

CLASS(E)

Módulo 6.

Asincronía II

Promesas

Asincronía

Hemos visto que el modelo de callbacks tiene varios problemas.

- Callback hell
- Gestión de errores engorrosa
- La asincronía es contagiosa


Promesas

Las promesas son un mecanismo de asincronía más avanzado que nos permite atajar dos de esos problemas:

- Callback hell
- Gestión de errores engorrosa

Callback hell

```
1 function hell(win) {
2   // for listener purpose
3   return function() {
4     loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5       loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6         loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7           loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8             loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9               loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                 loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                   loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                     loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                       async.eachSeries(SCRIPTS, function(src, callback) {
14                         loadScript(win, BASE_URL+src, callback);
15                       });
16                     });
17                   });
18                 });
19               });
20             });
21           });
22         });
23       });
24     });
25   };
26 }
```



Errores y callbacks

El catch **nunca** va a capturar el error.

Primero se evalúa el try/catch entero y luego, en otra rama, se evalúa el callback del setTimeout.

```
try {  
  setTimeout(() => {  
    console.log("Esto debería fallar")  
  }, 500)  
} catch(err) {  
  console.log("Ha habido un error!")  
}
```

Promesas

Una promesa representa **un valor futuro**.

Promesas

Una promesa es un **objeto** que hace de **intermediario**

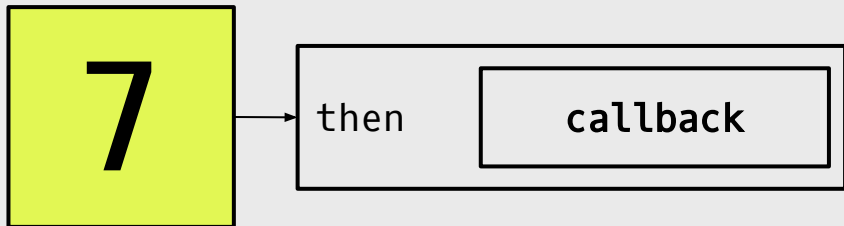
- entre el **productor** de un valor
- y sus **consumidores**

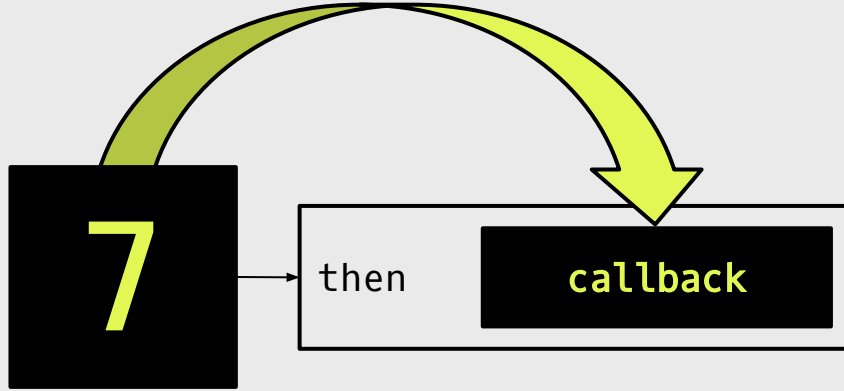
Para simplificar la gestión de procesos asíncronos.

Promesas

Una promesa es una caja que encierra un valor.

7





Promesas

Hay **tres** maneras de crear una promesa:

- `Promise.resolve(value)`
- `Promise.reject(error)`
- `new Promise(...)`

Promesas

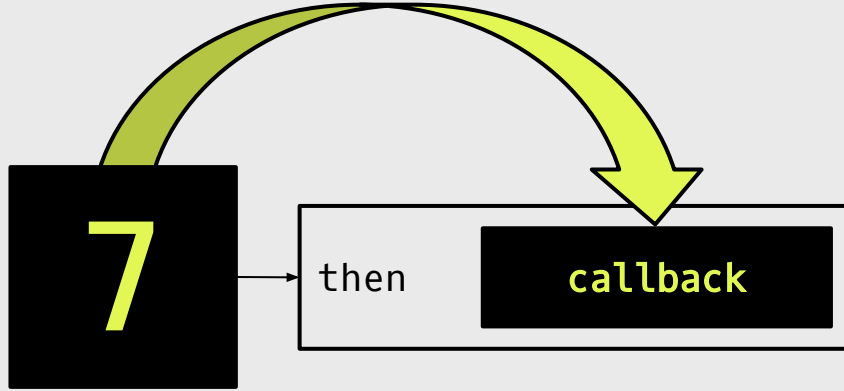
`Promise.resolve(value)`

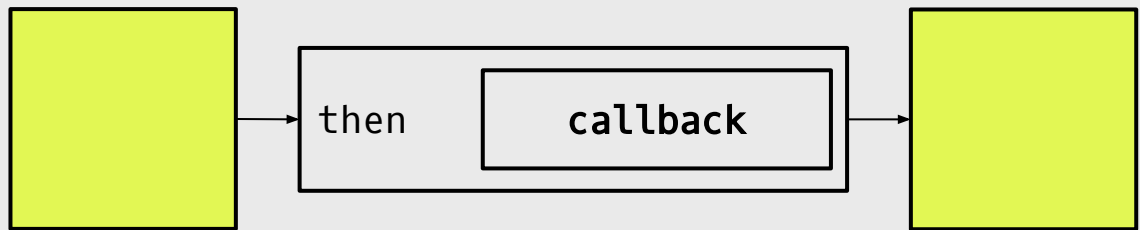
- Crea una promesa resulta
- Los `callbacks` de `.then` se ejecutan inmediatamente.

