**Asynchronous JavaScript**

Asynchronous JavaScript is a programming approach that enables the non-blocking execution of tasks, allowing concurrent operations, improved responsiveness, and efficient handling of time-consuming operations in web applications, JavaScript is a single-threaded and synchronous language. The code is executed in order one at a time, But Javascript may appear to be asynchronous in some situations.

There are several methods that can be used to perform asynchronous javascript tasks, which are

1. [Using callback](https://www.geeksforgeeks.org/asynchronous-javascript/#approach-1-using-callback)
2. [Using Promises](https://www.geeksforgeeks.org/asynchronous-javascript/#approach-2-using-promises)

JavaScript Callbacks

A callback is a function that is passed as an argument to another function and is executed after the completion of that main function. **In simple terms**, a callback function is called at the end of a task to either deliver results or perform an action. You pass this callback function to the main function, and once the main function completes, it invokes the callback to proceed with the **next steps**.

Use of callbacks

Callbacks are used for managing the outcomes of asynchronous tasks without blocking the program’s execution. Asynchronous tasks, like network requests or database queries, take time to finish. If these tasks were synchronous, the program would halt until they were done, resulting in a sluggish user experience.

With callbacks, though, you can keep the program running while these tasks happen in the background. When the task finishes, the callback function handles the result. This ensures the program stays responsive, enhancing the user experience.

**Important Points to Know About Callbacks**

**1. Asynchronous programming:**

Callbacks are used to handle the results of asynchronous operations, which means that the operation does not block the execution of the rest of the program. Instead, the program continues to run and the callback function is executed when the operation is complete.

**2. Non-blocking:**

Callbacks allow for non-blocking programming, which means that the program does not stop and wait for an operation to complete before continuing to execute. This is important for improving the performance and responsiveness of applications.

**3. Higher-order functions:**

A higher-order function is a function that takes one or more functions as arguments, or returns a function as a result. The main Function in the examples above is a higher-order function because it takes a callback function as an argument.

**4. Anonymous functions:**

Anonymous functions are functions that are not named and are often used as callbacks. The function passed to setTimeout in the first code example is an anonymous function.

**5. Closure:**

A closure is a function that has access to variables in its outer scope, even after the outer function has returned. This allows the callback function to access variables and information from the main function, even after the main function has completed its execution.

**Basic Callback Function**

function mainFunction(callback) {

console.log("Performing operation...");

// Use setTimeout to simulate an asynchronous operation

setTimeout(function() {

callback("Operation complete");

}, 1000);

}

// Define the callback function

function callbackFunction(result) {

console.log("Result: " + result);

}

// Call the main function with the callback function

mainFunction(callbackFunction);

**Callback with Array.forEach() Method**

let numbers = [1, 2, 3, 4, 5];

function mainFunction(callback) {

console.log("Performing operation...");

numbers.forEach(callback);

}

function callbackFunction(number) {

console.log("Result: " + number);

}

mainFunction(callbackFunction);

Function Sequence

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

Sequence Control

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  return sum;  
}  
  
let result = myCalculator(5, 5);  
myDisplayer(result);

**JavaScript Promise**

JavaScript promises might sound a bit complicated at first, but once you get a clear understanding of them, they make working with code that takes time to complete, like fetching data from a website or waiting for a timer, much easier to manage. Let’s break down what promises are and how you can use them.

A promise in JavaScript is like a **container** for a **future value**. It is a way of saying, “**I don’t have this value right now**, **but I will have it later**.” Imagine you order a book online. You don’t get the book right away, but the store promises to send it to you. While you wait, you can do other things, and when the book arrives, you can read it.

In the same way, **a promise** lets you keep working with your code while waiting for something else to finish, like **loading**data from a server. When the data is ready, the promise will deliver it.

A promise can be in one of three states:

* **Pending:**The promise is waiting for something to finish. For example, waiting for data to load from a website.
* **Fulfilled:**The promise has been completed successfully. The data you were waiting for is now available.
* **Rejected:** The promise has failed. Maybe there was a problem, like the server not responding.

When you create a promise, you write some code that will eventually tell the promise whether it was successful (fulfilled) or not (rejected).

let promise = new Promise(function(resolve, reject){  
 //do something  
});

let myPromise = new Promise(function(resolve, reject) {

// some code that takes time, like loading data

let success = true; // change this to false to check error

if (success) {

resolve("The data has loaded successfully!");

} else {

reject("There was an error loading the data.");

}

});

myPromise.then(function(message) {

// This runs if the promise is fulfilled

console.log(message);

}).catch(function(error) {

// This runs if the promise is rejected

console.log(error);

});

JavaScript Async

Introduced in ES8 (ES2017), async and await provide a more synchronous-looking way of writing asynchronous code. await makes JavaScript wait until a Promise is resolved or rejected.

* **async function**: Declares a function as asynchronous, which means it implicitly returns a Promise.
* **await**: Pauses the execution of the function until the Promise is resolved or rejected.

async function fetchData() {

try {

let data = await new Promise((resolve) => {

setTimeout(() => {

resolve('Data fetched');

}, 2000);

});

console.log(data); // Data fetched

} catch (error) {

console.error(error); // Error if Promise is rejected

}

}

fetchData();

**async functions**: They return a promise implicitly. So if you return a value from an async function, it's automatically wrapped in a promise.

async function sayHello() {

return 'Hello';

}

sayHello().then(console.log); // Outputs: 'Hello'

**await** can only be used inside async functions. It pauses the execution until the promise resolves, without blocking the event loop.

Using async/await helps avoid **callback hell**, which was common with nested callbacks.

**Handling Multiple Promises**

* **Promise.all()**: Used to run multiple promises in parallel. It waits until all promises are resolved (or one rejects).

let p1 = new Promise((resolve) => setTimeout(resolve, 1000, 'First'));

let p2 = new Promise((resolve) => setTimeout(resolve, 2000, 'Second'));

let p3 = new Promise((resolve) => setTimeout(resolve, 1500, 'Third'));

Promise.all([p1, p2, p3])

.then((results) => {

console.log(results); // ['First', 'Second', 'Third']

})

.catch((error) => {

console.error(error); // If any promise fails

});

**Promise.race()**: Resolves or rejects as soon as one of the promises settles (first one to complete wins the race).

Promise.race([p1, p2, p3])

.then((result) => {

console.log(result); // Logs whichever promise settles first

});

**The Event Loop**

JavaScript uses an **event loop** to handle asynchronous tasks. When an asynchronous function (like a setTimeout or network request) is executed, it gets offloaded to the browser or Node.js. Once completed, the callback is placed in the **callback queue**. The event loop will execute these callbacks only when the **call stack** is empty.

* **Call Stack**: A data structure where JavaScript keeps track of function calls. It operates in a Last-In-First-Out (LIFO) manner.
* **Callback Queue**: Functions waiting to be executed once the stack is empty.
* **Event Loop**: It constantly checks the call stack and the callback queue, pushing callbacks from the queue to the stack when it's empty.

console.log('Start');

setTimeout(() => {

console.log('Timeout');

}, 0);

console.log('End');

Even though the timeout is set to 0ms, "End" will be logged before "Timeout" because the event loop places the setTimeout callback in the queue, which runs only after the current call stack is empty.