## Debugging y manejo de errores

Debugging es el proceso de encontrar y corregir errores en el código. Es una parte importante del desarrollo de software y es esencial para garantizar que el código funcione correctamente. En JavaScript, hay varias herramientas y técnicas disponibles para ayudar en el debugging y manejo de errores.

Una de las herramientas más comunes para manejar errores en JavaScript es el bloque try-catch. Este bloque permite ejecutar código que podría generar un error, y capturar ese error en caso de que ocurra. Por ejemplo:

**try** {

//Código que podría generar un error

} **catch** (error) {

//Código que se ejecuta si ocurre un error

console.log(error);

}

La consola de JavaScript también es una herramienta útil para el debugging. La consola proporciona varios comandos para inspeccionar el código y entender el estado del programa. Por ejemplo, el comando “console.log()” permite imprimir valores en la consola para inspeccionarlos, el comando “console.table()” permite ver los valores de un objeto o array en forma de tabla, entre otros.

let array = [5,2,3,6,4];

console.table(array);

OUTPUT:

| (index) | Values |

--------------------

| 0 | 5 |

| 1 | 2 |

| 2 | 3 |

| 3 | 6 |

| 4 | 4 |

## Programación funcional

La programación funcional es un enfoque de programación que se centra en el uso de funciones y evita el uso de estado y efectos secundarios. En JavaScript, la programación funcional se basa en el uso de funciones puras, es decir, funciones que no tienen efectos secundarios y siempre devuelven el mismo resultado dado el mismo input.

Una de las características principales de la programación funcional es el uso de first-class functions (funciones como primer ciudadano), es decir, se consideran como valores y pueden ser asignados a variables, pasados como argumentos a otras funciones y retornadas como resultado de otras funciones. Esto permite crear funciones más pequeñas y reutilizables que se pueden combinar para crear funciones más complejas.

Por ejemplo, podemos crear una función “add” que toma dos números como argumentos y devuelve su suma:

**function** **add**(a, b) {

**return** a + b;

}

Luego, podemos asignar esta función a una variable y pasarla como argumento a otra función, como “makeOperation”:

Por ejemplo, la función map() es una función pura ya que toma un array de números como input y devuelve un nuevo array con los números transformados, sin modificar el array original.

# Asincronismo

# Arrays a profundidad

En esta clase, vamos a explorar algunas de las características avanzadas de los arrays en JavaScript, incluyendo la creación de arrays utilizando la clase constructora Array, copiar arrays, arrays bidimensionales y arrays de múltiples dimensiones.

# Creación de Arrays utilizando la Clase Constructora Array

En JavaScript, los arrays se pueden crear de una forma diferente a la tradicional utilizando la clase constructora Array. La clase constructora Array proporciona una forma fácil y flexible de crear arrays con diferentes elementos y longitudes.

**let** myArray = **new** Array();

También puedes especificar los elementos que deseas incluir en el array al crearlo, como se muestra a continuación:

**let** myArray = **new** Array(1, 2, 3, 4);

### Array.from()

La función Array.from() permite crear un nuevo array a partir de un objeto similar a un array o iterable. La sintaxis para utilizar Array.from() es la siguiente:

**let** myArray = Array.from('hello');

console.log(myArray); // ["h", "e", "l", "l", "o"]

### Array(n).fill()

La función fill() permite llenar un array con un valor específico. La sintaxis para utilizar fill() es la siguiente:

**let** myArray = **new** Array(5).fill(0);

console.log(myArray); // [0, 0, 0, 0, 0]

### Array.isArray()

La función Array.isArray() permite determinar si un objeto es un array o no. La sintaxis para utilizar Array.isArray() es la siguiente:

**let** myArray = [1, 2, 3, 4];

console.log(Array.isArray(myArray)); // true

## Copiando Arrays

En JavaScript, es común tener la necesidad de hacer una copia de un array existente, ya sea para manipular los datos sin afectar el array original o para trabajar con múltiples versiones del mismo array. Hay varias maneras de hacer esto en JavaScript, incluyendo la creación de una nueva instancia de un array, el uso de métodos incorporados y el uso del spread operator.

### Crear una nueva instancia del array:

Una forma de hacer una copia de un array es crear una nueva instancia del mismo array. Esto se logra usando el constructor Array:

**const** originalArray = [1, 2, 3, 4];

**const** copiedArray = **new** Array(...originalArray);

console.log(originalArray === copiedArray); // Output: false

En este ejemplo, se utiliza el operador spread para expandir los elementos del array original en una nueva instancia del constructor Array.