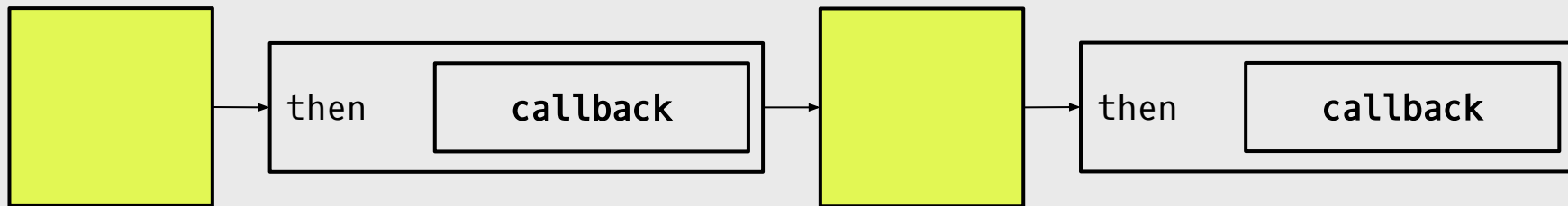
Promesas

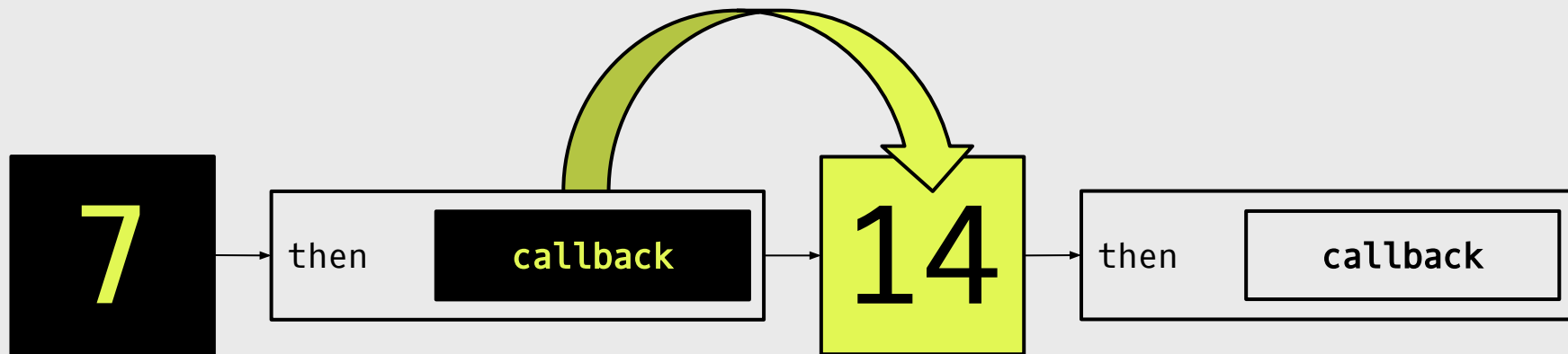
```
// Creamos caja con un 7
let promise = Promise.resolve(7)

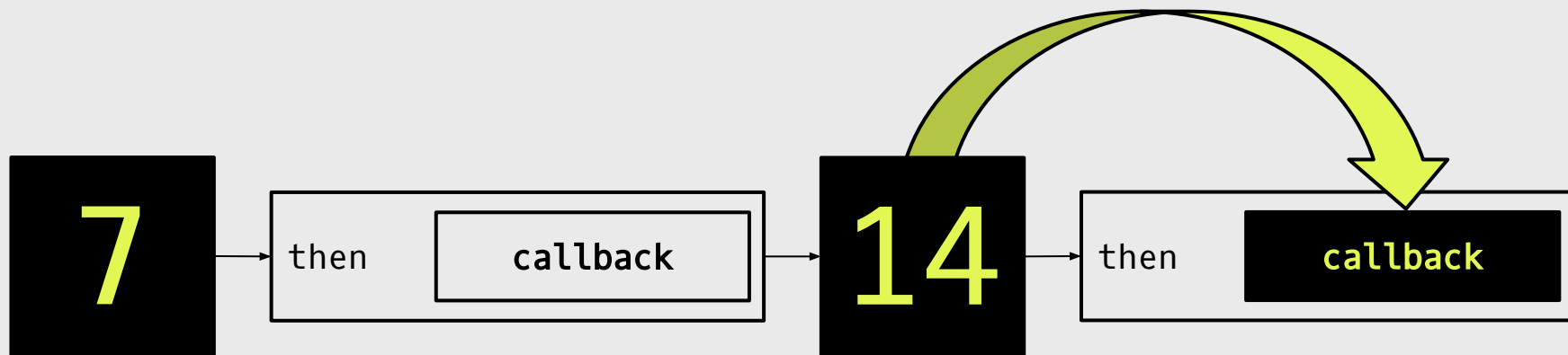
// Abrimos la caja
promise.then(result => {
  console.log(result) // 7
})
```









Promesas

```
let promise = Promise.resolve(7)

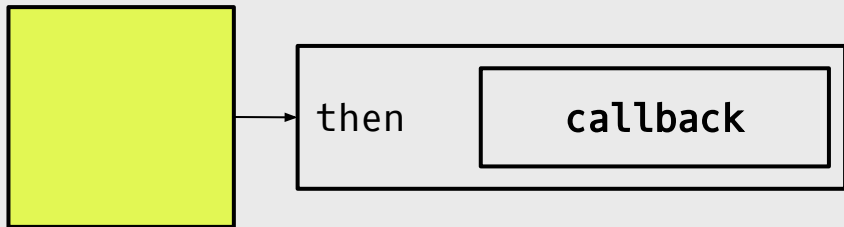
let promise2 = promise.then(result => {
  return result * 2
})

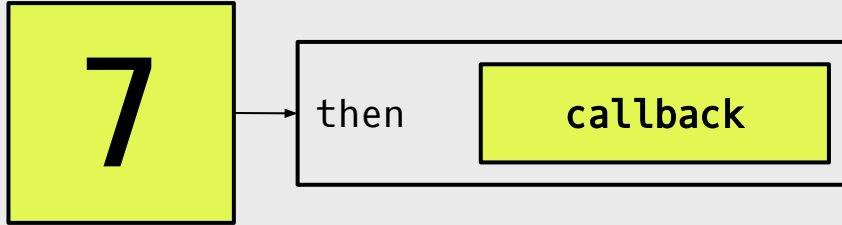
promise2.then(result => {
  console.log(result) // 14
})
```

