edureka!
a Veranda Enterprise

Full Stack Web Development Program

edureka!
a Veranda Enterprise

Day 22 – Callbacks in JavaScript

Titles

- Overview of Asynchronous Approach or Programming
- Introduction to Synchronous Programming
- Asynchronous Programming in JavaScript
- What is Callbacks?
- Pyramid of Doom
- Pyramid of Doom – Example
- Solutions for Pyramid of Doom

Learning Objectives

By the end of this module, you will be able to:

- Comprehend Asynchronous Programming in JavaScript and the concept of Callbacks.
- Evaluate the issues and challenges related to the Pyramid of Doom.

Power of Asynchronous Programming in JavaScript

edureka!

Overview of Asynchronous Approach or Programming

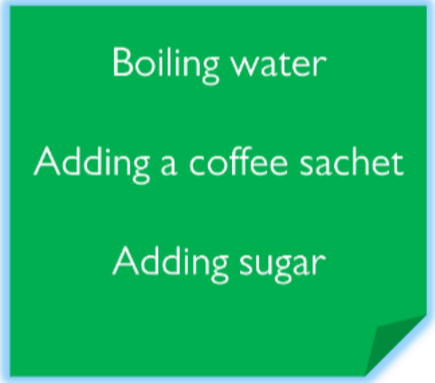
- To grasp the essence of **asynchronous programming**, envision a scenario where a group of chefs cooking together, each working on a different dish at the same time. This is like asynchronous programming, where different parts of a program run simultaneously, making things faster and more efficient.

- In traditional programming, tasks are done one after the other, like chefs waiting their turn to cook. But with asynchronous programming, everything happens at once - like all chefs cooking together.

- This way, a **computer program** can do multiple things at the same time, such as sending requests and processing user inputs, making it run smoother and quicker.

Introduction to Synchronous Programming

- **Synchronous programming** means that computers perform tasks sequentially, following the instructions in the exact order they were provided.
- Consider the process of preparing a coffee, with tasks such as boiling water, adding a coffee sachet, and adding sugar as per taste.



- Just like tackling these kitchen tasks one by one, waiting to complete each before starting the next, synchronous programming ensures that a computer finishes one job before proceeding to the next.
- This method simplifies the understanding and anticipation of the computer's actions at any moment.

Introduction to Synchronous Programming(contd.)

Example of synchronous code in JavaScript is mentioned below:

```
// Define three functions
function task1()
{
    console.log("Task 1");
}
function task2()
{
    console.log("Task 2");
}
function task3()
{
    console.log("Task 3");
}
// Execute the functions
task1();
task2();
task3();
```

Order of output of code will look like this:

```
``Task 1``
``Task 2``
``Task 3``
```

Introduction to Synchronous Programming(contd.)

- Synchronous programming may encounter issues, especially with operations that are time-consuming.

- Imagine a scenario where a synchronous application needs to wait for a reply from an external server. During this wait time, the application can't perform any other actions—it's essentially on hold until it receives the server response.

- This situation, known as **blocking**, can make the application seem unresponsive or **stuck/frozen** to the user.

Take a look at this piece of code:

