JavaScript

JavaScript es un lenguaje de programación interpretado que se utiliza principalmente para desarrollar aplicaciones web.

Es uno de los lenguajes más populares y ampliamente utilizados para el desarrollo web, y se ejecuta principalmente en el navegador web del usuario.

Aunque JavaScript fue originalmente creado para agregar interactividad y dinamismo a las páginas web, hoy en día se utiliza para una amplia variedad de propósitos, incluyendo el desarrollo de aplicaciones móviles, la creación de juegos y la realización de tareas en el lado del servidor.

JavaScript se puede utilizar junto con otras tecnologías como HTML y CSS para crear experiencias de usuario ricas y atractivas en la web.

1. Spread Operator

Útil para combinar arreglos y objetos

```
// Combinar arreglos
const arrA = [1,2,3];
```

```
const arrB = [4,5,6];

//         spread operator '...'
const arrC = [...arrA, ...arrB] // [1,2,3,4,5,6]

// Combinar objetos
const objA = { name: 'Elliot' }
const objB = { lastname: 'Alderson', age: 28 }

//         spread operator '...'
const objC = { ...objA, ...objB }
// { name: 'Elliot', lastname: 'Alderson', age: 28 }
```

2. Destructuring

Sirve para extraer datos o valores de objetos y arreglos

```
// Desestructurando arreglos
const arrA = ['pen', 'apple', 'apple-pen'];
const [ val1, val2, val3 ] = arrA;

console.log(val1); // 'pen'
console.log(val2); // 'apple'
console.log(val3); // 'apple-pen'

// Desestructurando objetos
const objA = { math: 97, music: 78, english: 79 };
const { math, english } = objA;

console.log(math); // 97
console.log(english); // 79
```

3. Hoisting

El hoisting es un comportamiento en JavaScript que hace que las declaraciones de variables y funciones se "levanten" al principio del bloque de código o al principio del archivo JavaScript. Esto significa que, aunque puedas haber escrito una declaración de variable o función en cualquier parte de tu código, el intérprete de JavaScript la tratará como si estuviera al principio del bloque de código o archivo.

El hoisting solo se aplica a declaraciones de variables y funciones, no a asignaciones de variables.

```
console.log(x);  // undefined
var x = 5;
```

```
console.log(x) // undefined
x = 5
```

```
console.log(x) // 5
var x = 'hello'
```

El código del ejemplo anterior pasaría a convertirse en lo siguiente:

```
var x = undefined;
console.log(x) // undefined
x = 5;
console.log(x);
x = 'hello'
```

4. const, let y var

- Las declaraciones con `var` son "hoisteadas" y despues auto-inicializadas a *undefined*.
- Las declaraciones con `const` y `let` también son "hoisteadas", pero no inicializadas, asi que no puedes acceder a ellas antes de su inicialización.
- `const`: declara una variable constante, lo que significa que no puede ser reasignada ni redefinida una vez que se ha asignado un valor. Las constantes deben tener un valor asignado al momento de su declaración.
- `let`: declara una variable que puede ser reasignada y redefinida más adelante en el código. A diferencia de las variables declaradas con var, las variables declaradas con `let` están sujetas al TDZ (Zona Muerta Temporal). Esto significa que no pueden ser accedidas ni asignadas hasta que se les asigna un valor.
- `var`: declara una variable que puede ser reasignada y redefinida más adelante en el código. A diferencia de las variables declaradas con `let` o `const`, las variables declaradas con `var` no están sujetas al TDZ y pueden ser accedidas y asignadas en cualquier momento durante la ejecución del código. Además, las variables declaradas con `var` tienen un comportamiento de alcance diferente al de `let` y `const`. Las variables declaradas con `var` tienen un alcance global o de función, mientras que las variables declaradas con `let` y `const` tienen un alcance de bloque.

> El TDZ (Temporal Dead Zone, o Zona Muerta Temporal) es un concepto en JavaScript que se refiere a una etapa durante la ejecución de un programa en la que una variable declarada con la palabra clave `let` o `const` está en un estado de "no definida" temporalmente.

5. Scope

5.1. scope global y scope local

En JavaScript, el término "scope" se refiere a la visibilidad y accesibilidad de las variables dentro de un programa.

Hay dos tipos de scope en JavaScript: el **scope global** y el **scope local**.

Tiene el beneficio de seguridad de que las variables solo pueden ser accedidas desde una cierta área del código.

