Promesas

UD5: Interacción con fuentes externas en Ionic

Objetivos de aprendizaje

- Comprender el concepto de asincronía y como lo tratan los distintos lenguajes de programación
- Aprender a utilizar y consumir promesas en Javascript

Asincronía

- Es uno de los conceptos principales que rige laprogramación en JS
- Cuando empezamos a programar, habitualmente realizamos tareas secuenciales, que se ejecutan una tras otra y cuyo flujo es sencillo de seguir

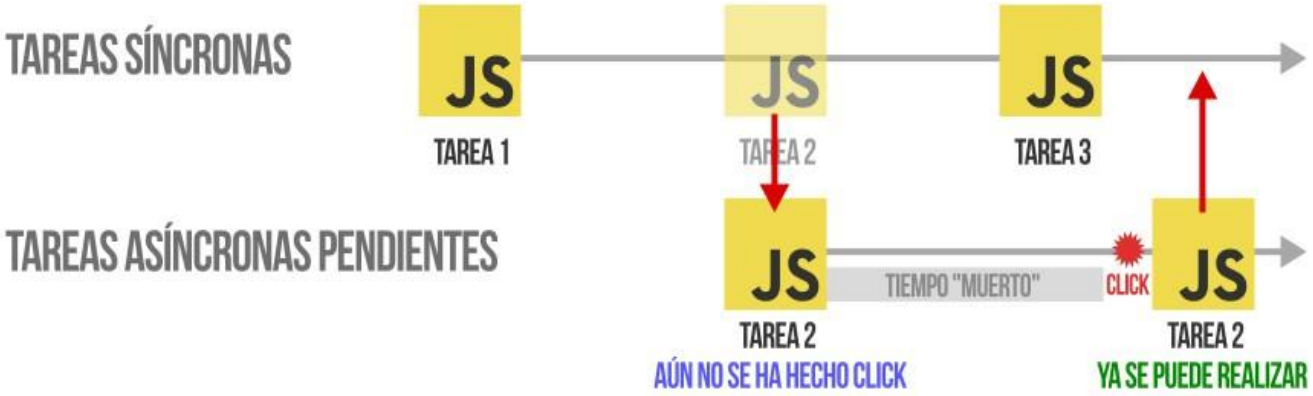
```
primera_funcion();      // Tarea 1: Se ejecuta primero
segunda_funcion();      // Tarea 2: Se ejecuta cuando termina
                        primera_funcion()
tercera_funcion();      // Tarea 3: Se ejecuta cuando termina
                        segunda_funcion()
```

Problemas asociados

- A veces tenemos muchas tareas que:
  - Pueden finalizar correctamente o no
  - Pueden depender unas de otras
- No sabemos cuanto tiempo va a tardar cada tarea

Lenguaje no bloqueante

- JS es un lenguaje **no bloqueante**: Las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas
- Ejemplo: Si `segunda_funcion()` realiza una tarea que depende de un click de ratón, no se queda bloqueado esperando a que finalice la tarea



Gestionando la asincronía

Método	Descripción
Callbacks	Forma clásica de gestionar asincronía
Promesas	Forma más moderna y actual
async/away	Una variación de las promesas con una variación de la sintaxis (azúcar sintáctico)

Función de callback (retrollamada)

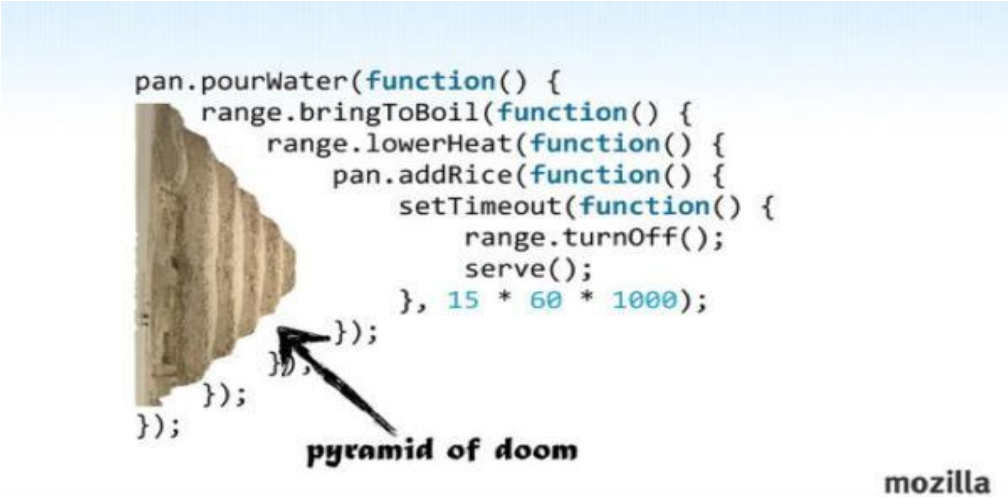
- Una función de *callback* es una función que:
  - Pasamos a otra función como argumento
- Se invoca en un momento posterior
- En muchos casos está asociada al éxito o fracaso de la primera función
- Ejemplos:

```
setTimeout(() => hazAlgo(), 3000);
```

```
window.addEventListener("load", (e) => cargaPagina());
```

