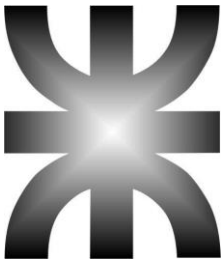

CALLBACKS

UNIDAD #5 - JS

TUP - Laboratorio de Computación II

MATERIAL COMPLEMENTARIO

Callbacks

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción.

Ejemplo:

```
function saludar(nombre) {
    alert('Hola ' + nombre);
  }

  function procesarEntradaUsuario(callback) {
    var nombre = prompt('Por favor ingresa tu nombre.');
    callback(nombre);
  }

  procesarEntradaUsuario(saludar);
```

El ejemplo anterior es una callback sincrónica, ya que se ejecuta inmediatamente.

Sin embargo, tenga en cuenta que las callbacks a menudo se utilizan para continuar con la ejecución del código después de que se haya completado una operación a sincrónica — estas se denominan devoluciones de llamada asincrónicas.

Pero veamos otro ejemplo. Imaginemos el siguiente bucle tradicional para recorrer un Array:

```
const list = ["A", "B", "C"];

for (let i = 0; i < list.length; i++) {
  console.log("i=", i, " list=", list[i]);
}
```

En `i` tenemos la posición del array que estamos recorriendo (*va de 0 a 2*) y con `list[i]` accedemos a la posición del array para obtener el elemento, es decir, desde `A` hasta `C`. Ahora veamos, como podemos hacer este mismo bucle utilizando el método `forEach()` del Array al cual le pasamos una función callback:

```
list.forEach(function(e,i) {
    console.log("i=", i, "list=", e);
  });
```

Esto se puede reescribir como:

```
["A", "B", "C"].forEach((e,i) => console.log("i=", i, "list=", e));
```

Lo importante de este ejemplo es que se vea que la **función callback** que le hemos pasando a `forEach()` se va a ejecutar por cada uno de los elementos del array, y en cada iteración de dicha **función callback**, los parámetros `e`, `i` van a tener un valor especial:

- `e` es el elemento del array
- `i` es el índice (*posición*) del array

Callbacks en Javascript

Una vez entendido esto, vamos a profundizar un poco con las **funciones callbacks** utilizadas para realizar tareas asíncronas. Probablemente, el caso más fácil de entender es utilizar un temporizador mediante la función `setTimeout(FUNCTION callback, NUMBER time)`.

Dicha función nos exige **dos parámetros**:

- La función `callback` a ejecutar
- El tiempo `time` que esperará antes de ejecutarla

Así pues, el ejemplo sería el siguiente:

```
setTimeout(function () {
    console.log("He ejecutado la función");
}, 2000);
```

Simplemente, le decimos a `setTimeout()` que ejecute la **función callback** que le hemos pasado por primer parámetro cuando transcurran `2000` milisegundos (*es decir, 2 segundos*). Utilizando **arrow functions** se puede simplificar el callback y hacer mucho más «fancy» y legible:

```
setTimeout(() => console.log("He ejecutado la función"), 2000);
```

Si lo prefieres y lo ves más claro (*no suele ser habitual en código Javascript, pero cuando se empieza suele resultar más fácil entenderlo*) podemos guardar el callback en una constante:

```
const action = () => console.log("He ejecutado la función");
setTimeout(action, 2000);
```

En cualquiera de los casos, lo importante es darse cuenta que estamos usando una **función callback** para pasársela a **setTimeout()**, que es otra función. En este caso, se trata de «programar» un suceso que ocurrirá en un momento conocido del futuro, pero muchas veces desconoceremos cuando se producirá (*o incluso si se llegará a producir*).

Si probamos el código que verás a continuación, comprobarás que el segundo **console.log()** se ejecutará **antes** que el primero, dentro del **setTimeout()**, mostrando primero **Código síncrono** y luego **Código asíncrono** en la consola del navegador:

```
setTimeout(() => console.log("Código asíncrono."), 2000);
console.log("Código síncrono.");
```

El último **console.log** del código se ejecuta primero (*forma parte del flujo principal de ejecución del programa*). El **setTimeout()** que figura en una línea anterior, aunque se ejecuta antes, pone en espera a la función callback, que se ejecutará cuando se cumpla una cierta condición (*transcurran 2 segundos desde ese momento*).

Esto puede llegar a sorprender a desarrolladores que llegan de otros lenguajes considerados **bloqueantes**; Javascript sin embargo se considera un lenguaje **asíncrono y no bloqueante**. ¿Qué significa esto? Al ejecutar la línea del **setTimeout()**, el programa no se queda bloqueado esperando a que terminen los 2 segundos y se ejecute la función callback, sino que continúa con el flujo general del programa para volver más adelante cuando sea necesario a ejecutar el **callback**, aprovechando así mejor el tiempo y **realizando tareas de forma asíncrona**.

