Javascript

Carrera Programador full-stack

Callbacks

Qué veremos!!

Tema principal funciones callbacks en Javascript

- Funciones sincrónicas y sus límites
- Asincronismo
- Que es un callback?
- Porque utilizarlas?.
- Gestión de errores
- Problemas comunes

Funciones sincrónicas

Hasta ahora las funciones que utilizamos se ejecutan de forma síncrona esto significa que se sigue una secuencia de ejecución y las instrucciones se ejecutan una después de otra.

Funciones sincrónicas

```
function suma(a, b) {
  return a + b;
}
console.log(suma(3, 5));
// ## Resultado en pantalla ##
// 8
```

En el ejemplo vemos una función normal, donde el resultado de la operación es devuelto usando la instrucción return; esto es llamado el estilo directo, y es la forma más común de devolver un resultado en una operación síncrona.

Funciones callback

El equivalente al ejemplo anterior usando una función callback.

```
function suma(a, b) {
 return a + b;
}

function oper(a, b, callback) {
 console.log(callback(a, b));
}

console.log('Antes de la ejecucion');
oper(3, 5, suma);
console.log('Despues de la ejecucion');

// ## Resultado en pantalla ##
// Antes de la ejecucion
// 8
// Despues de la ejecucion
```

Con este ejemplo se recibe suma con el nombre callback y se ejecuta

Se envía la función suma

Funciones callback

```
function resta(a, b) {
  return a - b;
}

function oper(a, b, callback) {
  console.log(callback(a, b));
}

console.log('Antes de la ejecucion');

oper(8, 5, resta);

console.log('Despues de la ejecucion');

// ## Resultado en pantalla ##
// Antes de la ejecucion
// 3
// Despues de la ejecucion
```

Con este ejemplo se recibe resta con el nombre callback y se ejecuta

Se envía la función resta

Funciones callback

```
function oper(a, b, operacion) {
  console.log(operacion(a, b));
}
console.log('Antes de la ejecucion');

oper(3, 5, function (a, b) {
    return a + b;
});

console.log('Despues de la ejecucion');
// ## Resultado en pantalla ##
// Antes de la ejecucion
// 8
// Despues de la ejecucion
```

No es necesario que el parámetro de la función se llame callback (puede tener cualquier nombre)

Cambiamos utilizando función anónima

Funciones callback

```
function oper(a, b, callback) {
  console.log(callback(a, b));
}
console.log('Antes de la ejecucion');

oper(3, 5, (a, b) => a + b);

console.log('Despues de la ejecucion');
// ## Resultado en pantalla ##
// Antes de la ejecucion
// 8
// Despues de la ejecucion
```

Con funciones flecha

Funciones sincrónicas

Esta forma de ver los códigos en la vida real sería como:

Esperando que se lave la ropa

Esperando que se laven los platos

Esperando que se cocine la comida

Funciones asincrónicas

Pero la realidad es que los lenguajes permiten la ejecución de sentencias de manera asincrónica esto significa que se pueden ejecutar instrucciones en paralelo, es decir “al mismo tiempo”.

Para realizar esta acción Javascript lo hace permitiendo enviar una función como parámetro de otra (llamados callbacks).

Funciones asincrónicas

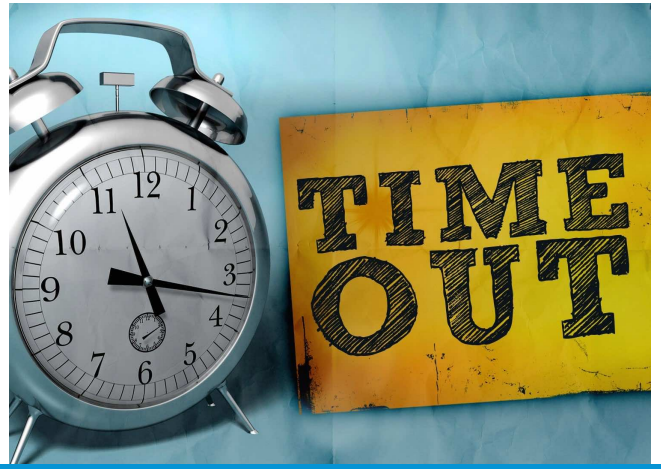
```
function oper(a, b, callback) {
  setTimeout(function() {
    console.log(callback(a, b));
  }, 500);
}
console.log('Antes de la ejecucion');

oper(3, 10, (a, b) => a + b);

console.log('Despues de la ejecucion?');
// ## Resultado en pantalla ##
// Antes de la ejecucion
// Despues de la ejecucion?
// 13
```