### Uso de métodos incorporados:

Otro enfoque para hacer una copia de un array es utilizar los métodos incorporados de JavaScript. Por ejemplo, puedes utilizar el método slice para hacer una copia de un array:

**const** originalArray = [1, 2, 3, 4];

**const** copiedArray = originalArray.slice();

console.log(originalArray === copiedArray); // Output: false

### Otro método incorporado es el método concat:

**const** originalArray = [1, 2, 3, 4];

**const** copiedArray = originalArray.concat();

console.log(originalArray === copiedArray); // Output: false

### spread operator:

El spread operator es una forma conveniente de hacer una copia de un array. El spread operator se utiliza para expandir los elementos de un array en una lista:

**const** originalArray = [1, 2, 3, 4];

**const** copiedArray = [...originalArray];

console.log(originalArray === copiedArray); // Output: false

En este ejemplo, se utiliza el spread operator para expandir los elementos del array original en una nueva instancia de un array.

Es importante tener en cuenta que, aunque estos métodos parecen hacer una copia de un array, solo hacen una copia superficial, lo que significa que los elementos dentro del array original y la copia comparten los mismos valores. Si necesitas hacer una copia profunda de un array, es necesario utilizar una biblioteca de terceros o escribir tu propia función para hacer una copia profunda.

## Arrays Bidimensionales

Los arrays bidimensionales son un tipo de estructura de datos en la que cada elemento es un array y cada uno de esos arrays contiene más elementos. En otras palabras, son arrays de arrays. Estos pueden ser útiles para representar datos en un formato más organizado, como una tabla o una matriz.

Para crear un array bidimensional, puedes inicializar un array vacío y luego agregar arrays a sus elementos. Aquí hay un ejemplo:

**const** matrix = [];

matrix[0] = [1, 2, 3];

matrix[1] = [4, 5, 6];

matrix[2] = [7, 8, 9];

console.log(matrix);

// Output: [

// [1, 2, 3],

// [4, 5, 6],

// [7, 8, 9]

// ]

También puedes crear un array bidimensional utilizando la clase constructora Array:

**const** matrix = **new** Array(3);

**for** (**let** i = 0; i < matrix.length; i++) {

matrix[i] = **new** Array(3);

}

matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[0][2] = 3;

matrix[1][0] = 4;

matrix[1][1] = 5;

matrix[1][2] = 6;

matrix[2][0] = 7;

matrix[2][1] = 8;

matrix[2][2] = 9;

console.log(matrix);

// Output: [

// [1, 2, 3],

// [4, 5, 6],

// [7, 8, 9]

// ]

Para acceder a un elemento específico en un array bidimensional, debes proporcionar dos índices. Por ejemplo, si deseas acceder al elemento 5 en el ejemplo anterior, puedes hacerlo de la siguiente manera:

console.log(matrix[1][1]);

// Output: 5

Además de crear y acceder a los elementos de un array bidimensional, también puedes copiar un array bidimensional utilizando el spread operator:

**const** originalMatrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

**const** copiedMatrix = [...originalMatrix];

console.log(originalMatrix === copiedMatrix);

// Output: false

En este ejemplo, el spread operator se utiliza para crear una copia profunda del array original, lo que significa que los elementos dentro del array también se copian.

Además de los arrays bidimensionales, también puedes crear arrays de más dimensiones. Por ejemplo, puedes crear un array tridimensional:

**const** cube = [];

**for** (let i = 0; i < 3; i++) {

cube[i] = [];

**for** (let j = 0; j < 3; j++) {

cube[i][j] = [];

**for** (let k = 0; k < 3; k++) {

cube[i][j][k] = i + j + k;

}

}

}

console.log(cube);

// Output:

//[

// [[0, 1, 2], [1, 2, 3], [2, 3, 4]],

// [[1, 2, 3], [2, 3, 4], [3, 4, 5]],

// [[2, 3, 4], [3, 4, 5], [4, 5, 6]]

//]

Como puedes ver, los arrays de más dimensiones siguen la misma lógica de los arrays bidimensionales, pero con más niveles de índices para acceder a los elementos.

