ASINCRONISMO EN JAVASCRIPT

Charly Cimino

Asincronismo en JavaScript

Charly Cimino

Versión 2023.08.06

Antes de empezar...

Los ejemplos que veremos en esta PPT han sido probados en Google Chrome, con un script embebido.

prueba.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <script type="text/javascript">
        // El código JS va aquí...
    </script>
</head>

<body></body>

</html>
```

Temas necesarios para entender la PPT

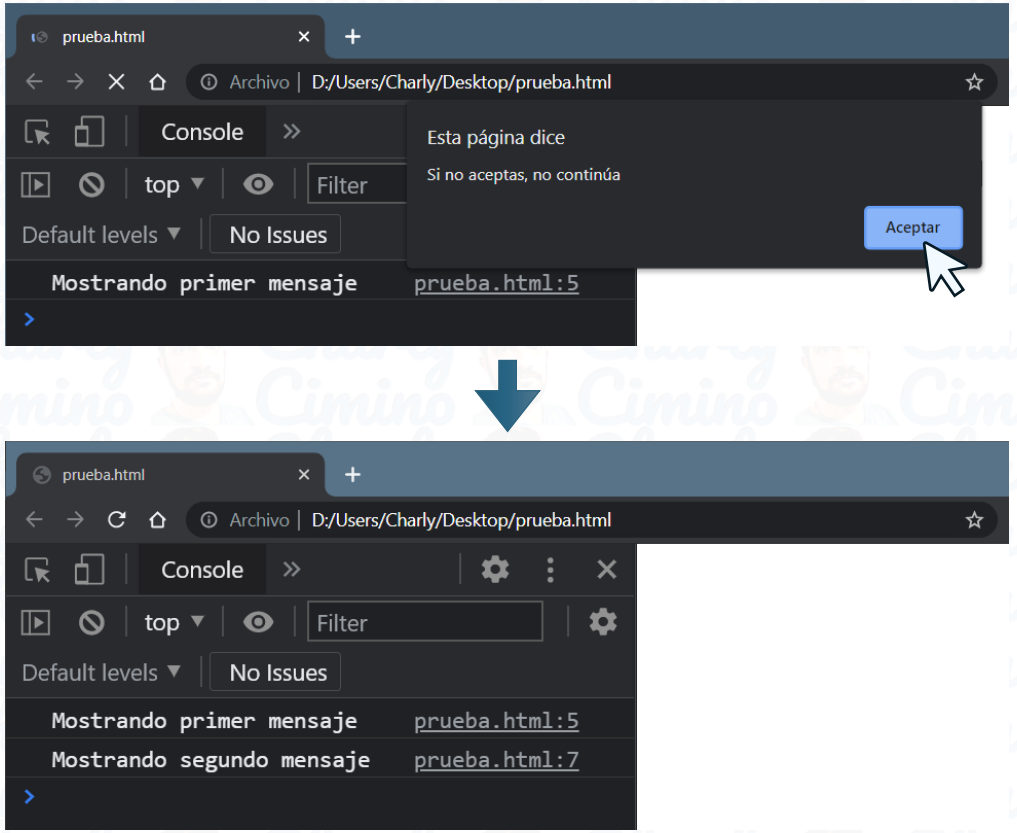
- Objetos literales.
- Funciones como elementos de primera clase.
- `let` / `const`.
- Plantillas literales.
- Funciones flecha.
- Función `reduce` de arrays.
- Concepto de JSON.

Muchos de los temas necesarios para entender asincronismo se encuentran en la PPT JavaScript Contemporáneo.

JavaScript es *single-threaded*

El motor de JavaScript dispone de un solo *thread* de ejecución, por lo que es fácil generar bloqueos.

```
console.log("Mostrando primer mensaje");
alert("Si no aceptas, no continúa");
console.log("Mostrando segundo mensaje");
```



https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts#javascript_is_single-threaded

Código bloqueante

Las operaciones que dependen del CPU son bloqueantes y síncronas.
No puede ejecutarse más de una tarea al mismo tiempo, salvo con la ayuda de Web Workers,
que permitirían usar otro *thread* (no cubiertos en esta PPT).

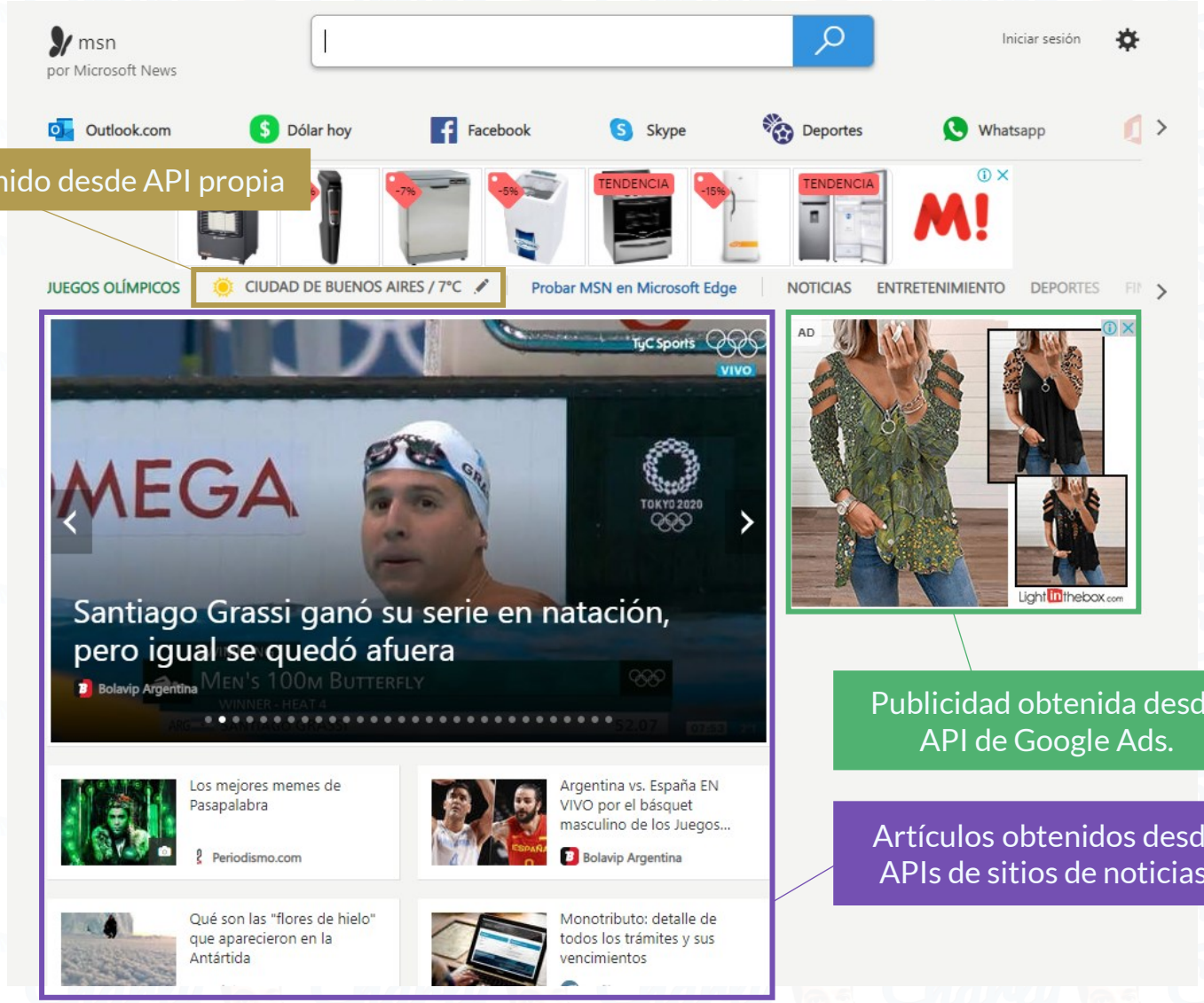
```
function tareaCostosaCPU() {
    for (let i = 0; i < 2000000; i++) {
        new Date().toString().toUpperCase();
    } // 2 millones de ciclos para simular carga
    return "elResultado";
}

