

Programación asíncrona

1 Permite iniciar una tarea de una duración prolongada , mientras se ejecuta otras tareas sin interponerse o detenerlas , el programa terminará cuando se haya terminado la tarea principal

un ejemplo son las funciones ejecutadas por los mismos navegadores , debido a que toman tiempo en ejecutarse son asíncronas para garantizar que la página funcione correctamente

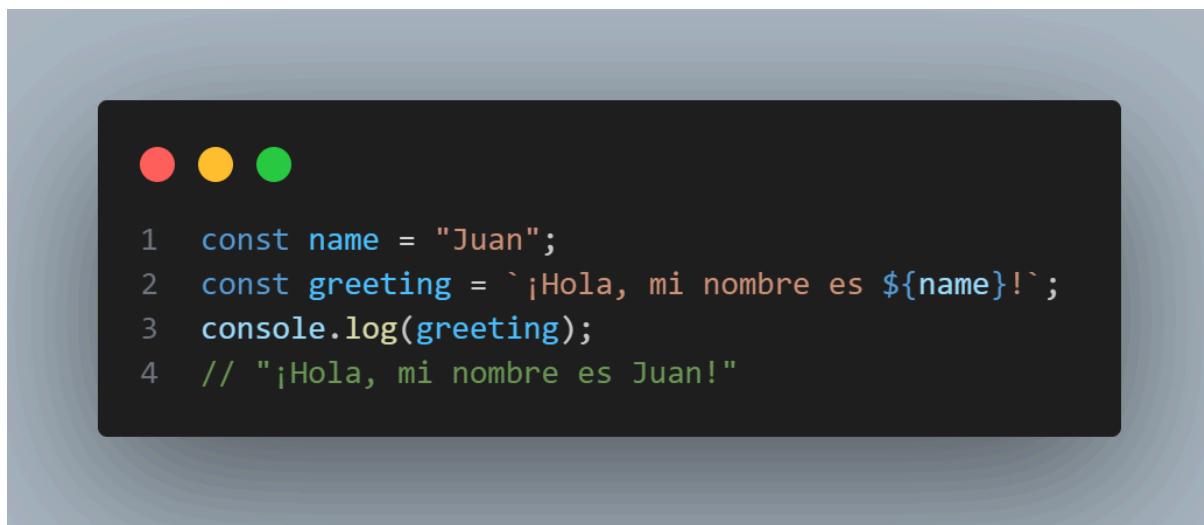
Ejemplos

Realizar peticiones HTTP utilizando **fetch()**

Acceder a la cámara o micrófono de un usuario mediante **getUserMedia()**

Pedir a un usuario que seleccione archivos usando **showOpen FilePicker()** (en-US)

Ejemplo de una función Síncrona



The screenshot shows a dark-themed code editor window. At the top left are three circular icons: red, yellow, and green. Below them is a code block with the following content:

```
1 const name = "Juan";
2 const greeting = `¡Hola, mi nombre es ${name}!`;
3 console.log(greeting);
4 // "¡Hola, mi nombre es Juan!"
```

```
const name = "Juan";
const greeting = `¡Hola, mi nombre es ${name}!`;
console.log(greeting);
// "¡Hola, mi nombre es Juan!"
```

Declara una cadena (**string**) con el nombre (**name**).

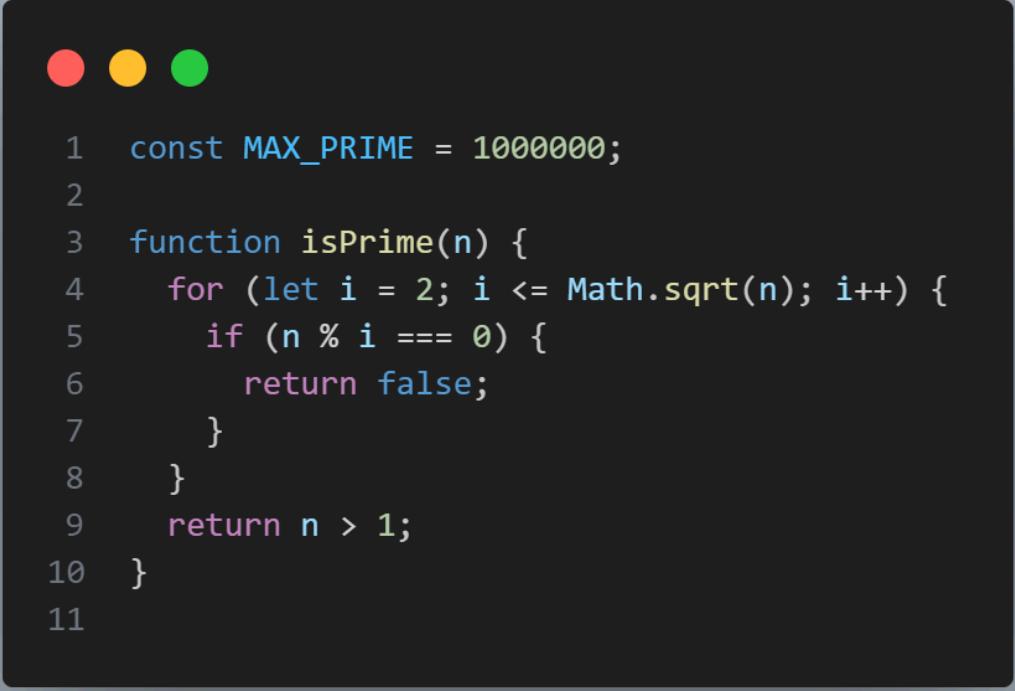
Declara otra cadena con el nombre (**greeting**), que utiliza (**name**).

Envía a la consola JavaScript el saludo.

En este caso, **makeGreeting** es una función sincrona porque quién la llama (**greeting**) tiene que esperar a que la función termine su trabajo y devuelva un valor antes poder continuar.

No genera muchos problemas la función sincrónica al ser sencilla , el efecto se aprecia en programas más complejos como:

Ejemplo:



```
1 const MAX_PRIME = 1000000;
2
3 function isPrime(n) {
4     for (let i = 2; i <= Math.sqrt(n); i++) {
5         if (n % i === 0) {
6             return false;
7         }
8     }
9     return n > 1;
10}
11
```

```
const MAX_PRIME = 1000000;
```

```
function isPrime(n) {
```

```

for (let i = 2; i <= Math.sqrt(n); i++) {
  if (n % i === 0) {
    return false;
  }
}
return n > 1;
}

```



```

1 const random = (max) => Math.floor(Math.random() * max);
2
3 function generatePrimes(quota) {
4   const primes = [];
5   while (primes.length < quota) {
6     const candidate = random(MAX_PRIME);
7     if (isPrime(candidate)) {
8       primes.push(candidate);
9     }
10   }
11   return primes;
12 }
13
14 const quota = document.querySelector("#quota");
15 const output = document.querySelector("#output");
16
17 document.querySelector("#generate").addEventListener("click", () => {
18   const primes = generatePrimes(quota.value);
19   output.textContent = `¡Finalizado! se han generado ${quota.value} números primos`;
20 });
21
22 document.querySelector("#reload").addEventListener("click", () => {
23   document.location.reload();
24 });

```

```
const random = (max) => Math.floor(Math.random() * max);
```

```

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {

```

```

const candidate = random(MAX_PRIME);
if (isPrime(candidate)) {
    primes.push(candidate);
}
}

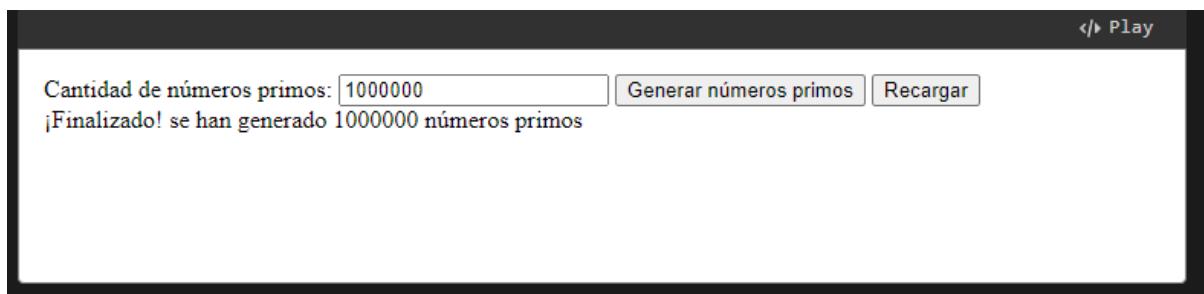
return primes;
}

const quota = document.querySelector("#quota");
const output = document.querySelector("#output");

document.querySelector("#generate").addEventListener("click", () => {
    const primes = generatePrimes(quota.value);
    output.textContent = `¡Finalizado! se han generado ${quota.value} números primos`;
});

document.querySelector("#reload").addEventListener("click", () => {
    document.location.reload();
});

```



Este programa genera una cantidad determinada de números primos e indica con un mensaje cuando esta acción haya finalizado

En dicha función a la hora de iniciar otra tarea mientras esta se ejecutada , no será posible debido a que se tendrá que esperar que la primera finalice

Ejemplo:

The screenshot shows a web page with a dark header bar containing a play button icon and the word "Play". Below the header is a form with a text input field labeled "Cantidad de números primos:" containing the value "1000000", a blue "Generar números primos" button, and a grey "Recargar" button. A large text area below the buttons contains the instruction: "Prueba a escribir algo aquí inmediatamente después de presionar el botón 'Generar números primos'".

Este es el problema básico de las funciones síncronas de larga duración. Por lo que se necesita buscar una forma de que nuestro programa:

- 1 Inicie una operación de larga duración llamando a una función.
- 2 Haga que esa función inicie la operación y regrese inmediatamente, de manera que nuestro programa pueda seguir respondiendo a otros eventos.
- 3 Notifique el resultado de la operación cuando se complete.

La importancia de los manejadores de eventos

Los manejadores de eventos son en realidad una forma de programación asíncrona debido a la naturaleza del lenguaje y del entorno en el que se ejecuta es decir :

Imagina que estás en una fila esperando tu turno para recibir un paquete. No sabes cuándo exactamente te tocará, pero estás atento. Cuando finalmente llega tu turno y recibes el paquete, eso sería como un evento en JavaScript.

En JavaScript, puedes hacer algo similar con funciones asíncronas. Digamos que tienes una tarea que llevará tiempo, como descargar un archivo grande de Internet. En lugar de esperar pasivamente a que se complete la descarga, puedes decirle a JavaScript: "Cuando la descarga termine, quiero hacer algo específico".

El siguiente ejemplo muestra esto en acción. Pulsamos "Pulse para iniciar la solicitud" para enviar una petición. Creamos un nuevo **XMLHttpRequest** y escuchamos su evento **loadend**. El manejador registra un mensaje "¡Finalizado!" junto con el código de estado.



```
1 const log = document.querySelector(".event-log");
2
3 document.querySelector("#xhr").addEventListener("click", () => {
4   log.textContent = "";
5
6   const xhr = new XMLHttpRequest();
7
8   xhr.addEventListener("loadend", () => {
9     log.textContent = `${log.textContent}Finalizado con el estado: ${xhr.status}`;
10    });
11
12   xhr.open(
13     "GET",
14     "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",
15   );
16   xhr.send();
17   log.textContent = `${log.textContent}Inicio de la solicitud XHR\n`;
18 });
19
20
21 document.querySelector("#reload").addEventListener("click", () => {
22   log.textContent = "";
23   document.location.reload();
24 });
```

```
const log = document.querySelector(".event-log");

document.querySelector("#xhr").addEventListener("click", () => {
  log.textContent = "";

  const xhr = new XMLHttpRequest();

  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}Finalizado con el estado: ${xhr.status}`;
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",
  );
  xhr.send();
  log.textContent = `${log.textContent}Inicio de la solicitud XHR\n`;
});

document.querySelector("#reload").addEventListener("click", () => {
  log.textContent = "";
```

```
document.location.reload();
});
```



este código muestra cómo manejar eventos en JavaScript en respuesta a acciones del usuario, como hacer clic en botones, y cómo trabajar con operaciones asíncronas, como hacer solicitudes HTTP, utilizando el objeto XMLHttpRequest

Set timeout , Set Interval

El método **setTimeout** permite ejecutar un fragmento de código, una vez transcurrido un tiempo determinado o mejor dicho para ejecutar un código o programa de JavaScript en un momento determinado.

