciones flecha). Por lo tanto, el hoisting de funciones solo se aplica a funciones declaradas con la palabra clave function.

Ahora que ya sabemos lo esencial sobre funciones vamos a ver algunos ejercicios