JavaScript
Funciones

CertiDevs

Índice de contenidos

Las **funciones** en **JavaScript** son bloques de código que se definen y luego se pueden invocar en diferentes partes del programa.

Las funciones permiten reutilizar código, modularizar la lógica y facilitar el mantenimiento del programa.

1. Declaración de Funciones

Hay varias formas de declarar funciones en JavaScript.

1.1. Funciones Declaradas (Function Declaration)

Las funciones declaradas se definen con la palabra clave function, seguida del nombre de la función, una lista de parámetros entre paréntesis y un bloque de código entre llaves.

```
function suma(a, b) {
    return a + b;
}
const resultado = suma(3, 5);
console.log(resultado); // 8
```

1.2. Funciones Expresadas (Function Expression)

Las funciones expresadas se definen asignando una función anónima a una variable.

Puedes utilizar la palabra clave function seguida de paréntesis y un bloque de código entre llaves.

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

const resultado = suma(3, 5);
console.log(resultado); // 8
```

1.3. Funciones Flecha (Arrow Functions)

Las **funciones flecha**, introducidas en **ES6**, permiten una sintaxis más concisa para la declaración de funciones.

Utilizan el operador flecha (`=>`) para separar los parámetros de la función del cuerpo de la función.

```
const suma = (a, b) => {
    return a + b;
};
const resultado = suma(3, 5);
```

```
console.log(resultado); // 8
```

Si la función flecha solo tiene un parámetro, puedes omitir los paréntesis alrededor del parámetro.

```
const cuadrado = x => {
    return x * x;
};
const resultado = cuadrado(4);
console.log(resultado); // 16
```

Si el cuerpo de la función flecha solo contiene una expresión de retorno, puedes omitir las llaves y la palabra clave return.

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

const resultado = suma(3, 5);
console.log(resultado); // 8
```

2. Parámetros y Argumentos

- Los **parámetros** son los nombres que se enumeran en la definición de la función.
- Los **argumentos** son los valores que se pasan a la función cuando se invoca.

2.1. Parámetros por Defecto

- Puedes asignar **valores por defecto** a los parámetros de una función. Si no se pasa un argumento para un parámetro con un valor por defecto, se utilizará el valor por defecto.

```
function saludo(nombre = 'Desconocido') {
    return `Hola, ${nombre}!`;
}
console.log(saludo('Ana')); // 'Hola, Ana!'
console.log(saludo()); // 'Hola, Desconocido!'
```

2.2. Rest Parameters

Utiliza el operador de propagación (···) seguido de un nombre de parámetro para recopilar todos los argumentos restantes en un array.

```
function numeros(...args) {
    console.log(args);
}
```

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

3. Retorno de Valores

Utiliza la palabra clave `return` para devolver un valor desde una función.

Si no se incluye una declaración `return` en la función, se devuelve `undefined` por defecto.

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

const resultado = suma(3, 5);
console.log(resultado); // 8

function sinRetorno() {
    console.log('Esta función no devuelve un valor.');
}
const resultadoSinRetorno = sinRetorno(); // 'Esta función no devuelve un valor.'
console.log(resultadoSinRetorno); // undefined
```

4. Funciones Anidadas

Puedes declarar funciones dentro de otras funciones. Estas **funciones anidadas** tienen acceso a las variables del ámbito que las contiene.

```
function funcionExterna() {
    const variableExterna = 'Variable en la función externa';

    function funcionInterna() {
        console.log(variableExterna);
    }

    funcionInterna(); // 'Variable en la función externa'
}
funcionExterna();
```

5. Funciones de Orden Superior (Higher-Order Functions)

Las **funciones de orden superior** son funciones que toman otras funciones como argumentos o devuelven funciones como resultado.

Ejemplos comunes de funciones de orden superior incluyen `map`, `filter` y `reduce`, que se utilizan

para manipular arrays en JavaScript.

```
const numeros = [1, 2, 3, 4, 5];

const cuadrados = numeros.map(x => x * x);
console.log(cuadrados); // [1, 4, 9, 16, 25]

const pares = numeros.filter(x => x % 2 === 0);
console.log(pares); // [2, 4]
const suma = numeros.reduce((acumulador, x) => acumulador + x, 0);
console.log(suma); // 15
```

6. Cierre (Closure)

El cierre o closure es un concepto importante en JavaScript que permite a una función acceder a variables del ámbito que la contiene, incluso después de que la función contenedora haya finalizado su ejecución.