Por ejemplo, el código siguiente imprimirá "Hola Mundo" en la consola de JavaScript después de que hayan pasado 2 segundos:



```
1  setTimeout(function () {  
2      console.log("Hola Mundo");  
3  }, 2000);  
4  
5  console.log("setTimeout() Ejemplo...");
```

```
setTimeout(function () {  
    console.log("Hola Mundo");  
}, 2000);  
  
console.log("setTimeout() Ejemplo...");
```

La sintaxis del método setTimeout() es la siguiente :

```
setTimeout(function, milisegundos, parametro, parametro, ...);
```

también puedes pasar parámetros adicionales al método **setTimeout()** que puede utilizar dentro de la función de la siguiente manera:

```
function saludos(nombre, rol){  
    console.log(`Hola, mi nombre es ${nombre}`);  
    console.log(`Yo soy ${rol}`);  
}  
  
setTimeout(saludos, 3000, "Juan", "Programador");
```

Cómo cerrar el método setTimeout

También puede evitar que el método **setTimeout()** ejecute la función utilizando el método **clearTimeout()**.

El método **clearTimeout()** requiere el id devuelto por **setTimeout()** para saber qué método **setTimeout()** debe cerrar:

```
clearTimeout(id);
```

Ejemplo de uso

```
const timeoutId = setTimeout(function() {
    console.log("Hola Mundo");
}, 2000);

clearTimeout(timeoutId);
console.log(`Tiempo de espera ID ${timeoutId} ha sido limpiado`);
```

Nota:

Si tienes varios métodos **setTimeout()**, entonces necesitas guardar los IDs devueltos por cada llamada al método y luego llamar al método **clearTimeout()** tantas veces como sea necesario para borrarlos todos.

Resumen

El método **setTimeout()** de JavaScript es un método incorporado que permite temporizar la ejecución de una determinada función. Es necesario pasar la cantidad de tiempo a esperar en **milisegundos**, lo que significa que para esperar un segundo, es necesario pasar mil **milisegundos**.

Setinterval

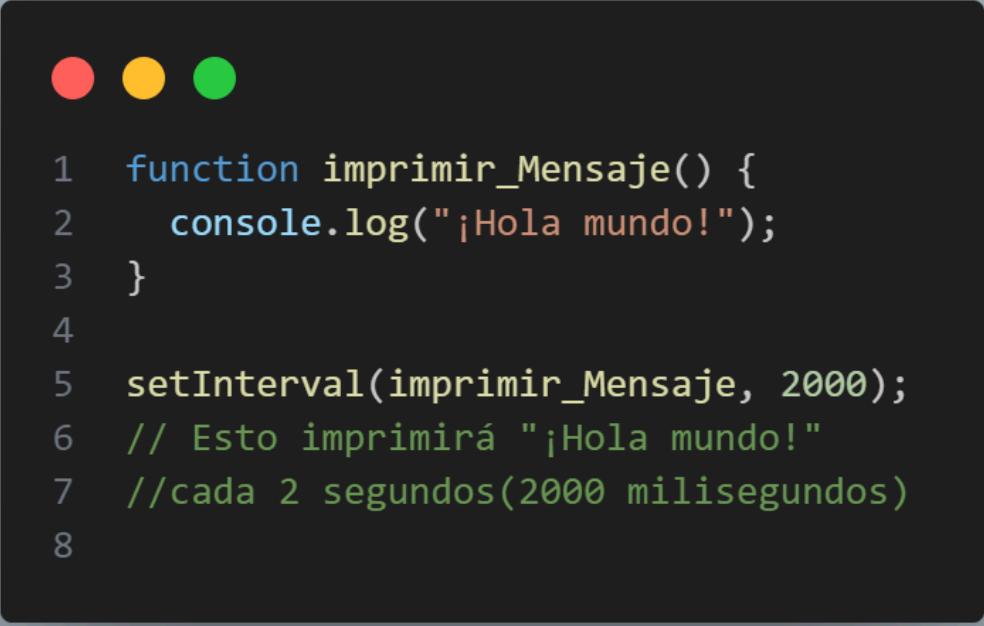
setInterval() para hacer que una función se repita con un tiempo de retraso entre cada ejecución es decir permite ejecutar un bloque de código repetidamente con un intervalo de tiempo específico entre cada ejecución.

Sintaxis :**setInterval(función, tiempo);**

Función a ejecutar: Debes proporcionar el nombre de la función que deseas ejecutar repetidamente. Esta función puede ser definida previamente o una función anónima (sin nombre).

Tiempo entre ejecuciones: Indicas el intervalo de tiempo en milisegundos (1 segundo = 1000 milisegundos). Es decir, cuánto tiempo quieras esperar antes de que se ejecute la función nuevamente.

Ejemplo



```
1 function imprimir_Mensaje() {  
2     console.log("¡Hola mundo!");  
3 }  
4  
5 setInterval(imprimir_Mensaje, 2000);  
6 // Esto imprimirá "¡Hola mundo!"  
7 //cada 2 segundos(2000 milisegundos)  
8
```

```
function imprimir_Mensaje() {  
    console.log("¡Hola mundo!");  
}
```

```
setInterval(imprimir_Mensaje, 2000); // Esto imprimirá "¡Hola mundo!" cada 2 segundos (2000 milisegundos)
```

Detener el intervalo: Puedes detener el intervalo usando la función **clearInterval**. Esta función requiere el identificador devuelto por **setInterval**.

```
let intervalo = setInterval(imprimirMensaje, 2000);  
clearInterval(intervalo); // Esto detiene la ejecución del intervalo
```

Ejemplo

```
let identificadorIntervaloDeTiempo;

function repetirCadaSegundo() {
    identificadorIntervaloDeTiempo = setInterval(mandarMensaje, 1000);
}

function mandarMensaje() {
    console.log("Ha pasado 1 segundo.");
}
```

Cuando tu código llama a la función **repetirCadaSegundo**, ejecutará **setInterval**. setInterval correrá la función **mandarMensaje** cada segundo (1000 ms).

Event loop(bucle de eventos)

El event loop en JavaScript es como un organizador de tareas que se encarga de manejar las acciones que ocurren de manera asíncrona, es decir, aquellas que no suceden en un orden predecible o inmediato.

Ejemplo:

Imagina que estás en una tienda y hay una fila (cola) de personas esperando para ser atendidas en la caja. Cada vez que alguien necesita ser atendido, se agrega al final de la fila y el cajero atiende a la persona que está al principio de la fila. El event loop funciona de manera similar, asegurándose de que las tareas se ejecuten en el orden correcto.

Cuando ocurre un evento o se llama a una función en JavaScript, ese trabajo se agrega a la cola del event loop.

El event loop se encarga de tomar cada tarea de la cola y ejecutarla, una por una. Si una tarea lleva mucho tiempo o está esperando algo externo (como cargar una imagen desde Internet), el event loop seguirá avanzando y atendiendo otras tareas mientras espera.

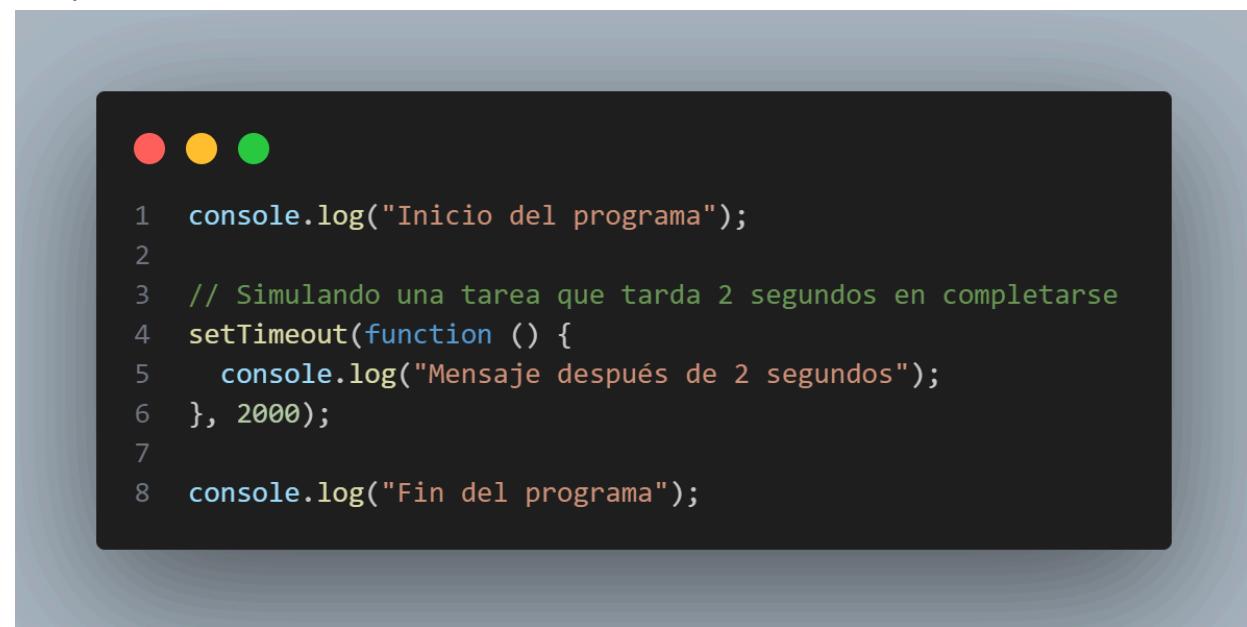
Es importante entender que JavaScript es un lenguaje de programación de un solo hilo, lo que significa que solo puede hacer una cosa a la vez.

Por eso es necesario que las funciones se ejecuten completamente antes de pasar a la siguiente tarea en la cola.

Esto garantiza que los datos estén en un estado consistente y que el código se ejecute de manera ordenada, incluso cuando se manejan múltiples tareas de manera simultánea.

Ejemplo:

Imagina que estás construyendo una aplicación web que muestra un mensaje después de un cierto tiempo de espera. En este caso, utilizaremos la función **setTimeout()** de JavaScript, que programará una tarea para ejecutarse después de un cierto período de tiempo.



```
● ● ●
1  console.log("Inicio del programa");
2
3  // Simulando una tarea que tarda 2 segundos en completarse
4  setTimeout(function () {
5      console.log("Mensaje después de 2 segundos");
6  }, 2000);
7
8  console.log("Fin del programa");
```

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

```
// Simulando una tarea que tarda 2 segundos en completarse
setTimeout(function() {
    console.log("Mensaje después de 2 segundos");
}, 2000);
```

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

Cuando ejecutas este código, verás que el mensaje "Inicio del programa" se muestra primero, seguido por "Fin del programa". Esto ocurre porque estas líneas de código se ejecutan de manera sincrónica, una después de la otra, en el orden en que están escritas.

Sin embargo, el mensaje "Mensaje después de 2 segundos" no se muestra inmediatamente después de "Fin del programa". Esto se debe a que `setTimeout()` programa una tarea para ejecutarse después de un retraso de 2 segundos, pero mientras tanto, el event loop sigue funcionando y ejecutando otras tareas.

Entonces, aunque la tarea de `setTimeout()` está lista para ejecutarse después de 2 segundos, el event loop aún debe esperar hasta que se complete el tiempo de espera antes de tomar esa tarea y ejecutarla. Esto es lo que permite que otras partes del código se sigan ejecutando mientras se espera que se complete el retraso.

Callbacks

Los callbacks (a veces denominados funciones de retrollamada o funciones callback) no son más que un tipo de funciones que se pasan por parámetro a otras funciones. El objetivo de esto es tener una forma más legible de escribir funciones, más cómoda y flexible para reutilizarlas.

Las funciones callback son solo funciones que se pasan como datos a otras funciones. Cuando usas una función como callback, los parámetros que se le pasan a esa función tienen un significado especial dentro de la función que la llama.

Nota : actualmente, controlar la asincronía únicamente mediante callbacks puede ser una práctica poco recomendable. Es preferible utilizar promesas, que generalmente es más adecuado.

Ejemplo:

Imaginemos el siguiente bucle tradicional para recorrer un : **Array**



```
1 const list = ["A", "B", "C"];
2
3 for (let i = 0; i < list.length; i++) {
4   console.log("i=", i, " list=", list[i]);
5 }
6
```

```
const list = ["A", "B", "C"];
```

```
for (let i = 0; i < list.length; i++) {
  console.log("i=", i, " list=", list[i]);
}
```

En la variable `i` tenemos la posición del array que estamos recorriendo. Este valor irá desde 0 hasta 2, mientras que con `list[i]` accedemos a la posición del array para obtener el elemento, es decir, desde A hasta C.

