Funciones Básicas

¿Qué es una función?

Una función es un bloque de código reutilizable diseñado para realizar una tarea específica. Permite dividir un programa en partes más manejables y comprensibles, facilitando la organización y el mantenimiento del código.

Declaración y uso de funciones

Las funciones en JavaScript se declaran usando la palabra clave `function`. Su sintaxis básica es:

```javascript
function nombreDeLaFuncion(param1, param2) {
    // Código a ejecutar
    return resultado;
}
```

Suma de dos números:

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

console.log(sumar(3, 4)); // Salida: 7
```

En este ejemplo:

- `sumar` es el nombre de la función.
- `a` y `b` son los parámetros.
- `return` devuelve el resultado de la operación.

Parámetros y valores de retorno

Los parámetros y los valores de retorno son conceptos fundamentales en las funciones de JavaScript. Permiten que las funciones sean flexibles y útiles en diversos escenarios.

Parámetros

Los parámetros son las entradas que recibe una función para realizar su tarea. Existen dos tipos principales:

1. **Obligatorios**: Son aquellos que deben proporcionarse cuando se llama a la función. Si no se pasan, el programa puede fallar o generar un comportamiento inesperado.
2. **Opcionales**: Son aquellos que tienen un valor predeterminado asignado. Si no se especifica un valor al llamar a la función, la función utilizará el valor predeterminado.

Esto hace que las funciones sean más robustas y versátiles al manejar diferentes casos de uso sin necesidad de validaciones adicionales.

Valores de retorno

Un valor de retorno es el resultado que una función produce después de realizar sus operaciones. Para devolver un valor, se utiliza la palabra clave `return`. Sin ella, la función devolverá `undefined` por defecto.

Función sin retorno:

```javascript
function saludar(nombre) {
    console.log(`Hola, ${nombre}!`);
}

saludar("Ana"); // Salida: Hola, Ana!
```

En este ejemplo:

- La función `saludar` recibe un parámetro `nombre` y ejecuta una acción (mostrar un mensaje en la consola).
- Como no tiene una declaración `return`, la función no devuelve ningún valor; simplemente realiza su tarea.

Parámetros opcionales:

```
function saludar(nombre = "amigo") {
    console.log(`Hola, ${nombre}!`);
}

saludar(); // Salida: Hola, amigo!
saludar("Luis"); // Salida: Hola, Luis!
```

En este caso:

- La función utiliza un valor predeterminado para el parámetro `nombre`. Si el usuario no proporciona un valor al llamar a la función, `nombre` tomará automáticamente el valor "amigo".

> **Info**
>
> - Las funciones pueden devolver cualquier tipo de dato: números, cadenas, objetos, funciones, etc.
> - Una función puede tener múltiples parámetros. Es fundamental respetar el orden en el que se declaran, ya que este define cómo se asignan los valores.
> - Es buena práctica proporcionar valores predeterminados para parámetros opcionales y evitar errores en caso de que no se pasen todos los argumentos esperados.

Funciones flecha (ES6)

Introducidas en ES6, las funciones flecha (`arrow functions`) ofrecen una sintaxis más concisa para escribir funciones. Esto nos permite utilizarlas en contextos donde su sintaxis no complica la lectura del código.

Sintaxis

```
const nombreDeLaFuncion = (param1, param2) => {
    // Código a ejecutar
    return resultado;
};
```

Ejemplo 1: Suma de dos números

```
const sumar = (a, b) => a + b;
console.log(sumar(3, 4)); // Salida: 7
```

Ventajas de las funciones flecha

- Concisión: Son más cortas y fáciles de leer
- Ámbito léxico de `this`**: No redefinen el valor de `this`, lo cual es útil en funciones de callback.

Ejemplo 2: Uso de `this`

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

    // Función tradicional
    this.saludar = function() {
        console.log(`Hola, soy ${this.nombre}`);
    };

    // Función flecha
```

```
    this.presentar = () => {
        console.log(`Hola, me llamo ${this.nombre}`);
    };
}

const juan = new Persona("Juan");
juan.saludar(); // Salida: Hola, soy Juan
juan.presentar(); // Salida: Hola, me llamo Juan
```

Ámbito de Variables

El ámbito de una variable define dónde puede ser accedida o utilizada en el código. En JavaScript, existen dos tipos principales de ámbito:

Variables globales

Son accesibles desde cualquier lugar del código. Se declaran fuera de cualquier función o bloque.

