UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Clase 4 - Asincronismo

Introducción (*)

JavaScript ejecuta el código secuencialmente en orden descendente. Sin embargo, hay algunos casos en los que el código se ejecuta (o debe ejecutarse) después de que ocurra otra cosa y también de forma no secuencial. Esto se llama **programación asíncrona**.

Resulta que JavaScript es un lenguaje de programación asincrono. Lo que quiere decir esto es que al ejecutar código JavaScript el hilo de ejecución continuará a pesar de encontrarse en situaciones en las que no obtenga un resultado inmediatamente. Por ejemplo, cuando hacemos el pedido de información a un servidor, la respuesta posiblemente demore un poco. Sin embargo, el hilo de ejecución de JavaScript continuará con las demás tareas que hay en el código.

Los callbacks aseguran que una función no se va a ejecutar antes de que se complete una tarea, sino que se ejecutará justo después de que la tarea se haya completado. Nos ayuda a desarrollar código JavaScript asíncrono y nos mantiene a salvo de problemas y errores.

Un ejemplo practico de esto seria una aplicación web que necesita llenar una tabla de datos, asi que el código hará un pedido al servidor de los datos que necesita llenar. Pero el hilo de ejecución no se detiene asi que ejecutará el código que pinta la tabla en el navegador. Esto se convierte en un problema ya que los datos del servidor llegan después de que la tabla se haya pintado en pantalla, una tabla sin tados obviamente.

Asi que trabajar con código asincrono puede tener muchas ventajas pero en casos como este presenta un gran problema. Pues bien, para solucionar esto algunas funciones de JavaScript tienen como parametro algo que se conoce como callback, que se exponen a continuación.

Callbacks (*)

Un callback es simplemente una función que se pasa como parametro a otra funcion. De forma más compleja: en Js, **las funciones son objetos**. Por ello, las funciones admiten otras como argumentos y pueden ser devueltas por otras funciones. Las funciones que hacen esto se denominan funciones de orden alto (high-order). Cualquier función que se pase como argumento se denomina función **Callback**.

En JS es posible construir funciones que usen callbacks de la siguiente manera.

```
function imprimir(callback){
    callback();
}

imprimir(function() {
    console.log('Texto impreso!');
});
```

En el ejemplo anterior la función **callback** se pasa en forma anónima a la función imprimir() que es la encargada de invocarala. Otra forma de verlo sería:

```
function imprimir(callback){
    callback();
}

const imprimir_a_consola = function() {
    console.log('Texto impreso!');
};

imprimir(imprimir_a_consola);
```

Función setTimeout() (*)

Una función muy conocida en Js que tiene un callback es **setTimeout()**. Esta función ejecuta el callback despúes de esperar cierto tiempo el cual también le pasamos como parámetro.

```
// setTimeout(funcion, tiempo_ms);
setTimeout(function(){
    console.log('Hola')
}, 2000);
```

Esta función ejecuta el callback solo despúes de que hayan pasado 2000 milisegundos.

Con ayuda de esta función podemos crear un código que nos permita visualizar la asincronia de JavaScript.

```
// Usando función anonima:
console.log('A')
setTimeout(function () {
    console.log('B')
}, 2000)
console.log('C')
/* Resultado del código anterior
 A
 C
 B  Despues de 2 segundo*/
```

La asincronia de JS nos permite visualizar en consola la letra C mucho antes de la B, el cual demora 2 segundos. Como se puede ver la ejecución no se detiene dos segundos, esta continua y luego de dos segundos aparece C.

Para tener un cierto control en el código asincrono de JavaScript existe un concepto muy interesante: **las promesas**.

Promesas (*)

Las promesas como mecanismo propio de JS llegan en la versión 6 del estandar EcmaScript, hasta entonces era necesario utilizar librerías (como JQuery) o frameworks externos para poder incorporarlas.

Una promesa es el objeto que representa la respuesta de una tarea asíncrona, es decir la respuesta a una tarea que de antemano no es posible saber cuándo se obtendrá. Por esta razón se deja preparado dentro de la promesa el código que se ejecutará cuando el resultado llegue o incluso cuando el resultado es un error. Un ejemplo típico es la obtención de respuesta a un pedido HTTP, que luego se analizará en detalle.

Toda promesa puede tener 4 estados:

- Pendiente: Es su estado inicial, no se ha cumplido ni rechazado.
- Cumplida: La promesa se ha resuelto satisfactoriamente.
- Rechazada: La promesa se ha completado con un error.
- Arreglada: La promesa ya no está pendiente. O bien se ha cumplido, o bien se ha rechazado.

Es posible crear una promesa de la siguiente manera.

```
let  x = 11
const p = new Promise((resolve, reject) =>{
    if(x == 10){
        resolve('La variable es igual a 10');
    }else{
        reject('La variable no es igual a 10');
    }
});
```

La función flecha del código anterior recibe dos parámetros, el primero **resolve** es una función que recibe como parametro el objeto que queremos que devuelva cuando el código tuvo el resultado que esperamos. Mientras que **reject** es una función que toma como parametro el objeto que devolverá si existe un error en el código asincronico.

