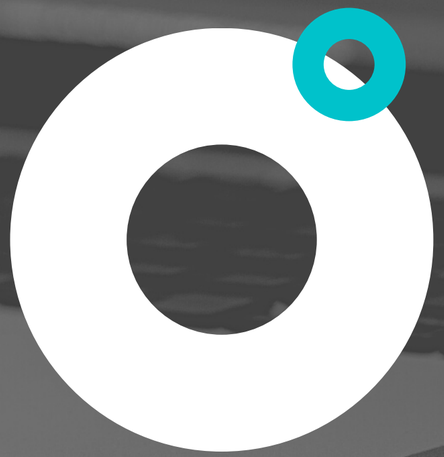
JAVASCRIPT CALLBACKS, ASYNC & AWAIT



BY
PRITEY MEHTA

cybercom creation

AGENDA

Callbacks
Async & Await

cybercom creation

WHAT IS CALLBACK?

A callback function in JavaScript is a function that is passed as an argument to another function, and is executed after the outer function has finished its execution. It is a way to handle asynchrony in JavaScript and is commonly used with event handlers, AJAX requests, and timers.

```
function parentFunction(cbFunc) {
  // some code here
  cbFunc();
}

function callbackFunction() {
  console.log("this is a callback function");
}

parentFunction(callbackFunction);
```

WHEN IT'S USEFUL?

Callbacks are very useful in JavaScript because they allow you to handle code that may take a long time to execute, such as an AJAX request, without freezing the UI or blocking other code from executing. The callback function is only executed when the data has been retrieved, allowing you to ensure that the rest of your code continues to run smoothly.

cybercom creation

SOME DRAWBACKS OF CALLBACKS

While callback functions are a very useful construct in programming, they can also have some drawbacks:

Readability: Callback functions can make code more difficult to read, especially when they are nested within multiple levels of functions. This can lead to what is sometimes referred to as "callback hell."

Debugging: Debugging can be more difficult with callback functions, especially when they are deeply nested or when they contain complex logic.

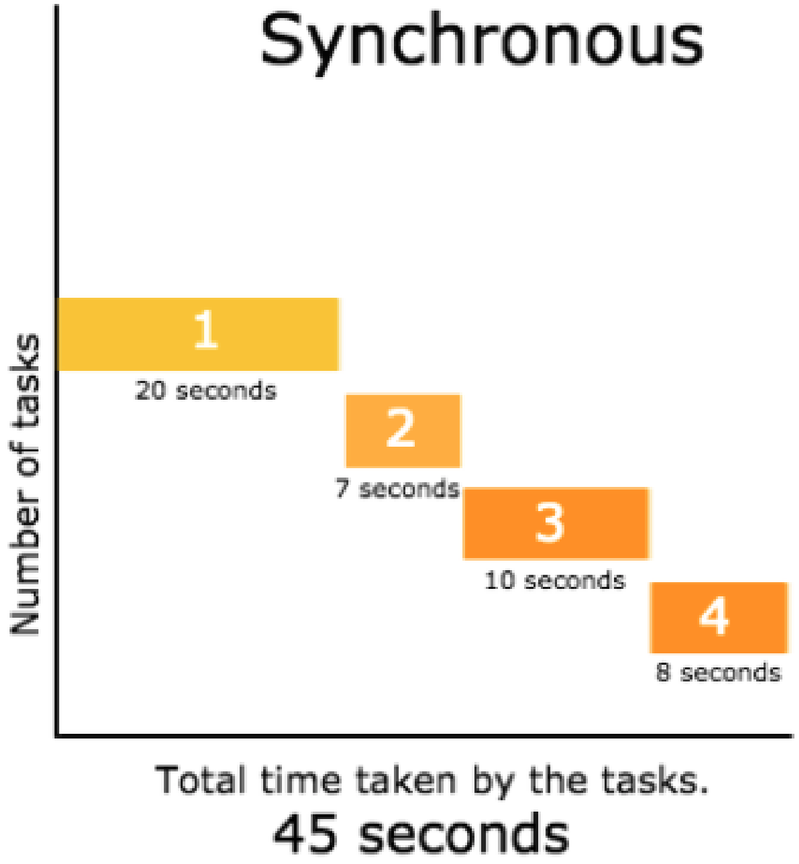
Sync Behaviour: Callback functions can introduce async behaviour into code, making it more difficult to understand and reason about the order of events in the program.