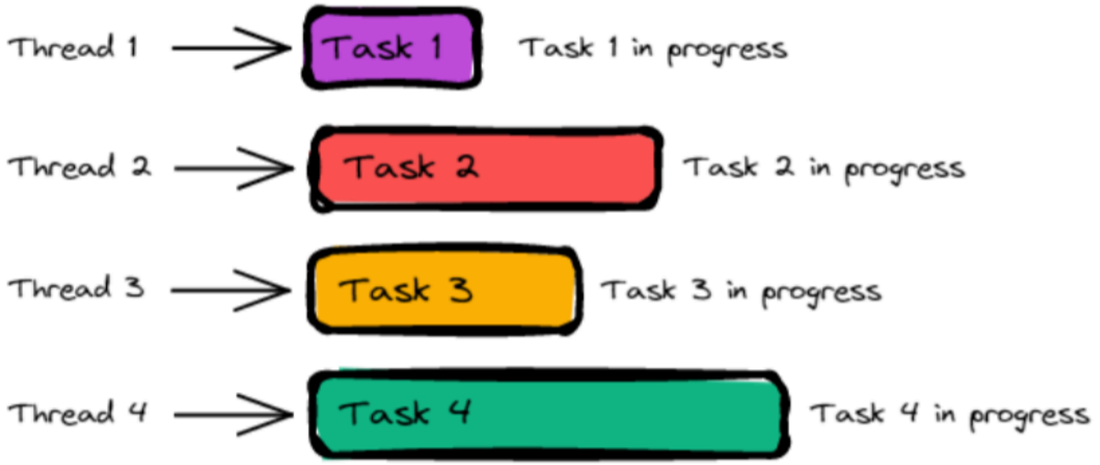
```
function someLongRunningFunction() {
    let start = Date.now();
    while (Date.now() - start < 5000) {
        // Looping without doing anything
    }
    return "Hello";
}

console.log('Starting...');

let result = someLongRunningFunction();
console.log(result);

console.log('...Finishing');
```

Introduction to Synchronous Programming(contd.)

- In this aforementioned code snippet:
  - The process kicks off with a **Starting...** message output to the console.
  - Next, it invokes **someLongRunningFunction**, mimicking a prolonged operation lasting 5 seconds, during which the program's subsequent execution is halted.
  - After finishing, the function returns **Hello**, which then gets printed to the console.
  - The sequence concludes with a **Finishing** message in the console.
- During the execution of **someLongRunningFunction()**, which lasts **5** seconds, the program is unable to proceed to the next line, rendering it unresponsive.
- This delay can significantly extend the program's overall runtime and negatively affect user experience by making the application appear frozen.
- By adopting asynchronous programming, applications can continue to run other tasks without pausing, thus remaining responsive while waiting for longer processes to finish.

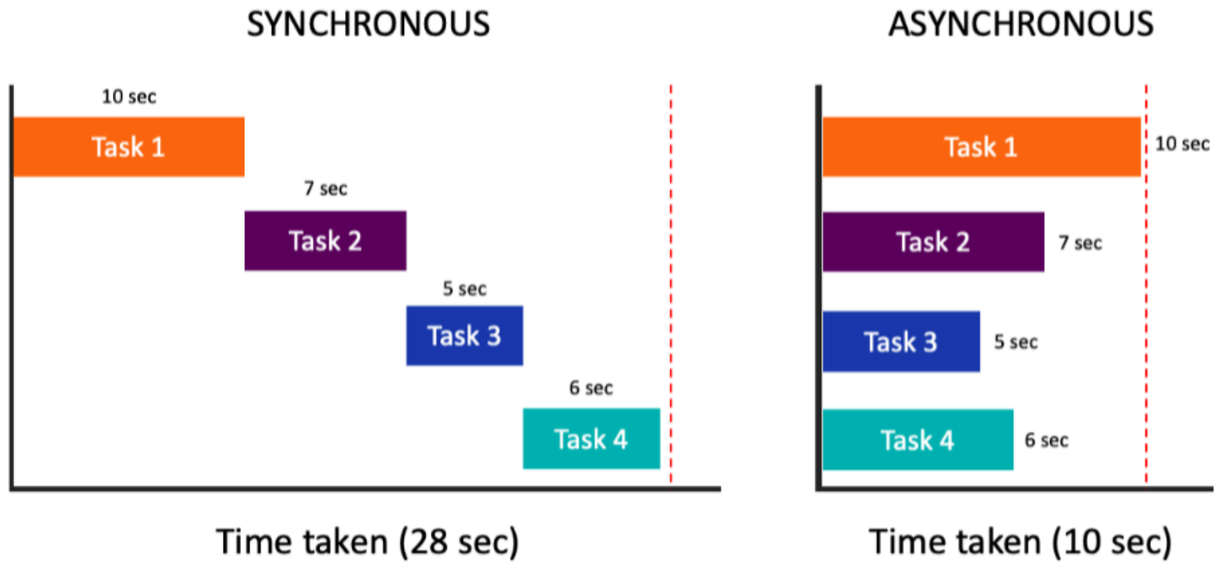
Asynchronous Programming in JavaScript

- **Asynchronous programming** is a paradigm that allows code execution to proceed without having to wait for potentially long-running operations to complete.
- This is particularly used in tasks such as fetching data from a server, reading files, or querying a database, which can take significant time to complete.