Ejemplo:

```
let mensaje = "Hola, mundo!";

function mostrarMensaje() {
    console.log(mensaje);
}

mostrarMensaje(); // Salida: Hola, mundo!
```

Variables locales

Solo son accesibles dentro de la función o bloque donde fueron declaradas.

Ejemplo:

```
function saludar() {
    let mensaje = "Hola desde dentro de la función";
    console.log(mensaje);
}

saludar(); // Salida: Hola desde dentro de la función
console.log(mensaje); // Error: mensaje no está definido
```

Variables con `var`, `let` y `const`

- `var`: Tiene ámbito global y puede ser redeclarada.
- `let` y `const`: Tienen ámbito de bloque y no pueden ser redeclaradas en el mismo ámbito.

Ejemplo:

```
if (true) {
    var x = 10;
    let y = 20;
    const z = 30;
}

console.log(x); // Salida: 10
console.log(y); // Error: y no está definido
console.log(z); // Error: z no está definido
```

Programación Asíncrona en JavaScript

JavaScript es un lenguaje de programación de un solo hilo, lo que significa que solo puede ejecutar una tarea a la vez. Sin embargo, muchas veces necesitamos realizar operaciones que toman tiempo, como solicitudes a servidores o tareas de lectura de archivos. Para manejar estas situaciones sin bloquear la ejecución del código, JavaScript utiliza mecanismos de programación asíncrona como callbacks y promesas.

Callbacks

Un callback es una función que se pasa como argumento a otra función, y que se invoca después de que la función receptora haya terminado su ejecución. Los callbacks son fundamentales en JavaScript, especialmente en operaciones asincrónicas como la lectura de archivos, las solicitudes HTTP o los temporizadores.

> **Info**
>
> 1. **Control de flujo**: Permiten manejar la ejecución de código de manera flexible.
> 2. **Reutilización**: Las funciones callback pueden ser reutilizadas en múltiples contextos.
> 3. **Manejo de tareas asincrónicas**: Facilitan la programación de procesos que no se completan de inmediato, como esperar una respuesta del servidor.

Veamos ahora un ejemplo de cómo utilizarlos

Sin Callback

```
function saludar(nombre) {
    console.log(`Hola, ${nombre}!`);
}

saludar("Ana"); // Salida: Hola, Ana!
```

En este caso, `saludar` es una función simple que se ejecuta inmediatamente cuando se llama.

Con Callback

```
function procesarUsuario(nombre, callback) {
    console.log(`Procesando usuario: ${nombre}`);
    callback(nombre);
}

function saludar(nombre) {
    console.log(`Hola, ${nombre}!`);
}

procesarUsuario("Luis", saludar);
```

Salida:

```
Procesando usuario: Luis
Hola, Luis!
```

- `procesarUsuario` toma dos argumentos: un nombre y una función `callback`.
- `callback` se invoca después de procesar al usuario.

En este caso `saludar` se invoca cuando ha terminado de ejecutarse la primera función.

Callbacks en Operaciones Asincrónicas

Una de las principales razones para usar callbacks es manejar tareas asincrónicas, como el acceso a bases de datos, la comunicación con servicios externos (APIs), o temporizadores.