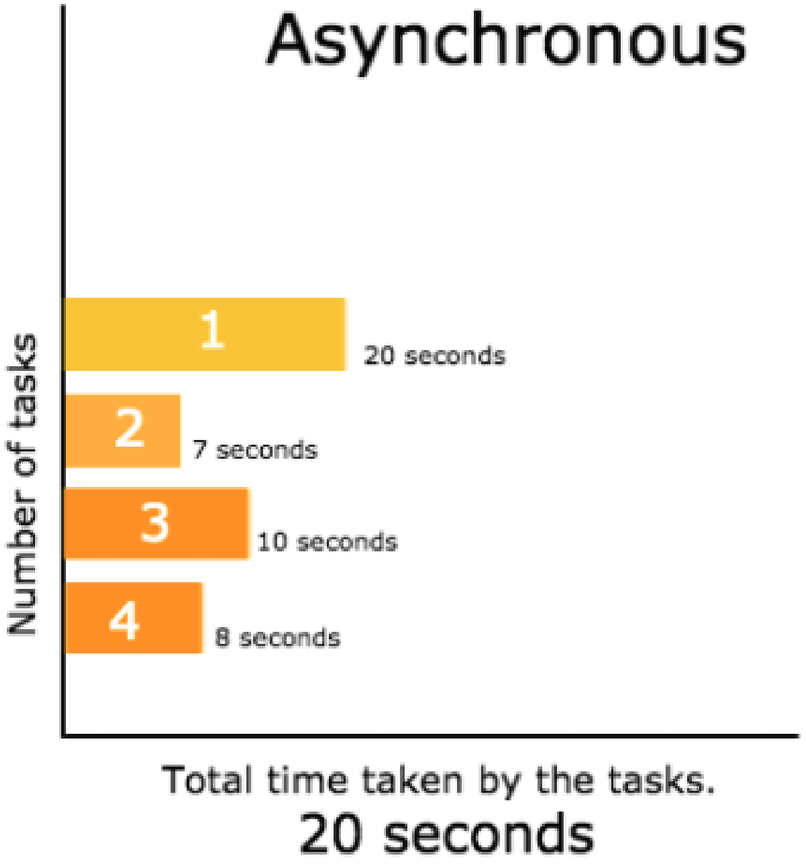
Memory Management: Callback functions can create closure variables that persist in memory, even after the function has completed execution. This can lead to memory leaks, especially in long-running programs.

ALTERNATE SOLUTIONS

There are several alternatives to using callback functions in programming, including:

- ✓ Promises
- ✓ Async/Await
- ✓ Generators


Synchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
45 seconds


Asynchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
20 seconds

cybercom creation

WHAT IS ASYNC/AWAIT?

Async/Await is a syntax for handling asynchronous code.

The async function declaration declares an async function where the await keyword is permitted within the function body.

JavaScript is always synchronous and single-threaded. If you're executing a JavaScript block of code on a page then no other JavaScript on that page will currently be executed.

JavaScript is only asynchronous in the sense that it can make, for example, Ajax calls. The Ajax call will stop executing and other code will be able to execute until the call returns.

Let's test with one example using setTimeout() function.

setTimeout() is non-blocking async function by default.

meaning that the timer function will not pause execution of other functions in the functions stack. In other words, you cannot use setTimeout() to create a "pause" before the next function in the function stack fires.


```
function _doSum(cbSub){
    setTimeout(function() {
        console.log("sum: after 5s")

    }, 5000);
    cbSub(); //callback function
}

function _doSub(){
     setTimeout(function() {
        console.log("sub: after 2s")

    }, 2000)
}


_doSum(_doSub);
```

