EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: INTRODUCCIÓN A CALLBACKS.
- EXERCISE 2: FUNCIONES ASÍNCRONAS.

EXERCISE 1: INTRODUCCIÓN A CALLBACKS

Como se definió en el Text Class Review, los callbacks son funciones que se pasan como argumentos en otra función. Revisemos un ejemplo que nos ayudará a entenderlos.

```
function saludo(nombre) {
        alert(`Hola ${nombre}`);
}

function procesaEntrada(callback) {
        var nombre = prompt('Ingresa tu nombre por favorM.');
        callback(nombre);
}
procesaEntrada(saludo);
```

Te recomendamos usar este código, y ver cómo funciona en tu consola, seguramente notarás que es un código bastante sencillo. En este ejemplo tenemos definidas dos funciones. La primera se llama `saludo()`, recibe un parámetro `"nombre"` como argumento, y muestra una alerta con la concatenación de un saludo con la variable `nombre`. Mientras que la segunda función `procesaEntrada()`, trabaja con lo ingresado por el usuario y recibe por parámetro un `"callback"`.

> Ten en cuenta que `"callback"` no es una palabra reservada, podríamos haber puesto cualquier otro nombre. Ésta solo fue utilizada para mostrar donde se coloca, y donde se llama dentro de la función en la que es utilizada. También recuerda que el `callback` en realidad representa la función `saludo()` en forma de parámetro.

El argumento `"callback"` después se llama dentro de la función `procesaEntrada()`, y se le pasa como argumento la variable `"nombre"` que solicita al usuario que ingrese su propio nombre mediante un `prompt`.

Al final, como muestra la línea 9, llamamos a `procesaEntrada()`, y le pasamos como parámetro la función `saludo`.

Un detalle importante sobre los callbacks es que, al pasar una función como parámetro de otra, no se le pueden colocar unos paréntesis a ese parámetro, pues dejaría de ser un callback y pasaría a ser ***una llamada*** a esa función. Entonces, al trabajar con ellos, nunca debemos hacer lo siguiente:

```
1  //Una llamada a la función
2  function procesaEntrada(callback()) { //incorrecto.
3          var nombre = prompt('Ingresa tu nombre por favor.');
4          callback(nombre);
5  }
6
7  //Un callback
8  function procesaEntrada(callback) { //correcto.
9          var nombre = prompt('Ingresa tu nombre por favor.');
10         callback(nombre);
11 }
```

Ahora, otra forma en que podemos usar callbacks que funcionan con su propio conjunto de parámetros, es colocando la función completa en el lugar en donde debería ir el parámetro, o usando funciones de flecha. Revisemos la primera forma.

Al llamar a una función que utiliza un callback, podemos hacerlo de 3 maneras: la primera es como ya lo hemos hecho, declarando una función y su callback por separado, y luego pasando el nombre del callback como un parámetro; la segunda forma es pasando el callback por completo como un parámetro; y la tercera es haciendo lo mismo, pero utilizando las funciones de fecha.

Veamos ahora la segunda forma. Usando las mismas funciones con las que hemos estado trabajando, pasaremos la función callback completa en el lugar donde solo iría su nombre.

```
1 function procesaEntrada(callback) {
2     var nombre = prompt('Ingresa tu nombre por favor.');
3     callback(nombre)
4 }
5
6 // Pasando el callback completo como parámetro
7 procesaEntrada(function (nombre) {
8     alert(`Hola ${nombre}`);
9 });
```

Como podemos ver, nuestra función callback que antes se llamaba `saludo()`, ahora se pasa completamente como el parámetro del `procesaEntrada()`. Se puede replicar este mismo procedimiento usando funciones de flecha como se muestra a continuación:

```
function procesaEntrada(callback) {
    var nombre = prompt('Ingresa tu nombre por favor.');
    callback(nombre)
}

// Usando arrow functions.
procesaEntrada((nombre) => {
    alert(`Hola ${nombre}`);
});
```

Esto cubre todas las características de callbacks sincrónicos. En el próximo ejercicio, aprenderemos acerca de los callbacks asincrónicos.

EXERCISE 2: FUNCIONES ASÍNCRONAS

Los callbacks son mayormente usados en funciones asíncronas, que son aquellas ejecutadas en paralelo en vez de en secuencia, donde una función se ejecuta solo después de que lo haga otra. Para mejorar nuestra comprensión de este tipo de funciones, primero demostraremos que JavaScript se ejecuta, por defecto, de forma secuencial de arriba a abajo.

```
let primeraAccion = () => console.log("primero");
let segundaAccion = () => console.log("segundo");

primeraAccion();
segundaAccion();
```

Si ahora nos dirigimos a nuestra consola, podremos ver que el mensaje `"primero"` aparece antes que el mensaje `"segundo"`. Esto es evidencia del comportamiento sincrónico de JS, ya que `primeraAccion()` debía terminar de ejecutarse antes de que `segundaAccion()`. Realmente no hay nada nuevo en esto, pues todo sucedió como se esperaba; luego, vamos a usar una función asincrónica que toma un callback como parámetro.

