Diplomatura en

GLOBALACADEMY

FullStack Developer

JavaScript
Sintaxis Avanzada y Modularización

Uso de this en un addEventListener

En JavaScript puede ser un poco complicado debido a cómo se maneja el contexto de ejecución. Cuando agregas un listener de eventos a un elemento, this dentro de la función del evento generalmente se refiere al elemento al cual el evento fue agregado. Sin embargo, el comportamiento puede variar si usas diferentes tipos de funciones (tradicional vs. flecha).

Uso de una Función Tradicional

Cuando usas una función tradicional como callback en addEventListener, this se refiere al elemento que recibe el evento. Este es el comportamiento predeterminado y es lo que se espera al manejar eventos en elementos DOM.

```
const boton = document.querySelector('button');

boton.addEventListener('click', function() {
  console.log(this);  // Aquí, `this` se refiere al elemento `button`.
  this.textContent = 'Click realizado';
});
```

Uso de una Función Flecha

Las funciones flecha no tienen su propio this, en su lugar capturan el this del contexto en el cual fueron creadas (binding léxico). Esto significa que si usas una función flecha como handler en un addEventListener, this no se referirá al elemento del evento, sino al this del contexto externo donde la función flecha fue definida.

```
const boton = document.querySelector('button');

boton.addEventListener('click', () => {
  console.log(this);  // `this` no se refiere al botón, sino al objeto `window` o al contexto en el cual la función flecha fue creada.
  // this.innerHTML = 'Click realizado';  // Esto no funcionaría como se espera.
});
```

Consejos para Manejar this en Event Listeners

Si necesitas que this se refiera al elemento del evento y deseas usar una función flecha, puedes acceder al elemento objetivo del evento a través del objeto de evento que se pasa automáticamente al handler:

```
const boton = document.querySelector('button');

boton.addEventListener('click', (event) => {
  console.log(event.target);  // `event.target` se refiere al elemento `button`.
  event.target.innerHTML = 'Click realizado';
});
```

Usar event.target te permite obtener el elemento que desencadenó el evento, lo cual es útil especialmente cuando this no se comporta como esperabas debido al uso de funciones flecha.

Operadores Avanzados y Técnicas de Manejo de Datos

1. Operadores Ternarios

Proporciona una manera concisa de ejecutar asignaciones o retornos basados en una condición.

```
condición ? expresión1 : expresión2
```

Ejemplo

```
let edad = 18;
let esAdulto = (edad >= 18) ? 'Sí' : 'No';
console.log(esAdulto); // 'Sí'
```

2. Operadores de Tipo

•**typeof**:

Retorna una cadena que describe el tipo de la variable.

Ejemplo

```
console.log(typeof "ChatGPT"); // 'string'
```

3. Operador de Propagación (Spread)

El Operador de Propagación (Spread Operator), denotado por tres puntos (...), es una característica muy útil en JavaScript que fue introducida en ECMAScript 2015 (ES6). Este operador permite expandir elementos de un iterable (como un arreglo) en lugares donde se esperan cero o más argumentos (en llamadas a funciones) o elementos (en arreglos).

Uso en Funciones

El operador de propagación puede ser utilizado para pasar los elementos de un arreglo como argumentos individuales a una función. Esto es especialmente útil cuando no sabes cuántos argumentos se pasarán a la función o cuando los datos ya están en un arreglo.Ejemplo:

Ejemplo

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

const numeros = [1, 2, 3];

// Pasando los elementos del arreglo como argumentos independientes
console.log(suma(...numeros)); // Salida: 6
```

Uso en Arreglos

El operador de propagación también puede ser usado para crear nuevos arreglos a partir de arreglos existentes, lo que facilita la concatenación y la manipulación de arreglos sin modificar los originales.