Ahora veamos, cómo podemos hacer este mismo bucle utilizando el método `forEach()`

```
1 const list = ["A", "B", "C"];
2
3 function action(element, index) {
4     console.log("i=", index, "list=", element);
5 }
6
7 list.forEach(action);
```

```
const list = ["A", "B", "C"];

function action(element, index) {
    console.log("i=", index, "list=", element);
}

list.forEach(action);
```

Esto se suele reescribir de la siguiente forma:

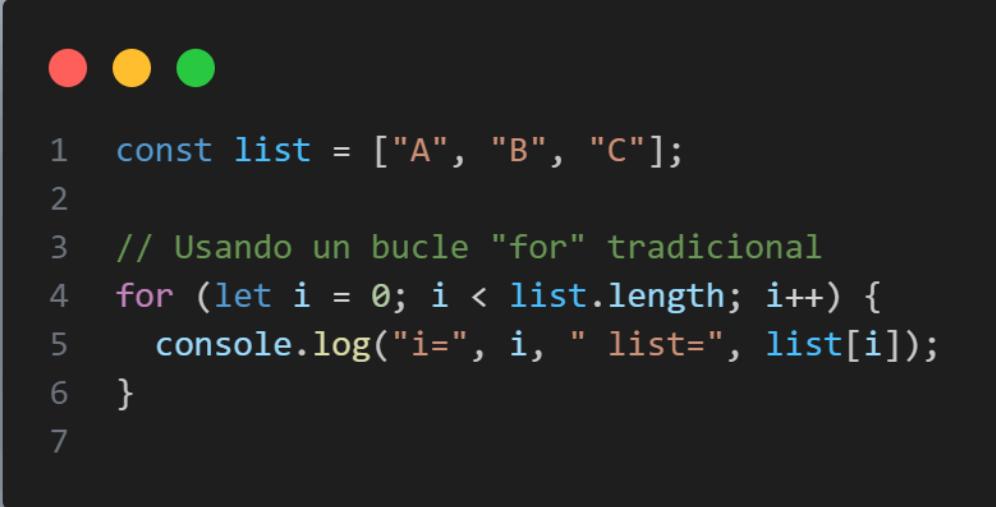
```
1 list.forEach((element, index) => {
2     console.log("i=", index, "list=", element)
3 });
4
```

```
list.forEach((element, index) => {
    console.log("i=", index, "list=", element)
});
```

Lo importante de este ejemplo es que se vea que la función callback que le hemos pasado por parámetro a `forEach()` se va a ejecutar por cada uno de los elementos del array, y en cada iteración de dicha función callback, los parámetros `element` e `index` van a tener un valor especial:

Ejemplo 2:

Imagina que tienes una lista de compras que quieras revisar uno por uno. Usando un bucle "for" tradicional, recorres la lista desde el primero hasta el último, utilizando un número para indicar la posición de cada artículo en la lista y accediendo a cada artículo usando esa posición.



```
1 const list = ["A", "B", "C"];
2
3 // Usando un bucle "for" tradicional
4 for (let i = 0; i < list.length; i++) {
5   console.log("i=", i, " list=", list[i]);
6 }
7
```

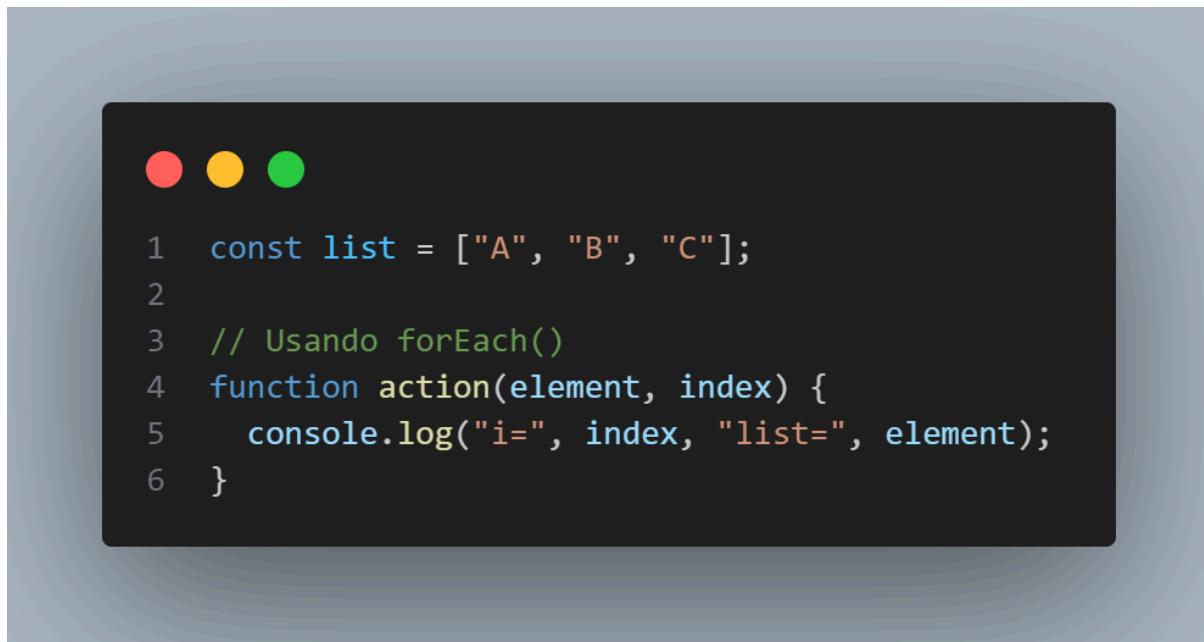
```
const list = ["A", "B", "C"];
```

```
// Usando un bucle "for" tradicional
for (let i = 0; i < list.length; i++) {
  console.log("i=", i, " list=", list[i]);
}
```

En el ejemplo, con el "for" tradicional, la variable "i" es como una etiqueta que indica en qué posición estás dentro de la lista, y "list[i]" es el artículo específico que estás revisando.

Luego, ves cómo hacer lo mismo usando el método "forEach()" en JavaScript.

Con "forEach()", puedes pasar una función que se ejecutará para cada elemento de la lista. Dentro de esta función, los parámetros "element" e "index" tienen un significado especial: "element" representa cada artículo de la lista y "index" representa su posición.



```
const list = ["A", "B", "C"];
```

```
// Usando forEach()
function action(element, index) {
  console.log("i=", index, "list=", element);
}

list.forEach(action);
```

También puedes usar una función de flecha en lugar de definir una función por separado:

```
const list = ["A", "B", "C"];
```



```
1 // Usando forEach() con una función de flecha
2 list.forEach((element, index) => {
3   console.log("i=", index, "list=", element)
4 });
5
6
```

```
// Usando forEach() con una función de flecha
list.forEach((element, index) => {
  console.log("i=", index, "list=", element)
});
```

Así que, en resumen, "forEach()" te permite recorrer la lista de una manera más elegante y clara, donde cada elemento se procesa utilizando una función que específicas, en lugar de tener que gestionar manualmente el índice y acceder a los elementos uno por uno.

Función de orden superior

- Toma una o más funciones como argumentos (callbacks).
- Devuelve una función como resultado.

Una función de orden superior es aquella que puede recibir otras funciones como argumentos o devolver una función como resultado. Esto permite crear funciones más flexibles y modulares, lo que es fundamental en la programación funcional.

Imagínate que tienes una función operar, y quieres que esa función realice una operación matemática específica, pero no quieres que la función operar tenga que saber cómo hacer esa operación en particular. En lugar de eso, le pasas a operar la función que realiza la operación específica como un argumento. Por ejemplo:

```

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

function restar(a, b) {
    return a - b;
}

function operar(a, b, operacion) {
    return operacion(a, b);
}

console.log(operar(5, 3, sumar)); // Salida: 8
console.log(operar(5, 3, restar)); // Salida: 2

```

En este ejemplo, operar es una función de orden superior porque recibe la función sumar o restar como argumento y la usa para realizar la operación deseada.

Esto significa que una función de orden superior puede recibir funciones como entrada o producir funciones como salida. Esta característica es fundamental en la programación funcional y proporciona flexibilidad y modularidad en el diseño del código.

```

function ejecutarOperacion(numero, operacion) {
    return operacion(numero);
}

function duplicar(num) {
    return num * 2;
}

function triplicar(num) {
    return num * 3;
}

console.log(ejecutarOperacion(5, duplicar)); // Salida: 10
console.log(ejecutarOperacion(5, triplicar)); // Salida: 15

```

Promesas

Promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:

- a promesa se cumple (**promesa resuelta**)
- La promesa no se cumple (**promesa rechazada**)
- La promesa se queda en un estado incierto indefinidamente (**promesa pendiente**)

Ejemplo:

Imagina que tienes una tarea por hacer, pero no sabes cuánto tiempo llevará completarla. Podría ser rápido o podría llevar más tiempo de lo esperado. Entonces, haces una "promesa" a ti mismo de que la completarás y le das a alguien un trozo de papel con tu promesa escrita.

Ahora, esa promesa puede estar en tres estados:

Pendiente: Significa que todavía no has completado la tarea, estás en proceso.

Aceptada: Significa que has cumplido tu promesa, has completado la tarea con éxito.

Rechazada: Significa que no pudiste cumplir tu promesa, algo salió mal mientras intentabas completar la tarea.

Ahora, en JavaScript, cuando trabajas con promesas, tienes métodos para manejar estos estados:

- **.Then(resolve):** Este método se ejecuta cuando tu promesa se cumple (es decir, cuando se completa con éxito). Aquí puedes especificar qué hacer después de que se cumpla la promesa.
- **.Catch(reject):** Este método se ejecuta cuando tu promesa es rechazada (es decir, algo sale mal durante la ejecución). Aquí puedes manejar el error y tomar medidas adecuadas.

- **.Then(resolve, reject)**: Este método es una forma abreviada de manejar tanto el caso de resolución como el de rechazo en un solo .then().
- **.Finally(end)**: Este método se ejecuta tanto si la promesa se cumple como si se rechaza. Es útil para realizar tareas que deben ocurrir independientemente del resultado de la promesa, como limpiar o cerrar recursos.

Consumir Promesa

Imagina que tienes una tarea que lleva tiempo, pero no sabes exactamente cuánto.

En JavaScript, puedes usar promesas para manejar estas tareas.

Digamos que tienes una tarea que involucra obtener datos de un servidor. Aquí es donde entra en juego fetch("/robots.txt").

Esta línea de código envía una solicitud al servidor para obtener un archivo llamado "robots.txt".

Pero cómo puede llevar tiempo obtener una respuesta del servidor, fetch() devuelve una promesa. Esto significa que promete darte la respuesta cuando esté disponible, pero no sabes exactamente cuándo será.