Lo mismo que utilizamos antes pero agregando un retraso en la ejecución del callback

Asincronismo con setTimeout

En el ejemplo anterior cambia el orden de la ejecución del código, mostrando al final el resultado de la suma. Usamos la función `setTimeout` para simular una operación asíncrona.

La función `setTimeout` toma como primer parámetro una función callback y como segundo parámetro el tiempo de demora en ejecutar el contenido de la función callback.



Asincronismo con setTimeout 2

Como `setTimeout` es una función asíncrona, no espera para ejecutar el código de la función callback; lo que hace es devolver el control a la función `oper`, y esta sigue la ejecución del programa, dejando la función callback a la espera en el ciclo de eventos de javascript para ser ejecutada.

Este es el funcionamiento por defecto de como javascript maneja las operaciones asíncronas.

Si quieres saber más sobre el ciclo de eventos de javascript puedes visitar:
https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop

Asincronismo en eventos

JavaScript no es un lenguaje asíncrono por naturaleza. Sin embargo, los callbacks de JavaScript suelen usarse para crear código asíncrono cuando los utilizamos con las APIs del entorno de ejecución de JavaScript, ya sea un navegador o el entorno de Node.js.

Por ejemplo, puedes pasar una función como callback a los eventos del navegador, como por ejemplo los eventos onClick, onMouseOver u onChange.

Asincronismo evento click

No sabrás cuando un usuario hará clic en un botón, pero puedes crear un handler que gestione el evento cuando suceda. El handler acepta una función como callback, que será ejecutada cuando se inicie el evento:

```
document.getElementById('boton').addEventListener('click', ()
=> {
  console.log('Se ha hecho clic en el botón');
});
```

Asincronismo evento load

También es muy habitual agregar código al evento load del objeto window del navegador, que ejecutará la función callback que definamos cuando la página se haya cargado y el DOM esté listo:

```
window.addEventListener('load', () => {
  console.log('Página cargada');
});
```

Problema de los callbacks

Las funciones callback son ideales para casos simples, pero no están exentas de problemas cuando el código se complica.

Cuando anidamos varios callbacks, cada uno de ellos agrega un nivel de profundidad a la cola de mensajes.

Si bien JavaScript puede gestionar estas situaciones sin problema, el código puede volverse difícil de leer.

Además, también será más difícil saber en dónde ha ocurrido un error.

Ej. Problema de los callbacks

A continuación anidamos cuatro callbacks, lo cual no es extraño, seguramente encuentres casos mucho más extremos:

```
window.addEventListener('load', () => {
  document.querySelector('#button').addEventListener('click',
() => {
    setTimeout(() => {
      items.forEach(item => {
        console.log(`Item: ${item}`);
      })
    }, 2000)
  });
});
```

Extremo del problema callbacks

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 la legibilidad.

```
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
        });
      });
    });
  });
});
```

Callbacks Hell

```
const lugaresVisitados = []
function irDePaseo() {
  setTimeout(function() {
    lugaresVisitados.push('Salta');
    setTimeout(function() {
      lugaresVisitados.push('Cordoba');
      setTimeout(function() {
        lugaresVisitados.push('Jujuy');
        setTimeout(function() {
          lugaresVisitados.push('Tierra del fuego');
          console.log(lugaresVisitados.toString())
        },1000);
      },1000);
    },1000);
  },1000);
}
irDePaseo();// OUTPUT : Salta, Cordoba, Jujuy, Tierra del fuego
```

¿Qué ocurre si cambiamos los tiempos por valores diferentes entre sí?.

Consejos

Implementar una función callback en javascript es fácil, una vez que entiendas cómo funciona. Si no la usas de forma correcta, a medida que tu código crece en complejidad, puedes terminar con un código muy ilegible, difícil de depurar al usar funciones callbacks en javascript.