```
function creaFuncion() {
    const mensaje = 'Mensaje desde el cierre';

    return function () {
        console.log(mensaje);
    };
}
const nuevaFuncion = creaFuncion();
nuevaFuncion(); // 'Mensaje desde el cierre'
```

En este ejemplo, `nuevaFuncion` es una función que se devuelve desde `creaFuncion`.

Aunque `creaFuncion` ya ha finalizado su ejecución, `nuevaFuncion` aún puede acceder a la variable mensaje en el ámbito de `creaFuncion`.

7. Funciones Autoinvocadas (Immediately Invoked Function Expression, IIFE)

Las **funciones autoinvocadas** son funciones que se ejecutan inmediatamente después de su declaración.

Se utilizan comúnmente para crear un ámbito aislado y evitar la contaminación del ámbito global.

Para crear una IIFE, envuelve la función en paréntesis y luego añade otro par de paréntesis al final para invocarla.

```
(function () {
```

```
const mensaje = 'Esta función se ejecuta inmediatamente después de su declaración';
console.log(mensaje);
})();

// 'Esta función se ejecuta inmediatamente después de su declaración'
```

8. Funciones de Retorno Implícito

Cuando una función flecha solo contiene una expresión, puedes omitir las **llaves** y la palabra clave **return**. La expresión se devuelve automáticamente.

```
const cuadrado = x => x * x;
const resultado = cuadrado(5);

console.log(resultado); // 25
```

9. Funciones como Objetos

En JavaScript, las funciones son **objetos** de primera clase, lo que significa que pueden tener propiedades y métodos, ser asignadas a variables, y pasar como argumentos a otras funciones.

```
function saludo() {
    console.log('Hola');
}

saludo.propiedad = 'Esta es una propiedad de la función saludo';

console.log(saludo.propiedad); // 'Esta es una propiedad de la función saludo'
```

10. Funciones Constructoras

Las **funciones constructoras** se utilizan para crear objetos en JavaScript.

Se declaran utilizando la notación de función declarada y, por convención, se nombran con la primera letra en mayúsculas.

Para crear un objeto a partir de una función constructora, utiliza la palabra clave `new`.

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;

    this.presentarse = function () {
        console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
```

```
    };
}
const juan = new Persona('Juan', 28);
juan.presentarse(); // 'Hola, mi nombre es Juan y tengo 28 años.'
```

11. Funciones Generadoras (Generator Functions)

Las **funciones generadoras** son un tipo especial de función en JavaScript que pueden pausar su ejecución y luego reanudarla en un momento posterior.

Se definen utilizando la palabra clave function* y se utilizan con frecuencia junto con el operador yield.

```
function* generador() {
    yield 1;
    yield 2;
    yield 3;
}

const iterador = generador();
console.log(iterador.next().value); // 1
console.log(iterador.next().value); // 2
console.log(iterador.next().value); // 3
```

12. Método call y apply

Los métodos **call** y **apply** te permiten invocar una función estableciendo explícitamente el valor de **this** y los argumentos que se pasarán a la función.

- `call` toma el valor de this seguido de una lista de argumentos.

```
function saludo(saludo, puntuacion) {
    console.log(`${saludo}, ${this.nombre}! Tu puntuación es ${puntuacion}.`);
}

const persona = {
    nombre: 'Ana'
};
saludo.call(persona, 'Hola', 42); // 'Hola, Ana! Tu puntuación es 42.'
```

- `apply` toma el valor de this seguido de un array de argumentos.

```
function saludo(saludo, puntuacion) {
```

```
    console.log(`${saludo}, ${this.nombre}! Tu puntuación es ${puntuacion}.`);
}

const persona = {
    nombre: 'Ana'
};
saludo.apply(persona, ['Hola', 42]); // 'Hola, Ana! Tu puntuación es 42.'
```

13. Método bind

El método **bind** te permite crear una nueva función que tiene su valor de this y argumentos predefinidos establecidos en valores específicos.

```
function saludo(saludo, puntuacion) {
    console.log(`${saludo}, ${this.nombre}! Tu puntuación es ${puntuacion}.`);
}

const persona = {
    nombre: 'Ana'
};
const saludoAna = saludo.bind(persona, 'Hola');
saludoAna(42); // 'Hola, Ana! Tu puntuación es 42.'
```