

Tema 4

Programación asíncrona

- **Introducción**
- Callbacks
- Promesas
- Async / Await
- Librerías asíncronas

- JavaScript se diseñó para añadir **interactividad a una página web**
- Una página web es como una Interfaz gráfica de usuario (GUI), que se suelen implementar con un **único hilo de ejecución** (hilo de despacho de eventos)
- Por este motivo, JavaScript inicialmente sólo se podía ejecutar en un **único hilo de ejecución** (*single-threaded*)
- No podía haber dos **funciones ejecutándose** a la misma vez de forma **concurrente**

- Y si se necesita hacer una **petición al servidor** al pulsar un botón?
- Si hay un único hilo y la llamada se bloqueara a la espera de respuesta, **la GUI se bloquearía**
- En vez de **esperar** a que llegue el resultado, se define **el código que se ejecutará** cuando llegue la respuesta
- JS implementa un modelo de programación **asíncrono**
- Las operaciones de entrada/salida **no son bloqueantes**

Introducción

El método ajax de jQuery no se bloquea a la espera de la respuesta

El código que se ejecutará cuando llegue la respuesta se define en una función (callback)

```
$.ajax({  
  url: "http://server.com/results.html",  
})  
.done(function( data ) {  
  console.log("Petición Recibida");  
  console.log("Data:", data);  
});  
  
console.log("Petición enviada");
```

```
Petición enviada  
Petición recibida  
Data: ...
```

Programación asíncrona

- Introducción
- **Callbacks**
- Promesas
- Async / Await
- Librerías asíncronas

- Se denomina **callback** a una función que se pasa como parámetro a otra función para que sea “**llamada de vuelta**” en algún momento posterior
- Tipos de callbacks
 - Continuaciones
 - Eventos
 - Otros tipos

- Las **continuaciones** (*continuations*) son aquellas callbacks que se ejecutarán cuando haya **terminado de ejecutarse** la **función** a la que se llama
- Habitualmente recibirán el **valor resultado** o un error
- Sólo se **ejecutan una vez** (al finalizar la función)

Continuaciones

ejem1

El método `readFile` no se bloquea a la espera de la respuesta

La callback que es llamada al terminar se suele denominar **continuation**

```
const fs = require('fs');  
fs.readFile('/etc/hosts', function (err, data) {  
    if (err) throw err;  
    console.log(data.toString());  
});
```

La mayoría de las callbacks en Node.js reciben un primer parámetro error para notificar de un error si existe

- Los manejadores de eventos (*handler*) se registran en los objetos que los generan
- Cada vez que se genera el **evento**, se ejecuta el *handler*
- En ocasiones se pueden registrar varios *handlers*
- En ocasiones se pueden desregistrar los *handler* previamente registrados

```
$("#button").on("click", ()=> {  
    console.log("clicked");  
});
```

```
$("#button").click(()=> {  
    console.log("clicked");  
});
```

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

var text = '';

var stream = fs.createReadStream('/etc/hosts');

stream.on('open', () => {
  console.log('The file is open');
});

stream.on('data', chunk => {
  console.log('Received ' + chunk.length + ' bytes: ' + chunk);
  text += chunk;
});

stream.on('end', () => {
  console.log('There will be no more data');
  console.log('Final data: ' + text);
});
```

Los eventos se registran con el método "on" ("addListener" suele ser común también)

Otros tipos de callbacks

- Existen **otros usos de los callbacks** que no puede categorizarse en ninguna de las secciones anteriores

```
setTimeout(() => {  
    alert("Hello");  
}, 3000);
```

Función que se ejecutará cuando pasen 3000ms

```
sleep(3000);  
alert("Hello");
```

En JS no se pueden tener llamadas bloqueantes

- Introducción
- Callbacks
- Eventos
- **Promesas**
- Async / Await
- Librerías asíncronas

- **Callback hell**



- Si usamos varias callbacks **continuación** una detrás de otra el código se vuelve complejo de entender y mantener

```
getData(function(a){
    getData(a, function(b){
        getData(b, function(c){
            getData(c, function(d){
                getData(d, function(e){
                    ...
                });
            });
        });
    });
});
```

Promesas

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err){
              if (err) console.log('Error writing file: ' + err)
            })
          }).bind(this))
        }
      })
    })
  }
})
```


Callback hell



```
function doSomething(params){
  $.get(url, function(result){
    setTimeout(function(){
      startAsyncProcess(function(){
        $.post(url, function(response){
          if(response.good){
            setStateasGoodResponse(function(){
              console.log('Hooray!')
            });
          }
        });
      });
    });
  });
}
```