const data = tareaCostosaCPU();
console.log(data);
console.log("Otra operación...");
```

Tarda unos cuantos segundos

Esperando que termine la función

https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

La web de hoy en día



Clima obtenido desde API propia

Publicidad obtenida desde API de Google Ads.

Artículos obtenidos desde APIs de sitios de noticias.

https://www.msn.com/es-ar

Las apps y sitios web modernos se integran con multitud de servicios externos.

¿Cómo se hace para que tales peticiones no bloqueen la carga del resto del contenido?

Ejecución síncrona

Si las peticiones se ejecutaran de forma síncrona, tardaría mucho más en cargarse completamente.


```
const mostrarNoticias = () => {
    console.log("Obtenidas noticias desde APIs externas...");
}

const mostrarClima = () => {
    console.log("Obtenido clima desde API propia...");
}

const mostrarPublicidad = () => {
    console.log("Obtenida imagen desde Google Ads...");
}

mostrarNoticias();
mostrarClima(); // Debe esperar a mostrarNoticias()
mostrarPublicidad(); // Debe esperar a mostrarClima()
```

Mientras no termine una operación, no se devuelve el control a la CPU.

```
Obtenidas noticias desde APIs externas...   codigo.js:24
Obtenido clima desde API propia...          codigo.js:27
Obtenida imagen desde Google Ads...         codigo.js:30
>
```

Los resultados se obtienen en el orden previsto.

https://msl-network.readthedocs.io/en/latest/concurrency_async.html

Callback

Es una función que será invocada por otra a modo de notificación cuando termine su tarea.

`setTimeout` es una función que establece un retardo en milisegundos.
El primer parámetro es la referencia a una función (el ***callback***), que será invocada cuando transcurran los milisegundos indicados como segundo parámetro.

```
const realizarAccion = () => {
    console.log("Acción realizada");
}

setTimeout(realizarAccion, 3000);
console.log("Otra acción");
```

Función que oficia de ***callback***.

```
Otra acción                codigo.js:6
>
```

3 segundos luego...

```
Otra acción                codigo.js:6
Acción realizada           codigo.js:2
>
```

Muchas veces, los *callbacks* se definen "*in situ*"

```
setTimeout(() => {
    console.log("Acción realizada");
}, 3000);
console.log("Otra acción");
```

https://lenguajejs.com/javascript/asincronia/callbacks/#callbacks-en-javascript

Ejecución asíncrona

Simulamos asincronismo con la función `setTimeout` que genera un retraso en la ejecución.

```
const mostrarNoticias = () => {
    setTimeout(() => {
        console.log("Obtenidas noticias desde API propia...");
    }, 7000); // Simulamos 7 segundos de demora
}

const mostrarClima = () => {
    setTimeout(() => {
        console.log("Obtenido clima desde weather-api...");
    }, 3000); // Simulamos 3 segundos de demora
}

const mostrarPublicidad = () => {
    setTimeout(() => {
        console.log("Obtenida imagen desde web externa");
    }, 5000); // Simulamos 5 segundos de demora
}

