Funciones callback

Una función callback no es más que una función que se pasa como parámetro a otra función para ser utilizada más adelante.

Es decir algo así:

```
function procesarEntradaUsuario(callback) {
  var usuario = prompt('Hola usuario. Dime tu nombre.');
  callback(usuario);
}
```

A la función **procesarEntradaUsuario** se le va a pasar como parámetro una función. La función que se pasa como parámetro se ha llamado callback y no lleva paréntesis.

La función **procesarEntradaUsuario** solicita al usuario que le proporcione su nombre, a continuación lo almacenará en la variable usuario y posteriormente llamará a la función que se ha pasado como parámetro y le pasará el usuario introducido como parámetro a dicha función.

Un detalle muy importante es que cuando pasamos una función como parámetro, lo único que se debe pasar como parámetro es el código de la función, pero no se debe ejecutar la función. Para ello se debe escribir el nombre de la función sin paréntesis, o bien escribir directamente el código de la función mediante una función anónima.

Un ejemplo típico de uso de una función callback lo podemos encontrar en el método forEach() de los arrays:

Definition and Usage

El método `forEach()` llama a una función para cada elemento en un array.
El método `forEach()` no se ejecuta para elementos vacíos.

See Also:

The Array map() Method
The Array filter() Method

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Arrays</h1>
<h2>The forEach() Method</h2>

<p>forEach() calls a function for each element in an array:</p>

<p id="demo"></p>

<script>
let text = "";
const fruits = ["apple", "orange", "cherry"];
fruits.forEach(myFunction);          //Se pasa “myFunction” como parámetro, y se ejecutará para
/                                    //cada elemento del array

document.getElementById("demo").innerHTML = text;

//myFunction recibirá dos parámetros, primero un elemento del array y en segundo lugar el
// índice de dicho elemento
function myFunction(item, index) {
  text += index + ": " + item + "<br>";
}
</script>

</body>
</html>
```

Utilidades de una función callback

a)Las callback ofrecen una forma de ampliar la funcionalidad de una función sin cambiar su código.

Es decir, conseguir que una función realice operaciones diferentes según nos convenga, sin tener que reescribirla.

Supongamos que tenemos una función que se encarga de saludar al usuario cuyo nombre se le pase como parámetro, mediante un mensaje con un alert:

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

Esta función la podríamos utilizar para pasarla como parámetro a nuestra función procesarEntradaUsuario(), la cual acepta como parámetro una función. Esto quedaría así:

```
procesarEntradaUsuario(saludar);
```

SALUDO1.HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
  <script> //Saludo1
     function procesarEntradaUsuario(callback){
      var usuario = prompt("Hola usuario. Dime tu nombre.");
      callback(usuario);
     }

     function saludarAlert(nombre){
       alert("Hola "+nombre);
     }


     procesarEntradaUsuario(saludarAlert);

  </script>
  </body>
</html>
```

Si resulta que también tenemos una función que se encarga de saludar al usuario, pero a través de la consola:

```
function saludarPorConsola(nombre) {
  console.log('Hola ' + nombre);
}
```

Tan solo tendríamos que pasar la nueva función como parámetro y listo.

SALUDO2.HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
  <script> //Saludo2
     function procesarEntradaUsuario(callback){
      var usuario = prompt("Hola usuario. Dime tu nombre.");
      callback(usuario);
     }

     function saludarAlert(nombre){
       alert("Hola "+nombre);
     }

     function saludarPorConsola(nombre) {
        console.log('Hola ' + nombre);
     }

     procesarEntradaUsuario(saludarPorConsola);

  </script>
  </body>
</html>
```

Una forma muy habitual de pasar funciones como parámetro es utilizar una función anónima. De esta forma se pasa como parámetro directamente el código de la función:

SALUDO3.HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
  <script> //Saludo3
     function procesarEntradaUsuario(callback){
      var usuario = prompt("Hola usuario. Dime tu nombre.");
      callback(usuario);
     }

     procesarEntradaUsuario(function(nombre){alert("Hola"+nombre);});
     procesarEntradaUsuario(function(nombre){console.log("Hola"+nombre);});

  </script>
  </body>
</html>
```

En estos casos es muy habitual utilizar funciones flecha:

SALUDO4.HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
  <script> //Saludo3
     function procesarEntradaUsuario(callback){
      var usuario = prompt("Hola usuario. Dime tu nombre.");
      callback(usuario);
     }

     procesarEntradaUsuario(nombre => alert("Hola"+nombre));
     procesarEntradaUsuario(nombre => console.log("Hola"+nombre));

  </script>
  </body>
</html>
```

b)Asegurarnos de que una función no se ejecuta hasta que otra termina.

Lo normal es que el código se ejecute de forma secuencial, y que las funciones que definimos se ejecuten una después de otra.

Por ejemplo, vamos a definir un par de funciones que den un mensaje por la consola:

Primero se ejecutará la función A y cuando esta termine se ejecutará la función B. Esta es la forma normal de hacer las cosas. Es una ejecución secuencial y **síncrona**.

Sin embargo, en JavaScript muy a menudo tenemos instrucciones que se ejecutan de forma **asíncrona**.

Un ejemplo típico de ejecución asíncrona es la gestión de eventos. Cuando definimos la gestión de un evento, por ejemplo una función que muestra un mensaje al pulsar un botón:

Este sería un ejemplo básico de gestión de eventos. En este caso no hay ninguna función callback.

