**Los conceptos son los siguientes:**

**¿Qué tipo de bucles hay en JS?**

Un bucle, o un loop en inglés, es un bloque de código que se ejecuta repetidamente hasta que se cumpla una cierta condición dada o mientras la condición sea verdadera. Son muy útiles para poder automatizar tareas repetitivas. En JS hay tres tipos principales de bucles: for, while y do…while.

Bucle for:

Los bucles for son útiles cuando se conoce la cantidad exacta de veces que se quiere repetir el bloque de código. Un uso común del bucle for es por ejemplo recorrer una lista, o también repetir una acción un número de veces especificado.

La sintaxis del bloque for empieza con la palara for y un paréntesis. En este paréntesis, de forma general, se comienza con una inicialización que se ejecuta únicamente al empezar el bloque el loop, continúa con la condición de que queremos que se cumpla, y termina con un incremento que se ejecuta cada vez que se completa un loop. Esto va seguido de llaves donde se encuentra la tarea que se quiere repetir. Esta es la estructura básica.

for (inicialización; condición; incremento) {

tarea que se quiere repetir;

}

Vamos a ver dos ejemplos básicos. En el primero vamos a imprimir los números del 0 al 5. Para ello, se va a comenzar inicializando la variable control con un valor inicial de 0 (let i = 0). Posteriormente se indica la condición, en este caso la condición es que el valor i sea menor que 6 (i < 6). Este apartado lo que hace es repetir el bloque de código dentro de las llaves hasta que i deje de ser menor que 6, es decir, cuando llegue a 6 el bucle se para y en caso de que hubiese más código, saldría del bucle y continuaría con las líneas siguientes. El tercer apartado de dentro del paréntesis es i++. Esta expresión se ejecuta cada vez que finaliza un bucle, es decir, cada vez que se termine de hacer una vez la tarea, se ejecuta este apartado, en este caso incrementa i en una unidad. Si no tuviésemos esta expresión, el valor de i seguiría siendo 0 y el bloque de código se ejecutaría de forma infinita. En este caso, como queremos imprimir los valores del 0 al 5, la tarea seria un console.log de i, ya que i es el número que queremos mostrar:

for (let i = 0; i < 6; i++) {

console.log(i);

}

0

1

2

3

4

5

En el segundo ejemplo de uso del bucle for, vamos a recorrer una lista. En este caso primero hay que tener la lista definida, es decir, antes del bucle. Vamos a ver un ejemplo de una lista de la compra, donde queremos imprimir cada elemento de la lista.

const listaCompra = ["patatas", "agua", "pasta"];

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

console.log(listaCompra[i]);

}

patatas

agua

pasta

En este caso, la condición en vez de ser un número fijo (por si se desconoce la longitud de la lista o se modifica) es la longitud de la lista usando length, el cual te devuelve la longitud exacta de la lista. Es decir, el tamaño de la lista es de 3, por lo tanto, comienza desde 0 que listaCompra[0] es el índice de patatas, y termina en listaCompra[2] que en este caso es pasta, y cuando i=3 termina de ejecutarse el bucle. Así conseguimos imprimir cada uno de los elementos de la lista usando el bucle for.

Además del bucle for clásico también existen variantes más actuales: for...of, for...in.

Bucle for...of:

Este tipo de bucle se usa para recorrer objetos iterables como arrays o strings. Es interesante cuando lo que te interesa es el valor del elemento, pero no su índice. Vamos a tomar de base la lista anterior listaCompra. Para imprimir cada uno de los elementos se haría de la siguiente forma:

for (const compra of listaCompra) {

console.log(compra);

}

patatas

agua

pasta

Dentro del paréntesis del for, const compra es una variable temporal, es decir, sólo existe en el bucle for mientras está ejecutándose, no es una variable que se pueda llamar desde el exterior del bucle. Esta es la que va a contener el valor del elemento que se itera. Of es una palabra clave, con esta defines que quieres iterar sobre los valores de la lista y no sobre los índices, como se hace con i en el bucle clásico, ni sobre claves de objetos. De esta forma evitas usar un contador e iteras hasta que se terminen los valores sin tener que definir el final, como por ejemplo length en el caso clásico.