Ahora, puedes usar **.then()** para decirle a JavaScript qué hacer cuando la promesa se cumpla, es decir, cuando obtengas la respuesta del servidor. Por ejemplo:

```
1  fetch("/robots.txt").then(function (response) {  
2      /* Aquí puedes hacer algo con la respuesta */  
3  });  
4
```

```
fetch("/robots.txt").then(function(response) {  
  /* Aquí puedes hacer algo con la respuesta */  
});
```

Puedes encadenar múltiples .then() si necesitas realizar más acciones después de que se cumpla la promesa. Y si algo sale mal y la promesa se rechaza, puedes usar .catch() para manejar ese error:



Y si necesitas hacer algo independientemente de si la promesa se cumple o se rechaza, puedes usar .finally():

```
fetch("/robots.txt")  
.then(function(response) {  
  /* Aquí puedes hacer algo con la respuesta */  
})  
.catch(function(error) {  
  /* Aquí manejas el error */  
})  
.finally(function() {
```

```
/* Esto se ejecuta independientemente de si la promesa se cumple o se rechaza */  
});
```



```
1  fetch("/robots.txt")  
2    .then(function (response) {  
3      /* Aquí puedes hacer algo con la respuesta */  
4    })  
5    .catch(function (error) {  
6      /* Aquí manejas el error */  
7    })  
8    .finally(function () {  
9      /* Esto se ejecuta independientemente de  
10        si la promesa se cumple o se rechaza */  
11    });  
12
```

Crear promesa

Ahora, ¿qué pasa si quieres crear tu propia promesa? Imagina que tienes una tarea que quieres encapsular en una promesa. Por ejemplo, lanzar un dado y obtener el resultado. Podrías hacerlo así:

En este código, lanzarDado() es una función que devuelve una promesa. La promesa se cumple (resolve) si no sacas un 6 y se rechaza (reject) si sacas un 6.

```
● ● ●

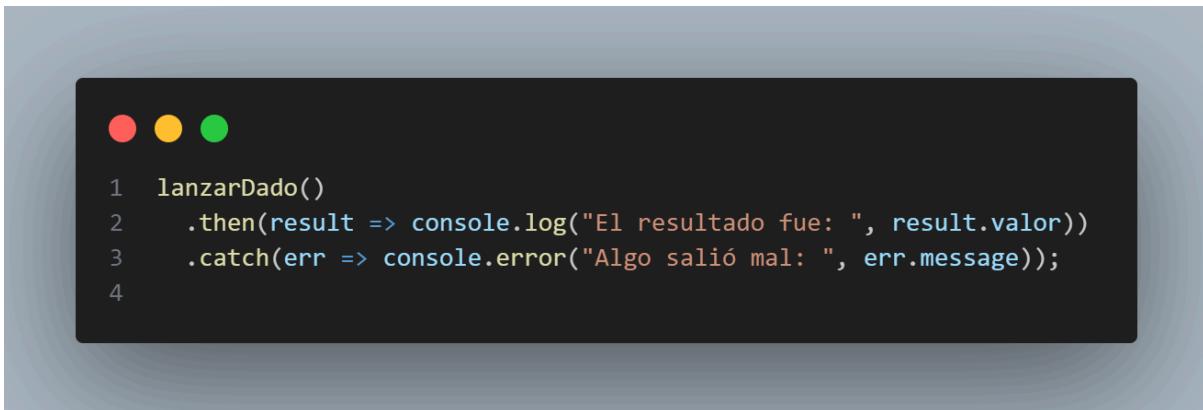
1 const lanzarDado = () => {
2   return new Promise((resolve, reject) => {
3     const resultado = 1 + Math.floor(Math.random() * 6);
4     if (resultado === 6) {
5       reject({
6         error: true,
7         message: "¡Sacaste un 6!"
8       });
9     } else {
10       resolve({
11         error: false,
12         valor: resultado
13       });
14     }
15   });
16 }
```

```
const lanzarDado = () => {
  return new Promise((resolve, reject) => {
    const resultado = 1 + Math.floor(Math.random() * 6);
    if (resultado === 6) {
      reject({
        error: true,
        message: "¡Sacaste un 6!"
      });
    } else {
      resolve({
        error: false,
        valor: resultado
      });
    }
  })
}
```

```
});  
};
```

Luego, puedes consumir esta promesa así:

```
lanzarDado()  
.then(result => console.log("El resultado fue: ", result.valor))  
.catch(err => console.error("Algo salió mal: ", err.message));
```



En resumen puedes imaginar casos más complejos donde necesites hacer tareas asíncronas y manejar sus resultados de manera ordenada y eficiente. Las promesas en JavaScript te permiten hacer exactamente eso.

Funciones Async Funciones Await Funciones asíncronas Async/await

Imagina que estás haciendo una tarea que tiene dos partes: una que lleva tiempo y otra que se puede hacer mientras esperas a que termine la primera parte. Por ejemplo, cocinar pasta. Mientras el agua hierva, puedes preparar la salsa.

Ahora, en la programación, a veces tienes tareas que toman tiempo, como cargar una imagen de internet. Para no bloquear el programa mientras esperas que se cargue la imagen, usas algo llamado "promesas". Es como decir: "Te prometo que te daré la imagen una vez que la haya cargado".

Antes, para manejar estas promesas, usábamos `.then()`. Sería como decir: "Cuando tengas la imagen, haz esto con ella".

Pero luego llegaron las palabras clave `async` y `await`, que son como atajos. Con ellas, puedes escribir el código de una manera más fácil de entender, más parecida a cómo hablamos. Es como decir: "Espera a que se cargue la imagen antes de hacer lo siguiente".

Así que, cuando ves `async` en una función, significa que esa función puede contener `await`, y **await** se usa dentro de la función para esperar a que las promesas se cumplan antes de continuar con el resto del código. En resumen, **async/await** es una forma más fácil y natural de manejar tareas asíncronas en programación.

, con **async/await**, estamos simplificando la forma en que escribimos código asíncrono, haciéndolo más legible y mantenible, pero aún estamos trabajando dentro del modelo no bloqueante de JavaScript.

Ejemplo:

Imagina que tienes un código que hace una solicitud a un servidor para obtener un archivo de texto llamado "**robots.txt**". Quieres imprimir ese texto en la consola. Originalmente, podrías hacerlo usando `.then()` de esta manera:



```
1  fetch("/robots.txt")
2    .then(response => response.text())
3    .then(data => {
4      console.log(data);
5      console.log("Código síncrono.");
6    });
7
```

Primero, haremos la tarea usando el antiguo método `.then()`:

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => {
    console.log(data);
    console.log("Código síncrono.");
  });
}
```

Aquí, JavaScript ejecuta las solicitudes y continúa ejecutando el código en paralelo, sin esperar a que la solicitud termine. Pero ahora, quieres hacer lo mismo usando `await`.



```
1  async function fetchData() {  
2      const response = await fetch("/robots.txt");  
3      const data = await response.text();  
4      console.log(data);  
5      console.log("Código síncrono.");  
6  }  
7  
8  fetchData();  
9
```

```
async function fetchData() {  
    const response = await fetch("/robots.txt");  
    const data = await response.text();  
    console.log(data);  
    console.log("Código síncrono.");  
}  
  
fetchData();
```

En este código, `await` detiene la ejecución en la línea donde se encuentra hasta que la promesa se resuelva. La función **fetchData()** es asíncrona, ya que contiene `await`. Cuando llamamos a **fetchData()**, JavaScript espera a que se complete antes de pasar a la siguiente línea de código.

Sin embargo, si intentamos hacer una función `request()`, nos dará un error:



```
1 function request() {  
2     const response = await fetch("/robots.txt"); // Error!  
3     const data = await response.text(); // Error!  
4     return data;  
5 }  
6  
7 request(); // Error!
```

```
function request() {  
    const response = await fetch("/robots.txt"); // Error!  
    const data = await response.text(); // Error!  
    return data;  
}  
  
request(); // Error!
```

Para solucionar esto, necesitamos hacer la función **request()** asíncrona:
Al marcar **request()** como **async**, le estamos diciendo a JavaScript que esta función contiene operaciones asíncrónicas y debe ser manejada de manera especial.



```
1  async function request() {  
2      const response = await fetch("/robots.txt");  
3      const data = await response.text();  
4      return data;  
5  }  
6  
7  request(); // Ahora funciona correctamente  
8
```

async

```
function request() {  
    const response = await fetch("/robots.txt");  
    const data = await response.text();  
    return data;  
}  
  
request(); // Ahora funciona correctamente
```

Procesamiento de un solo hilo

Generalmente implica una ejecución síncrona, donde las instrucciones se ejecutan una tras otra en orden secuencial.

Esto significa que una instrucción no comienza hasta que la anterior haya terminado.

```
1 // Definimos una función que realiza un trabajo pesado de manera sincrónica
2 function trabajoPesado() {
3     let resultado = 0;
4     for (let i = 0; i < 1000000000; i++) {
5         resultado += i;
6     }
7     return resultado;
8 }
9
10 // Llamamos a la función de trabajo pesado
11 console.log("Comenzando el trabajo pesado...");
12 const resultadoTrabajo = trabajoPesado();
13 console.log("El resultado del trabajo pesado es:", resultadoTrabajo);
14 console.log("Trabajo pesado completado.");
15
```

// Definimos una función que realiza un trabajo pesado de manera sincrónica

function trabajoPesado() {

let resultado = 0;

for (let i = 0; i < 1000000000; i++) {

resultado += i;

}

return resultado;

}

// Llamamos a la función de trabajo pesado

console.log("Comenzando el trabajo pesado...");

const resultadoTrabajo = trabajoPesado();

console.log("El resultado del trabajo pesado es:", resultadoTrabajo);

console.log("Trabajo pesado completado.");

Procesamiento de múltiples hilos

Puede ser síncrono o asíncrono. La diferencia radica en que, con múltiples hilos, se pueden realizar múltiples tareas simultáneamente.

Si los hilos están sincronizados, pueden ejecutarse de manera síncrona, donde una tarea espera a que otra termine antes de continuar. Pero la gran ventaja de los múltiples hilos es la capacidad de ejecutar tareas de manera asíncrona, lo que permite que múltiples tareas se ejecuten de manera concurrente sin esperar a que una termine antes de comenzar otra.

Operadores de entrada / salida

Bloqueante: Son operaciones que no devuelven el control a nuestra aplicación hasta que se ha completado. Por tanto el thread queda bloqueado en estado de espera.

No Bloqueante: Son operaciones que devuelven inmediatamente el control a nuestra aplicación, independientemente del resultado de esta. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario (si la operación no ha podido ser satisfecha) podría devolver un código de error.

Ejemplo:

Supongamos que queremos leer un archivo desde el sistema de archivos en Node.js:

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

// Operación bloqueante
const data = fs.readFileSync('archivo.txt');
console.log(data); // El programa se detiene aquí hasta que se complete la operación de lectura
                    // del archivo
```

En el ejemplo de operación bloqueante, el programa se detiene en la línea **fs.readFileSync('archivo.txt')** hasta que se complete la lectura del archivo.



```
1 const fs = require('fs');
2
3 // Operación bloqueante
4 const data = fs.readFileSync('archivo.txt');
5 console.log(data); // El programa se detiene aquí hasta que
6 //se complete la operación de lectura del archivo
```

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

// Operación no bloqueante
fs.readFile('archivo.txt', (err, data) => {
  if (err) {
    console.error('Error al leer el archivo:', err);
    return;
}
```

```
console.log(data); // Se muestra el contenido  
//del archivo cuando la operación se completa  
});  
console.log('La ejecución del programa continúa mientras se lee el archivo...');
```



