Callbacks and callback hell

Sure! Let's explore **Callbacks** and the concept of **Callback Hell** in detail.

🔁 What is a Callback?

A **callback** is a function that is **passed as an argument** to another function and is **invoked inside that function** to complete a certain task.

In JavaScript, callbacks are commonly used to **handle asynchronous operations** like reading files, making API requests, or setting timeouts.

🔧 Basic Example

```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback();
}

function sayBye() {
  console.log("Goodbye!");
}

greet("Abhi", sayBye);
```

Output:

```
Hello, Abhi
Goodbye!
```

- `sayBye` is passed as a **callback** to the `greet` function.

📦 Callbacks in Asynchronous Operations

Callbacks are powerful in **non-blocking** code (asynchronous tasks).

Example: Using `setTimeout`

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data fetched!");
    callback();
  }, 2000);
}

function processData() {
  console.log("Processing data...");
}

fetchData(processData);
```

Output (after 2 seconds):

```
Data fetched!
Processing data...
```

⚠️ What is Callback Hell?

Callback Hell refers to a situation where multiple nested callbacks make the code:

- Difficult to read
- Hard to debug
- Challenging to maintain

🔥 Example of Callback Hell

```
getUser(function(user) {
  getPosts(user.id, function(posts) {
    getComments(posts[0].id, function(comments) {
      console.log("User, Posts, and Comments loaded");
    });
  });
});
```

- Each function is nested inside the previous one.
- This “**Pyramid of Doom**” makes the code messy and hard to follow.

😬 Problems with Callback Hell

1. **Readability**: Hard to understand deeply nested logic.
2. **Maintainability**: Difficult to modify or extend.
3. **Error Handling**: Managing errors in multiple levels becomes complex.
4. **Inversion of Control**: You lose control over the program’s flow.

✅ How to Avoid Callback Hell

1. Modularize Callbacks

Break functions into separate named functions.

```
function onUserLoaded(user) {
  getPosts(user.id, onPostsLoaded);
}

function onPostsLoaded(posts) {
  getComments(posts[0].id, onCommentsLoaded);
}

function onCommentsLoaded(comments) {
  console.log("All data loaded!");
}

getUser(onUserLoaded);
```

2. Use Promises

```
getUser()
  .then(user => getPosts(user.id))
  .then(posts => getComments(posts[0].id))
  .then(comments => console.log("All data loaded!"))
  .catch(error => console.error(error));
```

3. Use async/await (Modern & cleaner)

```
async function loadData() {
  try {
    const user = await getUser();
    const posts = await getPosts(user.id);
    const comments = await getComments(posts[0].id);
    console.log("All data loaded!");
  } catch (error) {
    console.error(error);
  }
}

loadData();
```

💡 Summary

Concept	Description
Callback	A function passed to another function to be executed later
Used for	Handling async operations (APIs, file I/O, timeouts)
Callback Hell	Deep nesting of callbacks that makes code hard to read
Solutions	Named functions, Promises, `async/await`

Would you like to go deeper into **Promises** or move on to **Closures and Lexical Scoping** next?