- Las promesas se venían usando desde hace varios años en **JavaScript ES5**
- Existían **diferentes librerías** que implementan el mismo concepto, pero con ligeras diferencias
- Se han estandarizado en **ES6**

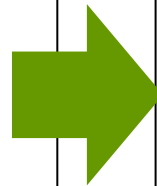
http://exploringjs.com/es6/ch_promises.html

<https://developers.google.com/web/fundamentals/getting-started/primers/promises>

- Una promesa es un objeto de la clase **Promise**
- El objeto **promesa** es devuelto por una función asíncrona (habitualmente sustituyendo a un **callback continuación**)
- El código que se ejecutará cuando esté disponible el resultado se define llamando al método **then** de la promesa

Continuation

```
asyncFunction(arg1, arg2,  
  result => {  
    console.log(result);  
  });
```



Promesa

```
asyncFunction(arg1, arg2)  
  .then(result => {  
    console.log(result);  
  });
```

- Encadenamiento de promesas
 - Si en el then de una promesa **se devuelve otra promesa**, se encadenan
 - Las promesas **evitan el callback hell** y hacen el código más lineal, más fácil de mantener

Continuations

```
getData(a => {  
  getMoreData(a, b => {  
    getMoreData(b, c => {  
      ...  
    });  
  });  
});
```

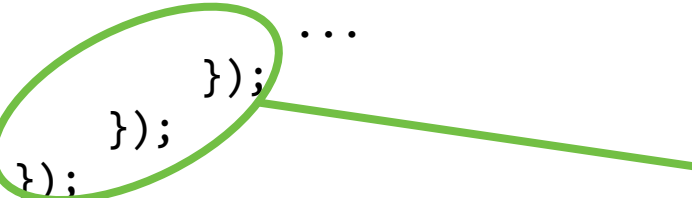
Promesas

```
getData()  
  .then(a => {  
    return getMoreData(a)  
  })  
  .then(b => {  
    return getMoreData(b)  
  })  
  .then(c => {  
    ...  
  })
```

- Encadenamiento de promesas
 - Si en el then de una promesa **se devuelve otra promesa**, se encadenan
 - Las promesas **evitan el callback hell** y hacen el código más lineal, más fácil de mantener

Continuations

```
getData(a => {  
  getMoreData(a, b => {  
    getMoreData(b, c => {  
      ...  
    });  
  });  
});
```



Promesas

```
getData()  
  .then(a => {  
    return getMoreData(a)  
  })  
  .then(b => {  
    return getMoreData(b)  
  })  
  .then(c => {  
    ...  
  })
```

- **Gestión de errores con continuaciones**

- Es habitual enviar el error como primer parámetro del callback

```
getData(function(e, a){  
    if(e) return console.log(e);  
    getData(a, function(e, b){  
        if(e) return console.log(e);  
        getData(b, function(e, c){  
            ...  
        });  
    });  
});
```

Si hay error, no se llama a la siguiente función

- Gestión de errores con promesas

- El error se puede gestionar en un único sitio, en una función **catch**, al final de los métodos **then**

```
getData()
  .then(a => {
    return getMoreData(a)
  })
  .then(b => {
    return getMoreData(b)
  })
  .then(c => {
    console.log(c);
  })
  .catch(e => {
    console.error(e);
  })
```

Si hay error en cualquier llamada, se ejecuta el catch (como con try/catch)