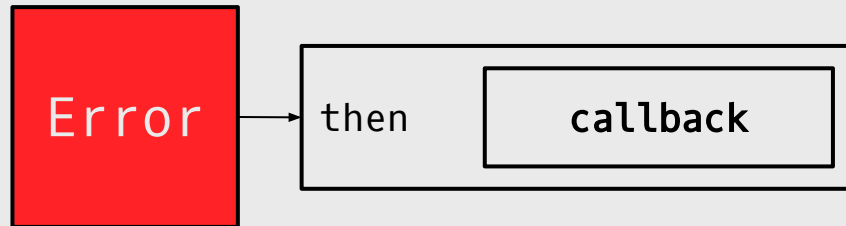
Promesas

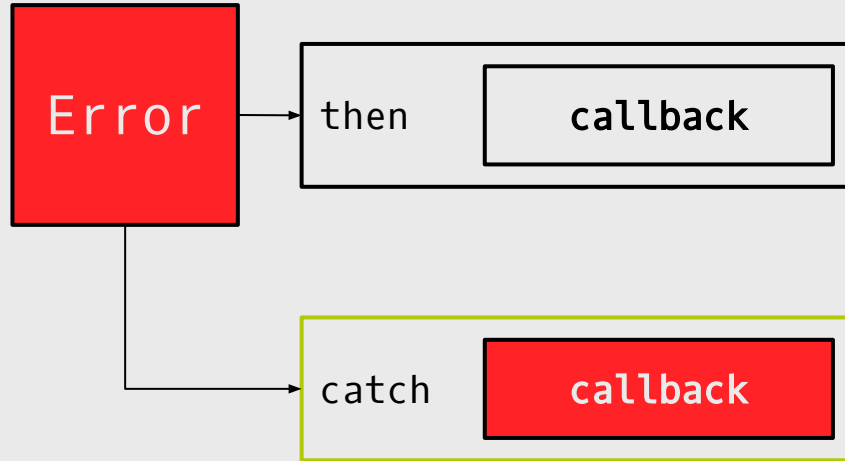
Una promesa tiene 3 estados posibles:

- pending
- fulfilled
- rejected









Promesas

Una promesa tiene 3 estados posibles:

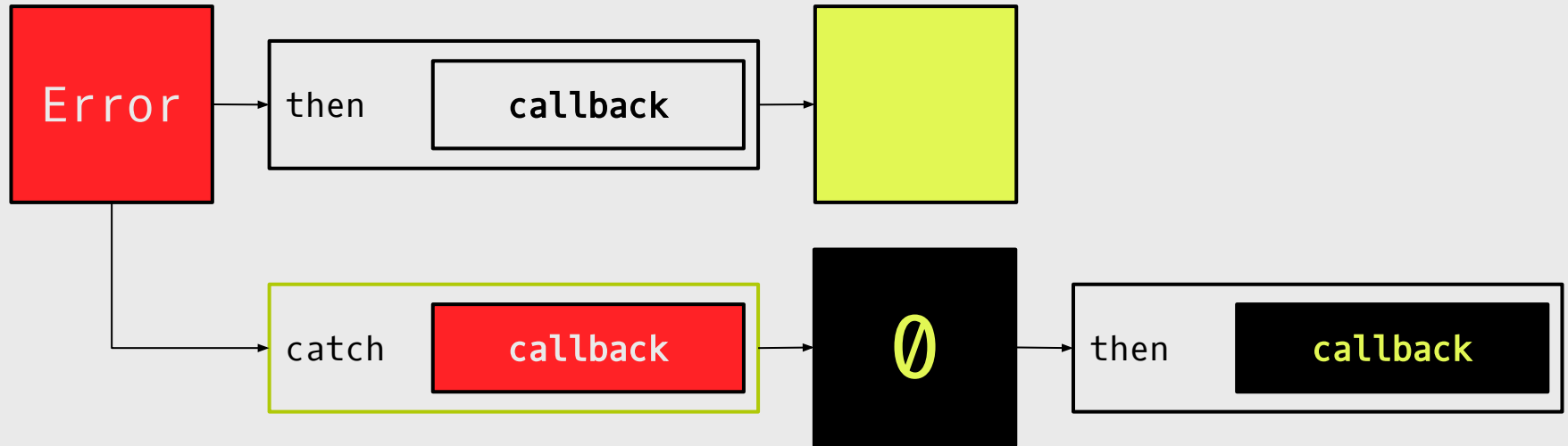
- pending
- fulfilled
- rejected

Cuando una promesa se resuelve o se rechaza, no puede volver a cambiar de estado

- se queda resuelta o rechazada para siempre.

Promesas

- Las llamadas a `.then()` y a `.catch()` devuelven una nueva promesa.
- Que representa el valor de retorno de sus callbacks.
- El callback de `.then()` se ejecuta cuando la promesa se resuelve.
- El callback de `.catch()` se ejecuta cuando la promesa se rechaza.



Promesas

Hay **tres** maneras de crear una promesa:

- `Promise.resolve(value)`
- `Promise.reject(error)`
- `new Promise(...)`

Promesas

`Promise.resolve(value)`

- Crea una promesa resulta
- Los `callbacks` de `.then` se ejecutan inmediatamente.

Promesas

`Promise.resolve(value)`

- Crea una promesa resulta
- Los `callbacks` de `.then` se ejecutan `inmediatamente`.

```
const p = Promise.resolve('ready');  
p.then(console.log);
```

Promesas

Promise.resolve(value)

- Crea una promesa resulta
- Los `callbacks` de `.then` se ejecutan `inmediatamente`.

```
const p = Promise.resolve('ready');  
p.catch(console.log); //?
```

Promesas

`Promise.reject(error)`

- Crea una promesa **rechazada**.
- Los **callbacks** de **`.catch`** se ejecutan **inmediatamente**.

Promesas

`Promise.reject(error)`

- Crea una promesa **rechazada**.
- Los **callbacks** de **`.catch`** se ejecutan **inmediatamente**.

```
const p = Promise.reject(new Error('doomed from the start'));  
p.catch(console.log);
```

Promesas

`Promise.reject(error)`

- Crea una promesa **rechazada**.
- Los **callbacks** de **`.catch`** se ejecutan **inmediatamente**.

```
const p = Promise.reject(new Error('doomed from the start'));  
p.then(console.log); //?
```

Promesas

`new Promise(callback)`

- Crea una promesa **pendiente**
- El callback se ejecuta **con delay 0**
- **callback** recibe dos parámetros:
 - **resolve**: callback de resolución
 - **reject**: callback de rechazo

Promesas

`new Promise(callback)`

- Crea una promesa **pendiente**
- El callback se ejecuta **con delay 0**
- **callback** recibe dos parámetros:
 - **resolve**: callback de resolución
 - **reject**: callback de rechazo

```
const p = new Promise((resolve, reject) => {  
  setTimeout(() => resolve('resolved'), 1000)  
});
```

```
p.then(console.log);
```

```
console.log('antes o después?')
```

Ejercicios promesas

Escribe una función `throwOneCoin` que devuelva una promesa que represente el lanzamiento de una moneda.

- La moneda tarda 2 segundos en caer.
- `50%` de las veces, la promesa se `resuelve` y devuelve `"cruz!"`.
- `50%` de las veces, la promesa se `rechaza` y devuelve `"cara..."`.

