JavaScript: What are Callbacks?

In JavaScript, a callback is a function that is passed as an argument to another function and is executed when that function has finished its work.

Using callbacks in this way allows us to write code that is asynchronous and non-blocking , which is important for performance and user experience.

Why use callbacks?

Callbacks are commonly used in JavaScript for asynchronous operations. When a function is executed asynchronously, it doesn't block the execution of the rest of the code. Instead, the function is executed in the background, and the rest of the code continues to execute. Once the asynchronous function has finished its work, it calls the callback function, which then handles the result of the operation.

Asynchronous operations are common in web development, where applications often make API calls to fetch data from a server. When a user clicks a button to trigger an API call, the JavaScript code doesn't wait for the response to come back before continuing to execute. Instead, it sets up the API call and registers a callback function to handle the response once it arrives. Here's an example of using a callback with an API call in JavaScript:

```
function fetchUserData(userId, callback) {
  // Make an API call to fetch user data
  fetch(`https://api.example.com/users/${userId}`)
    .then(response => response.json())
    .then(data => callback(data));
}

function displayUserData(userData) {
  // Display the user data on the page
  // ...
}

fetchUserData(123, displayUserData);
```

In this example, the `fetchUserData` function makes an API call to fetch user data for a specific user ID. Once the data is available, the `then` method is used to parse the response as JSON and call the callback function (`displayUserData`) with the data as an argument. The `displayUserData` function then handles the result of the API call by displaying the user data on the page.

Problems with callback hell

1) Callback Hell

2) Inversion of Control

1)Callback Hell -

Callback Hell is a situation in JavaScript where multiple nested callback functions make your code look like it's been through a blender on the highest setting.

Here's an example of what callback hell might look like:

```
asyncOperation1(function(result1) {
  asyncOperation2(result1, function(result2) {
    asyncOperation3(result2, function(result3) {
      // More nested callbacks ...
    });
  });
});
};
```

😨😨😨😨

As you can see, each asynchronous operation requires a callback function, and when you have multiple operations depending on each other, the code structure becomes deeply nested and hard to follow.

Callback hell can make it challenging to read, analyze, and maintain code. Additionally, it may make problems like error handling more complicated and prone to mistakes.

Let's illustrate this with a real-world example:

Meet Mr. Callback Bob, a JavaScript developer with a penchant for getting himself into sticky situations.

Bob's task for the day is to make a series of asynchronous calls to fetch data from a fictitious API called "Dad Joke Central." He wants to **fetch a random dad joke, translate it into another language, and then post it to a website**.

Bob starts off with good intentions but **ends up in Callback Hell:** 😳

```
fetchRandomJoke((joke) => {
    console.log(joke);

    translateJoke(joke, (translatedJoke) => {
        console.log(translatedJoke);

        postJoke(translatedJoke, () => {
            console.log("Joke posted successfully!");
        });
    });
});
```

Poor Bob! 😳 He's now stuck in a deep pit of callbacks, and his code looks like a staircase to nowhere.😄

The readability of the code has gone out the window, and Bob's sanity is hanging by a thread.😅

✨ Escaping Callback Hell:

1. Promises to the Rescue - With promises, he can chain asynchronous operations together and use “then” and “catch” methods to make the code more readable... 🥷

```
fetchRandomJoke()
    .then((joke) => {
        console.log(joke);
        return translateJoke(joke);
    })
    .then((translatedJoke) => {
        console.log(translatedJoke);
        return postJoke(translatedJoke);
    })
    .then(() => {
        console.log("Joke posted successfully!");
    })
    .catch((error) => {
        console.error("Something went wrong:", error);
    });
```

2. **Embrace the Power of Async/Await -**a newer JavaScript feature that makes asynchronous code look almost synchronous with better error-handling capabilities: 🤓

```
async function tellJoke() {
    try {
        const joke = await fetchRandomJoke();
        console.log(joke);

        const translatedJoke = await translateJoke(joke);
        console.log(translatedJoke);

        await postJoke(translatedJoke);
        console.log("Joke Translated & posted successfully!");
    } catch (error) {
        console.error("Something went wrong:", error);
    }
};

tellJoke();
```

With async/await, Bob's code is not only more readable but also easier to reason about.

😎😎

He can clearly see the flow of his asynchronous operations.

This makes it easier to model and implement complex workflows.

With the help of promises and async/await, you can rescue your code from the clutches of Callback Hell and make it more readable, maintainable, and enjoyable to work with.

2) Inversion of Control

pyramidical piece of code agitate people to utter terms like ***Callback hell,*** or is there something major going on behind the scenes? Yes, there is one great problem that is associated with callback hell that goes under the radar, and that is **"Inversion of Control".**

in an asynchronous JavaScript context is **"the first part of our program that executes now and the second part that executes in the callback and when we give that callback to someone else (like, for example, an external API). That's what inverts the control and puts them in charge of executing the second part of our program."**

A good short example of above is the **setTimeout()**,is an utility of the Timer API, and that API is responsible for calling the callback for us, which is **inverting the control**.