- **scope global**:
  - Se refiere a las variables que están disponibles para ser accedidas desde cualquier parte del programa.
  - Estas variables se definen fuera de cualquier función o bloque de código y a menudo se conocen como variables globales.
- **scope local**:
  - Se refiere a las variables que sólo están disponibles dentro de un bloque de código o función específico.
  - Estas variables se definen dentro de una función o bloque de código y a menudo se conocen como variables locales.

```
// Esta variable vive en el scope global
let name = 'Angela Moss';

function coolHello() {
  // Esta variable vive en el scope local
  // de la función 'coolHello'
  let greeting = 'Hello, my name is'

  // Se puede utilizar la variable 'name' porque
  // ha sido declarada en el scope global
  return greeting + name;
}
```

5.2. block scope y nested scope

Block scope y **nested scope** no se consideran tipos de scope separados en JavaScript. En cambio, son formas en las que el scope global y local existente pueden ser utilizados y aplicados dentro de un programa.

- **block scope**:
  - Se refiere a la visibilidad y accesibilidad de las variables dentro de un bloque de código, que es una sección de código rodeada por llaves "{ }".
  - `let` y `const` permiten definir variables que sólo están disponibles dentro del bloque de código en el que son definidas.

```
console.log(x) // undefined

if (true) {
  var x = 10
  console.log(x) // 10
}

console.log(x) // 10
```

```
console.log(x) // ReferenceError: x is not defined
```

```
if (true) {
  let x = 10
  console.log(x) // 10
}

console.log(x) // ReferenceError: x is not defined
```

- **nested scope**:
  - Se refiere al concepto de tener múltiples niveles de scope de bloque dentro de un programa.
  - Por ejemplo, se puede tener un bloque de código dentro de un bloque de código, creando un bloque de código anidado.

```
if (true) {
  // Esta variable sólo está disponible dentro de este bloque de código
  let message = 'Hello world!';
  if (true) {
    // Esta variable sólo está disponible dentro de este bloque de código anidado
    let nestedMessage = 'Hello world 2';
    console.log(message); // 'Hello world!'
  }

  console.log(nestedMessage); // ReferenceError: nestedMessage is not defined
}
```

6. Tipos de datos

Son los tipos de datos que una variable puede tener. Existen 9 tipos de datos en JavaScript.

- Siete **tipos de datos primitivos**:
  - `undefined`: valor asignado a variables que solo han sido declaradas y no tienen valor.
  - `boolean`: tipo de dato lógico que puede tener dos valores: `true` o `false`.
  - `number`: tipo de dato numérico.
  - `string`: secuencia de caracteres usando para representar un texto.
  - `bigint`: tipo de dato numerico que puede representar números enteros muy grandes.
  - `symbol`: nos permite obtener valores que no pueden volver a ser creados, es decir, son identificadores únicos e inmutables.
  - `Null`: representa una referencia que apunta, generalmente intencionalmente, a un valor inexistente.
- `Object`: estructura más compleja.
  - Todos los demás tipos se llaman *"primitivos"* porque sus valores pueden contener una sola cosa (ya sea una cadena, un número o lo que sea). Por el contrario, los objetos se utilizan para almacenar colecciones de datos y entidades más complejas.
- `Function`: fragmento de código que puede ser llamado por otro código o por sí mismo. Dichos valores pueden ser utilizados como identificadores (claves) de las propiedades de los objetos.

> Es importante tener en cuenta que el valor `null` no es lo mismo que `undefined`. `null` se utiliza para representar un valor inexistente intencionalmente, mientras que `undefined` se utiliza para representar

> una variable que no ha sido inicializada o una propiedad de un objeto que no existe.

7. Closures

Un *closure* es una función que se define dentro de otra función y que tiene acceso a las variables y parámetros de la función padre, incluso después de que la función padre haya terminado de ejecutarse.

```
function makeCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  }
}

let counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

```
function coolCounter(num, addition) {
  function add() {
    return num += addition;
  }
  return add;
}

let counter = coolCounter(4, 2);
console.log(counter()) // 6
console.log(counter()) // 8
console.log(counter()) // 10
```

8. Array: métodos

Los arrays de JavaScript tienen varios métodos integrados que te permiten manipular los elementos del array. Aquí hay algunos de los métodos de array más comúnmente utilizados:

- `.push()`: agrega uno o más elementos al final de un array y devuelve la nueva longitud del array
- `.pop()`: elimina el último elemento de un array y devuelve ese elemento
- `.shift()`: elimina el primer elemento de un array y devuelve ese elemento
- `.unshift()`: agrega uno o más elementos al principio de un array y devuelve la nueva longitud del array
- `.slice()`: devuelve un nuevo array que incluye una porción especificada de un array
- `.splice()`: agrega o elimina elementos de un array y devuelve los elementos eliminados
- `.sort()`: ordena los elementos de un array in situ y devuelve el array ordenado
- `.reverse()`: invierte el orden de los elementos de un array in situ y devuelve el array invertido
- `.concat()`: devuelve un nuevo array que combina los elementos de dos o más arrays

- `.join()`: devuelve una cadena que es el resultado de concatenar todos los elementos de un array, separados por una cadena separadora especificada