Callback hell

```
function getDate(cb) {  
  setTimeout(() => cb(Date.now()), 100);  
}
```

```
getDate((date) => {  
  getDate((date2) => {  
    getDate((date3) => {  
      // seguimos por aquí  
    });  
    // ???  
  });  
});
```

Promesas

```
function getDate() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve(Date.now()), 100)  
  });  
}
```

Promesas

```
function getDate() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve(Date.now()), 100)  
  });  
}
```

```
const datePromise = getDate();  
// seguimos por aquí!
```

Promesas

```
function getDate() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve(Date.now()), 100)  
  });  
}
```

```
const datePromise = getDate();  
const datePromise2 = getDate();  
const datePromise3 = getDate();  
// seguimos por aquí!
```

Promesas

```
function getDate() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve(Date.now()), 100)  
  });  
}
```

```
getDate()  
  .then(() => getDate())  
  .then(() => getDate())
```

Promesas

```
function futureValue(n) {  
  return new Promise(  
    resolve => setTimeout(() => resolve(n), 1000)  
  );  
}
```

```
futureValue(1)  
  .then(v => futureValue(v + 1))  
  .then(v => futureValue(v + 1))  
  .then(console.log); // ???
```

Promesas

`.then(...)`

- Crear secuencias de operaciones asíncronas.
- Manteniendo un flujo de ejecución claro.
- Sin necesidad de indentar cada paso.

Promesas

Una promesa se considera rechazada si se levanta una excepción o si se llama al callback `reject()`.

Promesas

```
const p1 = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!');  
});  
  
p1.catch(e => console.log('Captured:', e.message));
```

Promesas

`.catch(rejectCallback)`

- Devuelve una promesa.
- La promesa devuelta se comporta igual que la devuelta por `.then()`
- El valor de resolución será el valor retornado por `rejectCallback`.

Promesas

```
const p1 = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!');  
});
```

```
p1.catch((e) => {  
  console.log('Captured:', e.message);  
  return e;  
}).then(  
  () => console.log('All good!') //????  
);
```

Promesas

```
const p1 = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!');  
});
```

```
p1  
  .then(() => console.log('1...'))  
  .then(() => console.log('2...'))  
  .then(() => console.log('3...'))  
  .catch(() => console.log('Something bad happened')); ///?
```

Promesas

```
const p1 = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!');  
});
```

```
p1  
  .then(() => console.log('1...'))  
  .then(() => console.log('2...'))  
  .then(() => console.log('3...'))  
  .catch(() => console.log('Something bad happened'))  
  .then(() => console.log('Everything under control'));  
// ???
```

Promesas

En una cadena de promesas

- Los errores se propagan hacia abajo.
- Si se captura el error, la cadena se resuelve con normalidad a partir de ese punto.

Facilita el manejo de errores en procesos asíncronos.

Vamos a ver cómo funcionan
las promesas de forma
práctica

Ejercicio promesas II

Crea una función `wait` que reciba un número de milisegundos y devuelva una promesa.

- La promesa se debe resolver cuando pase el tiempo indicado.

```
wait(2000)
  .then(() => console.log("Han pasado dos segundos"))
```


Ejercicio promesas III

Crea una función `throwDice` que devuelva el resultado de tirar un dado de 6 caras al cabo de 1000ms a través de una promesa.

```
throwDice()  
  .then(result => console.log(result)) // 2
```

Ejercicio promesas IV

Crea una función `getPlayerScore` que devuelva el resultado de tirar dos dados (eg: [3,5])

- Utiliza la función `throwDice` del ejercicio anterior para calcular los valores de las tiradas.
- Devuelve el resultado a través de una promesa.

```
getPlayerScore()  
  .then(result => console.log(result)) // [3, 5]
```

Ejercicio promesas V

Crea una función `startGame` que devuelva los resultados de las tiradas de 3 jugadores (eg: `[[2,2],[4,6],[5,1]]`)

- Utiliza la función `getPlayerScore` del ejercicio anterior para calcular los valores de las tiradas.
- Devuelve el resultado utilizando una promesa.

```
startGame()  
  // [[2, 2], [4, 6], [5, 1]]  
  .then(result => console.log(result))
```

No parecemos estar
ganando mucho...