```
const parte1 = [1, 2, 3];
const parte2 = [4, 5, 6];

// Combinando dos arreglos en uno nuevo
const combinado = [...parte1, ...parte2];
console.log(combinado); // Salida: [1, 2, 3, 4, 5, 6]
```

Uso en Objetos

Desde ECMAScript 2018 (ES9), el operador de propagación también puede ser utilizado para copiar propiedades de objetos en otros objetos, lo que facilita la tarea de combinar o modificar objetos de manera inmutable (sin modificar el objeto original).

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };

// Combinando dos objetos en uno nuevo
const combinado = { ...obj1, ...obj2 };
console.log(combinado); // Salida: { a: 1, b: 3, c: 4 }
```

4. Valores Truthy y Falsy

En JavaScript, cada valor tiene una propiedad "booleana" inherente. Cuando se evalúa un valor en un contexto que espera un booleano (como en un condicional if, while, en operadores lógicos, etc.), JavaScript automáticamente convierte ese valor a true o false según si el valor es considerado truthy o falsy.

¿Qué son los Valores Falsy?

Los valores falsy son aquellos que se consideran como false cuando se convierten a booleano. Hay exactamente seis valores falsy en JavaScript:

1.false - El literal booleano false.

2.0 - El número cero.

3.-0 - El cero negativo.

4."" (cadena vacía) - Una cadena de caracteres vacía.

5.null - Indica una ausencia deliberada de valor.

6.undefined - Indica que una variable no ha sido asignada.

7.NaN - Representa "No es un Número", resultado de operaciones aritméticas indefinidas o erróneas.

¿Qué son los Valores Truthy?

Cualquier valor que no es falsy se considera truthy. Esto incluye:

- **"hello"** (cualquier cadena no vacía)
- **42** (cualquier número diferente de cero)
- **-42** (cualquier número negativo)
- **[]** (un array vacío)
- **{}** (un objeto vacío)
- **function() {}** (funciones)

Estos valores se evaluarán como true en contextos booleanos.

Ejemplos de Uso

El entendimiento de truthy y falsy es crucial para escribir condicionales efectivos y para usar de manera adecuada los operadores lógicos para control de flujo y asignaciones:

```
if ("hello") {
  console.log("La cadena es truthy");  // Se ejecuta porque "hello" es truthy
}

if ("") {
  console.log("Esta cadena no se mostrará");  // No se ejecuta porque "" es falsy
}

const nombre = "";
const saludo = nombre || "Invitado";
console.log(saludo);  // Imprime "Invitado" porque nombre es una cadena vacía, un valor falsy
```

5. Valores por Defecto con Operadores Lógicos en JavaScript

En JavaScript, los operadores lógicos como || (OR) y ?? (nullish coalescing) son comúnmente utilizados para establecer valores predeterminados de variables. Estos operadores permiten desarrollar código más conciso y fácil de mantener, especialmente en situaciones donde los valores de algunas variables pueden ser undefined o null.

Operador OR (||)

El operador OR (||) es ampliamente usado para asignar valores por defecto debido a su comportamiento con valores truthy y falsy. Si el primer operando es truthy, || retorna ese valor; de lo contrario, evalúa y retorna el segundo operando. Esto es útil para establecer un valor predeterminado cuando una variable podría ser falsy.

```
variable = valorPotencialmenteFalsy || valorPredeterminado;
```

```
function saludar(nombre) {
  nombre = nombre || 'Invitado';
  console.log(`Hola, ${nombre}`);
}

saludar();  // Output: "Hola, Invitado"
saludar('Elena');  // Output: "Hola, Elena"
```

En este ejemplo, si nombre es cualquier valor falsy (como undefined, null, "", etc.), nombre será asignado a "Invitado". Este comportamiento es especialmente útil para parámetros de funciones donde no se pasan valores o se pasan valores falsy.

Operador de Fusión de Nulos (??)

El operador de fusión de nulos (??) es una adición relativamente nueva a JavaScript que ofrece un control más fino sobre cómo se asignan los valores por defecto. A diferencia de ||, que devuelve el segundo operando si el primero es cualquier valor falsy, ?? solo devuelve el segundo operando si el primero es null o undefined.

```
variable = valorPotencialmenteNullOUndefined ?? valorPredeterminado;
```

```
function configurarColor(color) {
  color = color ?? 'azul';
  console.log(`Color seleccionado: ${color}`);
}

configurarColor();  // Output: "Color seleccionado: azul"
configurarColor('rojo');  // Output: "Color seleccionado: rojo"
configurarColor(null);  // Output: "Color seleccionado: azul"
```

En este ejemplo, color será "azul" solo si color es null o undefined. Esto es particularmente útil cuando se trabaja con tipos de datos donde 0, false, o "" (que son falsy pero pueden ser valores válidos y deseados) no deben ser reemplazados por un valor predeterminado.