En resumen usando una promesa es posible recibir el resultado que se necesita de una espera y ejecutar código luego de que el resultado llegue. Una forma sencilla de probar el uso de una promesa es mediante la función **setTimeout** mencionada anteriormente.

```
let mensaje = new Promise((resolve, reject)=>{
    setTimeout(function () {
        resolve('Este es el mensaje');
    }, 2000);
})
```

De esta forma se crea un objeto promesa con un mensaje como resultado favorable que se devolverá luego de 2 segundos. Ahora para controlar la promesa se utilizan los métodos: **then** y **catch** que vienen junto con las promesas.

```
mensaje.then(m =>{
    console.log(m)
}).catch(function () {
    console.log('error');
})
```

Entonces para capturar el resultado favorable de la promesa se usa then(). En este caso se trata del mensaje que se muestra usando un console.log(). Por otra parte el catch captura el resultado fallido o reject de la promesa. En este caso no implementamos ninguno código específico, por lo que sencillamente se ejecuta la función anónima que muestra la palabra *error* por consola.

El método Fetch (*)

Una promesa es similar a un tipo de dato y es por este motivo que muchas funciones de JavaScript y/o de librerias externas cuyo resultado es asincrono, o sea que demorará un tiempo en llegar están implementados en promesas. Uno de estos métodos es fetch.

La función **fetch** permite hacer una petición a un API y es justamente un callback. Por lo que tenemos que recibirlo usando then y catch de la siguiente forma.

```
const pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

pokemones.then(res => res.json())
    .then(data => {
        console.log(data.name);
    }).catch(error => console.log(error))
```

Esta es la forma que comunmente se utiliza a fetch pero hay que acomodarlo de tal manera que sea un poco más entendible.

```
let pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

pokemones
    .then(res => {
        return res.json()
    })
    .then(data => {
        console.log(data.name);
    }).catch(error => console.log(error))
```

Como puedes ver en el código anterior se pueden encadenar métodos then, pero se pueden hacer solo cuando el return del then anterior es una promesa. En este caso si nosotros vemos la implementación de json() veremos que este método devuelve una promesa. Por lo que el segundo then es la captura del resolve del json(). Asi que si intentamos descomponer más al código anterior tendríamos algo asi.

```
// primera promesa
let pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

// segunda promesa
let respuesta = pokemones.then(res => {return res.json()});

respuesta.then(data => {
    console.log(data.name);
})
```

Entonces si nosotros quisieramos ejecutar el fetch de manera ordenada y sincrona para ver los cinco primeros pokemones podemos hacer lo siguiente.

```
function obtener_pokemon(id){
    let url = "https://pokeapi.co/api/v2/pokemon/" + id;
    return fetch(url).then(res => {return res.json()});
}

obtener_pokemon(1).then(data => {
    console.log(data.name);
    return obtener_pokemon(2);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(3);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(4);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(5);
}).then(data =>{
    console.log(data.name);
})
```

Async Await (*)

Una de las caracteristicas más recientes de Javascript: **async / await**. Usamos el async para definir una función donde se encontrará el await que nos permitirá esperar una promesa de tal forma que podamos volver nuestro código sincrono.

```
function obtener_pokemon(id){
    let url = "https://pokeapi.co/api/v2/pokemon/" + id;
    return fetch(url).then(res => {return res.json()});
}

async function nombrar_pokemones() {
    let pokemon1 = await obtener_pokemon(1);
```

```
    console.log(pokemon1.name);
}
```

Usando el **async await** podemos lograr esto, que la variable espere por su resultado antes de ejecutar el console.log(). Y como podemos hacer que una variable espere podemos lograr incluso esto.

```
async function nombrar_pokemones() {
    for (let i = 1; i < 6; i++) {
        let pokemon = await obtener_pokemon(i);
        console.log(pokemon.name);
    }
}
```

Como se puede observar, el **async await** nos permite esperar el **resolve de una promesa** de tal manera que podamos obtener un resultado sincrono.

O incluso podemos hacerlo usando forEach de la siguiente manera.

```
function obtener_pokemones(){
    let url = "https://pokeapi.co/api/v2/pokemon/";
    return fetch(url).then(res => {return res.json()});
}

async function nombrar_pokemones() {
    let pokemones = await obtener_pokemones();
    pokemones.results.forEach(pokemon => {
        console.log(pokemon.name)
    })
}
```

En este caso lo que pedimos al api es una lista con los 20 primeros pokemones pero estos están como una lista de objetos dentro del objeto result en la respuesta json que obtendremos. Asi que usamos el forEach para recorrer cada elemento del objeto results. Sin embargo, muchas veces lo que buscaremos será llevar una lista de objetos directamente a un arreglo para eso existe el método **map**. El cual lo podemos utilizar de la siguiente forma.

```
async function nombrar_pokemones() {
    let pokemones = await obtener_pokemones();
    let arregloPokemones = pokemones.results.map(pokemon => pokemon.name)
    console.log(arregloPokemones)
}
```

El método map recibe una función flecha donde a partir del parametro que recibe que es cada elemento del results retorna lo que queremos que esté dentro del arreglo.