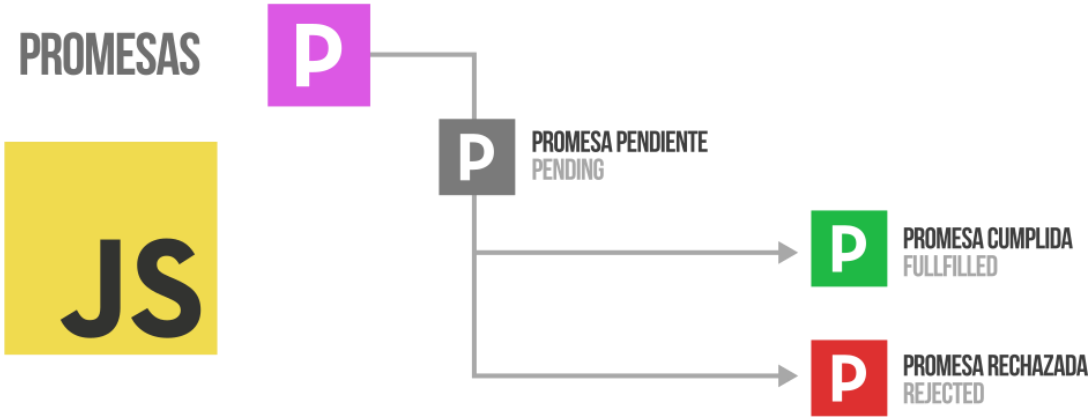
Problemas de los callbacks

- Si tenemos que gestionar la asincronía varias veces en una misma función se produce la estructura CallBack Hell (difícil de leer y depurar..)



Promesas

- Una promesa es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca estar disponible.



Una promesa puede estar en 3 estados: Pendiente, cumplida o rechazada

Definiendo una promesa

- El constructor recibe una función con dos parámetros

Parámetro	Descripción
`resolve()`	Cuando llamamos a `resolve(value)` se cambie el estado de la promesa de pendiente a resuelta (*fullfilled*), pasando value como resultado de la promesa. Lo usaremos cuando la operación se completa con éxito.
`reject()`	Cuando llamamos a `reject(reason)` se cambia el estado de la promesa de pendiente a rechazada (*rejected*) pasando la cadena *reason* como motivo del rechazo. Lo usamos cuando algo falla en la operación asíncrona

Son funciones ya definidas y provistas por el motor de JS. Solo necesitamos proporcionarlas como parámetro y usarlas

Consumiendo una promesa

ejemplos/1_promesas/1_promesa

```
function hazPizza(){
  const pizza = 'Pizza lista';
  const promesaPizza= new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(pizza);
    }, 2000);
  });
  return promesaPizza;
}


hazPizza()
  .then((resultado) => {
    console.log(resultado);
  });
```

El resultado se muestra cuando se cumple la promesa

Manejo de resultados

Método	Descripción
`.then(funtion resolve)`	Ejecuta la función `resolve` cuando la promesa se cumple
`.catch(funtion reject)`	Ejecuta la función `reject` cuando la promesa se rechaza
`.then(funtion resolve, function reject)`	Equivale a las dos anteriores en el mismo `.then`
`.finally(funtion end)`	Ejecuta la función end tanto si cumple como si se rechaza

Consumiendo una promesa (II)

ejemplos/1_promesas/2_errores

```
function hazPizza(ingredientes){
  const promesaPizza= new Promise((resolve, reject) => {
    if(ingredientes.includes("piña"))
  reject("La pizza no lleva piña!"); else{

  setTimeout(() => {
  resolve('Pizza lista con ' + ingredientes);
  }, 2000);
  }
  });
  return promesaPizza;
  }

  hazPizza(['tomate', 'queso', 'jamón','piña'])
  .then((resultado) => { console.log(resultado);
  })
  .catch((error) => { // Capturamos el error
  console.error(error);
  });
```

Encadenando promesas

```
hazPizza(['tomate', 'queso', 'jamón'])
.then((resultado) => {
console.log(resultado);
return hazPizza(['tomate', 'queso', 'jamón','pepperoni']);
})
.then((resultado) => { console.log(resultado);
})
.catch((error) => { // Capturamos el error
console.error(error);
});
```

ejemplos/1_promesas/3_encadenamiento

async/await

- Introducen azúcar sintáctico para gestionar las promesas de manera más sencilla
- Elaboramos el encadenamiento de `.then()` para usar uno en el que trabajamos de forma más tradicional

async/await

- async Se coloca previamente a la definición de una función para indicar que la definiremos como función asíncrona.
  - La invocación de una función async devuelve una promesa
- Para manejar una promesa declarada con async podemos usar .then o await

```
async function hazPizza(ingredientes){
...
return promesaPizza;
}
```

```
const resultado= await hazPizza(['tomate', 'queso', 'jamón'])
console.log(resultado);
```

ejemplos/1_promesas/4_async_await