mostrarNoticias();   // #3
mostrarClima();      // #1
mostrarPublicidad(); // #2
```

Los resultados se obtienen a medida que se van resolviendo las peticiones.

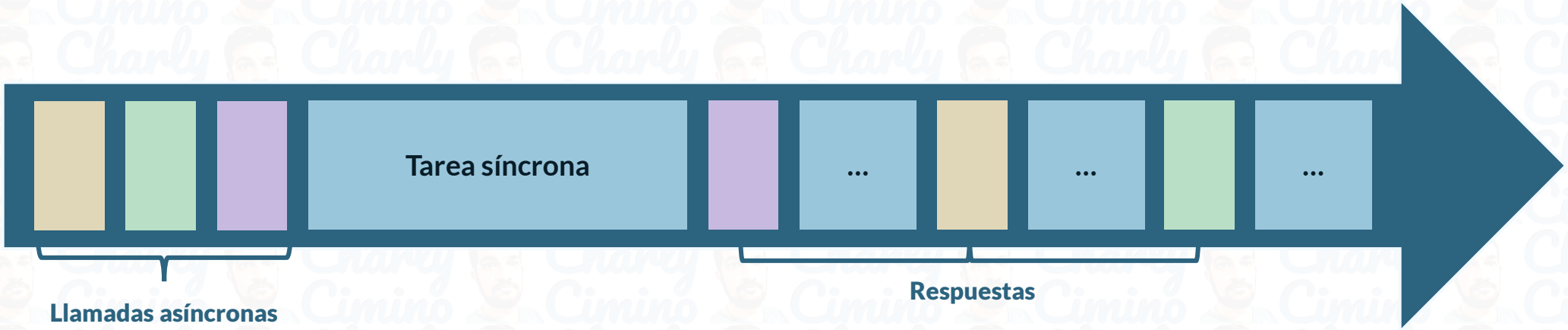
En total la demora fue de 7 segundos, en lugar de 15.

https://msl-network.readthedocs.io/en/latest/concurrency_async.html

Asincronismo ≠ Paralelismo

JavaScript tiene un solo hilo de ejecución.

No puede ejecutarse más de una tarea al mismo tiempo, salvo con la ayuda de Web Workers.



El asincronismo consiste en continuar con el resto de las operaciones, sin detenerse a esperar la respuesta de una operación externa. Cuando esté disponible, se procesará.

https://eloquentjavascript.net/11_async.html

Analogía con la cocina

Queremos cocinar milanesas con puré de papas.
Como cocineros solo tenemos dos manos. Podemos hacer una sola tarea a la vez.
Si quisiéramos hacer más de una al mismo tiempo, necesitamos ayudantes (analogía con los Web Workers).

Lo que sí podemos hacer es trabajar mientras otras cosas, que no dependen de nuestras manos, suceden en segundo plano (calentar aceite, hervir agua, descongelar algo en el microondas, etc.)








Poner agua a hervir...

¡Agua hirviendo!

Poner aceite a calentar...

¡Aceite caliente!

Operaciones no bloqueantes

Las operaciones de entrada/salida ocurren de forma asíncrona, por ende, no bloquean la ejecución.

Algunas formas de generar asincronismo

Función `setTimeout`	Función `setInterval`	Eventos
Peticiones HTTP	Acceso a BD	Leer y escribir en un archivo

Si JS es *single-thread*, ¿cómo es que estas operaciones corren en segundo plano?

Porque ocurren en el ámbito de las APIs implementadas por los navegadores junto al sistema operativo. Nos abstraemos.

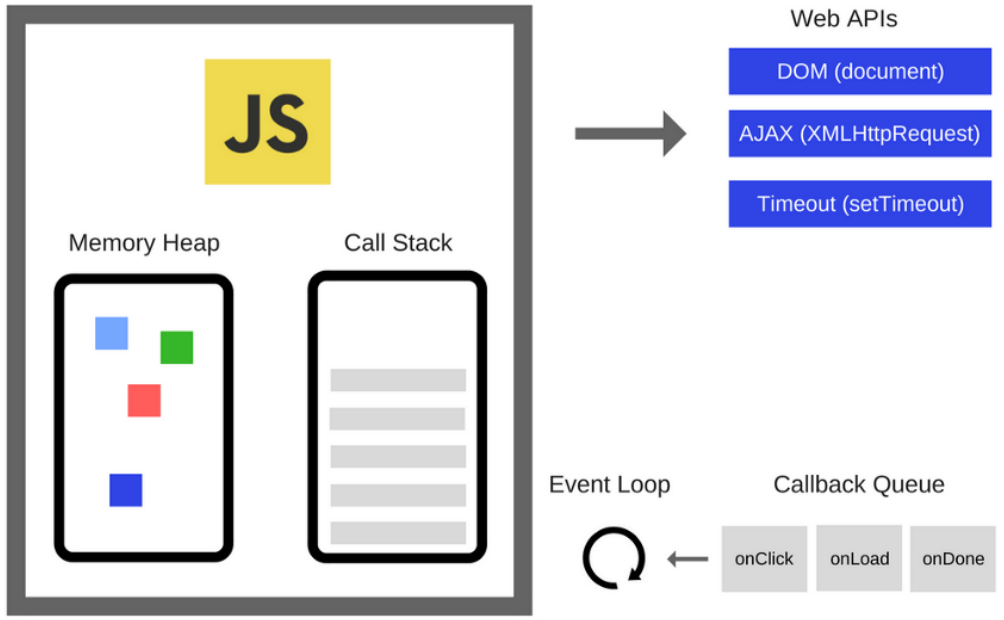
https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

Cómo funciona el asincronismo en JS

1. Se apila `console.log("Op. sync #1")`
2. Se imprime "Op. sync #1"
3. Se desapila `console.log("Op. sync #1")`
4. Se apila `setTimeout(function callback() {...}`
5. Se ejecuta `setTimeout`. El browser delega el control de tiempo al timer. No hay nada más que hacer.
6. Se desapila `setTimeout(function callback() {...})`
7. Se apila `console.log("Op. sync #3")`
8. Se imprime "Op. sync #3"
9. Se desapila `console.log("Op. sync #3")`
10. Tras dos segundos, el timer finaliza y pone el `callback()` en cola.
11. El Event Loop toma el primer elemento de la cola y lo apila.
12. Se ejecuta `callback()`, por ende, se apila `console.log("Op. async #2")`
13. Se imprime "Op. async #2“
14. Se desapila `console.log("Op. async #2")`
15. Se desapila `callback()`

Ver GIF

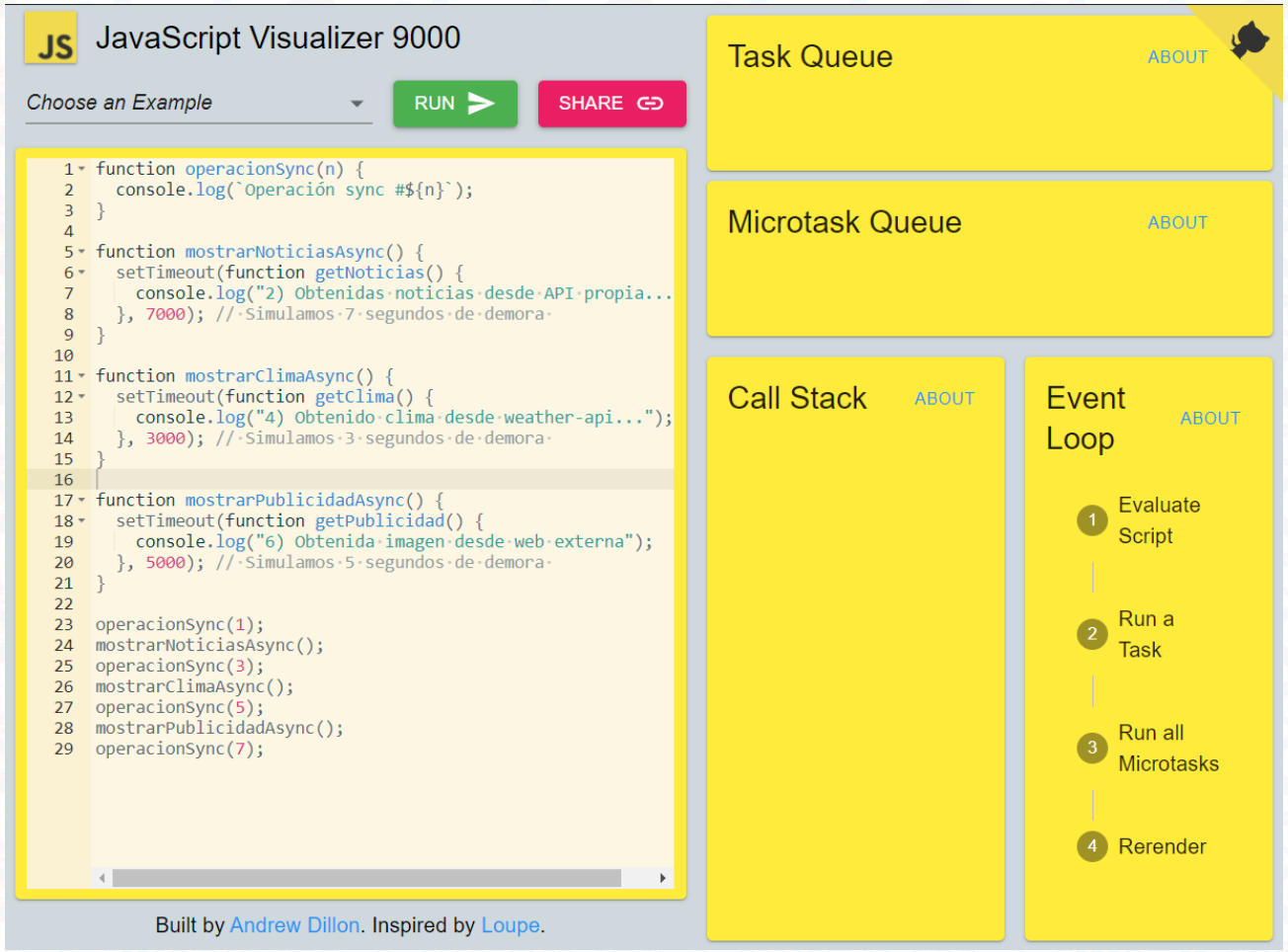
```
console.log("Op. sync #1");

setTimeout(function callback() {
    console.log("Op. async #2");
}, 2000); // 2 segundos de demora

console.log("Op. sync #3");
```



Paso a paso

En esta app web se puede comprobar de forma gráfica e interactiva cómo funciona paso a paso el *Event Loop* de JavaScript que gestiona el asincronismo.