Asincronía con callbacks

Las **funciones callback** pueden utilizarse como un primer intento de manejar la asincronía en un programa. De hecho, eran muy utilizadas en la época dorada de jQuery, donde muchas funciones o librerías tenían una estructura similar a esta (*en jQuery se usaba algo similar*):

```
function doTask(number, callback) {
    /* Código de la función */
```

```
}

doTask(42, function (err, result) {
    /* Trabajamos con err o result según nos interese */
});
```

Observa que **doTask()** es la función que realiza la tarea en cuestión. Puede tener los parámetros que se consideren adecuados, como cualquier otra función, la diferencia es que establecemos un **callback** que usaremos para controlar lo que se debe hacer.

Más adelante, llamamos a la función **doTask()** y en su parámetro **callback** pasamos una función con dos parámetros; **err** y **result**. El primero de ellos, **err**, utilizado para controlar un error y el segundo de ellos, **result**, utilizado para manejar los valores devueltos.

En primer lugar, veamos la implementación de la función **doTask**:

```
/* Implementación con callbacks */
const doTask = (iterations, callback) => {
    const numbers = [];
    for (let i = 0; i < iterations; i++) {
        const number = 1 + Math.floor(Math.random() * 6);
        numbers.push(number);
        if (number === 6) {
            /* Error, se ha sacado un 6 */
            callback({
                error: true,
                message: "Se ha sacado un 6"
            });
            return;
        }
    }
    /* Termina bucle y no se ha sacado 6 */
    return callback(null, {
        error: false,
        value: numbers
    });
}
```

Como se puede ver, estamos utilizando **arrow functions** para definir la función **doTask()**. Le pasamos un parámetro **iterations** que simplemente indica el número de iteraciones que tendrá el bucle (*número de lanzamientos del dado*). Por otro lado, el segundo parámetro es nuestro **callback**, que recordemos que es una función, por lo que podremos ejecutarla en momentos concretos de nuestro código. Lo hacemos en dos ocasiones:

- En el **if** cuando **number** es **6** (*detectamos como error cuando obtenemos un 6*). Le pasamos un objeto por parámetro que contiene un **error** y **message**, el mensaje de error.
- Tras el **for**, con dos parámetros. El primero NULL, ya que en este caso no hay error. El segundo parámetro un objeto que contiene un campo **value** con el array de resultados.

Teniendo claro esto, veamos la llamada a la función **doTask()**, donde le pasamos esa función **callback** e implementamos el funcionamiento, que en nuestro caso serán dos simples **console.error()** y **console.log()**:

```
doTask(10, function (err, result) {
    if (err) {
        console.error("Se ha sacado un ", err.message);
        return;
    }
    console.log("Tiradas correctas: ", result.value);
});
```

Esto es una forma clásica donde utilizamos una **función callback** para gestionar la asincronía y facilitar la reutilización, pudiendo reutilizar la función con la lógica, aplicando diferentes **funciones callback** según nos interese.

Observa que, aunque en este ejemplo se ha utilizado un parámetro **err** y otro **result** en el callback para gestionar un objeto de error y un objeto de resultados, esto puede modificarse a gusto del desarrollador, aunque lo habitual suele ser este esquema.

Desventajas de los callbacks

A pesar de ser una forma flexible y potente de controlar la asincronía, que permite realizar múltiples posibilidades, las **funciones callbacks** tienen ciertas desventajas evidentes. En primer lugar, el código creado con las funciones es algo caótico y (*quizás subjetivamente*) algo feo. Por ejemplo, el tener que pasar un NULL por parámetros en algunas funciones, no es demasiado elegante.

Pero, sobre todo, uno de los problemas evidentes viene a la hora de tener que gestionar la asincronía varias veces en una misma función, donde al introducir varias funciones con callbacks en su interior, conseguimos una **estructura anidada** similar a la siguiente:

```
firstTask(data, function(err, result) {
  secondTask(data, function(err, result) {
    thirdTask(data, function(err, result) {
      fourthTask(data, function(err, result) {
        fifthTask(data, function(err, result) {
          // Code
        });
      });
    });
  });
});
```

La forma triangular que produce es conocida como **Callback Hell** o **Pyramid of Doom**, debido a su forma, resultando un código muy poco elegante que se puede complicar demasiado de cara a la legibilidad. Es cuando entran en juego **las promesas.**

Otros Ejemplos

```
function calculate(num1, num2, callbackFunction) {
    return callbackFunction(num1, num2);
}

function calcProduct(num1, num2) {
    return num1 * num2;
}

function calcSum(num1, num2) {
    return num1 + num2;
}
// alerta a 75, producto de 5 y 15
alert(calculate(5, 15, calcProduct));
// alerta a 20, la suma de 5 y 15
alert(calculate(5, 15, calcSum));
```

Función Tradicional

```
function myDisplayer(some) {
    document.getElementById("demo").innerHTML = some;
}
```

```
function myCalculator(num1, num2) {
    let sum = num1 + num2;
    return sum;
}

let result = myCalculator(5, 5);
myDisplayer(result);
```

- o

```
function myDisplayer(some) {
    document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2) {
    let sum = num1 + num2;
    myDisplayer(sum);
}

myCalculator(5, 5);
```

Con Callbacks

```
function myDisplayer(some) {
    document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Referencias

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

https://developer.mozilla.org/es/docs/Glossary/Callback_function

https://lenguajejs.com/javascript/asincronia/callbacks/