Ejemplo: SetTimeout

El método `setTimeout` es una función nativa de JavaScript que permite ejecutar un bloque de código que le pasamos dentro de un callbak, después de un período de tiempo definido:

```
console.log("Inicio");

setTimeout(() => {
    console.log("Tarea asincrónica completada");
}, 2000);

console.log("Fin");
```

Salida:

```
Inicio
Fin
Tarea asincrónica completada
```

Veamos ahora los pasos que han sucedido:

- **Ejecución sincrónica inicial:**
  - El programa ejecuta `console.log("Inicio");` y se muestra inmediatamente en la consola.
- **Registro de la tarea asincrónica:**
  - `setTimeout` recibe dos argumentos:
    - Una función callback: `() => { console.log("Tarea asincrónica completada"); }`.
    - Un tiempo en milisegundos: `2000` (2 segundos).
  - El motor de JavaScript registra la tarea para que la callback se ejecute después del tiempo especificado, pero **no bloquea** la ejecución del programa mientras espera.
- **La ejecución continua:**
  - Mientras se espera el tiempo especificado, el programa continúa ejecutando las siguientes líneas. En este caso, se muestra `console.log("Fin");` en la consola.
- **Ejecución de la callback:**
  - Cuando pasan los 2 segundos, el motor de JavaScript saca la callback de la cola de eventos y la ejecuta, mostrando `Tarea asincrónica completada`.

El uso de una callback en `setTimeout` ilustra cómo podemos diferir la ejecución de una función hasta que se cumplan ciertas condiciones (en este caso, el tiempo). Esto es fundamental para gestionar tareas asincrónicas y evitar bloqueos, lo que es crucial en entornos que dependen de la interacción del usuario o de la red.

Callbacks Anidados

Sin embargo el uso excesivo de callbacks puede llevar a un código complicado y difícil de leer, conocido como "callback hell":

Ejemplo:

```
setTimeout(() => {
    console.log("Paso 1 completado");
    setTimeout(() => {
        console.log("Paso 2 completado");
        setTimeout(() => {
            console.log("Paso 3 completado");
        }, 1000);
    }, 1000);
}, 1000);
```

Salida:

```
Paso 1 completado
Paso 2 completado
Paso 3 completado
```

En estos casos el código se vuelve difícil de mantener y escalar. y es cuando soluciones modernas como las Promesas y `async/await` ayudan a evitar este problema.

Promesas

Una promesa es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Fue introducida en ES6 para simplificar la programación asíncrona y evitar el "callback hell" que se venía sufriendo hasta entonces.

Una promesa siempre ha de encontrarse en uno de los siguientes estados:

> **Estados de una promesa**
>
> 1. Pending (pendiente): La promesa se encuentra en espera.
> 2. Fulfilled (cumplida): La operación se completó exitosamente.
> 3. Rejected (rechazada): Hubo un error o fallo en la operación.

Sintaxis de una Promesa

```
const promesa = new Promise((resolve, reject) => {
    // Operación asíncrona
    let exito = true;

    if (exito) {
        resolve("Operación exitosa");
    } else {
        reject("Hubo un error");
    }
});

promesa
    .then(resultado => console.log(resultado))
    .catch(error => console.error(error));
```

Análisis:

- `resolve` y `reject` son funciones que se usan para cambiar el estado de la promesa.
- `then` maneja el caso exitoso.
- `catch` maneja errores.

Encadenamiento de Promesas

Las promesas se pueden encadenar para manejar secuencias de operaciones.

```
new Promise((resolve) => {
    resolve("Paso 1 completado");
})
    .then(resultado => {
        console.log(resultado);
        return "Paso 2 completado";
    })
    .then(resultado => {
        console.log(resultado);
        return "Paso 3 completado";
    })
    .then(resultado => {
        console.log(resultado);
    })
    .catch(error => console.error(error));
```

Salida:

```
Paso 1 completado
Paso 2 completado
Paso 3 completado
```

Como podemos ver, este sistema nos permite escribir un código que si bien sigue siendo secuencial ya que unos pasos dependen de la compleción de los anteriores, no es tan difícil de leer, escribir e interpretar.

Problemas de las Promesas (`.then`)

Aunque las **Promesas** resolvieron muchos de los problemas del Callback Hell, especialmente con la mejora en legibilidad y la capacidad de manejar errores de forma centralizada con `.catch`, tienen sus propios desafíos, lo que motivó la creación de `async/await`.