WHY "INVERSION OF CONTROL " IS A BIG DEAL?

when you pass a callback to a third-party library, you rely on them to execute your callback, that may cause some **problems** like:

- **A Callback may be called multiple times.**
- **A Callback would never get called.**
- **A Callback may be called synchronously.**

Let's understand this with a example

you have a third-party library that provides you with an utility called ***createOrder()*** that needs to be called with the order details. So the third-party library does all the analytics and stuff, and they are going to call your callback and let you know so that you can charge the customer and do all the payment-related stuff.

people start buying from your website, and everybody is happy with your work. Suddenly, one day, you get a call from your boss regarding a major crisis. John Cena tried to purchase a gaming console on our site, and his credit card was charged four times. Well, they've refunded his credit card, but your boss is not happy;

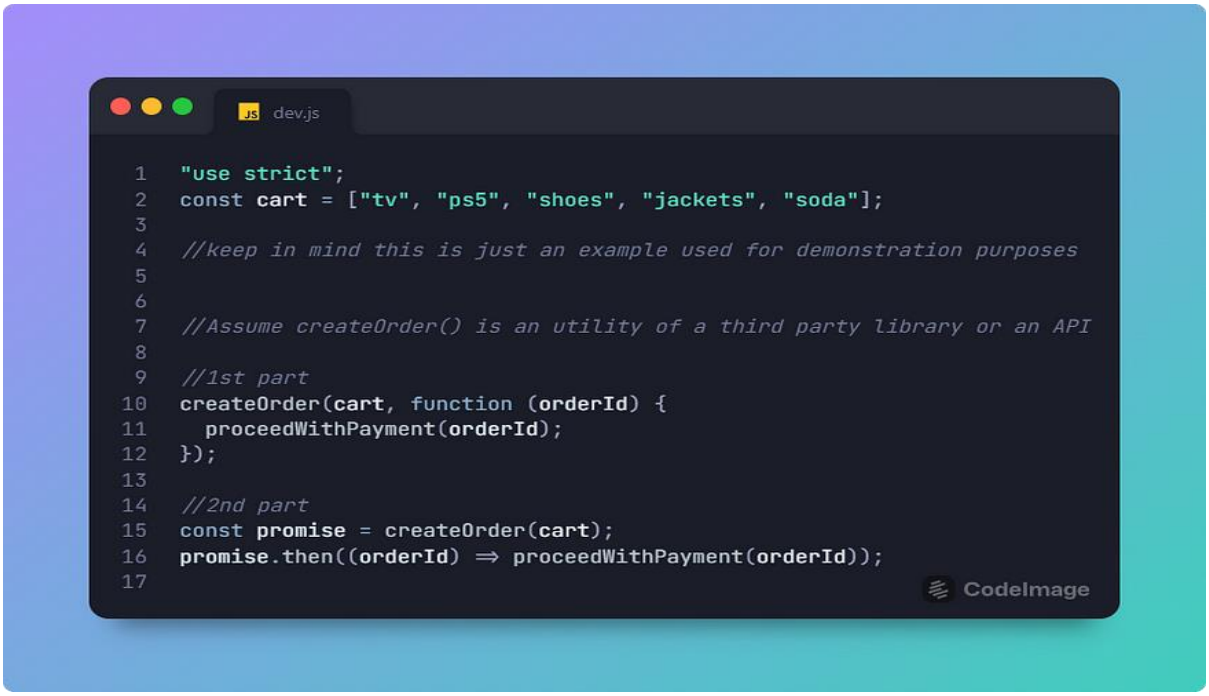
You eventually arrive at the conclusion that there's no other possible way for the credit card to have been charged four times other than for the ***createOrder()*** checkout utility to have called your callback four times.

HOW DO I SOLVE THIS PROBLEM ?

Promises are designed specifically to solve all these trust issues with callbacks. I'm not going to go in depth on what promises are, but if you don't know what a promise is, just think of it as a " **container for an asynchronously delivered value(An object that is used as a placeholder for the future result of an asynchronous operation)**".

Promises allow us to not rely on callbacks passed into asynchronous functions to handle asynchronous results.

Instead, we can chain Promises for a sequence of asynchronous operations, escaping the ***Callback hell.***



In the first part of the above example, we are **inverting the control** to the third-party library or API. We rely on the API to execute our callback, but keep in mind the problems that I told you about callbacks earlier.

In the second part, we are using a Promise object instead of a callback, so as soon as the Promise object gets filled, we can use the **'then()'** method to call our **proceedWithPayment()** function. So we are the **ones in control** this time and can easily make sure that our function gets called only once.