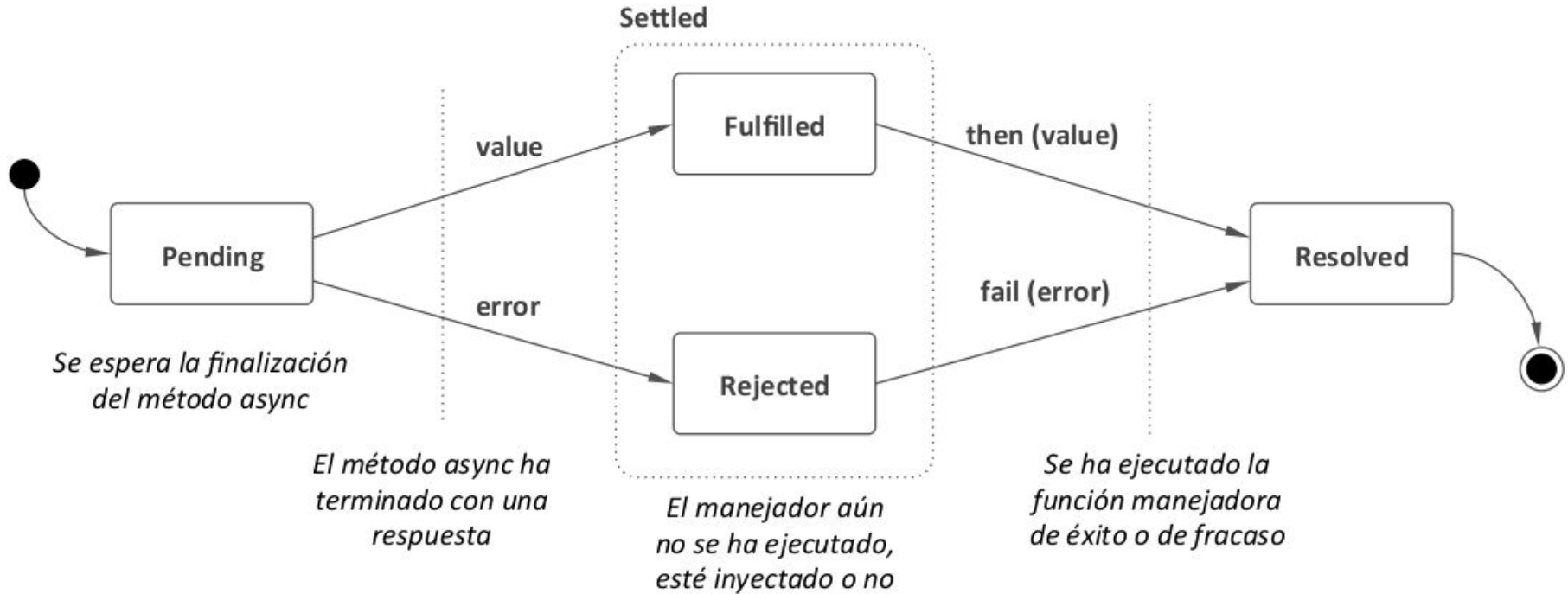
El código es más limpio porque evita poner una sentencia de control de errores en cada callback

- **Gestión de errores con promesas**

- Al igual que en los bloques try, también existe la función **finally**

```
getData()
  .then(a => {
    console.log(a);
  })
  .catch(e => {
    console.log(e);
  })
  .finally(() => {
    // finalizada (exitosa o rechazada)
  });
```


- Estado de una promesa



- **Estado de una promesa**

- Un objeto promesa puede estar en alguno de estos estados
 - **Pending (pendiente):** Está pendiente de recibir un resultado. Se está ejecutando.
 - **Settled (finalizada):** Ha finalizado su ejecución de forma satisfactoria (**fulfilled**) o con error (**rejected**) pero todavía no se han ejecutado los callbacks.
 - **Resolved (resuelta):** Ha finalizado y se han invocado los callbacks.

- **Implementación**

- Existen librerías cuyas funciones devuelven **promesas**, pero como se estandarizaron en ES6, todavía **no hay muchas**
- Otras librerías basadas en continuaciones se pueden **convertir** a promesas de forma automática (usando algunas utilidades)
- Nosotros podemos **implementar nuestras funciones** para que devuelvan promesas

- **Implementación**

- Para crear una promesa se crea un objeto de la clase **Promise**
- En el constructor se le pasa una función callback
- En esa función se reciben dos parámetros: `resolve` y `reject`
- Esos parámetros son funciones
- Se deberá llamar a una de estas funciones para resolver la promesa (`fulfill` o `reject`)

- Implementación

```
function process(){  
  return new Promise((resolve, reject) => {  
    // Hacer alguna cosa, posiblemente asíncrona  
    if (/* todo va bien... */) {  
      resolve("Resultado");  
    } else {  
      reject(Error("Motivo del error"));  
    }  
  });  
}
```

- Implementación
 - Timeout con promesas

```
function sleep(millis) {  
  return new Promise(resolve => {  
    setTimeout(() => resolve(), millis);  
  });  
}  
  
sleep(3000).then(() => console.log('He dormido 3s'));
```

Ejercicio 1

- Axios es una de las librerías NPM más utilizadas para hacer peticiones REST (67K ★ en GitHub)
- Está basada en promesas, lo que simplifica su uso
- Implementa una aplicación Node que cargue la información de la siguiente URL y la muestre por consola, usando promesas