Bucle for...in:

Este bucle, a diferencia del anterior, se usa mayoritariamente en objetos para recorrer sus propiedades. Por ejemplo, vamos a crear un objeto de coche

const coche = {

color: "rojo",

marca: "toyota",

año: "2018"

};

for (const key in coche) {

console.log(key + ": " + coche[key]);

}

color: rojo

marca: toyota

año: 2018

Como se puede observar, se puede imprimir tanto la clave (key) como el valor de la clave (coche[key]) iterando sobre el objeto.

Método forEach:

Además de estos bucles, hay un método propio de arrays llamado forEach que, a pesar de no ser un bucle como tal, cumple una función similar y es muy utilizado para recorrer los elementos de un array, ejecuntando una función callback por cada elemento. Siguiendo con la lista de la compra, se puede realizar tanto con una arrow function o function y se pueden conseguir tanto los elementos como sus índices:

listaCompra.forEach((compra, index) => {console.log(compra, index);})

patatas 0

agua 1

pasta 2

listaCompra.forEach(function(compra, index) {

    console.log(compra, index);

});

patatas 0

agua 1

pasta 2

Bucle while:

En un bucle while, el código se ejecuta mientras una condición siga siendo verdadera. La condición se evalúa antes de cada nueva iteración. Cuando la condición es falsa, se deja de ejecutar. A diferencia del bucle for, este se utiliza cuando no sabes cuantas veces se va a ejecutar, porque la condición va cambiando. En este caso se coloca un contador antes de comenzar con el bloque del bucle. Vamos a imprimir los números del 0 al 5:

let i = 0;

while (i < 6) {

console.log(i);

i++;

}

0

1

2

3

4

5

Como se puede observar, es muy similar al bloque for anterior para imprimir estos números, con la diferencia de el contador se inicia fuera y antes del bucle while y el incremento se realiza dentro del cuerpo.

Bucle do…while:

Este bucle es muy similar al while, pero mientras en el while primero se mira la condición, en do…while siempre se ejecuta mínimo una vez porque la condición se evalua después de la primera ejecución. Es decir, en un bucle while, se podría ver una condición falsa y no recorrer ni una sola vez el bucle, pero en do…while siempre se va a hacer una primera ejecución. En este caso, la sintáxis es:

let i = 0;

do {

console.log(i);

i++;

} while (i < 6)

0

1

2

3

4

5

**¿Cuáles son las diferencias entre const, let y var?**

const, let y var son palabras clave que se usan para declarar variables. A pesar de que todas cumplen esta misma función, tienen algunas diferencias que vamos a ver a continuación.

Var

var es la forma más antigua de declarar una variable.

Ambito de uso: El uso de var da alcance de función a la variable. Esto significa que, en una función, al crear una variable con var, hace que sea accesible desde cualquier parte de la función sin tener en cuenta otros bloques como por ejemplo los bucles o condicionales. Es decir, aunque se declare dentro de uno de estos bloques, al salir del bloque (dentro de la función) sigue siendo accesible. Por ejemplo:

function nombreVar() {

if (true) {

var nombre = "Pepito"; //Dentro del bloque if

console.log(nombre); //Imprime Pepito

}

console.log(nombre); //Fuera del bloque if, también imprime Pepito

}

nombreVar();

console.log(nombre); // ReferenceError poque está fuera de la función

Cuando una variable se declara fuera de una función, es decir, en un ámbito global, la variable está disponible en todos los lados.

Además, estas declaraciones tienen hoisting, lo que significa que puedes usar la variable antes de declararla sin que muestre un error. Otra característica que tiene es que se puede cambiar su valor fácilmente sin causar errores:

console.log(nombre); //Es indefinido, pero no muestra error por hoisting

var nombre = "Pepito";

console.log(nombre);

var nombre = "Antonio";

console.log(nombre);

undefined

Pepito

Antonio

El mayor problema con el uso de la declaración con var es que al ser tan sencillo redefinir su valor, se puede hacer sin darte cuenta en otro punto del script, lo que puede generar muchos errores. Eso hace que let y cost sean necesarios.