```javascript
function operacionSync(n) {
    console.log(`Operación sync #${n}`);
}

function mostrarNoticiasAsync() {
    setTimeout(function getNoticias() {
        console.log("2) Obtenidas noticias desde API propia...");
    }, 7000); // Simulamos 7 segundos de demora
}

function mostrarClimaAsync() {
    setTimeout(function getClima() {
        console.log("4) Obtenido clima desde weather-api...");
    }, 3000); // Simulamos 3 segundos de demora
}

function mostrarPublicidadAsync() {
    setTimeout(function getPublicidad() {
        console.log("6) Obtenida imagen desde web externa");
    }, 5000); // Simulamos 5 segundos de demora
}

operacionSync(1);
mostrarNoticiasAsync();
operacionSync(3);
mostrarClimaAsync();
operacionSync(5);
mostrarPublicidadAsync();
operacionSync(7);
```



https://www.jsv9000.app/

Primero lo *sync*, luego lo *async*

JavaScript no procesará los *callbacks* de las llamadas asíncronas hasta que no se completen las llamadas síncronas, aun cuando éstas últimas demoren más tiempo.

```
const procesarSync = (ciclos) => {
    for (let i = 0; i < ciclos; i++) {
        new Date().toString().toUpperCase();
    } // Solo produce carga para la CPU
    return `${ciclos} ciclos procesados.`
}

const procesarAsync = () => {
    setTimeout(() => {
        console.log("Op. async");
    }, 0); // No demora nada, pero es async
}

console.log(procesarSync(10));
procesarAsync();
console.log(procesarSync(1000000));
```

Probar ejecución en JSV9000

Mientras no termine `procesarSync(1000000)`, el *callback* de `procesarAsync()` deberá esperar.

```
10 ciclos procesados.
>
```

Varios segundos luego...

```
10 ciclos procesados.
1000000 ciclos procesados.
Op. async
>
```

Esperar por valor *async*

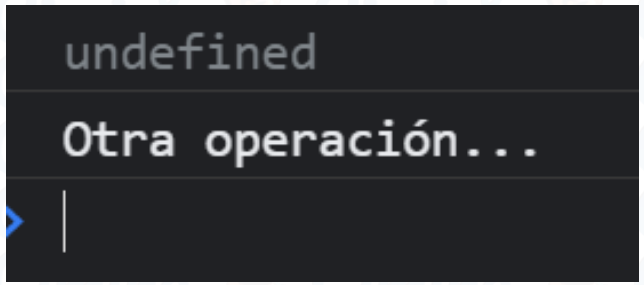
Es muy común requerir datos de forma externa a nuestra app y tener que esperarlos para procesar algún resultado.

¿Cómo aguardamos que concluya la operación para evitar este escenario?

```
const getResultadoAsync = () => {
    let resultado;
    setTimeout(() => {
        resultado = "elResultado";
    }, 5000);
    return resultado;
}

const data = getResultadoAsync();
console.log(data); // undefined
console.log("Otra operación...");
```

Tarda 5 segundos

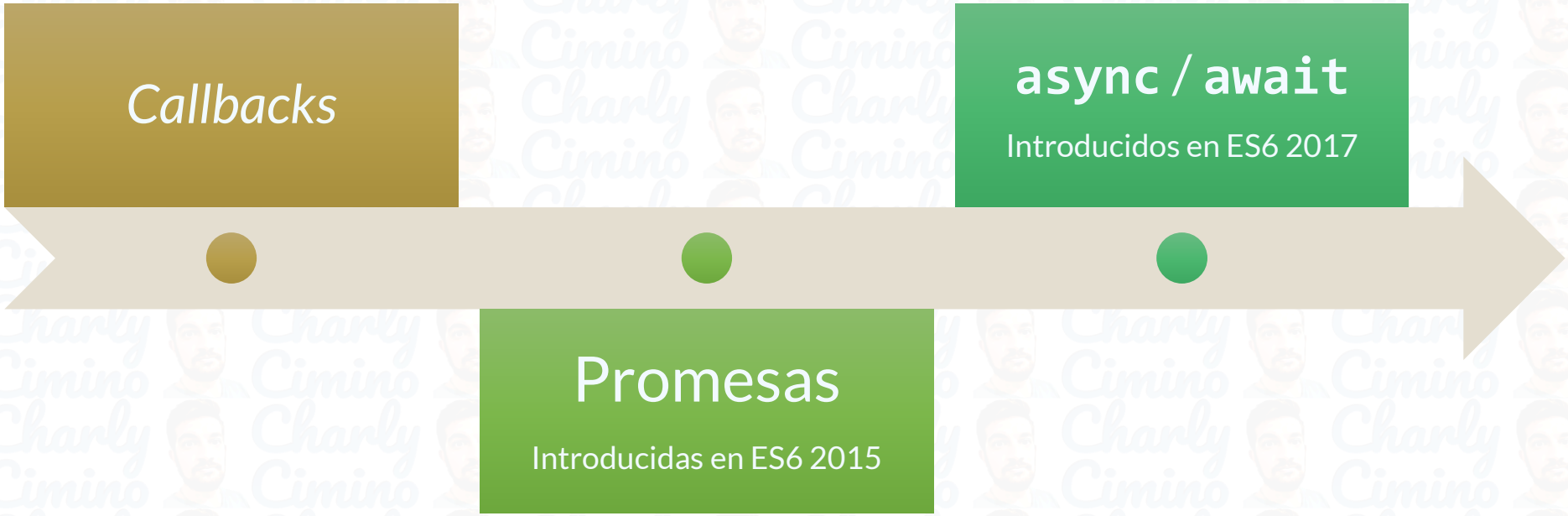
Aún no se sabe



Captura de la consola

Manejar asincronismo en JavaScript

La sintaxis para manejar asincronismo en JavaScript ha ido evolucionando a lo largo del tiempo.



https://anexsoft.com/javascript-callbacks-vs-promise-vs-asyncawait

Asincronismo con *callbacks*

La función `realizarAccionAsync` es quien ejecutará la tarea asíncrona, recibiendo parámetros como cualquier otra, pero en especial también la referencia a una función que oficie de ***callback***.

Cuando se complete la tarea asíncrona, se invocará al ***callback*** con el resultado obtenido.