Asynchronous Programming in JavaScript (contd.)

- It helps in creating responsive, fast, and efficient applications by allowing these time-consuming operations to run in the background, thereby not blocking the main thread of execution.
- JavaScript implements asynchronous programming through several mechanisms, including
  - **Callbacks**
  - **Promises**
  - **Async/Await**



What is Callbacks?

- Imagine you're in a coffee shop, ordering your favorite drink. Instead of waiting at the counter, you sit and browse your phone. When your order's ready, the barista calls your name (*acting as the callback*) for you to collect your drink and move on. This way, you're not idly waiting but are free to engage in other activities.

- In programming, especially in JavaScript, a **callback** works on a similar principle. It's a way to say, *Hey, go ahead and do this task, and when you're done, let me know by calling this function (the callback)*. While waiting for the task to complete, your program isn't just sitting idle; it can continue executing other tasks.

What is Callbacks? (contd.)

- A callback is a function that is passed to another function as an argument.
- These functions are predominantly utilized in JavaScript for crafting asynchronous code.
- Although JavaScript typically executes code in a sequential, top-down manner, certain scenarios require the code to execute in a specific sequence.
- Callback functions facilitate this requirement, embodying the essence of asynchronous programming.
- Functions that accept other functions as arguments are termed **higher-order functions**.

```
do_sum(10, 20, function(result){
    document.write(result);
});

function do_sum(num1, num2, callback)
{
    var sum = 0;
    sum = num1 + num2;

    /*call callback function*/
    callback(sum)
}
```

Callback Function

What is Callbacks? (contd.)

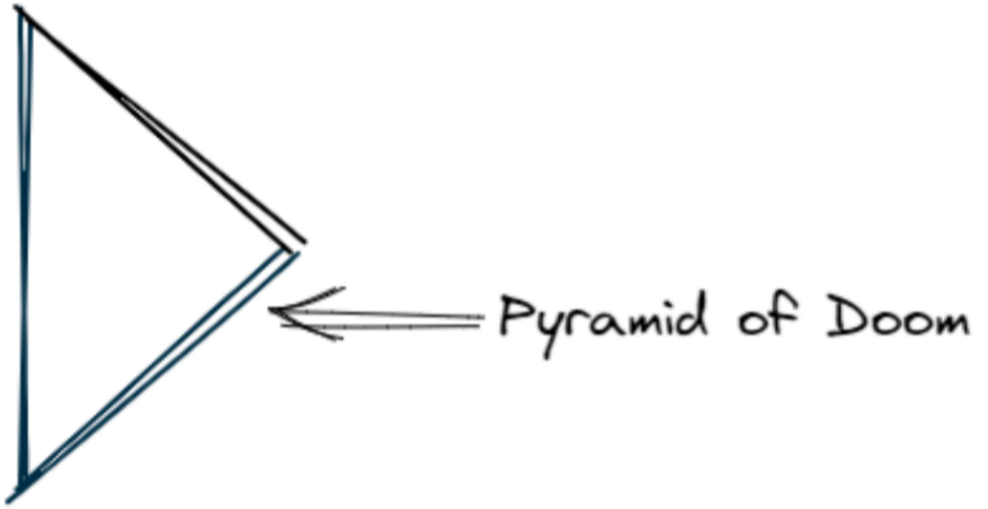
- In the simplest terms, a callback is a function that is passed as an argument to another function and is executed after some operation has been completed.
- Here's a very basic example:

```
function greet(name, callback) {
    console.log('Hi ' + name);
    callback();
}

greet('Alice', function() {
    console.log('Callback function is called.');
});
```

- In this example, the greet function takes a name and a callback function as arguments. It first logs a greeting message using the name, and then it calls the callback function, which logs another message.