Let:

A diferencia de var, let tiene alcance de bloque. Al declarar una función con let, la variable sólo es accesible desde el bloque en el que se encuentra, dentro de las llaves donde se ha creado. Es decir, si declaro la variable dentro de un bloque condicional e intento imprimirla una vez finalizado ese bloque, aunque esté en la misma función la variable no va a existir, ya que solo existe en el bloque declarado.

function ejemploLet() {

if (true) {

let nombre = "Pepito";

console.log(nombre); // Imprime Pepito

}

console.log(nombre); // ReferenceError: nombre is not defined

}

A diferencia de let, a pesar de que se elevan a la parte superior, no puede ser llamada antes de declararla debido a que mostraría un error de referencia en vez de retornarnos un undefined.

Otra diferencia es que, aunque sí se muede modificar, no puedes redeclarar la variable en el mismo ámbito. Esta función hace que let sea una opción recomendable para poder evitar posibles nuevas declaraciones de la misma variable. Pero si se le puede reasignar un valor. Para explicar esto, vamos a poner un ejemplo:

Esto sí que se puede hacer:

let nombre = “Pepito”;

nombre = “Juan”;

Pero esto va a retornar SyntaxError: Identifier 'nombre' has already been declared.

let nombre = "Pepito";

let nombre = "Juan";

A pesar de ello, en diferentes ámbitos sí que se puede definir sin tener errores, como por ejemplo:

let nombre = "Pepito";

if (true) {

let nombre = "Juan";

console.log(nombre); // Imprime Juan

}

console.log(nombre); // Imprime Pepito

Const:

La declaración de variable usando const es muy similar a let. Esta forma también es exclusiva de bloque y además no puede ser llamada antes de ser declarada. Es decir, al igual que con let, con const solo se pueden acceder a las declaraciones dentro del bloque donde se encuentra declarado.

La diferencia es que en este caso no se pueden ni reasignar su valor ni volver a declararla. Lo que significa que no se puede realizar ninguna actualización, por ejemplo, no se puede hacer ninguna de estas dos:

const nombre = "Pepito";

nombre = "Juan";

console.log(nombre) // TypeError: Assignment to constant variable.

const nombre = "Pepito";

let nombre = "Juan";

console.log(nombre) // SyntaxError: Identifier 'nombre' has already been declared

A pesar de ello, cuando se trata de objetos, las propiedades de éstos sí se pueden actualizar:

const persona = {

nombre: "Pepito",

apellido: "Martinez",

}

//Esto no se puede hacer

persona = {

nombre: "Marta",

apellido: "Rodriguez",

}

//Esto sí se puede hacer para actualizar el valor de nombre sin error

persona.nombre = "Marta";

Respecto al hoisting, como let, se elevan pero no se inicializan, es decir, tampoco se pueden llamar antes de ser declaradas.

**¿Qué es una función de flecha?**

Una función es un bloque de códigos que realiza una tarea específica. Son instrucciones reutilizables que se ejecutan al llamar o invocar la función. Es decir, cuando se va a hacer una tarea de forma repetitiva en el código, en vez de repetir todo el rato las instrucciones, se reutilizan las creadas dentro de la función.

Las funciones flechas son una forma de declarar funciones que se han vuelto populares en el JS moderno debido a su sintaxis más simple. Son una forma concisa de escribir expresiones de función. A diferencia de las declaraciones de función más tradicionales, las funciones flecha son no tienen un nombre identificador en su propia declaración, se nombran al asignarlas a una variable. Al igual que las expresiones de función tradicionales, no tienen hoisting, es decir, no se pueden llamar antes de definirlas en el código. A diferencia de las declaraciones o expresiones de función regulares que usan la palabra clave function, las funciones de flecha usan el operador de flecha “=>”. Estas funciones siempre se almacenan en variables o se pasan directamente como argumento y su estructura base es la siguiente:

let nombreVariable = (parametros de la función) => {

//cuerpo de la función a ejecutar

};

Por ejemplo, si queremos una función que diga buenos días:

let buenosDias = () => {

    console.log("¡Buenos días!");

};

buenosDias();

¡Buenos días!

En este caso, como la función no toma argumentos, el paréntesis se queda en blanco.