```
1 const fs = require('fs');  
2  
3 // Operación no bloqueante  
4 fs.readFile('archivo.txt', (err, data) => {  
5   if (err) {  
6     console.error('Error al leer el archivo:', err);  
7     return;  
8   }  
9   console.log(data); // Se muestra el contenido  
10  //del archivo cuando la operación se completa  
11 });  
12  
13 console.log('La ejecución del programa continúa mientras se lee el archivo...');
```

En el ejemplo de operación no bloqueante, el programa continúa ejecutándose después de la llamada a `fs.readFile('archivo.txt')`, y el contenido del archivo se maneja en una función de devolución de llamada que se ejecuta cuando la operación de lectura se completa.

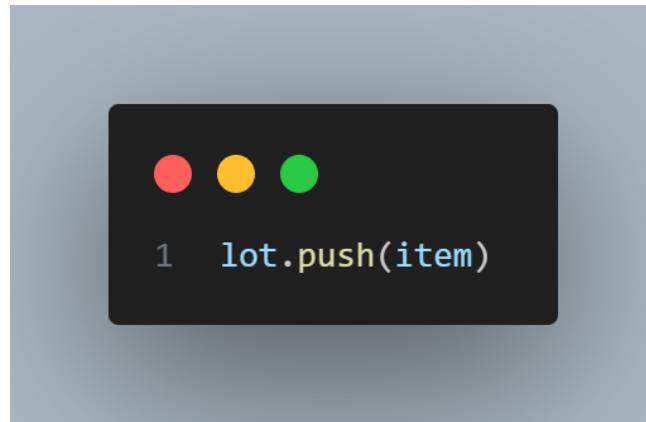
Esto permite que el programa siga siendo receptivo mientras espera que se complete la operación de lectura del archivo.

LIFO en JS Last In , First Out (Stack)

Quiere decir que el último en entrar, es el primero en salir.

Algunos ejemplos de la vida real pueden ser los siguientes:

Para esto se usará el método **push()** que permitirá agregar datos a la pila. Y cada vez que se agregue un nuevo elemento a la lista, esta quedará al final de la misma.



- **push:** Agrega un nuevo valor a la pila, **ubicándolo al final de ésta.**
- **pop:** Retorna el último valor ingresado a la pila, **sacándolo de ésta.**
- **peek:** Retorna el último valor ingresado a la pila, **sin sacarlo de ésta.**
- **size:** Retorna el número de elementos que contiene la pila.
- **print:** Muestra el contenido de la pila.

Ejemplos:

Montaña de pancakes - montas pancake sobre pancake y empiezas a comer por el último que montaste

Montaña de sillas - acomodar las sillas una encima de la otra, pero para poder sacar una tienes que sacar la última que pusiste y así sucesivamente

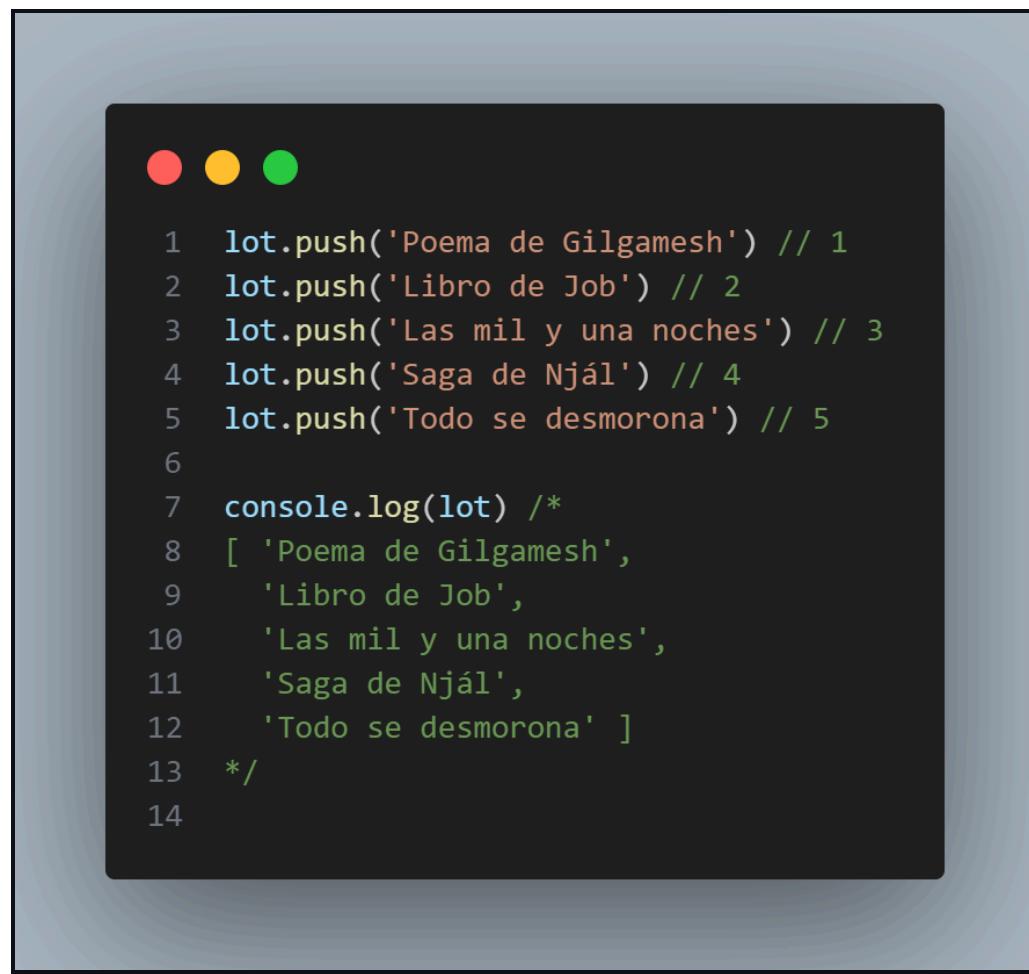
Ejemplo código:

Por ejemplo, agregar un libro a la lista.

```
lot.push('Poema de Gilgamesh') // 1
```

```
lot.push('Libro de Job') // 2  
lot.push('Las mil y una noches') // 3  
lot.push('Saga de Njál') // 4  
lot.push('Todo se desmorona') // 5
```

```
console.log(lot) /*  
[ 'Poema de Gilgamesh',  
  'Libro de Job',  
  'Las mil y una noches',  
  'Saga de Njál',  
  'Todo se desmorona' ]  
*/
```



```
1 lot.push('Poema de Gilgamesh') // 1  
2 lot.push('Libro de Job') // 2  
3 lot.push('Las mil y una noches') // 3  
4 lot.push('Saga de Njál') // 4  
5 lot.push('Todo se desmorona') // 5  
6  
7 console.log(lot) /*  
8 [ 'Poema de Gilgamesh',  
9   'Libro de Job',  
10  'Las mil y una noches',  
11  'Saga de Njál',  
12  'Todo se desmorona' ]  
13 */  
14
```

El objetivo de la pila LIFO es obtener cada vez el último elemento de la lista, en este caso usamos el método `pop()` que se encargara de obtener el último elemento de la lista y eliminarla de la lista.

Por ejemplo, bajo el mismo ejemplo anterior.

```
const book = lot.pop()

console.log(book) // "Todo se desmorona"

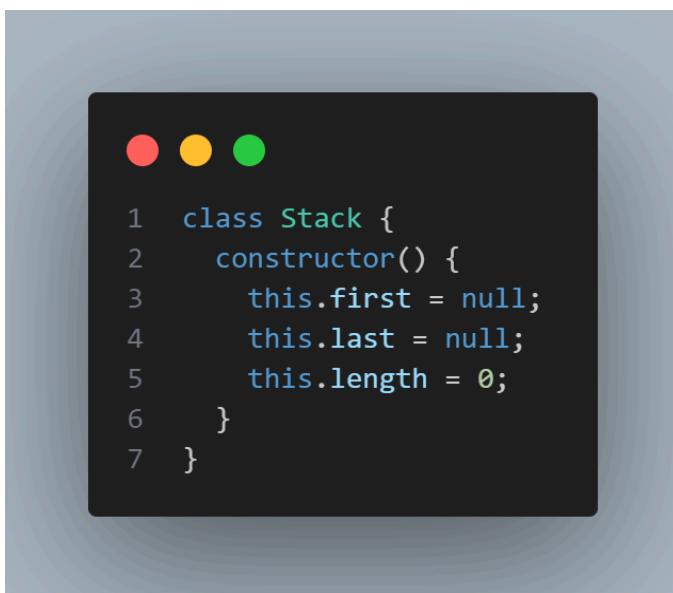
console.log(lot) /*
[ 'Poema de Gilgamesh',
  'Libro de Job',
  'Las mil y una noches',
  'Saga de Njál' ]
*/
```



1. Se crea un array llamado `lot` (que parece ser una abreviatura de "lista" o "lote") y se agregan cinco elementos a esta lista usando el método `push()`:
 - 'Poema de Gilgamesh'
 - 'Libro de Job'
 - 'Las mil y una noches'
 - 'Saga de Njál'
 - 'Todo se desmorona'
2. Se imprime el contenido actual de la lista `lot` utilizando `console.log(lot)`. Esto muestra todos los elementos de la lista en el orden en que fueron agregados.

3. Se ejecuta `lot.pop()`, que elimina el último elemento de la lista `lot` y devuelve ese elemento. En este caso, el último elemento es 'Todo se desmorona', por lo que esa cadena se guarda en la variable `book`.
4. Se imprime el valor de `book`, que es el elemento que fue eliminado de la lista.
5. Se imprime nuevamente el contenido de la lista `lot`, que ahora contiene los cuatro elementos restantes después de eliminar 'Todo se desmorona'.

Nota: En JavaScript, esta estructura no existe, por lo que comúnmente se utiliza un arreglo en su lugar. Podemos utilizar los métodos de `Array.prototype.push()` y `Array.prototype.pop()`



En JavaScript, no existe una estructura de datos específica llamada "pila" como en otros lenguajes de programación. Por eso, a menudo usamos arreglos para simular una pila.

Ejemplo: Imagina una pila de libros. Si agregas un nuevo libro, este se coloca encima de los demás. Para sacar un libro, tomas el que está en la parte superior (último libro que colocaste).

Métodos `push()` y `pop()` para simular una Stack:

Los arreglos en JavaScript tienen los métodos `push()` y `pop()`, que permiten agregar un elemento al final del arreglo y eliminar el último elemento, respectivamente. Estos métodos se pueden usar para emular una pila.

```
const pila = [] // Creamos una pila vacía usando un arreglo
```

```
pila.push(1); // Agregamos un elemento a la pila  
pila.push(2); // Agregamos otro elemento a la pila  
const elementoEliminado = pila.pop(); // Eliminamos el último elemento agregado  
console.log(elementoEliminado); // Esto imprimirá 2
```



```
1 const Stack = []; // Creamos una Stack vacía usando un arreglo  
2 Stack.push(1); // Agregamos un elemento a Stack  
3 Stack.push(2); // Agregamos otro elemento a la Stack  
4 const elementoEliminado = Stack.pop(); // Eliminamos el último elemento agregado  
Problema: los arreglos no tienen métodos para simular una Stack:  
6
```

Aunque usar arreglos con **push()** y **pop()** funciona para simular una Stack, los arreglos tienen otros métodos que no son propios de una pila. Esto puede llevar a errores si accidentalmente usamos esos métodos adicionales.

Implementación de una clase para una Stack:

Para evitar este problema, podemos implementar nuestra propia clase de Stack que restringe el acceso solo a los métodos necesarios para un Stack, es decir, **push()** y **pop()**.

```
class Stack {  
    constructor() {  
        this.first = null; // Puntero al primer elemento de la pila  
        this.last = null; // Puntero al último elemento de la pila  
        this.length = 0; // Tamaño de la pila  
  
    }  
    // Métodos para agregar y quitar elementos de la pila...  
}
```



```
1  
2 class Stack {  
3     constructor() {  
4         this.first = null; // Puntero al primer elemento de la pila  
5         this.last = null; // Puntero al último elemento de la pila  
6         this.length = 0; // Tamaño de la pila  
7     }  
8     // Métodos para agregar y quitar elementos de la pila...  
9 }
```

Nodos para representar elementos individuales:

Para modelar los elementos individuales que componen la pila, podemos usar una clase Node. Cada nodo contiene su valor y una referencia al nodo anterior en la pila.



```
1 class Node {  
2     constructor(val) {  
3         this.val = val; // Valor del nodo  
4         this.prev = null; // Referencia al nodo anterior en la pila  
5     }  
6 }  
7
```

Symbols

Según la especificación, sólo dos de los tipos primitivos pueden servir como clave de propiedad de objetos:

Un símbolo es único. Esto quiere decir que, cada vez que creamos uno nuevo, es completamente diferente y no se compara con ningún otro símbolo, incluso si se crean a partir del mismo valor.

Para crear un símbolo, se utiliza la función `Symbol()`. Opcionalmente, se puede pasar una cadena como argumento, que se utiliza como descripción del símbolo. Esta descripción no es accesible en ningún lugar del código y solo se utiliza para fines de depuración.

```
const mySymbol = Symbol()  
const mySymbolWithDescription = Symbol('descripción del símbolo')
```

```
// los símbolos son únicos  
Symbol() === Symbol() // false  
// incluso con la misma descripción
```

`Symbol('a') === Symbol('a') // false`

```
● ● ●  
1 const mySymbol = Symbol()  
2 const mySymbolWithDescription = Symbol('descripción del símbolo')  
3  
4 // los símbolos son únicos  
5 Symbol() === Symbol() // false  
6 // incluso con la misma descripción  
7 Symbol('a') === Symbol('a') // false  
8
```

string, o
symbol.

Si se usa otro tipo, como un número, este se auto convertirá a string. Así, `obj[1]` es lo mismo que `obj["1"]`, y `obj[true]` es lo mismo que `obj["true"]`.

El valor de “**Symbol**” representa un identificador único.
Un valor de este tipo puede ser creado usando **Symbol()**:

Se garantiza que los símbolos son únicos. Aunque declaremos varios Symbols con la misma descripción, éstos tendrán valores distintos. La descripción es solamente una etiqueta que no afecta nada más.
Por ejemplo, aquí hay dos Symbols con la misma descripción... pero no son iguales:



```
1 let id = Symbol("id");
```



```
1 let sym1 = Symbol();
2 let sym2 = Symbol("foo");
3 let sym3 = Symbol("foo");
4
```

Si quieres crear un símbolo a partir de una cadena de texto, puedes usar el método **Symbol.for()**. Esto crea un símbolo globalmente compartido, lo que significa que siempre se devuelve el mismo símbolo para una clave dada.