La función `setTimeout()` es un ejemplo perfecto, ya que toma dos parámetros: un callback y una cantidad de tiempo en milisegundos. Ésta retrasa la ejecución de un callback. Si pasamos `primeraAccion()` como

callback de `setTimeout()`, veremos cómo cambian el orden de los mensajes. Nuestro código ahora se ve así:

```
let primeraAccion = () => console.log("primero");
let segundaAccion = () => console.log("segundo");

// pasamos primeraAccion como callback
setTimeout(primeraAccion,3000)
segundaAccion();
```

Y nuestra consola muestra lo siguiente:

```
segundo                                    script.js:5
primero                                    script.js:4
>
```

Como podemos ver, sucedió algo muy interesante. JavaScript ejecutó la primera función, pero como tuvo que esperar tres segundos, siguió adelante y ejecutó la segunda al mismo tiempo. Es por eso que la palabra "segundo" aparece exactamente tres segundos antes que la palabra "primero". El hecho que estas dos funciones se ejecuten en paralelo, constituye una función sincrónica, y dado el hecho de que estábamos usando callback, se denominan ***funciones de callback asincrónicas***.

Ahora consideraremos un ejemplo similar, usando la función `setInterval()`, que toma los mismos parámetros que la función `setTimeout()` que acabamos de usar, y ejecuta un callback en intervalos o "cada cierta cantidad de tiempo". En este caso, se empleará esta función para generar un "reloj" que mostrará la hora, y la actualizará cada segundo. Para hacerlo, agregaremos un nuevo elemento a nuestro HTML:

```
<h1 id="reloj"></h1>
```

Además, en nuestro archivo JS incluiremos el siguiente código:

```
function mostrarHora() {
    let d = new Date();

document.getElementById("reloj").innerHTML=`${d.getHours()}:${d.getMinutes(
)}:${d.getSeconds()}`;
}

setInterval(mostrarHora, 1000);
```

Esto mostrará el siguiente resultado en nuestro navegador:

Como podemos apreciar, cada vez que llamamos a la función `setInterval()`, estamos ejecutando el callback `mostrarHora()` cada segundo (o cada 1000 milisegundos). Este es solo otro ejemplo de JavaScript ejecutando ambas funciones: `setInterval()` y el callback al mismo tiempo.

Los ejemplos que hemos analizado hasta el momento son sencillos, pues tienen la intención de facilitar el aprendizaje sobre los callbacks, y las funciones asincrónicas que hacen uso de ellos. Un ejemplo más real, es cuando se desea ejecutar una acción mientras se carga un recurso desde un servidor.

En el siguiente ejemplo, usaremos un callback para mostrar un recurso de un archivo local, el cual simulará cómo lo haría un programa al interactuar con un servidor o al esperar un archivo.

```
// Definimos nuestro callback
function mostrar(algo) {
    document.getElementById("aqui").innerHTML = algo;
}

// Metodo para consumir un recurso externo
function consumirArchivo(myCallback) {
    // Se inicializa el objeto XMLHttpRequest
    let req = new XMLHttpRequest();
    // Se configura una solicitud de tipo GET
    req.open('GET', "referencia.html");

    // En esta funcion se carga los datos del doc.
    req.onload = function () {
        // Si existe el recurso, usar el contenido.
        if (req.status == 200) {
            myCallback(this.responseText);
        } else {
            // Si no existe, mostrar un error.
            myCallback("Error: " + req.status);
        }
    }

    // Inicializa el request.
    req.send();
}

consumirArchivo(mostrar);
```

En este código definimos 2 funciones. La primera es nuestro callback, y simplemente establece la información que le pasamos por parámetro en el documento HTML. Nuestro segundo método es el que usaremos para consumir una fuente externa, simulando cómo un programa interactuaría con un servidor. Aunque parezca complicado, en realidad no lo es, revisémoslo con atención. Inicializamos el objeto `XMLHttpRequest`, y luego configuramos una solicitud de tipo GET a una ruta, que en este caso solo es un archivo en la raíz de nuestro proyecto llamado `recurso.html`. ¿Qué contiene este archivo?: lo siguiente

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <p>Este recurso simula el consumo de un servidor usando un
callback.</p>
</body>

</html>
```

Posteriormente, trabajamos con los datos obtenidos del archivo. Si el archivo existe, desplegará el contenido; pero si no existe, mostrará un error. Por último, se inicializa la solicitud de tipo GET, concluyendo con la llamada a este método pasándole nuestro callback como parámetro.

Se utilizan los objetos `XMLHttpRequest` para interactuar con los servidores. Puede recuperar datos de una URL sin tener que actualizar la página completa. Esto permite que una página web actualice solo una parte de una página, sin interrumpir lo que está haciendo el usuario.

Al haber creado el archivo X en la raíz de nuestro proyecto, debemos esperar ver el siguiente resultado en nuestro navegador:

Este recurso simula el consumo de un servidor usando un callback.

Si no existiera el archivo, se desplegaría lo siguiente:

Error: 404

De esta forma queda demostrado cómo utilizar los callbacks y las funciones asíncronas, y también cómo unir estos dos conocimientos para realizar peticiones asíncronas con callbacks.