1. Encadenamiento Extenso y Jerarquía de `.then`

Cuando se trabajan con múltiples promesas encadenadas, el código puede volverse difícil de seguir, especialmente si hay muchas operaciones secuenciales. Aunque no tiene el nivel de anidación del Callback Hell, puede convertirse en una "escalera de `.then`":

```
realizarOperacion()
    .then((resultado1) => {
        return otraOperacion(resultado1);
    })
    .then((resultado2) => {
        return otraOperacionMas(resultado2);
    })
    .then((resultado3) => {
        console.log("Resultado final:", resultado3);
    })
    .catch((error) => {
        console.error("Error:", error);
    });
```

Esto sigue siendo más legible que el Callback Hell, pero con muchas operaciones la jerarquía de `.then` puede ser difícil de manejar y mantener.

2. Manejo de Errores

El manejo de errores con Promesas se realiza mediante `.catch`. Sin embargo, si hay múltiples errores posibles en diferentes pasos, puede ser difícil rastrear exactamente dónde ocurrió el problema:

```
realizarOperacion()
    .then((resultado1) => {
        return otraOperacion(resultado1);
    })
    .catch((error) => {
        console.error("Error en otraOperacion:", error);
    })
    .then((resultado2) => {
        return otraOperacionMas(resultado2);
    })
    .catch((error) => {
        console.error("Error en otraOperacionMas:", error);
    });
```

Esto puede resultar en duplicación de código o en confusión si no se manejan los errores correctamente.

3. Poca Similitud con Código Sincrónico

Las Promesas requieren un paradigma mental diferente, ya que el flujo de ejecución no se parece al del código sincrónico. Esto puede ser más difícil de entender para principiantes:

```
realizarOperacion()
    .then((resultado1) => {
        console.log("Operación completada:", resultado1);
    });
```

La lógica de "esperar y continuar" no es explícita, ya que los desarrolladores deben recordar que las operaciones dentro de un `.then` son asincrónicas.

4. Falta de Concisión en Operaciones Complejas

Cuando tienes operaciones condicionales o bucles, el código basado en Promesas puede ser complicado y extenderse innecesariamente:

```
let resultados = [];
operacion1()
    .then((res1) => {
        resultados.push(res1);
        return operacion2();
    })
    .then((res2) => {
        resultados.push(res2);
        if (res2 === "condicion") {
            return operacion3();
        } else {
            return operacion4();
        }
    })
    .then((resFinal) => {
        resultados.push(resFinal);
        console.log("Resultados:", resultados);
    })
    .catch((error) => {
        console.error("Error:", error);
    });
```

El flujo condicional o basado en iteraciones se vuelve tedioso de leer y escribir.

El Surgimiento de `async/await`

Para resolver estas limitaciones, se introdujo `async/await` en JavaScript (ES2017). Esto hace que el código asincrónico se vea y se comporte más como código sincrónico, abordando directamente los problemas de las Promesas.

Ventajas de `async/await` sobre `.then`

Con `async/await`, el flujo del programa es más lineal y fácil de seguir:

```
async function ejecutarOperaciones() {
   try {
      const resultado1 = await realizarOperacion();
      const resultado2 = await otraOperacion(resultado1);
      const resultadoFinal = await otraOperacionMas(resultado2);
      console.log("Resultado final:", resultadoFinal);
   } catch (error) {
       console.error("Error:", error);
   }
}

   ejecutarOperaciones();
```

El código se parece al código sincrónico, lo que facilita la comprensión y se lee de una manera mucho mas intuitiva debido a la misma semántica de las palabras `async`, `try`, `await` y `catch`.

Además, en lugar de múltiples `.catch`, puedes usar un único bloque `try-catch` para manejar todos los errores en el flujo asincrónico:

En definitiva el modelo de Promesas (`.then`) fue un avance significativo sobre los callbacks, pero todavía tenía problemas de legibilidad y escalabilidad en escenarios complejos. `async/await` se introdujo como una mejora que combina el poder de las Promesas con una sintaxis más simple y legible, alineando mejor el código asincrónico con el estilo del código sincrónico. Esto ha convertido a `async/await` en el enfoque preferido para trabajar con tareas asincrónicas en JavaScript.