```
const sharedSymbol = Symbol.for('shared symbol')
const sameSharedSymbol = Symbol.for('shared symbol')

sharedSymbol === sameSharedSymbol // true
```



```
1 const sharedSymbol = Symbol.for('shared symbol')
2 const sameSharedSymbol = Symbol.for('shared symbol')
3
4 sharedSymbol === sameSharedSymbol // true
5
```

Una vez que se ha creado un símbolo, se puede utilizar como clave en un objeto. Pero ten cuidado porque los símbolos no se enumeran cuando se itera sobre las propiedades de un objeto. Se podría decir que son como propiedades privadas, en cierto modo.

```
const mySymbol = Symbol('property of object')
const myObject = {
  [mySymbol]: 'This is the value'
}

console.log(myObject[mySymbol]); // 'This is the value'
```



```
1
2 const mySymbol = Symbol('property of object')
3 const myObject = {
4   [mySymbol]: 'This is the value'
5 }
6
7 console.log(myObject[mySymbol]); // 'This is the value'
```

Los símbolos también se pueden utilizar como eventos únicos. Por ejemplo, si queremos crear una función que se ejecute cuando se hace clic en un elemento del DOM, podemos utilizar un símbolo para nombrar de forma única el evento y evitar conflictos con otros eventos que puedan estar en el mismo elemento.

```
const clickEventSymbol = Symbol('Click event')

element.addEventListener(clickEventSymbol, () => {
  console.log('Click event triggered')
})
```



```
1
2 const clickEventSymbol = Symbol('Click event')
3
4 element.addEventListener(clickEventSymbol, () => {
5   console.log('Click event triggered')
6 })
```

Sets



```
1 const set = new Set(); // Set({})(Conjunto vacío)
2 const set = new Set([5, 6, 7, 8, 9]); // Set({5, 6, 7, 8, 9})(Conjunto con 5 elementos)
3 const set = new Set([5, 5, 7, 8, 9]); // Set({5, 7, 8, 9})(Conjunto con 4 elementos)
4 set.constructor.name; // "Set"
```

Los Set en Javascript son estructuras de datos nativas muy interesantes para representar conjuntos de datos. La característica principal es que los datos insertados no se pueden repetir.

Un conjunto (Set) es una colección de valores únicos, sin orden específico. No permite valores duplicados, es decir es una colección de valores únicos, lo que significa que no puede haber elementos duplicados en un Set.

Como hemos dicho, la característica principal de los conjuntos es que es una estructura que no permite valores repetidos

Creación: Se puede crear un Set utilizando el constructor Set o la notación literal {...}.

Ejemplo:



```
1 const mySet = new Set([1, 2, 3, 4, 1]); // {1, 2, 3, 4}
2 const anotherSet = { 1, 2, 3, 4, 1}; // {1, 2, 3, 4}
3
```



```
1 set.add(1);
2 set.add(2);
3 set.add(3);
4 set.add(2); // No se agregará, ya que 2 ya está en el set
5
6
7 console.log(set); // Output: Set { 1, 2, 3 }
8
```

Un Set nunca tendrá el mismo elemento almacenado, despreocupándote por ejemplo, si tienes que asegurarte de que no existan elementos duplicados, ya que es un caso que no puede ocurrir.

Propiedad o Método	Descripción
NUMBER .size	Propiedad que devuelve el número de elementos que tiene el conjunto.
SET .add(element)	Añade un elemento al conjunto (si no está repetido) y devuelve el set. Muta
BOOLEAN .has(element)	Comprueba si element ya existe en el conjunto. Devuelve si existe.
BOOLEAN .delete(element)	Elimina el element del conjunto. Devuelve si lo eliminó correctamente.
.clear()	Vacía el conjunto completo.

Propiedad size

Si quieres saber cuántos elementos tienes en el conjunto, puedes utilizar la propiedad .size, que funciona de forma muy similar al .length de los array, por ejemplo.



```
1 const set = new Set();
2 set.size;    // 0
3 const set = new Set([5, 6, 7, 8]);
4 set.size;    // 4
5 const set = new Set([5, 6, 7, 8, 8]);
6 set.size;    // 4 (El 8 sólo se inserta una vez)
```

Los diferentes métodos que tienen las estructuras de conjuntos Set:

Añadir elementos ADD():

En primer lugar, el método `.add()` permite añadir un elemento al conjunto. Recuerda que aunque hasta ahora hemos utilizado sólo números, en el conjunto pueden insertarse otros tipos de elementos.

```
● ● ●  
1 const set = new Set();  
2  
3 set.add(5);  
4 set.add("A");  
5 set.add(5); // No se inserta  
6  
7 set; // Set({5, "A"})
```

Observa que si intentamos añadir un elemento ya existente, no nos dará error, pero simplemente no se volverá a insertar. El método `.add()` devuelve el set con la inserción realizada, es decir, devuelve una referencia al conjunto.

Comprobar si existen Has():

Para comprobar si un elemento existe en un conjunto, podemos utilizar el método `.has()`. Este método devuelve un `true`, por lo que sí existe, nos devolverá `true`. De lo contrario, `false`.



```
1 const set = new Set([1, 2, 3]);
2 set.has(2);      // true
3 set.has(34);    // false
4 set.add(34);
5 set.has(34);    // true
```

Borrar elementos (Delete)

Si necesitamos borrar algún elemento del conjunto, podemos utilizar el método `.delete()`. Al igual que el anterior, devuelve un `.
Si el borrado se realizó con éxito, devolverá true, si no pudo realizarse (no existe el elemento), devolverá false.`

```
const set = new Set([1, 2, 3]);
set.delete(3); // true
set.delete(39); // false
set; // Set({1, 2})
```



```
1 const set = new Set([1, 2, 3]);
2 set.delete(3); // true
3 set.delete(39); // false
4
5 set; // Set({1, 2})
```

Vaciar Conjuntos(Clear):

Si por otro lado, queremos hacer un borrado completo de los elementos, utilizaremos el método `.clear()`, que no devuelve nada. Simplemente borrará todos los elementos del conjunto y lo dejará vacío.

```
const set = new Set([1, 2, 3]);
set.clear();
set.size; // 0
```



Convertir Arrays

Una de las cosas más interesantes y útiles de los Set, es que al ser una estructura iterable (se puede recorrer), es muy sencillo utilizar desestructuración y convertirlo a un array (o viceversa):

```
const set = new Set([5, "A", [99, 10, 24]]);
set.size; // 3 (Contiene 3 elementos)
set.constructor.name; // "Set"
const array = [...set];
array.constructor.name; // "Array"
array; // [5, "A", [99, 10, 24]]
```



```
1 const set = new Set([5, "A", [99, 10, 24]]);
2 set.size; // 3 (Contiene 3 elementos)
3 set.constructor.name; // "Set"
4 const array = [...set];
5 array.constructor.name; // "Array"
6 array; // [5, "A", [99, 10, 24]]
```

Notas: Cuidado cuando tengas conjuntos con tipos de datos más complejos con elementos anidados (arrays, objetos, etc...). Recuerda que son referencias y modificar un elemento referenciado, modificará el original.

Para evitar esto de forma sencilla, puedes utilizar la función **structuredClone()**



```
1 const set = new Set([5, "A", [99, 10, 24]]);
2 set.size; // 3
3 const clonedArray = [...structuredClone(set)];
4 const array = [...set];
5 clonedArray[2][0] = "Modified";
6 [...set][2][0]; // 99 (El original se mantiene intacto)
7 array[2][0] = "Modified";
8 [...set][2][0]; // "Modified" (El original ha mutado)
```

Además, también puedes hacer la operación inversa, para convertir un array en un Set:

```
const array = [5, 4, 3, 3, 4];
const set = new Set(array);
set; // Set({ 5, 4, 3 })
```



```
1 const array = [5, 4, 3, 3, 4];
2 const set = new Set(array);
3 set; // Set({ 5, 4, 3 })
```

Operaciones:

add(valor): Agrega un nuevo valor al conjunto

delete(valor): Elimina un valor del conjunto

has(valor): Comprueba si un valor existe en el conjunto

size: Devuelve la cantidad de elementos en el conjunto

clear(): Elimina todos los elementos del conjunto.

values(): Devuelve un iterador sobre los valores del conjunto.

forEach(callback): Ejecuta una función para cada valor del conjunto.

```
const mySet = new Set([1, 2, 3, 4, 1]);
```

```
mySet.add(5); // {1, 2, 3, 4, 5}
mySet.delete(2); // {1, 3, 4, 5}
console.log(mySet.has(3)); // true
console.log(mySet.size); // 4
mySet.clear(); // {}
```

```
for (const value of mySet.values()) {
  console.log(value); // No se imprime ningún valor
}
```

```
1
2 const mySet = new Set([1, 2, 3, 4, 1]);
3
4 mySet.add(5); // {1, 2, 3, 4, 5}
5 mySet.delete(2); // {1, 3, 4, 5}
6 console.log(mySet.has(3)); // true
7 console.log(mySet.size); // 4
8 mySet.clear(); // {}
9
10 for (const value of mySet.values()) {
11   console.log(value); // No se imprime ningún valor
12 }
13
```

WeakSets

A grandes rasgos, los WeakSet son otro tipo de estructura de conjuntos, muy similar a Set (también impide introducir elementos duplicados), sin embargo, tiene algunos matices y diferencias. Veamos esas diferencias.

Los **Set** son una estructura de datos poco restrictiva, ya que puedes insertar cualquier tipo de elemento. Los **WeakSet** no permiten insertar datos primitivos:

```
1 // *** Set
2 const set = new Set([1, "A", true]); // OK
3 const set = new Set([{ name: "Manz" }, [2, 30]]); // OK
4 // *** WeakSet
5
6 const set = new WeakSet([1, "A", true]);
7 // ERROR: Uncaught TypeError: Invalid value used in weak set
8 const set = new WeakSet([{ name: "Manz" }, [2, 30]]); // OK
```

Por otro lado, los **WeakSet** utilizan **referencias débiles** a un objeto, es decir, si ese objeto no se utiliza (no está referenciado) en ninguna otra parte del código, se eliminará del **WeakSet** en cuanto el **Garbage Collector** (Recolector de basura) lo decida para liberar memoria:

```
● ● ●  
1 let element = { name: "Manz" };  
2 const set = new WeakSet([element]);  
3 set; // WeakSet({ { name: "Manz" } })  
4 element = null;  
5 set; // WeakSet({})
```

Observa que la penúltima línea, reasignamos a **null** la variable **element**. En ese caso, la zona de memoria donde está guardada la información **{ name: "Manz" }** no está referenciada en ninguna otra variable, por lo que Javascript considera que ya no es útil, y la borra del **WeakSet** y de memoria.

Cuadro comparativo

Característica	Set	WeakSet
Se pueden insertar elementos repetidos	✗	✗
Se pueden insertar elementos primitivos	✓	✗
Si no se usa el elemento, se elimina del set	✗	✓
Se puede convertir a array (es iterable)	✓	✗
Propiedad .size	✓	✗
Método .add()	✓	✓
Método .has()	✓	✓
Método .delete()	✓	✓
Método .clear()	✓	✗

Maps

Los Map en Javascript son estructuras de datos nativas que permiten implementar una estructura de tipo mapa, es decir, una estructuras donde tiene valores guardados a través de una clave para identificarlos. Comúnmente, esto se denomina pares clave-valor.

Similar a un diccionario en otros lenguajes Un mapa (Map) es una colección de pares clave-valor, donde cada clave es única.