Para ver un ejemplo de una función de flecha con argumentos, vamos a crear una función de saludo que tome de argumento el nombre y el apellido de la persona que va a saludar:

let saludo = (nombre, apellido) => {

    console.log(`¡Hola ${nombre} ${apellido}!`);

};

saludo("Marta", "Rodríguez");

¡Hola Marta Rodríguez!

Además de la sintaxis, una diferencia muy importante de estas funciones en comparación con las funciones tradicionales es el manejo de this.

En las funciones tradicionales, el valor de this no se define cuando se escribe la función se define al ejecutarla. Esto hace que this sea dinámico y va a depender de la forma en la que se llama la función.

En el siguiente ejemplo se ve una función normal que va a ser llamada como un método de un objeto. Vamos a hacer otra función de saludo. En esta situación, this se refiere al objeto:

const persona = {

    nombre: "Marta",

    apellido: "Rodríguez",

    saludo: function() {

        console.log(`¡Hola ${this.nombre} ${this.apellido}!`);

    }

};

persona.saludo();

¡Hola Marta Rodríguez!

En cambio, en las funciones flecha el valor de this se mantiene fijo y es el mismo que el del entorno de donde se ha creado la función. Si por ejemplo intentásemos realizar la misma función anterior, pero esta vez con una función flecha:

const persona = {

    nombre: "Marta",

    apellido: "Rodríguez",

    saludo: () => {

        console.log(`¡Hola ${this.nombre} ${this.apellido}!`);

    }

};

persona.saludo();

¡Hola undefined undefined!

Como se puede observar, this no se comporta de la misma manera. Como no está apuntando al objeto de persona, sino al contexto global de donde se ha creado persona, aparece como undefined (no definido).

El uso común de this en las funciones flecha es cuando actúan como calbacks o con métodos que ya tienen this definido y se quiere conservar.

const persona = {

    nombre: "Marta",

    apellido: "Rodríguez",

    saludo: function()  {

        console.log(`¡Hola ${this.nombre} ${this.apellido}!`);

        setTimeout(() => {

            console.log(`¡Adiós ${this.nombre} ${this.apellido}!`);

        }, 1000);

    }

};

persona.saludo();

¡Hola Marta Rodríguez!

// 1 segundo después….

¡Adiós Marta Rodríguez!

En este caso tenemos un añadido a la función anterior. Como ejemplo la primera función muestra el ejemplo anterior de this en una función simple y posteriormente, en un timeout se muestra el uso de this en una función flecha. Esta función lo que hace es que posteriormente a imprimir el primer apartado del saludo, espera 1 segundo e imprime la despedida fijada dentro de la función flecha de setTimeout. Al estar dentro del contexto global, el uso de this se hereda del this de persona, por lo que aquí no nos saldría indefinido.

**¿Qué es la deconstrucción de variables?**

La deconstrucción de variables es una foma de extraer arrays o propiedades de los objetos y asignarlos a variables de forma individual. Vamos a seguir con el objeto persona anterior como ejemplo y vamos a ver cómo se extraen los datos con y sin deconstrucción de objetos:

Esta sería la forma de hacerlo sin usar la deconstrucción de variables:

const persona = {

    nombre: "Marta",

    apellido: "Rodríguez",

};

const nombreUno = persona.nombre;

const apellidoUno= persona.apellido;

console.log(nombreUno, apellidoUno);

Marta Rodríguez

Y esta otra con la deconstrucción:

const persona = {

    nombre: "Marta",

    apellido: "Rodríguez",

};

const {nombre, apellido} = persona

console.log(nombre, apellido);

Marta Rodríguez

De esta forma se extraen de forma más concisa tanto el nombre como el apellido. Además, no solo se pueden extraer, sino que también se les puede asignar nuevos nombres de variables

**¿Qué hace el operador de extensión en JS?**

El operador de extensión (…), como su nombre indica, permite expandir elementos iterables (arrays, strings, objetos) en lugares donde se esperan cero o más argumentos, o elementos (para literales de array) o pares clave-valor (para literales de objeto). Facilita mucho la manipulación y combinación tanto de arrays como de objetos, sin usar otros métodos como por ejemplo concat().