Pyramid of Doom

- Callbacks offer a practical approach for managing asynchronous tasks.
- Yet, the readability and complexity of the code can deteriorate when numerous callbacks are nested within each other.
- This issue arises from linking several callbacks in succession, leading to a layered, pyramid-shaped indentation pattern known as **Callback Hell** or the **Pyramid of Doom**.



Pyramid of Doom - Example

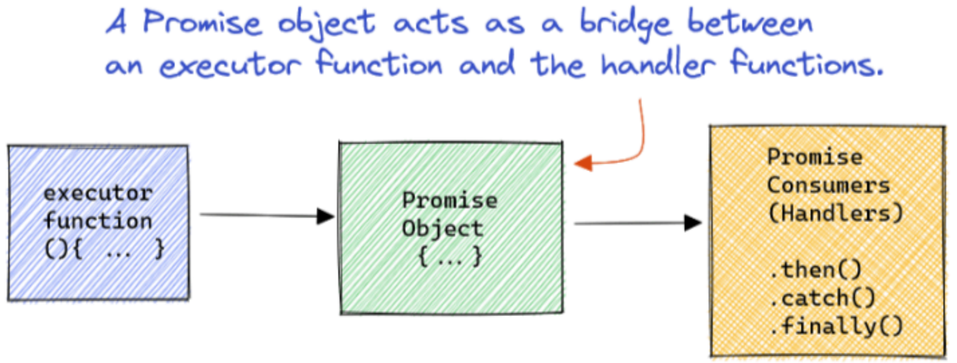
Example:

```
fetchData(function(a)
{
      fetchMoreData(a, function(b)
      {
            fetchEvenMoreData(b, function(c)
            {
                  fetchEvenEvenMoreData(c, function(d)
                  {
                        fetchFinalData(d, function(finalData)
                        {
                              console.log(finalData);
                        });
                  });
            });
      });
});
```

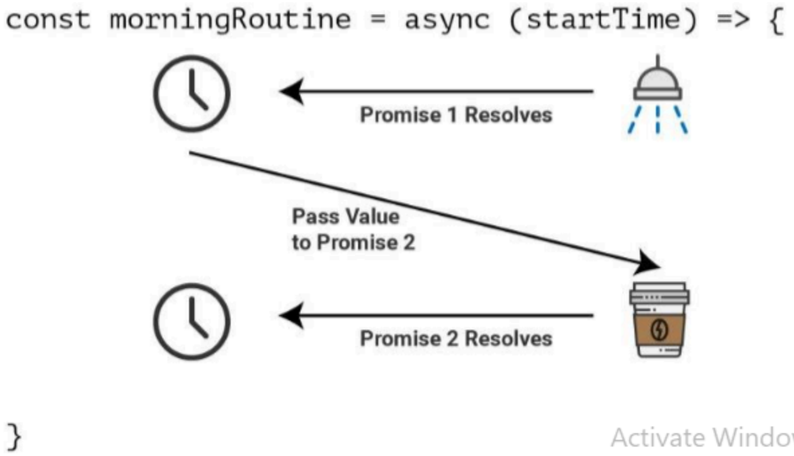
This structure, resembling a pyramid, is known as **Callback Hell**

Solutions for Pyramid of Doom

- To address the challenges posed by callback hell, modern JavaScript introduces **Promises** and **Async/Await**, which offer cleaner and more manageable ways to handle asynchronous operations.



- However, understanding callbacks is fundamental as they form the basis of these more advanced concepts.



- Callbacks are a foundational concept in JavaScript, enabling asynchronous programming by allowing functions to execute after certain operations complete, thereby improving the efficiency and responsiveness of applications.