```
1 let map = new Map();
2
3 map.set('key1', 'value1');
4 map.set('key2', 'value2');
5
6 console.log(map.get('key1')); // Output: value1
7
```

```
let map = new Map();
map.set('key1', 'value1');
map.set('key2', 'value2');
console.log(map.get('key1')) // Output: value1
const map = new Map(); // Map(()) (Mapa vacío)
const map = new Map([[1, "uno"]]); // Map({ 1=>"uno" })
const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]); // Map({
1=>"uno", 2=>"dos", 3=>"tres" })
map.constructor.name; // "Map"
```



```
1 const map = new Map(); // Map(()) (Mapa vacío)
2 const map = new Map([[1, "uno"]]); // Map({ 1=>"uno" })
3 const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]); // Map({ 1=>"uno", 2=>"dos", 3=>"tres" })
4 map.constructor.name; // "Map"
```

En este ejemplo, creamos un elemento **map**, que no es más que un mapa de pares clave-valor. El primer **map** se define como un mapa vacío, el segundo, es un mapa con un solo elemento, y el tercero con 3 elementos. Para inicializar los mapas con datos, se introduce como parámetro un array de entradas (un array de arrays), que en nuestro tercer caso tiene estas combinaciones:

- **Clave:** **NUMBER** 1 => **Valor:** **STRING** "uno"
- **Clave:** **NUMBER** 2 => **Valor:** **STRING** "dos"
- **Clave:** **NUMBER** 3 => **Valor:** **STRING** "tres"

Por lo tanto, si consultamos **map** con la clave 2, nos devolverá un **STRING** "dos".

Los tipos de datos Map son muy similares a los Objetos de Javascript, ya que estos últimos se pueden usar como estructuras de diccionario mediante **pares clave-valor**.

Una estructura de tipo **Map** tiene las siguientes propiedades o métodos:

Propiedad o Método	Descripción
NUMBER .size	Propiedad que devuelve el número de elementos que tiene el mapa.
MAP .set(key, value)	Establece o modifica la clave key con el valor value . Muta
BOOLEAN .has(key)	Comprueba si key ya existe en el mapa y devuelve si existe o no.
OBJECT .get(key)	Obtiene el valor de la clave key del mapa.

Ten en cuenta que al contrario que los **OBJECT**, los **MAP** pueden utilizar como clave cualquier tipo de dato. En el caso de los **OBJECT** debes utilizar un **STRING** o un **SYMBOL**.

Propiedad Size

Si quieres saber cuántos elementos tiene un mapa, puedes utilizar la propiedad `.size`, que funciona de forma muy similar al `.length` de los array, por ejemplo.

```
● ● ●  
1 const map = new Map();  
2 map.size; // 0  
3 const map = new Map([[1, "uno"], [2, "dos"]]);  
4 map.size; // 2  
5 const map = new Map([[1, "uno"], [2, "dos"], [1, "tres"]]);  
6 map.size; // 2 (El 1->"tres" sobreescribe al anterior)
```

Observa que si introducimos un nuevo par clave-valor que tiene la misma clave que otro (tenemos dos que comparten la clave 1), se sobreescibirá. No pueden existir dos pares clave-valor con la misma clave.

Establecer elementos (SET)

El método `.set()` fija un par clave-valor en el mapa. Observa que hay un pequeño matiz muy importante de diferencia entre el concepto «añadir» (`.add()`) y el concepto «establecer» o «fijar» (`.set()`):

- Si usamos `.set()` para una clave que no existe, se añade al mapa.
- Si usamos `.set()` para una clave que ya existe, la sobreescibe

```
● ● ●  
1 const map = new Map();  
2 map.set(5, "cinco");  
3 map.set("A", "letra A");  
4 map.set(5, "cinco sobreescrito"); // Sobreescrige el anterior  
5 map; // Map({ 5=>"cinco sobreescrito", "A"=>"letra A" })
```

Para comprobar si un elemento existe en un mapa, se debe hacer a través de su clave, y se utiliza el método `.has()`. Este método devuelve un **Boolean**, por lo que si existe la clave, nos devolverá `true`, y en caso contrario, nos devolverá `false`



```
1 const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
2 map.has(2); // true
3 map.has(34); // false
4 map.set(34, "treinta y cuatro");
5 map.has(34); // true
```

Comprobar elementos (Delete)

Si necesitamos borrar algún elemento del mapa, lo podemos hacer mediante el método `.delete()`. Devuelve un Boolean a `true` si lo consigue eliminar, en caso contrario, devolverá `false`.



```
1 const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);
2 map.delete(3); // true
3 map.delete(39); // false
4 map; // Map({ 1=>"uno", 2=>"dos" })
```

Vaciar conjunto(Clear)

utilizando el método `.clear()` borraremos todos los elementos del mapa, dejándolo vacío. Este método no devuelve nada

```
● ● ●  
1 const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
2 map.clear();  
3 map.size // 0
```

Convertir en Arrays

Si tenemos claro el proceso de desestructuración, podemos convertir los **Map** en **ARRAY** o incluso en **OBJECT** de forma muy sencilla. Eso sí, antes te recomiendo mirarte el artículo de Iteradores en Objetos:

```
● ● ●  
1 const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);  
2 map.size; // 3 (Contiene 3 elementos)  
3 map.constructor.name; // "Map"  
4 const entries = [...structuredClone(map)];  
5 entries.constructor.name; // "Array"  
6 entries; // [[1, "uno"], [2, "dos"], [3, "tres"]]
```

```
const myMap = new Map([
  ['nombre', 'Juan'],
  ['edad', 30],
  ['profesion', 'Desarrollador'],
]); // Map { 'nombre' => 'Juan', 'edad' => 30, 'profesion' => 'Desarrollador' }
const anotherMap = {
  nombre: 'Juan',
  edad: 30,
  profesion: 'Desarrollador',
}; // Map { 'nombre' => 'Juan', 'edad' => 30, 'profesion' => 'Desarrollador' }
```

- `set(clave, valor)`: Agrega un nuevo par clave-valor al mapa.
- `get(clave)`: Obtiene el valor asociado a una clave específica.
- `has(clave)`: Comprueba si una clave existe en el mapa.
- `delete(clave)`: Elimina un par clave-valor del mapa.
- `size`: Devuelve la cantidad de pares clave-valor en el mapa.
- `clear()`: Elimina todos los pares clave-valor del mapa.
- `keys()`: Devuelve un iterador sobre las claves del mapa.
- `values()`: Devuelve un iterador sobre los valores del mapa.
- `entries()`: Devuelve un iterador sobre los pares clave-valor del mapa.
- `forEach(callback)`: Ejecuta una función para cada par clave-valor del mapa.

Weakmaps

Al igual que ocurre con los **Set** y los **WeakSet**, con los **Map** tenemos una estructura denominada **WeakMap**. La idea es la misma: se trata de una estructura derivada, muy similar a los **Map**, pero con algunas diferencias.

Diferencias con los Maps

Al margen de algunas diferencias que detallaremos más adelante, la diferencia principal de los **Map** con los **WeakMap** es que estos últimos, no permiten utilizar tipos primitivos (**BOOLEAN** , **STRING** , **NUMBER**) como **clave**, mientras que el **Map** si lo permite:

```
1 // *** Map
2 const map = new Map([[1, "uno"]]); // OK
3 const map = new Map([[{ id: 1, type: "number" }, "uno"]]); // OK
4 // *** WeakMap
5 const map = new WeakMap([[1, "uno"]]);
6 // ERROR: Uncaught TypeError: Invalid value used in weak map key
7 const map = new WeakMap([[{ id: 1, type: "number" }, "uno"]]); // OK
```

Weaksets

Un WeakSet es una colección de objetos débiles. A diferencia de los Sets, los objetos en un WeakSet no son rastreados por el garbage collector de JavaScript, evitando fugas de memoria. Solo puede almacenar objetos, no valores primitivos.

Característica	Map	WeakMap	Object
Se pueden insertar claves repetidas	✗	✗	✗
Se pueden insertar claves con tipos primitivos	✓	✗	Sólo STRING o SYMBOL
Si no se usa el elemento, se elimina del map	✗	✓	✗
Se puede convertir a array (es iterable)	✓	✗	✗ Object.entries(obj)
Pueden colisionar algunas claves *	✗	✗	✓
Las claves garantizan un orden por inserción	✓	✓	✗
Propiedad .size	✓	✗	✗ Object.keys(obj).length
Método .set()	✓	✓	✗ Se usa asignación por clave
Método .get()	✓	✓	✗ Se usa acceso a la clave
Método .has()	✓	✓	✗ Object.keys(obj).includes(key)
Método .delete()	✓	✓	✗
Método .clear()	✓	✗	✗

Iterables

Un iterable es un objeto que puede ser utilizado con un ciclo `for...of` para acceder a sus elementos de forma secuencial.

En Javascript, los **iterables** son objetos que **contienen una colección de elementos y proporcionan un mecanismo para acceder a ellos uno a uno**.

Ejemplos de iterables:

- **Arreglos:** `[1, 2, 3, 4, 5]`
- **Cadenas de texto:** `"¡Hola, mundo!"`
- **Sets:** `{1, 2, 3, 4, 5}`
- **Mapas:** `{ "nombre": "Juan", "edad": 30 }`
- **Objetos con propiedad `Symbol.iterator`:** Cualquier objeto que implemente este símbolo especial.

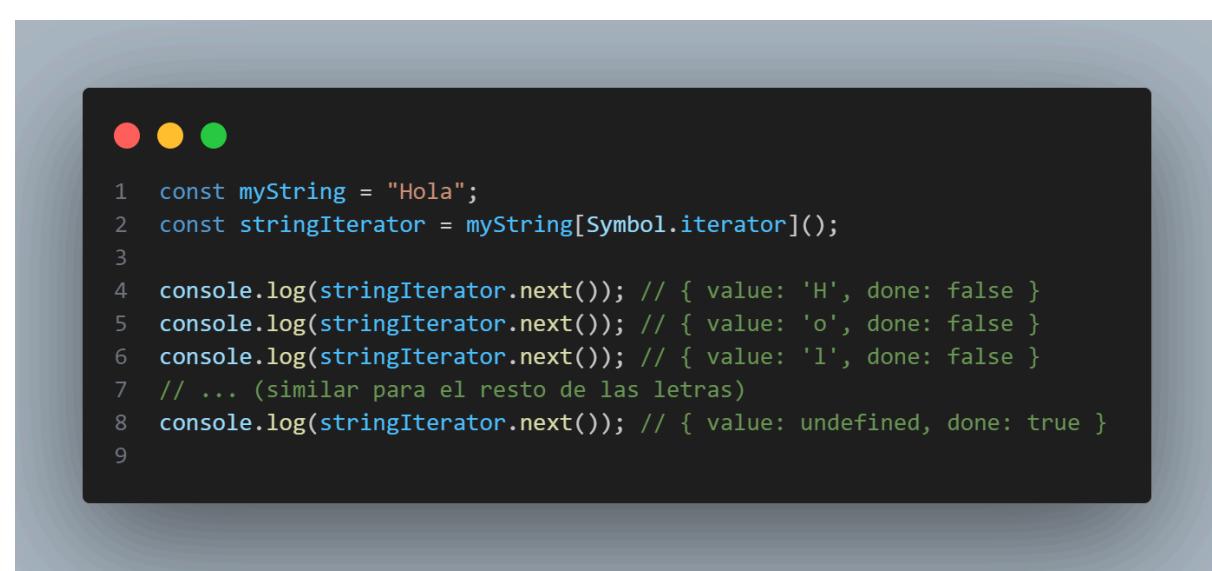


```
1 const myString = "Hola";
2 for (const char of myString) {
3     console.log(char); // Imprime H, o, l, a
4 }
5
6 const myNumbers = [1, 2, 3, 4];
7 for (const number of myNumbers) {
8     console.log(number); // Imprime 1, 2, 3, 4
```

Iterators

Un iterador es un objeto que permite recorrer los elementos de un iterable. Posee un método `next()` que devuelve el siguiente elemento del iterable y un atributo `done` que indica si se ha llegado al final de la iteración.

Un iterador es un concepto que se repite, y en el ámbito de la programación, se suele referir a algo que te permite recorrer una estructura de datos por todos sus apartados o miembros.



