🗐 Understanding Callbacks in JavaScript: The Good & The Bad

JavaScript uses **callbacks** as a core part of asynchronous programming. But like everything, callbacks come with both benefits and drawbacks.

✅ Two Sides of Callbacks

1. 🌟 The Good Side

- Callbacks are **essential** for writing asynchronous code in JavaScript.
- They allow execution of code after an operation completes (e.g., after an API call, file read, timer, etc.).

2. ⚠️ The Bad Side

Using callbacks can lead to:

- 😵 **Callback Hell** A deeply nested and hard-to-read structure when callbacks depend on other callbacks.

- 📶 **Inversion of Control** We lose control over execution when we rely on third-party or external functions to execute our callbacks.

💡 Core JavaScript Nature

JavaScript is a **synchronous**, **single-threaded** language. It has **one call stack**, and it can run only **one operation at a time**.

```
console.log("Namaste");
console.log("JavaScript");
console.log("Season 2");
```

🖨️ **Output:**

```
Namaste
JavaScript
Season 2
```

> ⏱️ JavaScript executes code quickly, without waiting. Like they say: **"Time, tide, and JavaScript wait for none."**

⏳ Delaying Execution with Callbacks

```
console.log("Namaste");

setTimeout(function () {
    console.log("JavaScript");
}, 5000);

console.log("Season 2");
```

🖨️ **Output:**

```
Namaste
Season 2
JavaScript
```

Here, `setTimeout()` delays execution of `"JavaScript"` by 5 seconds using a **callback**.

🛍️ e-Commerce Callback Example

Imagine a user placing an order with cart items:

```
const cart = ["shoes", "pants", "kurta"];
```

Steps to place an order:

1. ✅ Create Order
2. 💳 Proceed to Payment

❌ Problem without Callback (No Guarantee of Sequence)

```
api.createOrder();
api.proceedToPayment();
```

Here, there's **no guarantee** that `createOrder()` finishes before `proceedToPayment()` starts.

✅ Fixing It with Callback

```
api.createOrder(cart, function () {
    api.proceedToPayment();
});
```

Now, `proceedToPayment()` runs **only after** `createOrder()` is complete.

🔁 Chaining More Actions

Now, you want to:

1. Show order summary after payment.
2. Update wallet after summary.

```
api.createOrder(cart, function () {
    api.proceedToPayment(function () {
        api.showOrderSummary(function () {
            api.updateWallet();
        });
    });
});
```

⚠️ This is **Callback Hell** – aka Pyramid of Doom

- Deep nesting makes code **hard to read**, **debug**, and **maintain**.
- Happens often in real apps with many async steps (like file uploads, data processing, APIs).

🔄 Inversion of Control

When we pass a callback, we hand over **control** to another function and **trust** it will:

- Call our function
- Do it correctly
- Do it once (not zero or twice!)

Example:

```
api.createOrder(cart, function () {
    api.proceedToPayment();
});
```

Here, we **blindly trust** `createOrder()` to execute `proceedToPayment()`.

But what if:

- The developer of `createOrder()` forgets to call the callback?
- It gets called twice or never?
- Bugs creep in because of mismanagement?

🧠 **This is called "Inversion of Control".** We're giving away control, and that's risky!

🎯 Key Takeaway

- Callbacks are **powerful**, but overusing them can lead to **messy**, **unreliable** code.
- These issues led to the evolution of **Promises** and **async/await** — which we’ll explore next.

🌐 Learn More

🔗 Visit: callbackhell.com 📺 Watch: Live session on YouTube (link below if available)