```
let fruits = ['apple', 'banana', 'orange'];

// Agregar un nuevo elemento al final del array
fruits.push('mango'); // ['apple', 'banana', 'orange', 'mango']

// Eliminar el último elemento del array
let removed = fruits.pop(); // removed = 'mango', fruits = ['apple', 'banana',
'orange']

// Agregar un nuevo elemento al principio del array
fruits.unshift('strawberry'); // ['strawberry', 'apple', 'banana', 'orange']

// Eliminar el primer elemento del array
removed = fruits.shift(); // removed = 'strawberry', fruits = ['apple', 'banana',
'orange']

// Ordenar los elementos del array
fruits.sort(); // ['apple', 'banana', 'orange']

// Invertir el orden de los elementos del array
fruits.reverse(); // ['orange', 'banana', 'apple']
```

9. Arrays: funciones de orden superior

Los arrays de JavaScript tienen varios métodos integrados que te permiten realizar operaciones en los elementos del array y devolver un nuevo array. Estos métodos se llaman funciones de array o funciones de orden superior. Aquí hay algunos ejemplos de funciones de array comunes:

- `.map()`: crea un nuevo array aplicando una función a cada elemento del array original
- `.filter()`: crea un nuevo array con elementos que pasan una prueba especificada por una función
- `.reduce()`: reduce los elementos de un array a un único valor aplicando una función a cada elemento sucesivamente
- `.forEach()`: ejecuta una función proporcionada una vez por cada elemento del array
- `.every()`: verifica si todos los elementos del array pasan una prueba especificada por una función y devuelve un valor booleano
- `.find()`: devuelve el primer elemento del array que pasa una prueba especificada por una función, o undefined si ningún elemento pasa la prueba
- `.some()`: verifica si al menos un elemento del array pasa una prueba especificada por una función y devuelve un valor booleano

```
let numbers = [1, 2, 3, 4, 5];

// Crear un nuevo array con los cuadrados de los números
let squares = numbers.map(n => n * n); // squares = [1, 4, 9, 16, 25]
```

```
// Crear un nuevo array con solo los números pares
let evens = numbers.filter(n => n % 2 == 0); // evens = [2, 4]

// Reducir el array a la suma de sus elementos
let sum = numbers.reduce((acc, n) => acc + n, 0); // sum = 15

// Ejecutar una función para cada elemento del array
numbers.forEach(n => console.log(n)); // muestra 1, 2, 3, 4, 5 en la consola

// Comprobar si todos los elementos del array son mayores que 0
let allPositive = numbers.every(n => n > 0); // allPositive = true

// Encontrar el primer elemento del array que es divisible por 2
let firstEven = numbers.find(n => n % 2 == 0); // firstEven = 2

// Comprobar si algún elemento del array es divisible por 3
let anyMultipleOf3 = numbers.some(n => n % 3 == 0); // anyMultipleOf3 = true
```

10. Callbacks

Un callback en JavaScript es una función que se pasa a otra función como un argumento, y se ejecuta después de que la función principal ha terminado.

```
function miFuncion(callback) {
  // código de la función principal
  console.log("La función ha terminado");

  // ejecuta el callback
  callback();
}

// llama a la función y pasa una función anónima como callback
miFuncion(function() {
  console.log("Este mensaje se mostrará después de que la función principal haya
terminado");
});
```

Los callbacks son útiles porque permiten que las funciones se ejecuten en el momento adecuado, en lugar de tener que esperar a que una función termine antes de ejecutar otra.

11. Asincronismo

El asincronismo se refiere a que no hará todas las tareas al mismo tiempo, es decir, que se harán varias tareas en paralelo o que aunque vayan de forma secuencial no debes esperar para poder seguir haciendo algo más.

Código síncrono (Bloqueante): Python

```
import requests

res = requests.get('https://api.com/endpoint')
print(res.text)
print('Impreso después del request')
```

Código asíncrono (No Bloqueante): JavaScript