```
● ● ●
1 const myString = "Hola";
2 const stringIterator = myString[Symbol.iterator]();
3
4 console.log(stringIterator.next()); // { value: 'H', done: false }
5 console.log(stringIterator.next()); // { value: 'o', done: false }
6 console.log(stringIterator.next()); // { value: 'l', done: false }
7 // ... (similar para el resto de las letras)
8 console.log(stringIterator.next()); // { value: undefined, done: true }
9
```

```
const myString = "Hola";
const stringIterator = myString[Symbol.iterator]();
```

```
console.log(stringIterator.next()); // { value: 'H', done: false }
console.log(stringIterator.next()); // { value: 'o', done: false }
console.log(stringIterator.next()); // { value: 'l', done: false }
// ... (similar para el resto de las letras)
console.log(stringIterator.next()); // { value: undefined, done: true }
```

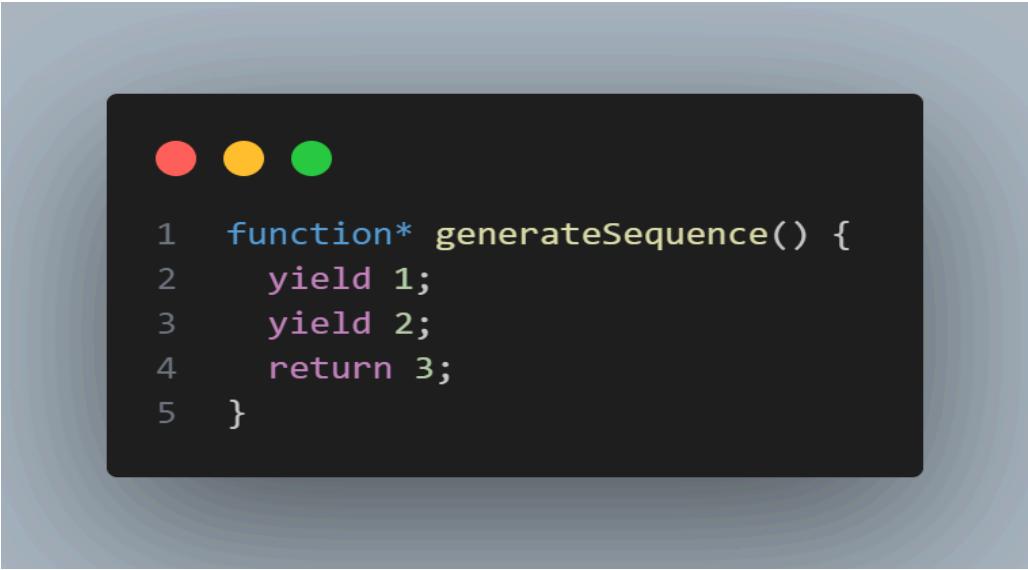
Generadores

Las funciones regulares devuelven solo un valor único (o nada).

Los generadores pueden producir (“**yield**”) múltiples valores, uno tras otro, a pedido. Funcionan muy bien con los iterables, permitiendo crear flujos de datos con facilidad.

Funciones Generadoras

Para crear un generador, necesitamos una construcción de sintaxis especial: **function***, la llamada “función generadora”.



```
1 function* generateSequence() {
2     yield 1;
3     yield 2;
4     return 3;
5 }
```

Las funciones generadoras se comportan de manera diferente a las normales. Cuando se llama a dicha función, no ejecuta su código. En su lugar, devuelve un objeto especial, llamado “objeto generador”, para gestionar la ejecución.



```
1 function* generateSequence() {
2     yield 1;
3     yield 2;
4     return 3;
5 }
6
7 function* generateSequence() {
8     yield 1;
9     yield 2;
10    return 3;
11 }
12
13 // "función generadora" crea "objeto generador"
14 let generator = generateSequence();
15 alert(generator); // [object Generator]
```

El método principal de un generador es next(). Cuando se llama, se ejecuta hasta la declaración yield <value> más cercana (se puede omitir value, entonces será undefined). Luego, la ejecución de la función se detiene y el value obtenido se devuelve al código externo.

El resultado de next() es siempre un objeto con dos propiedades:

value: el valor de yield.

done: true si el código de la función ha terminado, de lo contrario false.

Proxies

JavaScript's Proxy es una capacidad que permite la creación de objetos capaces de modificar y personalizar las operaciones básicas realizadas en otros objetos.

Para establecer un objeto Proxy, son necesarios dos componentes: un objeto de destino y un objeto controlador. El objeto objetivo es aquel en el que se deben interceptar las operaciones, mientras que el objeto controlador es responsable de sostener las trampas o métodos utilizados para capturar estas operaciones.

```
const target = {
  name: 'John',
  age: 25,
};
```

```

const handler = {
  get: function(target, prop) {
    console.log(`Getting property ${prop}`);
    return target[prop];
  },
};

const proxy = new Proxy(target, handler);

console.log(proxy.name);
// Getting property name
// John

```

En este ejemplo, generamos un objeto objetivo que tiene dos características: nombre y edad. También generamos un objeto controlador que tiene una trampa para capturar cualquier esfuerzo para leer una propiedad en el objeto de destino. Después de eso, producimos un objeto Proxy al proporcionar el objetivo y los objetos del controlador al constructor Proxy. Por último, recuperamos la propiedad de nombre del objeto Proxy, que invoca la trampa get y envía un mensaje a la consola.

Proxies Revocables

Los objetos proxy poseen una característica fascinante que les permite ser invalidados, lo que hace que sus trampas ya no intercepten operaciones en el objeto objetivo. Para construir un objeto Proxy que pueda ser invalidado, utilice el `Proxy.revocable()` función.

```

const target = {
  name: 'John',
  age: 25,
};

const handler = {
  get: function(target, prop) {
    console.log(`Getting property ${prop}`);
    return target[prop];
}

```

```

        } ,
};

const {proxy, revoke} = Proxy.revocable(target, handler);

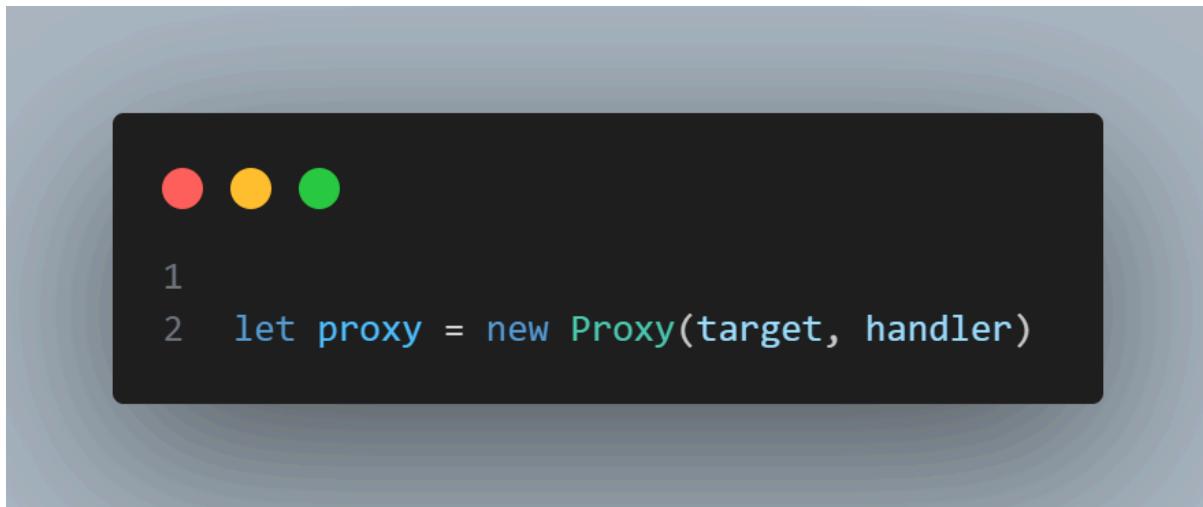
console.log(proxy.name);
// Getting property name
// John

revoke();

console.log(proxy.name);
// Uncaught TypeError: Cannot perform 'get' on a proxy that
has been revoked

```

Un objeto Proxy envuelve (es un “wrapper”: envoltura, contenedor) a otro objeto e intercepta sus operaciones (como leer y escribir propiedades, entre otras). El proxy puede manejar estas operaciones él mismo o, en forma transparente permitirle manejarlas al objeto envuelto.



Los proxys son usados en muchas librerías y en algunos frameworks de navegador. En este artículo veremos muchas aplicaciones prácticas.

target – es el objeto a envolver, puede ser cualquier cosa, incluso funciones.

handler – configuración de proxy: un objeto que “atrapa”, métodos que interceptan operaciones. Ejemplos, la trampa get para leer una propiedad de target, la trampa set para escribir una propiedad en target, entre otras.

Un proxy es un objeto que actúa como intermediario entre otro objeto (objeto objetivo) y su código que accede a él. Permite interceptar y modificar operaciones sobre el objeto objetivo, como leer, escribir o eliminar propiedades.

Se crea utilizando la función `Proxy()`, pasando el objeto objetivo y un objeto de configuración que define las trampas a interceptar.



```
1 const obj = { nombre: 'Juan', edad: 30 };
2
3 const proxy = new Proxy(obj, {
4   get: function (target, property) {
5     console.log(`Accediendo a la propiedad "${property}" de ${target}`);
6     return Reflect.get(target, property);
7   },
8   set: function (target, property, value) {
9     console.log(`Modificando la propiedad "${property}" de ${target} a ${value}`);
10    return Reflect.set(target, property, value);
11  }
12 });
13 );
14
15 console.log(proxy.nombre); // Imprime "Accediendo a la propiedad \"nombre\""
16 // de [ nombre: 'Juan', edad: 30 ] " y luego "Juan"
```



```
1
2 // Define un objeto 'obj' con una propiedad 'nombre' y 'edad'
3 obj = { nombre: 'Juan', edad: 30 };
4
5 // Crea un Proxy al objeto 'obj'
6 proxy = new Proxy(obj, {
7   // Define el handler para la operación 'get' en el Proxy
8   get: function (target, property) {
9     // Muestra un mensaje indicando el acceso a la propiedad y devuelve el valor real
10    Mostrar mensaje`Accediendo a la propiedad "${property}" de ${target}`;
11    return Obtener el valor real de la propiedad utilizando Reflect.get(target, property);
12  },
13  // Define el handler para la operación 'set' en el Proxy
14  set: function (target, property, value) {
```

```
const obj = { nombre: 'Juan', edad: 30 };

const proxy = new Proxy(obj, {
  get: function(target, property) {
    console.log(`Accediendo a la propiedad "${property}" de ${target}`);
    return Reflect.get(target, property);
  },
  set: function(target, property, value) {
    console.log(`Modificando la propiedad "${property}" de ${target} a ${value}`);
    return Reflect.set(target, property, value);
  }
});
```

```
console.log(proxy.nombre); // Imprime "Accediendo a la propiedad \"nombre\" de {  
nombre: 'Juan', edad: 30 }" y luego "Juan"  
proxy.nombre = 'Pedro'; // Imprime "Modificando la propiedad \"nombre\" de { nombre:  
'Juan', edad: 30 } a \"Pedro\""  
console.log(proxy.nombre); // Imprime "Pedro"
```

Propiedades dinámicas de los objetos

Las propiedades dinámicas son aquellas que se pueden agregar, eliminar o modificar en tiempo de ejecución, sin necesidad de estar predefinidas en la estructura del objeto.

```
const obj = {};
obj.propiedadDinamica = 'Valor dinámico';
obj['otraPropiedadDinamica'] = 'Otro valor dinámico';
```

Se pueden eliminar utilizando el operador `delete`:

```
delete obj.propiedadDinamica;
```

Se pueden modificar utilizando la misma notación que para crearlas:

```
obj.propiedadDinamica = 'Nuevo valor dinámico';
```



The screenshot shows a code editor window with a dark theme. At the top left are three colored circular icons: red, yellow, and green. The code editor displays the following JavaScript code:

```
1  obj.propiedadDinamica = 'Nuevo valor dinámico';
2
3
4  const obj = {};
5
6  function crearPropiedadDinamica(nombre, valor) {
7    obj[nombre] = valor;
8  }
9
10 crearPropiedadDinamica('nombre', 'Juan');
11 crearPropiedadDinamica('edad', 30);
12
13 console.log(obj.nombre); // Imprime "Juan"
14 console.log(obj.edad); // Imprime 30
15
16
17
```

```
const obj = {};
```

```
function crearPropiedadDinamica(nombre, valor) {  
    obj[nombre] = valor;  
}
```

```
crearPropiedadDinamica('nombre', 'Juan');  
crearPropiedadDinamica('edad', 30);
```

```
console.log(obj.nombre); // Imprime "Juan"  
console.log(obj.edad); // Imprime 30
```

Método This

Los objetos son creados usualmente para representar entidades del mundo real, como usuarios, órdenes, etc.



```
let user = {  
    name: "John",  
    age: 30  
};
```

```
1
2 let user = {
3   name: "John",
4   age: 30
5 };
6
7 user.sayHi = function () {
8   alert("¡Hola!");
9 };
10
11 user.sayHi(); // ¡Hola!
```

Aquí simplemente usamos una expresión de función para crear la función y asignarla a la propiedad user.sayHi del objeto.

Entonces la llamamos user.sayHi(). ¡El usuario ahora puede hablar!
Una función que es la propiedad de un objeto es denominada su método.
Así, aquí tenemos un método sayHi del objeto user.

También podríamos usar una función pre-declarada como un método, parecido a esto:

```
1 let user = {
2   // ...
3 };
4
5 // primero, declara
6 function sayHi() {
7   alert("¡Hola!");
8 };
9
10 // entonces la agrega como un método
11 user.sayHi = sayHi;
12
13 user.sayHi(); // ¡Hola!
```