Promesas

Si las promesas únicamente dependen del valor anterior, podemos solucionar el callback hell:

```
fetch(API_URL)
  .then(response => response.json())
  .then(data => fetch(data.nextPageURL))
  .then(response => response.json())
  .then(data => console.log(data))
```

Promesas

Si el resultado de una serie de promesas debe compilar los valores devueltos por cada promesa, seguimos teniendo que anidar el código.

¡No hemos solucionado el callback hell!

Promesas

Necesitamos tener acceso a todos los resultados al final de la serie.

```
return new Promise((resolve) => {  
  getPlayerScore().then((result) => {  
    getPlayerScore().then((result2) => {  
      getPlayerScore().then((result3) => {  
        resolve([result, result2, result3]);  
      });  
    });  
  });  
});
```

Promesas

Hasta ahora hemos lanzado promesas de forma secuencial. Una detrás de otra.

Una ventaja de las promesas es que podemos lanzarlas en paralelo fácilmente.

Promesas

`Promise.all` recibe un array de promesas y `devuelve una promesa` que resuelve todos los resultados.

```
const p1 = futureValue(1)
const p2 = futureValue(2)

Promise.all([p1, p2])
  .then(results => console.log(results)) // [1, 2]
```

Ejercicio promesas VI

Resuelve los ejercicios anteriores utilizando `Promise.all`

- `getPlayerScore`
- `startGame`

Promesas

`Promise.all` falla si hay un error en cualquiera de las promesas recibidas.

Promesas

Promise tienen más mecanismos de paralelización.

```
// devuelve el resultado de la primera promesa que *resuelva*  
Promise.any([promises])
```

```
// resuelve cuando la *primera* promesa de la lista se resuelva  
// si la primera promesa falla, Promise.race también falla  
Promise.race([promises])
```

Promesas

En definitiva, las promesas nos ayudan:

- A facilitar la gestión de errores (catch).
- A decidir cuando queremos romper el flujo de ejecución.
- Lidiar con el callback hell en algunos casos (paralelización).

Promesas

A pesar de las ventajas, las promesas tienen inconvenientes:

- Las cadenas de promesas pueden ser difíciles de entender.
- No resuelven todos los casos de anidación elegantemente.
- Los bloques try/catch siguen sin funcionar

Habréis notado que el código
asíncrono es difícil de seguir.

Si pudiéramos tratar a las
promesas como si fueran
código síncrono...

ES6 introduce una nueva
forma de gestionar la
asincronía

Async/await

Async/await

Await nos permite transformar esto...

en esto:

```
fetch(url)
  .then(response => {
    console.log(response)
    return response.json()
  }).then(data => {
    console.log(data)
  })
```

```
const response = await fetch(url)
const data = await response.json()
console.log(response)
console.log(data)
```

Async/await

Await bloquea la ejecución del código hasta que resuelva la promesa que está esperando.

```
console.log("uno")    // a los 0 ms  
await wait(1000)      // bloquea la ejecución  
console.log("dos")    // a los 1000 ms  
console.log("tres")   // a los 1000 ms
```

¡Nos permite tratar el código
como si fuera síncrono!

Async/await

No podemos bloquear todo el programa entero cada vez que hacemos un await.

Por ese motivo, await solo funciona dentro de funciones `async`.

```
async function getJSON(url){  
  const response = await fetch(url)  
  return await response.json()  
}
```

Async/await

Por debajo, async transforma la función getJSON para que devuelva una promesa.

```
function getJSON(url){  
  return new Promise((resolve, reject) => {  
    // ... ejecuta toda la función  
    // ... resuelve con el valor del *return*  
  })  
}
```

Async/await

Es decir, todas las funciones async devuelven una promesa.

¡Estas promesas también se pueden awaitear!

```
const data = await getJSON(url)
```


Ejercicio async/await

Implementa los siguientes ejercicios anteriores con async/await:

- throwDice
- getPlayerScore
- startGame

Ejercicio async/await II

```
const urls = [url1, url2]
const results = await asyncMap(urls, getJSON)
```

Implementa `asyncMap` utilizando `async/await`.