<https://jsonplaceholder.typicode.com/posts/1>

- El contenido es un JSON con información de un post

<https://github.com/axios/axios>

- Varias promesas se puede sincronizar entre sí
 - **Promise.all(...)**: Devuelve una promesa que se resuelve cuando todas las promesas se han resuelto. Devuelve error en cuanto alguna de ellas es rechazada (rejected)

```
var promise1 = ...  
var promise2 = ...  
var promise3 = ...  
  
Promise.all([promise1,promise2,promise3])  
  .then(values => {  
  
    var valP1 = values[0];  
    var valP2 = values[1];  
    var valP3 = values[2];  
  
  }).catch(error => console.error(error));
```


- Varias promesas se puede sincronizar entre sí
 - **Promise.allSettled(...)**: Devuelve una promesa que se cumple cuando todas finalizan (cumplidas o rechazadas). Nunca devuelve error.

```
var promise1 = ...
var promise2 = ...
var promise3 = ...

Promise.allSettled([promise1,promise2,promise3])
  .then(results => {

    if (results[0].status === 'fulfilled'){
      console.log(results[0].value);
    } else { //results[0].status === 'rejected'
      console.error(results[0].reason);
    }
  });
```

- Introducción
- Callbacks
- Eventos
- Promesas
- **Async / Await**
- Librerías asíncronas

- Las promesas proporcionan un **mejor modelo de programación** asíncrona que las callbacks continuación
- Pero el código **no es tan limpio** como el código síncrono
- Las funciones **declaradas como asíncronas** (**async**) pueden usar funciones con promesas como si fueran **funciones síncronas**

- Desde ES7 / ES2016 una función se puede declarar como **async**
- Dentro de su código, podemos usar el operador **await**

```
function main() {  
  
    sleep(3000).then(() => {  
        console.log('3s')  
    });  
}  
  
main()
```

```
async function main() {  
  
    await sleep(3000)  
    console.log('3s')  
}  
  
main()
```

- También podemos usar **expresiones asíncronas** para usar el await
 - *Immediately Invoked Async Function Expression (IIAFF)*

```
(async function () {  
    await sleep(3000)  
    console.log('3s')  
})();
```

<https://github.com/tc39/proposal-top-level-await>

Async / Await

- Si la promesa devuelve un valor, podemos asignar ese valor a una variable
- Para gestionar promesas rechazadas, usamos try/catch

```
getData()  
  .then(a => {  
    return getMoreData(a)  
  })  
  .then(b => {  
    return getMoreData(b)  
  })  
  .then(c => {  
    console.log(c);  
  })  
  .catch(e => {  
    console.error(e);  
  })
```



```
try {  
  
  let a = await getData();  
  let b = await getMoreData(a);  
  let c = await getMoreData(b);  
  console.log(c);  
  
} catch(e) {  
  console.error(e);  
}
```

Ejercicio 2

- Convierte el Ejercicio 1 para que use async / await

- Definir una función como “async” es suficiente para que devuelva una promesa

```
async function asyncFunction() {  
    // Async call success...  
    return true;  
}  
  
asyncFunction().then(value => {  
    console.log(value);  
});
```


- Para rechazar la promesa, basta con elevar una excepción

```
async function asyncFunction() {  
    throw Error("Any kind of error");  
}  
  
asyncFunction().then(value => {  
    console.log('SUCCESS');  
}).catch(error => {  
    console.error('FAIL');  
});
```

- Con **Promise.all** podemos esperar a múltiples funciones async ejecutándose “en paralelo”

```
async function async1() { ... }  
async function async2() { ... }  
async function async3() { ... }  
  
let [v1, v2, v3] = Promise.all([async1(), async2(), async3()])  
  
console.log(v1);  
console.log(v2);  
console.log(v3);
```

- Introducción
- Callbacks
- Eventos
- Promesas
- Async / Await
- **Librerías asíncronas**

Librerías asíncronas

- Existen librerías de alto nivel para controlar la asincronía
- La más usada en RxJS
- Permite definir el procesamiento de múltiples eventos de forma reactiva



<https://rxjs-dev.firebaseio.com/>

- Procesamiento de eventos con RxJS



```
fromEvent(this.movieSearchInput.nativeElement, 'keyup').pipe(  
  map(event => event.target.value)  
  // if character length greater than 2  
  ,filter(res => res.length > 2)  
  // Time in milliseconds between key events  
  ,debounceTime(1000)  
  // If previous query is different from current  
  ,distinctUntilChanged()  
  // subscription for response  
) .subscribe(text => {  
  this.searchGetCall(text).subscribe((res)=>{  
    this.apiResponse = res;  
  },err =>{  
    console.log('error',err);  
  });  
});
```

Estas librerías se basan en la programación reactiva funcional