JavaScript Callbacks

Last Updated : 06 Feb, 2025

In JavaScript, callbacks play an essential role in handling asynchronous tasks like reading files, making API requests, and executing code after certain events. If you've ever heard the phrase "I will call back later!", that's exactly how callbacks work.

What is a Callback Function?

A callback function is a function that is passed as an argument to another function and executed later.

- A function can accept another function as a parameter.
- Callbacks allow one function to call another at a later time.
- A callback function can execute after another function has finished.

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

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

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

Output

```
Hello, Ajay
Goodbye!
```

Here, sayGoodbye() is passed as a callback to greet(), which executes after the greeting.

How Do Callbacks Work in JavaScript?

JavaScript executes code line by line (synchronously), but sometimes we need to delay execution or wait for a task to complete before running the next function. Callbacks help achieve this by passing a function that is executed later.

Callbacks for Asynchronous Execution

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

setTimeout(function () {
    console.log("Inside setTimeout");
}, 2000);

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

Output

```
Start
End
Inside setTimeout  (after 2 seconds)
```

- setTimeout() is an asynchronous function that takes a callback to execute after 2 seconds.
- The rest of the code continues executing without waiting.

Where Are Callbacks Used?

1. Handling Asynchronous Operations

Callbacks are widely used in

- API requests (fetching data)
- Reading files (Node.js file system)
- Event listeners (clicks, keyboard inputs)
- Database queries (retrieving data)

2. Callbacks in Functions Handling Operations

When a function needs to execute different behaviors based on input, callbacks make the function flexible.

```
function calc(a, b, callback) {
```

```
    return callback(a, b);
}

function add(x, y) {
    return x + y;
}

function mul(x, y) {
    return x * y;
}

console.log(calc(5, 3, add));
console.log(calc(5, 3, mul));
```

Output

```
8
15
```

- calculate() receives two numbers and a function (add or multiply).
- The passed function is executed inside calculate().

3. Callbacks in Event Listeners

JavaScript is event-driven, and callbacks handle user interactions like clicks and key presses.

```
document.getElementById("myButton").addEventListener("click",
function () {
    console.log("Button clicked!");
});
```

Here, the anonymous function is a callback that runs when the button is clicked.

4. Callbacks in API Calls (Fetching Data)

Callbacks are useful when retrieving data from APIs.

```
function fetch(callback) {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
        .then(response => response.json())
        .then(data => callback(data))
        .catch(error => console.error("Error:", error));
}

function handle(data) {
    console.log("Fetched Data:", data);
```

```
}

fetch(handle);
```

fetchData() gets data from an API and passes it to handleData() for processing.

Features of JavaScript Callbacks

- **Asynchronous Execution:** Handle async tasks like API calls, timers, and events without blocking execution.
- **Code Reusability:** Write modular code by passing different callbacks for different behaviors.
- **Event-Driven Programming:** Enable event-based execution (e.g., handling clicks, keypresses).
- **Error Handling:** Pass errors to callbacks for better control in async operations.
- **Non-Blocking Execution:** Keep the main thread free by running long tasks asynchronously.

Problems with Callbacks

Although callbacks are useful, they have some drawbacks.

1. Callback Hell (Nested Callbacks)

When callbacks are nested deeply, the code becomes unreadable and hard to maintain.

```
function step1(callback) {
    setTimeout(() => {
        console.log("Step 1 completed");
        callback();
    }, 1000);
}

function step2(callback) {
    setTimeout(() => {
        console.log("Step 2 completed");
        callback();
    }, 1000);
}

function step3(callback) {
    setTimeout(() => {
        console.log("Step 3 completed");
        callback();
    }, 1000);
```

```
}

step1(() => {
    step2(() => {
        step3(() => {
            console.log("All steps completed");
        });
    });
});
```

As the number of steps increases, the nesting grows deeper, making the code difficult to manage.

2. Error Handling Issues in Callbacks

Error handling can get messy when dealing with nested callbacks.

```
function divide(a, b, callback) {
    if (b === 0) {
        callback(new Error("Cannot divide by zero"), null);
    } else {
        callback(null, a / b);
    }
}

function result(error, result) {
    if (error) {
        console.log("Error:", error.message);
    } else {
        console.log("Result:", result);
    }
}

divide(10, 2, result);
divide(10, 0, result);
```

Output

```
Result: 5
Error: Cannot divide by zero
```

Handling errors inside callbacks can complicate code readability.

Alternatives to Callbacks

1. Promises (Fixing Callback Hell)

Promises provide a better way to handle asynchronous tasks without deep nesting.

```
function step1() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Step 1 completed");
            resolve();
        }, 1000);
    });
}

function step2() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Step 2 completed");
            resolve();
        }, 1000);
    });
}

function step3() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Step 3 completed");
            resolve();
        }, 1000);
    });
}

    .then(() => console.log("All steps completed"));
```


Promises make code more readable by chaining .then() instead of nesting callbacks.

2. Async/Await (Cleaner Alternative)

async/await provides an even cleaner way to handle asynchronous code.

```
async function processSteps() {
    await step1();
    await step2();
    await step3();
    console.log("All steps completed");
}

processSteps();
```

async/await makes code look synchronous, improving readability.

When to Use and Avoid Callbacks?

Use callbacks when

- Handling asynchronous tasks (API calls, file reading).
- Implementing event-driven programming.
- Creating higher-order functions.

Avoid callbacks when

- Code becomes nested and unreadable (use Promises or async/await).
- You need error handling in asynchronous operations (Promises are better).

Next Article
JavaScript typedArray.values() with Examples

Similar Reads

What is Callback Hell in JavaScript ?

One of the primary ways to manage asynchronous operations in JavaScript is through callback functions that execute after a certain...

15+ min read

JavaScript Coding Questions and Answers

JavaScript is the most commonly used interpreted, and scripted Programming language. It is used to make web pages, mobile...

15+ min read

Type Conversion and Type Coercion in JavaScript

Data types in JavaScript are flexible due to which the type of the variables can be changed when the program runs. Type Conversion and Type...