```
const request = require('request');

request('https://api.com/endpoint', (err, res, body) => {
  console.log(body);
});

console.log('Impreso después del request');
```

12. Promesas

- Las promesas son una forma de manejar el resultado de operaciones asíncronas.
- Una promesa representa el resultado de una operación que aún no ha terminado.
- Te permiten hacer otras cosas mientras esperas a que una tarea se complete, y te avisan cuando la tarea esté lista para que puedas manejar el resultado de forma adecuada.

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    // realizamos una llamada a una API externa
    fetch('https://api.com/endpoint')
      .then(response => response.json())
      .then(data => resolve(data))
      .catch(error => reject(error));
  });
}

// utilizamos la promesa
fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

13. async-await

`async` y `await` son palabras clave que se utilizan para trabajar con funciones asíncronas. La palabra clave `async` se utiliza para definir una función asíncrona. Se coloca antes de la declaración de la función, como en el siguiente ejemplo:

```
async function myAsyncFunction() {
  // código de la función
}

const anotherAsyncFunction = async () => {
  // código de la función
}
```

La palabra clave `await` se utiliza dentro de una función asíncrona para esperar a que una promesa se complete. Por ejemplo:

```
async function myAsyncFunction() {
  const result = await someAsyncOperation();
  // continuar con la ejecución de la función utilizando el resultado de la
operación asíncrona
}
```

Cuando se utiliza `await`, la ejecución de la función se detiene hasta que la promesa se resuelve. Una vez que se resuelve la promesa, el valor de esta se asigna a la variable `result` y la ejecución de la función continúa después de la línea donde se utiliza `await`.

> Es importante tener en cuenta que `await` sólo se puede utilizar dentro de una función asíncrona. Si se intenta utilizar fuera de una función asíncrona, se obtendrá un error.

14. Event Loop

El event loop es un mecanismo que controla la ejecución de tareas y eventos en JavaScript. Es como una especie de coordinador que se encarga de hacer que todo funcione de manera adecuada en el lenguaje.

Cuando necesitas realizar una tarea en JavaScript, es posible que tenga que esperar un tiempo para que se complete. Por ejemplo, si quieres cargar datos de un servidor, puede llevar algún tiempo. En lugar de tener que esperar a que la tarea se complete, el event loop se encarga de hacer que la tarea se realice en segundo plano mientras haces otras cosas. Así, puedes seguir trabajando mientras se completa la tarea.

15. this

En JavaScript, el keyword `this` se refiere al objeto que es propietario del método en el que se está utilizando. Puede ser útil para acceder a las propiedades y métodos de ese objeto de manera más concisa.

```
const name = 'Peter'

const obj = {
  name: 'Tony',
  getName: function() {
    return this.name // hace referencia a la propiedad 'name' del objeto y no a la
variable global
  },
```

```
}

console.log(obj.getName()) // 'Tony'
```

16. bind, call y apply

16.1 bind

Este método crea una nueva función con el mismo código que la función original, pero con el valor de `this` establecido de manera explícita. La nueva función se puede llamar más tarde y el valor de `this` se mantendrá fijo, independientemente del contexto en el que se llame. Por ejemplo:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    return this.name;
  }
}

function greeting() {
  console.log('Hello, my name is ' + this.getName())
}

// Se crea una instancia de la clase Person con el nombre "John"
const person = new Person('John');

// Se utiliza el método "bind" de la función "greeting" para vincular el objeto
"person" como el valor de "this" en la función
// El método "bind" devuelve una nueva función con el objeto "person" vinculado
como el valor de "this"
const sayHi = greeting.bind(person);

// Se llama a la función "sayHi", que es la función "greeting" con el objeto
"person" vinculado como el valor de "this"
sayHi();
```

16.2. call

Este método **llama** a una función y establece el valor de `this` de manera explícita. Además, permite pasar parámetros a la función de manera dinámica. Por ejemplo:

```
class Person {
  constructor(name) {
    this.name = name;
  }
```

```
  getName() {
    return this.name;
  }
}

function greeting(text) {
  console.log(text + this.getName())
}

const person = new Person('John');

greeting.call(person, 'Hello, my name is ') // 'Hello, my name is John'
```

16.3. apply

Este método funciona de manera similar a `call`, pero en lugar de pasar los parámetros a la función como argumentos separados, se pasa un array con los parámetros. Por ejemplo:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    return this.name;
  }
}

function greeting(text) {
  console.log(text + this.getName())
}

greetingA.apply(person, ['Hi, my name is ']) // 'Hello, my name is John'
```