**Ejemplos de uso en arrays:**

- Para copiar arrays sin modificar el original:

const arrayOriginal = [1,2,3];

const arrayCopia = [...arrayOriginal];

console.log(arrayCopia); // [1,2,3]

- Para combinar arrays en una nueva:

const array1= [1,2,3];

const array2= [4,5,6];

const arrayCombinado = [...array1,...array2];

console.log(arrayCombinado); // [1,2,3,4,5,6]

- Para añadir elementos a un array existente:

const array1 = [1, 2, 3];

const array2 = [0, ...array1, 4];

console.log(array2); // [0, 1, 2, 3, 4]

**Ejemplos de uso en objetos:**

- Copiar objetos:

const objetoOriginal = { a: 1, b: 2 };

const objetoCopia = { ...objetoOriginal };

console.log(objetoCopia); // { a: 1, b: 2 }

**- Combinar objetos:**

const objeto1 = { a: 1, b: 2 };

const objeto2 = { b: 3, c: 4 };

const objetoCombinado = { ...objeto1, ...objeto2 };

console.log(objetoCombinado); // { a: 1, b: 3, c: 4 }

**¿Qué es la programación orientada a objetos?**

La programación orientada a objetos, también conocida como POO, es una forma de programar pensado en el conjunto de objetos que interactúan entre sí. Es decir, el diseño del programa se crea alrededor de objetos y no de funciones ni otros tipos de lógica. Un objeto es una instancia de una clase que puede contener propiedades o atributos y métodos.

Las clases son como unas plantillas que se usan para crear los objetos. Aquí se definen las propiedades y los métodos del objeto.

Vamos a ver varios ejemplos de estas definiciones:

// Clase: El plano para crear objetos de, en este caso, Persona

class Persona {

constructor(nombre, apellido) {

this.nombre = nombre;

this.apellido = apellido;

}

saludar() {

console.log(`¡Hola ${this.nombre} ${this.apellido}!`);

}

}

// Objetos: Instancias de la clase Persona

const persona1 = new Persona("Ana", "Bilbao");

const persona2 = new Persona("Pepito", "Rodríguez");

persona1.saludar(); // Esto imprime: ¡Hola Ana Bilbao!

persona2.saludar(); // Esto imprime: ¡Hola Pepito Rodríguez!

Además de las clases y los objetos, POO tiene cuatro pilares principales:

1. Abstracción: Simplifica algo complejo definiendo lo que hace un objeto. Se enfoca en qué hace un objeto y no tanto en cómo lo hace.
2. Encapsulación: Esto es el principio de agrupar las propiedades y funciones (métodos) dentro del mismo objeto o de la clase. Aquí se permite que se exponga solo lo que se quiere mostrar, ocultando los detalles internos y el funcionamiento complejo a quienes no necesitan verlo.
3. Herencia: Esto permite que una clase herede las propiedades y métodos de otra clase, a la que se le llama clase padre (o superclase). Así se puede reutilizar código y se crean relaciones de jerarquía entre las clases (por ejemplo, "un gato es un animal").

class Animal {

 constructor(nombre) {

 this.nombre = nombre;

 }

 hacerSonido() {

 console.log('El animal hace un sonido');

 }

}

//Clase derivada de la clase Animal

class gato extends Animal {

    constructor(nombre, raza) {

        super(nombre);

        this.raza = raza;

    }

}

const miGato = new gato('Kiwi', 'persa');

miGato.hacerSonido();

console.log(`Mi gato se llama ${miGato.nombre} y es de raza ${miGato.raza}.`); // Mi gato se llama Kiwi y es de raza Persa.

1. Polimorfismo. Esto permite que objetos que pertenecen a distintas clases puedan responder a un mismo método de distintas maneras. Por ejemplo, si creásemos otra clase llamada Perro que ladrase en hacerSonido() y el Gato maullase, los dos actuarían de forma distinta ante un mismo método, esto es el polimorfismo.

A poster of a program

AI-generated content may be incorrect.