También podemos asignar directamente valores por defecto a los parámetros en la declaración de la función. Esto hace que el código sea más claro y reduce la necesidad de comprobaciones adicionales dentro de la función.

```
function saludar(nombre = 'Invitado') {
  console.log(`Hola, ${nombre}`);
}

saludar(); // Hola, Invitado
saludar('Elena'); // Hola, Elena
```

Operador de Encadenamiento Opcional (?.)

El operador de encadenamiento opcional ?. es una adición relativamente nueva a JavaScript, introducida en ES2020, que permite leer el valor de una propiedad profundamente anidada dentro de una cadena de objetos sin tener que validar explícitamente cada referencia en la cadena. Este operador es especialmente útil para tratar con objetos que pueden tener partes no definidas o nulas sin tener que utilizar múltiples sentencias de verificación.

Funcionamiento del Operador ?.

El operador ?. funciona de manera similar al operador de acceso a propiedades (.), pero con una característica clave: si la referencia a la izquierda de ?. es null o undefined, la expresión a la derecha del operador no se evaluará y la expresión completa devolverá undefined en lugar de lanzar un error de tipo.

```
let usuario = {
  nombre: 'Carlos',
  detalles: {
      edad: 30,
      direccion: {
          calle: 'Principal',
          numero: 123
      }
  }
};

// Uso del operador de encadenamiento opcional
let calle = usuario.detalles?.direccion?.calle;
console.log(calle); // 'Principal'

// Sin el operador de encadenamiento opcional
let calleErronea = usuario.detalles.direccion.calle;
console.log(calleErronea); // 'Principal', pero si detalles o direccion son undefined, causaría error
```

Aplicaciones del Operador ?.

Acceso a Propiedades Anidadas:

Evita errores de TypeError cuando alguna parte de la cadena de acceso es undefined o null.

Ejemplo: let codigoPostal = usuario.detalles?.direccion?.codigoPostal;

Llamadas de Métodos Seguras:

Permite realizar llamadas de métodos solo cuando el método existe.
Ejemplo: usuario.metodoInexistente?.(); devuelve undefined si métodoInexistente no está definido.

Acceso a Elementos de Arrays:

Proporciona una manera segura de acceder a índices de arrays que podrían no estar definidos.

Ejemplo: let primerElemento = arrayInseguro?.[0];

Beneficios del Uso de ?.

- **Mejora la Seguridad del Código:** Reduce el riesgo de errores en tiempo de ejecución causados por referencias null o undefined.
- **Código Más Limpio y Conciso:** Reduce la necesidad de múltiples comprobaciones if o uso de operadores lógicos para evitar errores de tipo.
- **Facilita la Lectura y Mantenimiento:** El código que utiliza ?. es más fácil de leer y mantener, especialmente en aplicaciones grandes con estructuras de datos complejas.

Simplificando la Asignación de Datos con Destructuración en ES6

¿Qué es la Destructuración?

- Definición: La destructuración es una expresión de JavaScript que permite extraer datos de arrays, objetos, y mapas en variables separadas de una manera más legible y eficiente.

- Ventajas: Reduce la necesidad de acceder a los valores a través de índices o claves específicas, simplificando el código y haciéndolo más mantenible.

Conceptos Básicos de la Destructuración

- Puedes destructurar objetos y arrays.

- La sintaxis varía ligeramente entre destructuración de objetos y arrays debido a su estructura.

Ejemplo de Destructuración de Objetos

```
const usuario = { nombre: 'Laura', edad: 29 };
const { nombre, edad } = usuario;
console.log(nombre); // Imprime 'Laura'
console.log(edad); // Imprime 29
```

nombre y edad se asignan a las variables correspondientes del objeto usuario directamente.

Ejemplo de Destructuración de Arrays

```
const frutas = ['manzana', 'banana', 'cereza'];
const [primera, segunda] = frutas;
console.log(primera); // Imprime 'manzana'
console.log(segunda); // Imprime 'banana'
```

primera y segunda capturan los dos primeros elementos del array frutas directamente.