**Array.prototype.every()**

[El método every](https://platzi.com/clases/2461-arrays/41333-every/) es una función que se utiliza en los arrays en JavaScript para comprobar si todos los elementos del array cumplen con una determinada condición. Devuelve un valor booleano que indica si todos los elementos del array cumplen con la condición especificada.

Por ejemplo, supongamos que tienes un array de números y deseas comprobar si todos los números son mayores que cero:

**const** numbers = [1, 2, 3, 4, 5];

**const** allPositive = numbers.every(**function**(element) {

**return** element > 0;

});

console.log(allPositive);

// Output: true

En este ejemplo, el método every se utiliza para comprobar si todos los elementos en el array numbers son mayores que cero. La función que se pasa como argumento a every devuelve true si el elemento es mayor que cero y false en caso contrario. Si todos los elementos devuelven true, every devuelve true. De lo contrario, devuelve false.

También puedes utilizar una arrow function para hacer lo mismo:

**const** numbers = [1, 2, 3, 4, 5];

**const** allPositive = numbers.every(element => element > 0);

onsole.log(allPositive);

// Output: true

Además de comprobar si todos los elementos cumplen con una determinada condición, también puedes utilizar el método every para realizar otras tareas en cada elemento del array. Por ejemplo, supongamos que tienes un array de objetos y deseas comprobar si todos los objetos tienen una propiedad específica:

**const** objects = [{name: 'John'}, {name: 'Jane'}, {name: 'Jim'}];

**const** allHaveName = objects.every(**function**(element) {

**return** element.hasOwnProperty('name');

});

console.log(allHaveName);

// Output: true

El método every es una herramienta útil para comprobar si todos los elementos en un array cumplen con una determinada condición. Es una forma sencilla y eficiente de procesar los elementos en un array y realizar tareas específicas en ellos.

**Array.prototype.Find()**

[El método find](https://platzi.com/clases/2461-arrays/40879-find-y-findindex/) se puede utilizar en arrays para encontrar un elemento que cumpla con una condición determinada. Este método devuelve el primer elemento que cumple con la condición o undefined si no se encuentra ninguno.

El método find toma un callback como argumento. Este callback es llamado para cada elemento del array hasta que se encuentre un elemento que cumpla con la condición.

Aquí hay un ejemplo de cómo utilizar el método find para encontrar el primer número mayor que 5 en un array:

**const** numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

**const** result = numbers.find(number => number > 5);

console.log(result);

// Output: 6

En este ejemplo, el callback number => number > 5 se utiliza para determinar si un número es mayor que 5. Si se encuentra un número que cumpla con esta condición, el método find devuelve ese número y detiene la búsqueda.

También es posible utilizar el método find con objetos. Por ejemplo, aquí hay un ejemplo de cómo utilizar el método find para encontrar el primer objeto con una propiedad específica:

**const** people = [

{ name: 'John', age: 32 },

{ name: 'Jane', age: 28 },

{ name: 'Jim', age: 40 }

];

**const** result = people.find(person => person.age === 32);

console.log(result);

// Output: { name: 'John', age: 32 }

En este ejemplo, el callback person => person.age === 32 se utiliza para determinar si una persona tiene una edad de 32 años. Si se encuentra un objeto que cumpla con esta condición, el método find devuelve ese objeto y detiene la búsqueda.

**Array.prototype.findIndex()**

[El método findIndex](https://platzi.com/clases/2461-arrays/40879-find-y-findindex/) es una función que se utiliza para buscar un elemento en un array y devolver el índice de su primera aparición. Este método acepta un callback como argumento y la ejecuta para cada elemento en el array hasta que se encuentra un elemento que cumpla con la condición especificada. En ese momento, findIndex devuelve el índice del elemento encontrado.

A diferencia del método find, en este caso, solo se devuelve el primer indice del elemento que cumpla la condición, por ejemplo:

**const** numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

**const** index = numbers.findIndex(num => num > 5);

console.log(index); // Output: 5

En este ejemplo, la función de callback recibe cada elemento del array numbers y devuelve true si el elemento es mayor que 5. El método findIndex devuelve el índice del primer elemento que cumple con esta condición, que es 5.

Si no se encuentra ningún elemento que cumpla con la condición, findIndex devuelve -1. Por ejemplo:

**const** numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

**const** index = numbers.findIndex(num => num > 100);

console.log(index); // Output: -1

En este ejemplo, no hay ningún elemento en el array numbers que sea mayor que 100, por lo que findIndex devuelve -1.