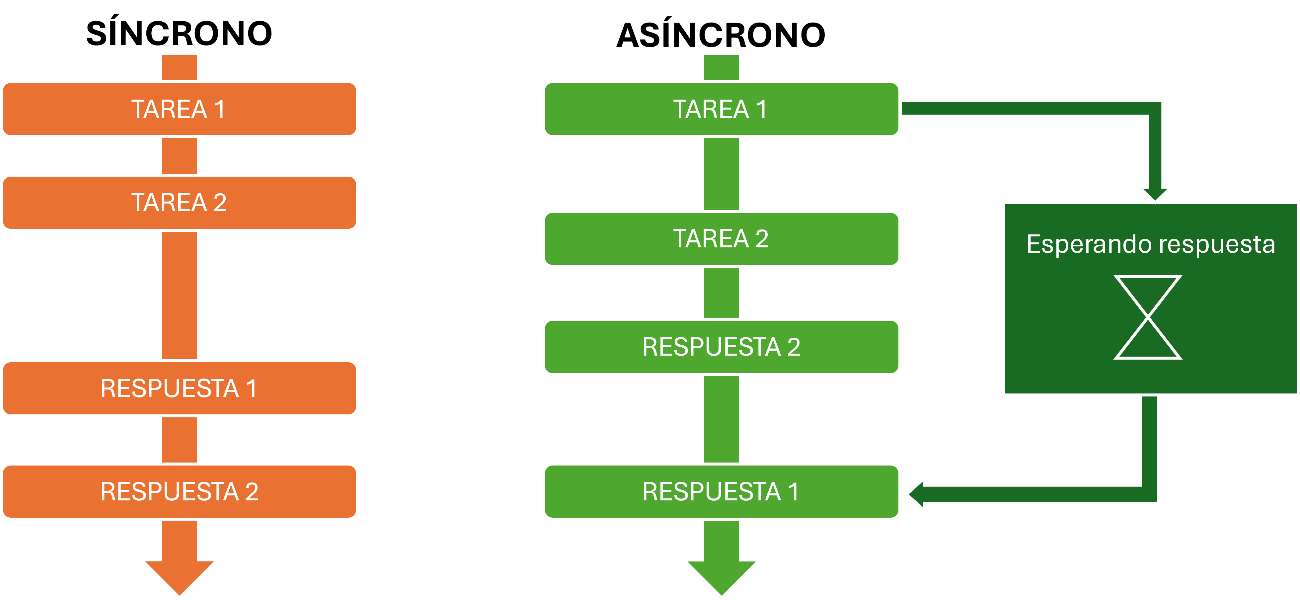
Imagen: https://desarrollodesoftware.dev/c-software-para-el-desarrollo-de-aplicaciones-orientada-a-objetos

**¿Qué es una promesa en JS?**

En JS, promesa (promise) es un objeto que representa la finalización o fallo de una operación asíncrona. Estas permiten escribir el código asíncrono de forma estructurada y fácil de leer, evitando el anidamiento excesivo, que también se llama callback hell (infierno de callbacks).

Asincronía

Primero, vamos a recordar qué significa la asincronía. Cuando el programa encuentra una tarea que está tardando mucho, en vez de quedarse esperando a que termine y bloquearse, el programa sigue ejecutando otras cosas mientras tanto y cuando finaliza la tarea lenta, se ejecuta una función llamada callback (llamada de vuelta) para manejar el resultado.



Estados de promesa

Una promesa puede tener tres estados:

Pentiente (pending): Estado inicial en el que la operación asíncrona no ha finalizado.

Cumplida (fullfilled o resolved): Aquí la operación asíncrona se ha cumplido exitosamente y la promesa devuelve un valor.

Rechazada (rejected): En este caso, la operación asíncrona ha fallado y la respuesta de la promesa es la razón del error que ha sucedido.

Cuando la promesa pasa al estado de cumplida o rechazada, se considera que ya se ha resuelto y su estado no va a volver a cambiar.

Métodos principales

En las promesas, hay dos métodos principales:

.then(): Se ejecuta al cumplirse la promesa. Se usa para realizar operaciones sucesivas cuando la promesa se resuelve

.catch(): Se ejecuta al rechazarse la promesa. Se usa para el manejo de errores.

Ejemplos

1. Ejemplo básico. Vamos a imaginar que estamos haciendo un bizcocho y lo hemos metido en el horno. Estamos esperando a que se termine de hornear, pero no sabemos si va a salir bien o se va a quemar. Vamos a ver esta situación en una promesa:

const bizcocho = new Promise((resolve, reject) => {

    const exito = true;

    if (exito) {

        resolve("El bizcocho está perfecto.");

    } else {

        Reject("Oh no, se ha quemado el bizcocho.");

    }

});

bizcocho

    .then(resultado => {

        console.log(resultado);

    })

    .catch(error => {

        console.error(error);

    });

// Mientras sea true, output = El bizcocho está perfecto.