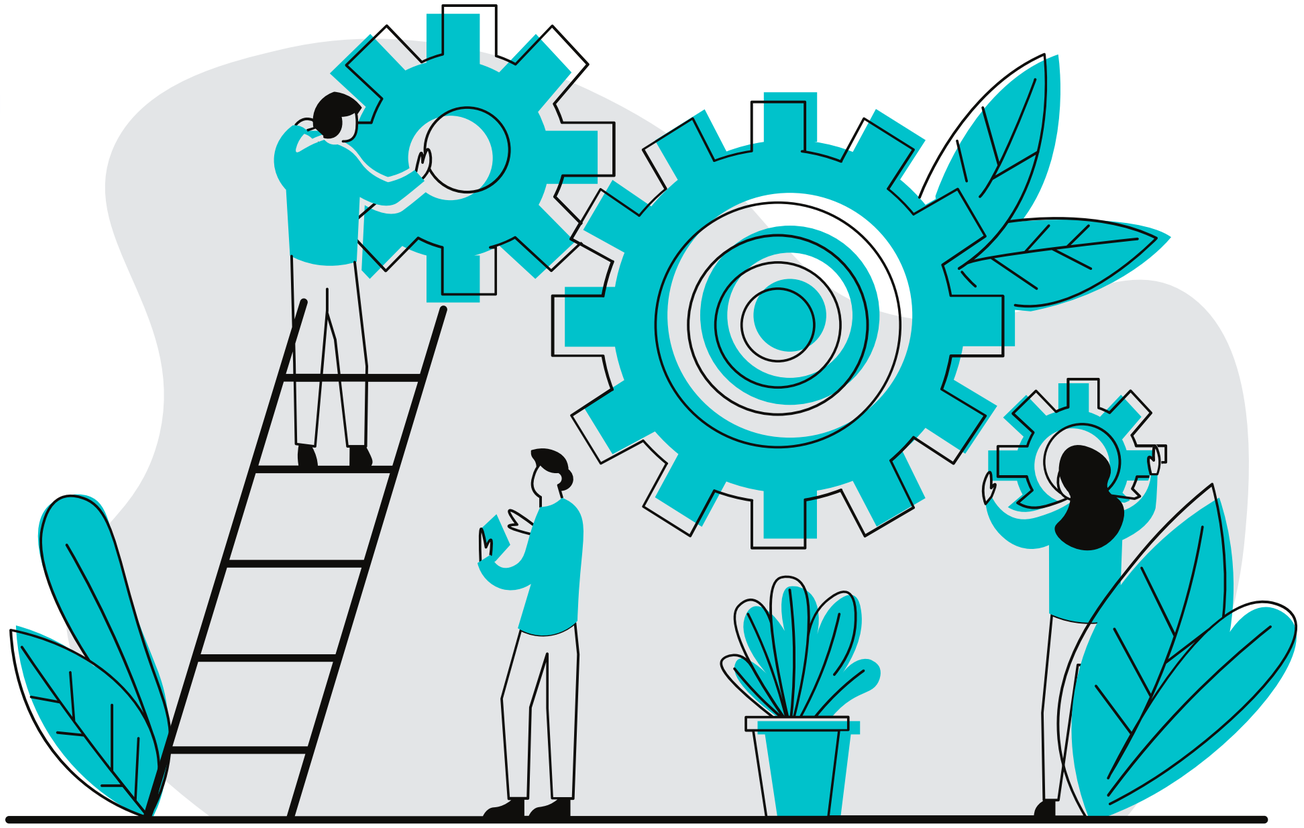
cybercom creation

TEST EXERCISE

Prepare a tea using
setTimeout() & Callbacks

steps:

1. boil water (5s)
2. boil tea leaves (2s)
3. add milk.



cybercom creation

SO WHAT'S THE SOLUTION?

you can use the **await** keyword to pause the execution of the function until a promise is resolved. The await keyword can only be used inside an async function.

By using async/await, you can write asynchronous code that is more readable and easier to understand than equivalent code written using callbacks or promises.

TRY IT !

await operator pauses the execution of the current async function until the operand Promise is resolved. When the Promise is resolved, the execution is resumed and the resolved value is used as the result of the await .

```
async function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function showDelayMessage() {
  console.log('Delaying...');
  await delay(3000);
  console.log('Delay complete.');
}

showDelayMessage();
```

PROMISE OVERVIEW

In JavaScript, a promise is an object representing the eventual completion or failure of an asynchronous operation. A promise can be in one of 3 states:

1. Pending: The initial state of a promise, representing that the asynchronous operation has not yet completed.
2. Fulfilled: The state of a promise representing that the asynchronous operation has completed successfully and the promise has a resolved value.
3. Rejected: The state of a promise representing that the asynchronous operation has failed and the promise has a rejected reason.

A promise can be created using the Promise constructor, which takes a function as its argument.

This function, known as the "executor function," is automatically executed and passed two arguments: **resolve and reject.**

resolve is called when the asynchronous operation is completed successfully and reject is called when the asynchronous operation has failed.

Here's an example of creating a promise that resolves after a specified amount of time

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("The operation was successful");
  }, 2000);
});
```

Promises can be consumed using the then and catch methods. The then method is called when the promise is fulfilled and takes a success callback as an argument. The catch method is called when the promise is rejected and takes a failure callback as an argument.

Here's an example of consuming the promise created earlier:

```
myPromise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error(error);
  });
```

cybercom creation

WHEN TO USE PROMISES?

Promises are commonly used in JavaScript to handle asynchronous operations, such as

API calls
file I/O operations
timers, among others.

They provide a way to handle the results of these operations in a clean and predictable manner, making it easier to write asynchronous code that is easy to read and maintain.

cybercom creation

Keep it Up!

Scan to Get Promises Example