```
const realizarAccionAsync = (param1, param2, callback) => {
  /*   Se realizan las acciones asíncronas.
       Se invoca a callback() cuando finaliza.
       Por convención:
       Si hubo errores: callback(elError, undefined);
       Si hubo éxito:   callback(null, data);
  */
}

const procesarResultado = (err, res) => {
    /* Usar 'err' o 'res' según sea el caso */
}

realizarAccionAsync("arg1", "arg2", procesarResultado);
// realizarAccionAsync( "arg1", "arg2", procesarResultado() );
```

En general, si existe un error, la respuesta será `undefined`, de lo contrario, será un valor válido.

Por convención, el ***callback*** espera recibir dos valores como parámetros: un error y un resultado, en ese orden.

No poner paréntesis.
Queremos enviar la referencia de la función, no queremos ejecutarla aún.

Nombrar a los parámetros `callback`, `err` o `res` son convenciones/costumbres. Podrían nombrarse de cualquier forma.

https://lenguajejs.com/javascript/asincronia/callbacks/#asincron%C3%ADa-con-callbacks

Simulacro de asincronismo con *callback*

La función `getPersonaByID` simulará una conexión con una BD para buscar un registro por ID y devolverlo.
Si no existe, devolverá un objeto que representa un error.

```
const getPersonaByID = (id, callback) => {
    console.log(`Vamos a buscar ID ${id} en la BD. Paciencia...`);
    setTimeout(() => {
        if (id > 0) {
            const per = { id, nombre: "Pepe" }; // ES6
            // const per = {id: id, nombre: "Pepe"}; //ES5
            callback(null, per);
        } else {
            const unError = {
                error: true,
                msg: `No se encontró persona con ID ${id}`
            }
            callback(unError); // Equivalente: callback(unError, undefined);
        }
    }, 3000); // Demora 3 segundos
}

const procesarPersona = (err, res) => {
    if (err) { // Equivalente: (err != undefined)
        console.error(`ERROR: ${err.msg}`);
    } else {
        console.log("¡Se encontró 1 registro!", res);
    }
}

getPersonaByID(10, procesarPersona);
getPersonaByID(-4, procesarPersona);
```

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:25
Vamos a buscar ID -4 en la BD. Paciencia...   codigo.js:27
>
```

3 segundos luego...

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:25
Vamos a buscar ID -4 en la BD. Paciencia...   codigo.js:27
¡Se encontró 1 registro!                      codigo.js:21
▶ {id: 10, nombre: "Pepe"}
✕ ▶ ERROR: No se encontró persona con ID -4   codigo.js:19
>
```

Uso de XMLHttpRequest para peticiones HTTP

Gracias a objetos de tipo `XMLHttpRequest` de AJAX es posible recibir datos de un servidor de forma asíncrona, sin recargar la página web.

Se crea un objeto de tipo XMLHttpRequest

Se envía la referencia a una función para ser invocada cuando ocurra el evento readystatechange.

```
const xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    // Aquí van las operaciones...
};
xhttp.open("GET", laUrl, true);
xhttp.send();
```

Se envía la solicitud

Inicializa la solicitud enviando qué método HTTP usar (GET, POST, etc.), la URL y un booleano que indica si la operación será asíncrona o no (por defecto, `true`)

Para comprobar que la petición fue correcta

```
xhttp.onreadystatechange = function () {
    if (this.readyState == 4) {
        // https://www.w3schools.com/tags/ref_httpmessages.asp
        if (this.status == 200) {
            // Petición exitosa
            // this.responseText: Respuesta del servidor como String
            // this.responseXML: Respuesta del servidor como XML
            // JSON.parse(this.responseText): Parseado a JSON
        } else {
            // Error en la petición
        }
    } else {
        // Aún no está listo
    }
};
```

https://developer.mozilla.org/es/docs/Web/Guide/AJAX/Getting_Started

APIs de prueba

nationalize.io es un API que predice la nacionalidad en base a un nombre de pila.

```
{
    "name": "manolo",
    "country": [{
            "country_id": "ES",
            "probability": 0.3459614259041762
        },
        {
            "country_id": "EC",
            "probability": 0.15046485347831998
        },
        {
            "country_id": "PE",
            "probability": 0.09927136127621772
        }
    ]
}
```

https://api.nationalize.io/?name=manolo

Más APIs públicas y gratuitas:

https://github.com/public-apis/public-apis

REST Countries es un API que devuelve información sobre países (*JSON de ejemplo recortado*).

```
[
    {
        "name": {
            "common": "Spain",
            "official": "Kingdom of Spain",
            "nativeName": {
                "spa": {
                    "official": "Reino de España",
                    "common": "España"
                }
            }
        },
        "capital": [
            "Madrid"
        ],
        "languages": {
            "spa": "Spanish"
        },
        "flags": {
            "png": "https://flagcdn.com/w320/es.png",
            "svg": "https://flagcdn.com/es.svg"
        }
    }
]
```

https://restcountries.com/v3.1/alpha/es

Uso de *callback* para app de nombre

```
const getRespuestaHttp = (url, callback) => {
    const OK = 200; const DONE = 4;
    const xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function () { // Se usa function para que el this quede ligado
        if (this.readyState == DONE) {
            if (this.status == OK) {
                const resp = JSON.parse(this.responseText);
                callback(null, resp);
            } else
                callback(this.response ? JSON.parse(this.response) : { error: "Desconocido" });
        }
    };
    xhttp.open("GET", url, true);
    xhttp.send();
}

const averiguarPais = nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    getRespuestaHttp(url, (err, res) => { // Esta es la función callback
        if (err)
            alert(`Error: ${err.error}`);
        else {
            let paisMasProb = res.country.reduce((a, b) => {
                    return a.probability > b.probability ? a : b;
            }, 0);
            alert(`País más probable: ${paisMasProb.country_id}`);
        }
    });
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.
Mostramos el ID del país que tenga la máxima probabilidad.

Modificá la `url` para que sea incorrecta y así poder testear cómo se comporta el *callback* ante un error, por ejemplo:

```
const url = `https://api.nationalize.io/?nazme=${nombre}`;
const url = `https://api.nawtionalize.io/?name=${nombre}`;
```

Anidamiento de callbacks

```
const getRespuestaHttp = (url, callback) => {
    const OK = 200; const DONE = 4;
    const xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function () { // Se usa function para que el this quede ligado
        if (this.readyState == DONE) {
            if (this.status == OK) {
                const resp = JSON.parse(this.responseText);
                callback(null, resp);
            } else
                callback(this.response ? JSON.parse(this.response) : { error: "Desconocido" });
        }
    };
    xhttp.open("GET", url, true);
    xhttp.send();
}

const averiguarPais = nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    getRespuestaHttp(url, (err, res) => {
        if (err)
            alert(`Error: ${err.error}`);
        else {
            let paisMasProb = res.country.reduce((a, b) => {
                    return a.probability > b.probability ? a : b;
            }, 0);
            url = `https://restcountries.com/v3.1/alpha/${paisMasProb.country_id}`;
            getRespuestaHttp(url, (err, res) => {
                if (err)
                    alert(`Error: ${err.error}`);
                else {
                    alert(`Probablemente seas de ${res[0].translations.spa.common}`);
                }
            });
        }
    });
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.

Tomamos el ID del país que tenga la máxima probabilidad y obtenemos el nombre del país en español, gracias a la API de REST Countries.

"Callback Hell"

Cuando se anidan llamadas asíncronas, donde cada una depende del resultado de la anterior, se obtiene un código difícil de leer y mantener, conocido como "*Callback Hell*" (Infierno de las devoluciones de llamada) o "*Pyramid of Doom*" (Pirámide del Doom)