Casos de Uso Prácticos

•Extraer múltiples propiedades de un objeto recibido de una API.

•Facilitar la lectura de múltiples valores de retorno de una función.

Destructuración de Arrays

La destructuración de arrays permite asignar elementos de un array a variables individuales de una forma compacta y directa. Simplifica el acceso a los elementos del array y mejora la legibilidad del código.

Sintaxis Básica

```
const numeros = [1, 2, 3];
const [primero, segundo, tercero] = numeros;
// primero, segundo y tercero se asignan a 1, 2, y 3 respectivamente.
```

Omitiendo Elementos

```
const colores = ['rojo', 'verde', 'azul', 'amarillo'];
const [primero, , tercero] = colores;
console.log(primero); // 'rojo'
console.log(tercero); // 'azul'
```

La coma adicional indica que el segundo elemento (verde) no se asigna a ninguna variable.

Usando el Operador Rest para Recolectar Elementos Restantes

```
const frutas = ['manzana', 'banana', 'cereza', 'durazno'];
const [primera, ...resto] = frutas;
console.log(primera); // 'manzana'
console.log(resto); // ['banana', 'cereza', 'durazno']
```

El operador ... se utiliza para asignar todos los elementos restantes a una variable resto.

Intercambio de Variables sin Variables Temporales

```
let a = 1, b = 2;
[a, b] = [b, a];
console.log(a); // 2
console.log(b); // 1
```

La destructuración también puede ser usada para intercambiar los valores de las variables de manera eficiente y sin necesidad de una variable temporal.

Destructuración Anidada

La destructuración anidada permite acceder a propiedades más profundas dentro de objetos y arrays en una sola expresión, facilitando la extracción de datos de estructuras complejas.

```
const usuario = {
  nombre: 'Carlos',
  detalles: {
      edad: 30,
      direccion: {
          calle: 'Principal',
          ciudad: 'Ciudad Central'
      }
  }
};

const { detalles: { direccion: { calle, ciudad } } } = usuario;
console.log(calle, ciudad); // "Principal", "Ciudad Central"
```

Ejemplo de Destructuración Anidada en Arrays:

```
const colores = [["rojo", "verde"], ["azul", "amarillo"]];
const [[primero, segundo], [tercero]] = colores;
console.log(primero, segundo, tercero); // "rojo", "verde", "azul"
```

Modularización

Los módulos en JavaScript ES6+ ofrecen una forma de encapsular código en unidades separadas, facilitando la reutilización y la gestión de las dependencias.

Introducción a Módulos en ES6+

- Los módulos ES6+ permiten dividir el código JavaScript en archivos separados, cada uno definido con su propio alcance.
- Cada módulo puede exportar funciones, clases, objetos o variables para ser utilizadas en otros módulos.

Exportar en Módulos

- **Sintaxis para exportar:**

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

Importar en Módulos

- **Sintaxis para importar:**

```
import { suma } from './suma.js';
console.log(suma(2, 3)); // 5
```

Exportaciones Nombradas vs. Exportaciones por Defecto

Exportaciones Nombradas

•Permiten exportar múltiples valores desde un módulo.

•Deben ser importados usando exactamente el mismo nombre que fueron exportados, a menos que se utilice as para renombrar.

•Ejemplo:

```
// archivo utilidades.js
export const pi = 3.14159;
export function cuadrado(x) {
    return x * x;
}

// archivo main.js
import { pi, cuadrado } from './utilidades.js';
```

Exportaciones por Defecto

•Cada módulo puede tener una única exportación por defecto.

•Se importa sin necesidad de usar llaves y se puede nombrar como se desee al importar.

•**Ejemplo:**

```
function procesarTexto(texto) {
    // Realiza alguna operación con el texto
    return texto.toUpperCase();
}

export default procesarTexto;
```

```
// Importa la función default del módulo y la nombra a gusto
import procesar from './procesador.js';

const textoOriginal = "Hola Mundo!";
const textoProcesado = procesar(textoOriginal);
console.log(textoProcesado);  // Salida: "HOLA MUNDO!"
```