Consejos

- **Callback al final:** Usa la función callback como último parámetro. La mayoría de las funciones callbacks son funciones anónimas, creadas donde pasas los parámetros. Al ubicar la función callback al final y creas una función anónima, tu código quedará más legible.

- **Evita las callbacks hell:** Los callbacks hell son un antipatrón muy común cuando usas callbacks en javascript. Esto sucede, cuando tienes muchas callbacks anidadas, dando como resultado la dificultad de leer o seguir la estructura del código. Para evitarlas intenta no crear muchos niveles de anidación, usa funciones normales en vez de funciones anónimas para separar las callbacks o utiliza otra soluciones como promesas y async/await (una solución más moderna de cómo manejar operaciones asíncronas que veremos más adelante).

Consejos

- **Prefiere el estilo directo:** Si usas solo funciones puramente síncronas, es recomendable usar el **estilo directo**, en vez de, propagar el resultado usando una función callback.

- **Conoce el estilo de la función:** Siempre lee la documentación de la función que usa la callback para saber que tipo de estilo está usando. Conociendo el estilo de la función puedes determinar el diseño de tu código.

Ejercicios de integración

Ejercicio para realizar en clase.

Veamos un ejemplo donde integremos HTML, CSS y JS en la utilización de callbacks.

```
<body>
    <div class="container">
        <h2 class="word">Programación</h2>
        <h2 class="word">FullStack</h2>
        <h2 class="word">TdF</h2>
    </div>
</body>
```

Ejercicios de integración

```css
.word {
    color: #5e5ce6;
    font-size: 4rem;
    transition: all .5s ease-in;
    margin: 0 5px;
    transform: translateY(3.8rem);
    opacity: 0;
}

body {
    display: flex;
    justify-content: center;
    align-items: center;
    width: 95vw;
    height: 95vh;
}
```

```css
.container {
    overflow: hidden;
    display: flex;
    flex-direction: column;
    align-items: center;
}

.animate {
    opacity: 1;
    transform: translateY(0);
}
```

Ejercicios de integración

```
let words = document.querySelectorAll(".word");
const animateAll = (animate) => {
    setTimeout(() => {
        animate(words[0]);
        setTimeout(() => {
            animate(words[1]);
            setTimeout(() => {
                animate(words[2]);
            }, 1000)
        }, 1000)
    }, 1000)
}
const animate = (word) => {
    word.classList.add("animate");
}

animateAll(animate);
```

Ejercicio en clase

Del ejercicio visto juega cambiando los tiempos de los setTimeout para analizar qué pasa.

Piensa cómo podría hacerse para modificar el ejercicio utilizando una iteración en lugar de llamar a cada elemento.

Información

Fuentes

https://www.freecodecamp.org/news/what-is-a-callback-function-in-javascript/

https://www.neoguias.com/callbacks-javascript/

https://antonioweb.dev/articulo/funcion-callback-javascript-guia-completa

Ejercicios

Carrera Programador full-stack

Ejercicio 1

```
// Main function
const mainFunction = (callback) => {
  setTimeout(() => {
      callback([2, 3, 4]);
  }, 2000)
}

// Add function
const add = (array) => {
  let sum = 0;
  for(let i of array) {
      sum += i;
  }
  console.log(sum);
}

// Calling main function
mainFunction(add);
```

1. Analiza el siguiente código y describe que realiza.
2. Cómo modificarías para que se muestren los elementos de la lista por consola?.
3. Y para que se listen en diferente orden utilizando el asincronismo?. ¿Cómo harías para que setTimeout se ejecute con diferentes tiempos?.

Ejercicio 2

Crea una función con callback con la cual se puedan realizar las operaciones básicas de una calculadora (suma, resta multiplicación y división) enviando diferentes funciones para cada operación.
Agrega los códigos html y css para realizar una calculadora donde los valores los ingresa el usuario y tenga una estética aceptable.

Ejercicio 3

Realiza los códigos de una página sencilla como la vista en clase donde utilizando eventos y funciones callbacks permitas al usuario generar cajas con algún efecto, el usuario debe poder definir la cantidad de cajas que se realizarán y el tiempo de demora que tendrán en mostrarse en la página. Utiliza inputs para permitirle al usuario ingresar la información y un botón para que ejecute la acción una vez completo los parámetros.