```
const averiguarPais = (nombre) => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    getRespuestaHttp(url, (err, res) => {
        if (err)
            alert(`Error: ${err.error}`);
        else {
            let paisMasProb = res.country.reduce((a, b) => {
                    return a.probability > b.probability ? a : b;
            }, 0);
            url = `https://restcountries.com/v3.1/alpha/${paisMasProb.country_id}`;
            getRespuestaHttp(url, (err, res) => {
                if (err)
                    alert(`Error: ${err.error}`);
                else {
                    alert(`Probablemente seas de ${res[0].translations.spa.common}`);
                }
            });
        }
    });
};
```

Esto motivó la creación de las Promesas en ES6

Muchas librerías aún no han sido actualizadas y funcionan solo con *callbacks*.

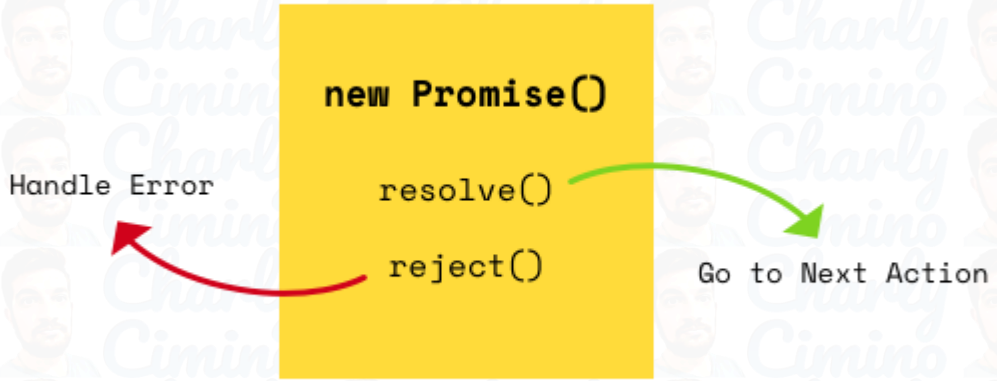
https://blog.nearsoftjobs.com/the-callback-hell-6cc184ce8704

Promesas

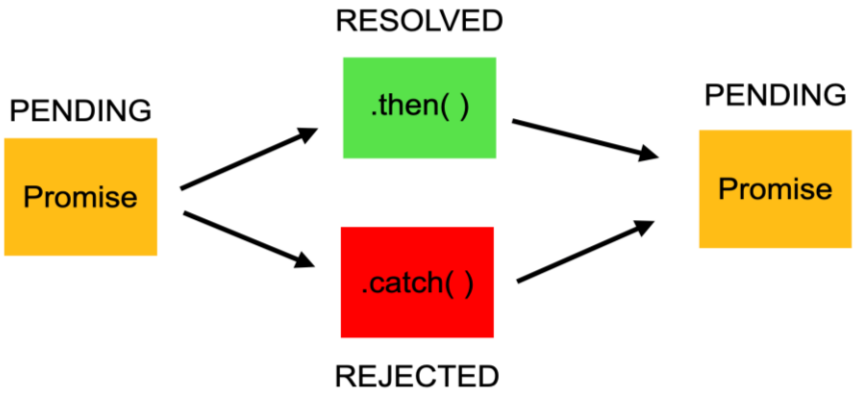
Una promesa es un objeto, que representa el resultado de una operación asíncrona, la cual puede conocerse al instante, a futuro o nunca.

A la hora de crear una promesa, se le envía una función que recibe dos *callbacks*: uno que se llamará en caso de éxito (*resolve*) y otro que se llamará en caso de error (*reject*).

```
new Promise( function(resolveCallback, rejectCallback) { ... } );
```



https://media.vlpt.us/images/edie_ko/post/e32022c2-24fb-49dc-8d93-3e014daeacef/Creating-Promises.png



https://www.freecodecamp.org/news/content/images/2020/06/Ekran-Resmi-2020-06-06-12.21.27.png

A la hora de consumir una promesa, se invoca al método **then** enviando un *callback* para manejar un resultado exitoso y otro para manejar un posible error.

```
laPromesa.then(callbackExito, callbackError);
```

Como **then** devuelve una nueva promesa, se puede obtener el mismo efecto encadenando un llamado al método **catch**, para hacerlo más legible.

```
laPromesa.then(callbackExito).catch(callbackError);
```

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise

Asincronismo con Promesas

La función `realizarAccionAsync` es quien ejecutará la tarea asíncrona, recibiendo parámetros como cualquier otra, pero devolviendo una **promesa** de que habrá pronto una respuesta.

```
const realizarAccionAsync = (param1, param2) => {
    return new Promise((resolve, reject) => {
        /*   Se realizan las acciones asíncronas.
             Si hubo errores: reject(elError);
             Si hubo éxito:   resolve(data);
        */
    });
}

realizarAccionAsync("arg1", "arg2")
    .then(res => {
        // Hacer algo con el resultado
    })
    .catch(err => {
        // Hacer algo con el error
    })
```

La función que recibe ambos *callbacks* y se envía al constructor de la promesa se denomina **ejecutor** (*executor*)

Al método `then` se le envía el *callback* a invocar en caso éxito (*resolve*)

Al método `catch` se le envía el *callback* a invocar en caso de error (*reject*)

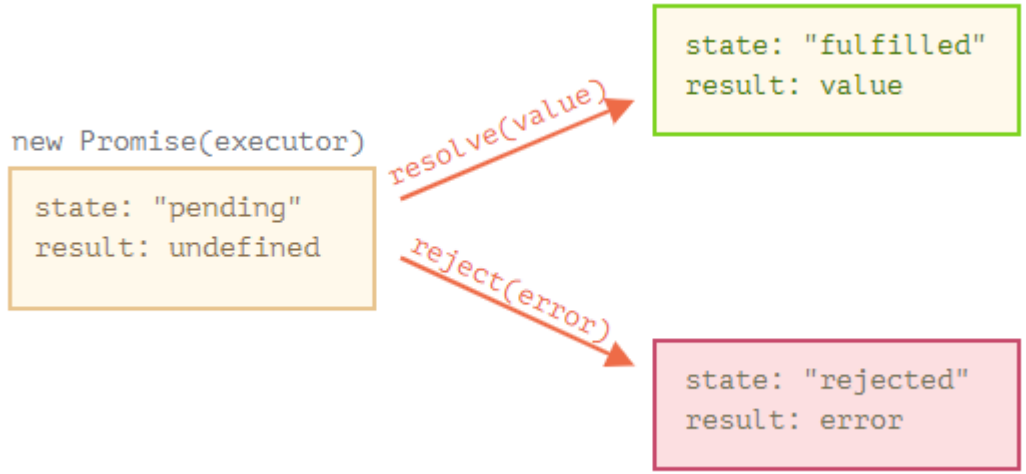
Usar nombres como `resolve` o `reject` son convenciones/costumbres. `then` y `catch` deben invocarse así por definición de la clase `Promise`.

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

Estado de una promesa

Toda promesa tiene un estado definido, entre otras cosas, por los atributos `state` y `result` (no accesibles directamente).