```
<!DOCTYPE html>
<html>
<body>

<button id="myBtn" onclick="myFunction()">¡Saluda al mundo!</button>

<p id="demo"></p>

<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World!";
}
</script>

</body>
</html>
```

En este otro ejemplo, se utiliza otra forma de gestión de eventos y sí se utiliza una función callback:

```
<script>
const element = document.getElementById("myBtn");
element.addEventListener("click", myFunction);

function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World";
}
</script>
```

Con addEventListener() se indica el evento que se quiere controlar (“click”) y qué función se ejecutará cuando se produzca dicho evento (“myFunction”). Como se puede observar, “myFunction” se pasa como parámetro a addEventListener(). Esto es una función callback. Pero además la ejecución será asíncrona, ya que “myFunction” no se ejecutará en el momento en el que se ejecuta addEventListner(), sino que habrá que esperar hasta que se produzca el evento (“click”) y entonces se vuelve a llamar a la función “myFunction”, y es en ese momento cuando se ejecuta. Esto ya no es una ejecución secuencial.

Otro ejemplo de una ejecución asíncrona se produce cuando desde el cliente se solicita un dato a un servidor. Este es el comportamiento típico de la comunicación AJAX: la comunicación asíncrona.

Si una función solicita un dato a un servidor, la función quedará esperando hasta que reciba la respuesta del servidor. Pero en este caso, no se detiene la ejecución de todo el programa, sino que se continuará la ejecución del mismo. Cuando se reciba la respuesta del servidor entonces se reanudará la ejecución de esa función que se había quedado esperando la respuesta del servidor.

La clave de usar callbacks está en que la callback se le pasa como parámetro a la función asíncrona para que así, la función asíncrona pueda ejecutar la callback una vez que haya concluido su tarea.

Ejemplo: loadScript(1)

Echa un vistazo a la función loadScript(src), que carga un código script src dado:

```
function loadScript(src) {
  // crea una etiqueta <script> y la agrega a la página
  // esto hace que el script dado: src comience a cargarse y ejecutarse cuando se
  //complete
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Esto inserta en el documento una etiqueta nueva, creada dinámicamente, <script src =" ... "> con el código src dado. El navegador comienza a cargarlo automáticamente y lo ejecuta cuando la carga se completa.

Esta función la podemos usar así:

```
// cargar y ejecutar el script en la ruta dada
loadScript('/my/script.js');
```

El script se ejecuta “asincrónicamente”, ya que comienza a cargarse ahora, pero se ejecuta más tarde, cuando la función ya ha finalizado.

El código debajo de loadScript (...), no espera que finalice la carga del script.

```
loadScript('/my/script.js');
// el código debajo de loadScript
// no espera a que finalice la carga del script
// ...
```

Digamos que necesitamos usar el nuevo script tan pronto como se cargue. Declara nuevas funciones, y queremos ejecutarlas.

Si hacemos eso inmediatamente después de llamar a loadScript (...), no funcionará:

```
loadScript('/my/script.js'); // el script tiene a "function newFunction() {…}"
newFunction(); // no hay dicha función!
```

Puedes comprobarlo con el ejemplo **cargarScript1.html.**

Ejemplo: loadScript(2)

Aquí está la variante basada en callback:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script para $ {src}`));

  document.head.append(script);
}
```

Puedes comprobar que funciona en el ejemplo **cargarScript2.html.**

Lo que hemos hecho en este ejemplo ha sido pasarle a la función que tiene una ejecución asíncrona (la fundión loadScript) una callback.
La función loadScript llamará a la función que le hemos pasado como parámetro (la función callback) cuando haya terminado de cargar el script, esto lo podremos averiguar cuando se dispare el evento script.onload.

Una función que hace algo de forma asincrónica debería aceptar un argumento de callback donde ponemos la función por ejecutar después de que se complete.

Ejemplo de AJAX y función callback:
https://www.w3schools.com/xml/tryit.asp?filename=tryajax_callback

Algunos vídeos en YouTube:
https://www.youtube.com/watch?v=zQVnDW8SaA0

https://www.youtube.com/watch?v=p3Oq3AfuteA&list=RDLVzQVnDW8SaA0&index=25