// Si cambia a false, output = Oh no, se ha quemado el bizcocho.

1. Ahora vamos a ver un ejemplo que tenga una operación asíncrona con setTimeout. Aquí por ejemplo el setTimeout sería lo que el bizcocho tarda en hornearse. Vamos a darle una probabilidad de éxito de conseguir el bizcocho perdecto del 75%.

const hornearBizcocho = new Promise((resolve, reject) => {

    setTimeout(() => {

        const exito = Math.random() > 0.25; //Aquí está la probabilidad del 75%

        if (exito) {

            resolve("Después de esperar, el bizcocho está perfecto.");

        } else {

            reject("Oh no, después de esperar el bizcocho está quemado.");

        }

    }, 2000); // Le damos 2 segundos

});

console.log("El bizcocho está en el horno..."); //Este mensaje aparece primero

hornearBizcocho

    .then(resultado => {

        console.log(resultado);

    })

    .catch(error => {

        console.error(error);

    });

// Como es al azar, al esperar 2 segundos la consola puede retornar cualquiera de estas:

// Si exito = true: "¡El bizcocho salió perfecto después de esperar!"

// Si exito = false: "¡Se quemó el bizcocho tras esperar!"

**¿Qué hacen async y await por nosotros?**

Las palabras clave async y await se usan para escribir el código asíncrono de forma en la que parezca síncrona para que el código sea más fácil de leer y se simplifique el manejo de las promesas. Vamos a ver lo que significa cada una:

* async: Se usa antes de una función para decir que se va a devolver una promesa y esto permite poder usar la palabra await dentro de la función.
* await: Se usa dentro de las funciones async para pausar la ejecución mientras la promesa se está resolviendo (tanto resuelta como rechazada).

Gracias a estas palabras, no hace falta usar múltiples encadenamientos con .then() y .catch(). Esto permite que el código sea más limpio y comprensible para el mantenimiento y manejo de errores. Mientras el código esté en espera en una operación asíncrona de async/await, se pueden ir ejecutando otras tareas.

Vamos a continuar con el mismo ejemplo del bizcocho en el horno para ver el uso de async y await. En este caso, en lugar de usar .then() y .catch(), se usan try y catch.

function hornearBizcocho() {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            const exito = Math.random() > 0.3; // 70% de éxito

            if (exito) {

                resolve("Después de esperar, el bizcocho está perfecto.");

            } else {

                reject("Oh no, después de esperar el bizcocho está quemado.");

            }

        }, 2000);

    });

}

//Se comienza con async al principio de la función

async function prepararBizcocho() {

    console.log("El bizcocho está en el horno.");

    try {

        const resultado = await hornearBizcocho();

        console.log(resultado);

    } catch (error) {

        console.error(error);

    }

}

prepararBizcocho();

En este código, al haber comenzado la función de prepararBizcocho declarando la función con async, se permite usar await. Cuando se menciona await hornearBizcocho(), el código lo mantiene en espera hasta resolver o rechazar la promesa, pero no se bloquea el código. Al igual que anteriormente, primero se imprime “El bizcocho está en el horno.”. Si se cumple la promesa, al de dos segundos nos devolvería “Después de esperar, el bizcocho está perfecto.”, pero si se rechaza, obtendríamos “Oh no, después de esperar el bizcocho está quemado.”

**Y por último debes realizar el siguiente ejercicio práctico, y subirlo a tu repositorio en Git-Hub para revisarlo**

**-Cree un bucle for en JS que imprima cada nombre en esta lista. miLista =**[**“velma”, “exploradora”, “jane”, “john”, “harry”**](https://support.devcamp.com/tickets)

**-Cree un bucle while que recorra la misma lista y también imprima los nombres. Nota: Recuerda crear un contador para que el ciclo no sea infinito.**

**-Cree una función de flecha que devuelva "Hola mundo".**

**Esta es toda la asignación, ¡mucha suerte!**