```
new Promise((resolve, reject) => {
    /*   Se realizan las acciones asíncronas.
         Si hubo errores: reject(elError);
         Si hubo éxito:   resolve(data);
    */
});
```



La promesa cambiará de estado según a qué *callback* invoque (o será pendiente para siempre si no lo hace).

Una promesa no cambia de estado una vez que haya pasado a ser cumplida (*fulfilled*) o rechazada (*rejected*).

https://javascript.plainenglish.io/exploring-javascript-promises-in-depth-d07541c8ca9f

Simulacro de asincronismo con Promesa

La función **`getPersonaByID`** simulará una conexión con una BD para buscar un registro por ID y devolverlo.
Si no existe, devolverá un objeto que representa un error.

```
const getPersonaByID = id => {
    return new Promise((resolve, reject) => {
        console.log(`Vamos a buscar ID ${id} en la BD. Paciencia...`);
        setTimeout(() => {
            if (id > 0) {
                const per = { id, nombre: "Pepe" }; // ES6
                // const per = {id: id, nombre: "Pepe"}; //ES5
                resolve(per);
            } else {
                const unError = {
                    error: true,
                    msg: `No se encontró persona con ID ${id}`
                }
                reject(unError);
            }
        }, 3000); // Demora 3 segundos
    });
}

getPersonaByID(4).then(res => {
    console.log("Mostrando resultado en 'then'", res);
}).catch(err => {
    console.err("Mostrando error en 'catch'", err);
});

getPersonaByID(-1).then(res => {
    console.log("Mostrando resultado en 'then'", res);
}).catch(err => {
    console.error("Mostrando error en 'catch'", err);
});
```

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:3
Vamos a buscar ID -1 en la BD. Paciencia...   codigo.js:3
>
```

3 segundos luego...

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:3
Vamos a buscar ID -1 en la BD. Paciencia...   codigo.js:3
Mostrando resultado en 'then'   codigo.js:21
▶ {id: 10, nombre: "Pepe"}
⊗ ▶ Mostrando error en 'catch'   codigo.js:29
  ▶ {error: true, msg: "No se encontró persona con ID -1"}
>
```

Promesas vs. *callbacks*

Comparemos el siguiente código usando una u otra metodología

```
operacionAsync1()
    .then(res => {
        return operacionAsync2(res);
    })
    .then(res2 => {
        return operacionAsync3(res2);
    })
    .then(resFinal => {
        console.log(`Obtenido el res final: ${resFinal}`);
    })
    .catch(errorCallback);
```

Usando promesas

```
operacionAsync1(res => {
    operacionAsync2(res, res2 => {
        operacionAsync3(res2, resFinal => {
            console.log(`Obtenido el res final: ${resFinal}`);
        }, errorCallback);
    }, errorCallback);
}, errorCallback);
```

Usando *callbacks*

Nótese que, usando promesas, solo hubo que enviar una sola vez la referencia al *callback* que maneja el error.

Si ocurre algún error en la cadena de `then`, se saltará al `catch`, como ocurre tradicionalmente con try-catch.

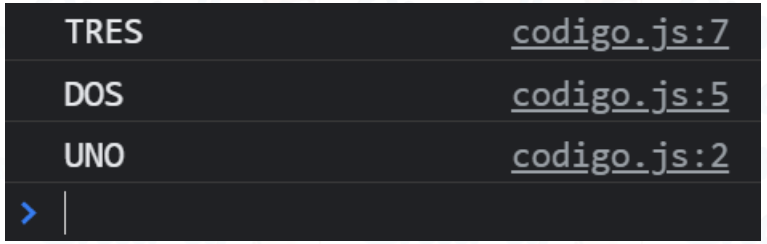
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

Las promesas tienen prioridad

Es común usar promesas en la comunicación con APIs externas que no queremos retrasar o mezclar con otras funciones asíncronas como eventos del usuario o retardos.

ES6 establece una *"microstask queue"* que gestiona los *callbacks* de las promesas con mayor prioridad.

```
// Llamada asíncrona con callback.
setTimeout(() => console.log("UNO"), 0);

// Llamada asíncrona con promesa.
Promise.resolve().then(() => console.log("DOS"));

// Llamada síncrona
console.log("TRES");
```



Probar ejecución en JSV9000

https://javascript.info/microtask-queue

Uso de fetch() para peticiones HTTP

La función `fetch` es la evolución de `XMLHttpRequest` de AJAX.

Se le envía una URL y retorna una **promesa**.

```
fetch(laURL)
    .then(res => {
        console.log(res);
        return res.json();
    })
    .then(resJSON => {
        console.log(resJSON);
    })
    .catch(err => {
        console.error(err);
    })
```

`res` es un objeto de tipo Response

`json()` es un método de Response que devuelve una nueva promesa.

Como el anterior `then` retornó una nueva promesa, se encadena otro.

El `catch` captura cualquier error durante la cadena de promesas.

Para comprobar que la petición fue correcta

```
fetch(laURL)
    .then(res => {
        if (res.ok) {
            // Continuar con las operaciones
        } else {
            throw new Error( /*Elegir qué enviar*/ );
        }
    })
    .catch((error) => {
        // Manejar el error
    });
```

https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

Uso de promesa para app de nombre

```
const handleFetch = url => {
    return fetch(url)
        .then(handleError);
}

const handleError = (res) => {
    if (!res.ok) throw new Error(res.statusText);
    return res;
}

const averiguarPais = nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    handleFetch(url)
        .then(res => res.json())
        .then(resJSON => {
            let paisMasProb = resJSON.country.reduce((a, b) => {
                return a.probability > b.probability ? a : b;
            }, 0);
            alert(`País más probable: ${paisMasProb.country_id}`);
        })
        .catch(err => {
            alert(err);
        });
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.
Mostramos el ID del país que tenga la máxima probabilidad.