- `asyncMap` recibe una función `que devuelve una promesa`.
- `asyncMap` ejecuta las funciones en `paralelo`.

Ejercicio async/await III

```
const urls = [url1, url2]
const results = await asyncSequentialMap(urls, getJSON)
```

Implementa `asyncSequentialMap` utilizando `async/await`.

- `asyncMap` recibe una función que devuelve una promesa.
- `asyncMap` ejecuta las funciones secuencialmente.

Async/await

Todas las funciones async devuelven una promesa.

```
const data = await getJSON(url)
```

Async/await

Si sucede un error dentro de una función async, se bloquea la ejecución del código, igual que pasaría en código síncrono.

```
// en el caso de que getJSON lance un error...  
const data = await getJSON(url)  
console.log("No se imprime nunca")
```

Async/await

Por lo tanto ¡podemos volver a utilizar try/catch!

```
try{  
  const data = await getJSON(url)  
} catch(err) {  
  console.log(err)  
}
```

Ejercicio async/await IV

Implementa la función `promiseAllSafe` que recibe una lista de promesas y devuelve una lista con los resultados.

- Si la promesa se resuelve, añade el resultado a la lista.
- Si la promesa falla, añade null a la lista e imprime el error por consola.
- Las promesas se deben lanzar `secuencialmente`

Ejercicio async/await V

Implementa la función `retry` que repite una función asíncrona hasta que resuelve o se acaba el límite de intentos.

```
async function retry(func, retries){  
  // ...  
}  
  
const result = retry(async() => {  
  return await getJSON(url)  
}, 3)
```


Ejercicio async/await VI

Implementa el ejercicio `filesystem III` utilizando `async/await`

- Utiliza la librería `fs.promises`
 - `const fs = require("fs").promises;`
- No utilices métodos síncronos

Async/await

En definitiva, async/await:

- Resuelve el callback hell en todos los casos.
- Nos permite razonar como si el código fuera síncrono.
- Simplifica el uso de la asincronía.

Abstracción

Tenemos un problema (asincronía) y una herramienta primitiva (callbacks) para solucionarlo.

¡Construimos abstracciones para descomplicar el código!

- callbacks → promesas → async/await

Abstracción

Problema:

- Aplicar transformaciones en estructuras de datos.

Herramientas primitivas:

- Bucles, condicionales.

Abstracciones:

- map, filter, reduce, union, difference, chunk...

Abstracción

Problema:

- Recorrer estructuras de árbol.

Herramientas primitivas:

- Funciones, condicionales.

Abstracciones:

- flatten, deepFlatten, sumDeep, assignDeep...

Abstracción

Problema:

- Facilitar/optimizar comunicación con el back end.

Herramientas primitivas:

- Intervalos, callbacks.

Abstracciones:

- throttle, debounce, retry, Promise.race, getJSON...

Abstracción

Si tuviéramos que programar en ensamblador, montar una web app compleja sería un proyecto **muy costoso**.

¡También sería muy difícil de entender!

Abstracción

Toda la computación está basada en capas incrementales de abstracción.

En orden ascendente:

- Partículas elementales
- Átomos
- Moléculas
- Transistores
- Puertas lógicas
- Circuitos (eg.: multiplexores)
- CPU
- Lenguaje ensamblador
- V8 engine (C++)
- Javascript
- React

Abstracción

Si tuviéramos que montar servidores web modernos utilizando **puertas lógicas** ¡serían proyectos **inasumibles**!

Abstracción

Es mucho más sencillo a largo plazo implementar ensamblador, luego un lenguaje de alto nivel como JavaScript, y después una librería como Express.

Abstracción

La misma lógica aplica partiendo de JavaScript. El uso de abstracciones es vital para reducir el tiempo de desarrollo, facilitar la comprensión del código y reducir la redundancia.