Modificá la `url` para que sea incorrecta y así poder testear cómo se comporta la promesa ante un error, por ejemplo:

```
const url = `https://api.nationalize.io/?naZme=${nombre}`;
const url = `https://api.naWtionalize.io/?name=${nombre}`;
```

Encadenamiento de promesas

```
const handleFetch = url => {
    return fetch(url)
        .then(handleError);
}

const handleError = (response) => {
    if (!response.ok) throw new Error(response.statusText);
    return response;
}

const averiguarPais = nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    handleFetch(url)
        .then(res => res.json())
        .then(resJSON => {
            let paisMasProb = resJSON.country.reduce((a, b) => {
                return a.probability > b.probability ? a : b;
            }, 0);
            return paisMasProb.country_id;
        })
        .then(codPais => {
            url = `https://restcountries.com/v3.1/alpha/${codPais}`;
            handleFetch(url)
                .then(res => res.json())
                .then(resJSON => {
                    alert(`Probablemente seas de ${resJSON[0].translations.spa.common}`);
                })
        })
        .catch(err => {
            alert(err);
        });
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Error común: Si `fetch` falla, no se captura el error y se rompe la cadena. (Ver más)
Solución en la siguiente diapositiva.

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.

Tomamos el ID del país que tenga la máxima probabilidad y obtenemos el nombre del país en español, gracias a la API de REST Countries.

Encadenamiento de promesas (Mejorado)

```
const handleFetch = url => {
    return fetch(url)
        .then(handleError);
}

const handleError = (response) => {
    if (!response.ok) throw new Error(response.statusText);
    return response;
}

const averiguarPais = nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    handleFetch(url)
        .then(res => res.json())
        .then(resJSON => {
            let paisMasProb = resJSON.country.reduce((a, b) => {
                return a.probability > b.probability ? a : b;
            }, 0);
            return paisMasProb.country_id;
        })
        .then(codPais => {
            url = `https://restcountries.com/v3.1/alpha/${codPais}`;
            return handleFetch(url);
        })
        .then(res => res.json())
        .then(resJSON => {
            alert(`Probablemente seas de ${resJSON[0].translations.spa.common}`);
        })
        .catch(err => {
            alert(err);
        });
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Solución: Retornar la promesa y mantener la cadena plana. (Ver más)

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.

Tomamos el ID del país que tenga la máxima probabilidad y obtenemos el nombre del país en español, gracias a la API de REST Countries.

async / await

async y **await** son dos nuevas palabras clave que agregan azúcar sintáctico al manejo de asincronismo con promesas.

`async` hace que se retorne una promesa.

```
async function funcionAsync() {
    return "Resultado";
}

console.log( funcionAsync() );

funcionAsync()
    .then(res => console.log(res));
```

Muestra la promesa por consola.

Como en cualquier promesa, el resultado se procesa en el *callback* del `then`.

La palabra **async** delante de una función hará que esta devuelva una promesa.

```
                                    codigo.js:5
▶ Promise {<fulfilled>: "Resultado"}
Resultado                           codigo.js:7
>
```

```
async function funcionAsync() {
    return "Resultado";
}

async function mostrarResultado() {
    let res = await funcionAsync();
    console.log(res);
}

mostrarResultado();
```

Solo puede usarse `await` dentro de una función `async`.

Esperamos el resultado con `await` sin usar `then`.

La palabra **await** aguarda por el resultado de una promesa.

```
Resultado                           codigo.js:7
>
```

https://www.w3schools.com/js/js_async.asp

Simulacro de asincronismo con async / await

La función `getPersonaByID` simulará una conexión con una BD para buscar un registro por ID y devolverlo.
Si no existe, devolverá un objeto error.

```
const getPersonaByID = id => {
    return new Promise((resolve, reject) => {
        console.log(`Vamos a buscar ID ${id} en la BD. Paciencia...`);
        setTimeout(() => {
            if (id > 0) {
                const per = { id, nombre: "Pepe" }; // ES6
                // const per = {id: id, nombre: "Pepe"}; //ES5
                resolve(per);
            } else {
                const unError = {
                    error: true,
                    msg: `No se encontró persona con ID ${id}`
                }
                reject(unError);
            }
        }, 3000); // Demora 3 segundos
    });
}

async function testear(id) {
    try {
        let res = await getPersonaByID(id);
        console.log("Mostrando resultado en el 'try'", res);
    } catch(e) {
        console.error("Mostrando error capturado en 'catch'", e);
    }
}

testear(10);
testear(-1);
```

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:3
Vamos a buscar ID -1 en la BD. Paciencia...   codigo.js:3
>
```

3 segundos luego...

```
Vamos a buscar ID 10 en la BD. Paciencia...   codigo.js:3
Vamos a buscar ID -1 en la BD. Paciencia...   codigo.js:3
Mostrando resultado en el 'try' ▶{id: 10, nombre: "Pepe"}   codigo.js:23
▶Mostrando error capturado en 'catch'   codigo.js:25
  ▶{error: true, msg: "No se encontró persona con ID -1"}
>
```

Promesas vs. async / await

Comparemos el siguiente pseudocódigo usando una u otra sintaxis.

```
operacionAsync1().then(res => {
        return operacionAsync2(res);
    })
    .then(res2 => {
        return operacionAsync3(res2);
    })
    .then(resFinal => {
         console.log(`Obtenido el res final: ${resFinal}`);
    })
    .catch(errorCallback);
```

Usando promesas

```
async function probar() {
    try {
        let res = await operacionAsync1();
        let res2 = await operacionAsync2(res);
        let resFinal = await operacionAsync3(res2);
        console.log(`Obtenido el resultado final: ${resultadoFinal}`);
    } catch (error) {
        console.error(`Ocurrió un error: ${error}`);
    }
}
```

Usando async / await

Con async / await se logra un código más cercano a la forma tradicional síncrona.

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/async_function#reescritura_de_una_cadena_de_promesas_con_una_funci%C3%B3n_async

Uso de async/await para app de nombre

```
const handleFetch = async url => {
    const res = await fetch(url);
    return await handleError(res);
}

const handleError = (res) => {
    if (!res.ok) throw new Error(res.statusText);
    return res;
}

const averiguarPais = async nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    try {
        const res = await handleFetch(url);
        const resJSON = await res.json();
        let paisMasProb = resJSON.country.reduce((a, b) => {
            return a.probability > b.probability ? a : b;
        }, 0);
        alert(`País más probable: ${paisMasProb.country_id}`);
    } catch (err) {
        alert(err);
    }
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.
Mostramos el ID del país que tenga la máxima probabilidad.

Modificá la `url` para que sea incorrecta y así poder testear cómo se comporta la promesa ante un error, por ejemplo:

```
const url = `https://api.nationalize.io/?naZme=${nombre}`;
const url = `https://api.naWtionalize.io/?name=${nombre}`;
```

```
const handleFetch = async url => {
    const res = await fetch(url);
    return await handleError(res);
}

const handleError = (res) => {
    if (!res.ok) throw new Error(res.statusText);
    return res;
}

const averiguarPais = async nombre => {
    let url = `https://api.nationalize.io/?name=${nombre}`;
    try {
        let res = await handleFetch(url);
        let resJSON = await res.json();
        let paisMasProb = resJSON.country.reduce((a, b) => {
            return a.probability > b.probability ? a : b;
        }, 0);
        const codPais = paisMasProb.country_id;
        url = `https://restcountries.com/v3.1/alpha/${codPais}`;
        res = await handleFetch(url);
        resJSON = await res.json();
        alert(`Probablemente seas de ${resJSON[0].translations.spa.common}`);
    } catch (err) {
        alert(err);
    }
};

let nombre = prompt("¿Cuál es tu nombre?");
averiguarPais(nombre);
```

Encadenamiento de promesas con async / await

Pedimos el nombre al usuario (se espera sin espacios) y obtenemos un listado de nacionalidades probables, gracias a la API de nationalize.io.

Tomamos el ID del país que tenga la máxima probabilidad y obtenemos el nombre del país en español, gracias a la API de REST Countries.

FIN DE LA PRESENTACIÓN

Encontrá más como estas en mi sitio web.