

WEB WEB WEB
DEVELOPMENT DEVELOP

JavaScript

**Asynchronous
JavaScript**

Gagan Saini

Asynchronous

Not all tasks can be executed sequentially. That's where **asynchronous** programming comes into play. It allows multiple tasks to **run concurrently** without blocking the execution of other code. In simpler terms, it enables your web applications to handle tasks in the background while the main thread remains responsive.



Requirement

If we always use synchronous programming then some functions like fetching data from external sources, accessing user camera or mic might take some time to get data.

And then we have to wait till the get back to perform other functions. Which makes our website unresponsive.

That's where we need asynchronous functionality in our website.



Method

By default there are some functions which is already asynchronous like

**fetch(), setTimeout(), setInterval(),
addEventListener()** etc.

How to make async functions.

- Callbacks
- Promises
- Async/Await



WEB WEB WEB
DEVELOPMENT DEVELOP

JavaScript

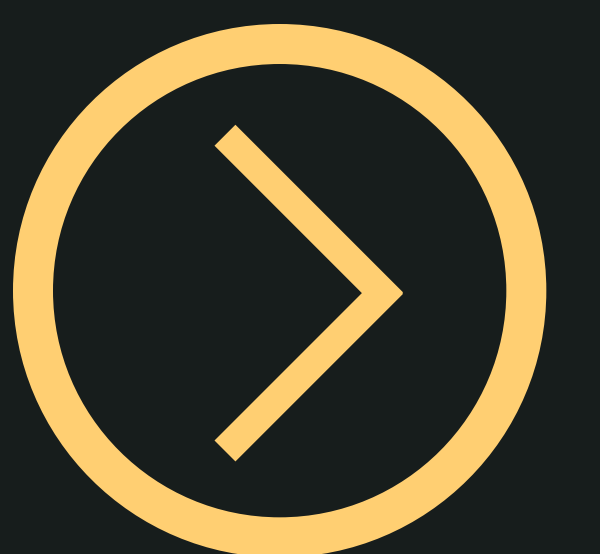
Callback functions

Gagan Saini

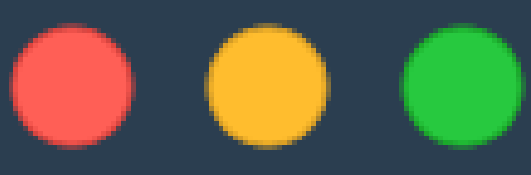
Callbacks

Callbacks are functions that you pass to other functions as an argument. These functions promise to execute when a certain task completes.

A callback is a function passed as an argument to another function.



Example



```
function fetchUserData(userId, callback) {  
  setTimeout(() => {  
    const userData = { id: userId, name: 'Gagan Saini' };  
    callback(userData);  
  }, 3000);  
}  
  
function displayUser(user) {  
  console.log(`User ID: ${user.id}`);  
  console.log(`Name: ${user.name}`);  
}  
  
fetchUserData(123, displayUser);
```

User ID: 123 and Name: Gagan Saini is logged after 3 seconds.



Example Explanation

The **fetchUserData()** function simulates an asynchronous operation using **setTimeout()** to introduce a delay.

The **displayUser()** function is a callback that takes the fetched user data and displays it.

When **fetchUserData()** completes its asynchronous task (after **3 seconds** in this case), it invokes the provided callback **displayUser()** function with the fetched user data.



Pros

Asynchronous Control: They allow you to specify what should happen after a task is completed, making it easier to handle non-blocking operations.

Modularity: Functions can be written independently and then combined using callbacks to achieve specific functionality,

Event Handling: Callbacks are commonly used to handle events, like button clicks, form submissions, or any interaction-driven tasks in web applications. This allows you to define behavior based on user actions.



Pros

Error Handling: Callbacks enable better error handling in asynchronous operations. You can pass error parameters to callbacks to handle different error scenarios gracefully.

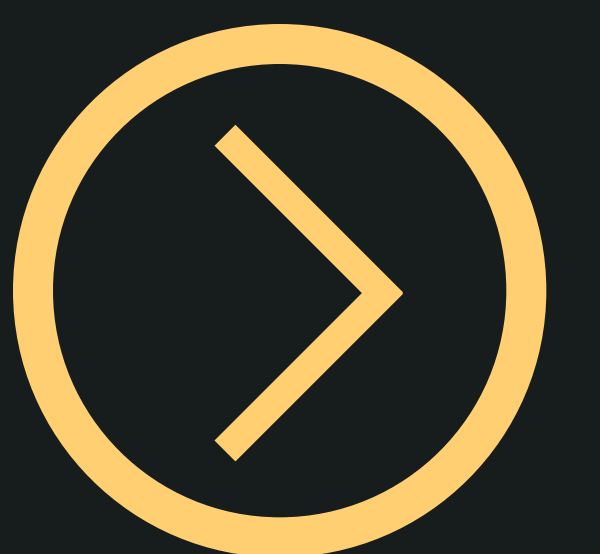
Simple Structure: Callbacks can be simple to implement, especially in scenarios where you only need to handle one or two asynchronous tasks.



Cons

Callback Hell: Deeply nested callbacks can lead to unreadable and hard-to-maintain code. This can happen when you have multiple levels of nested callbacks, making the code difficult to follow.

Inversion of Control: Where the main program's flow is determined by the callbacks rather than the linear sequence of code. This can sometimes make code harder to understand reason about.



Cons

Loss of Context: The context (**this** value) can become unpredictable in callbacks. The value of this inside a callback might not be what you expect, especially when dealing with object methods.

Limited Parallelism: Traditional callbacks can sometimes make it difficult to handle parallel execution of multiple asynchronous tasks. The need for multiple nested callbacks can complicate the logic.



Cons

Difficulty in Error Handling: In complex scenarios, error handling with callbacks can become convoluted. Managing errors across multiple levels of callbacks can be challenging and error-prone.

Readability: If not used carefully, callbacks can reduce code readability, especially when the callbacks themselves are long or contain complex logic.



WEB WEB WEB

DEVELOPMENT DEVELOP

JavaScript

Asynchronous JS Promises

Gagan Saini

Promise

The Promise object represents the eventual completion or failure of an asynchronous operation and its resulting value.

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

This lets async methods return values like synchronous methods. Instead of immediately returning the final value, the async method returns a promise to supply the value at some point in the future.



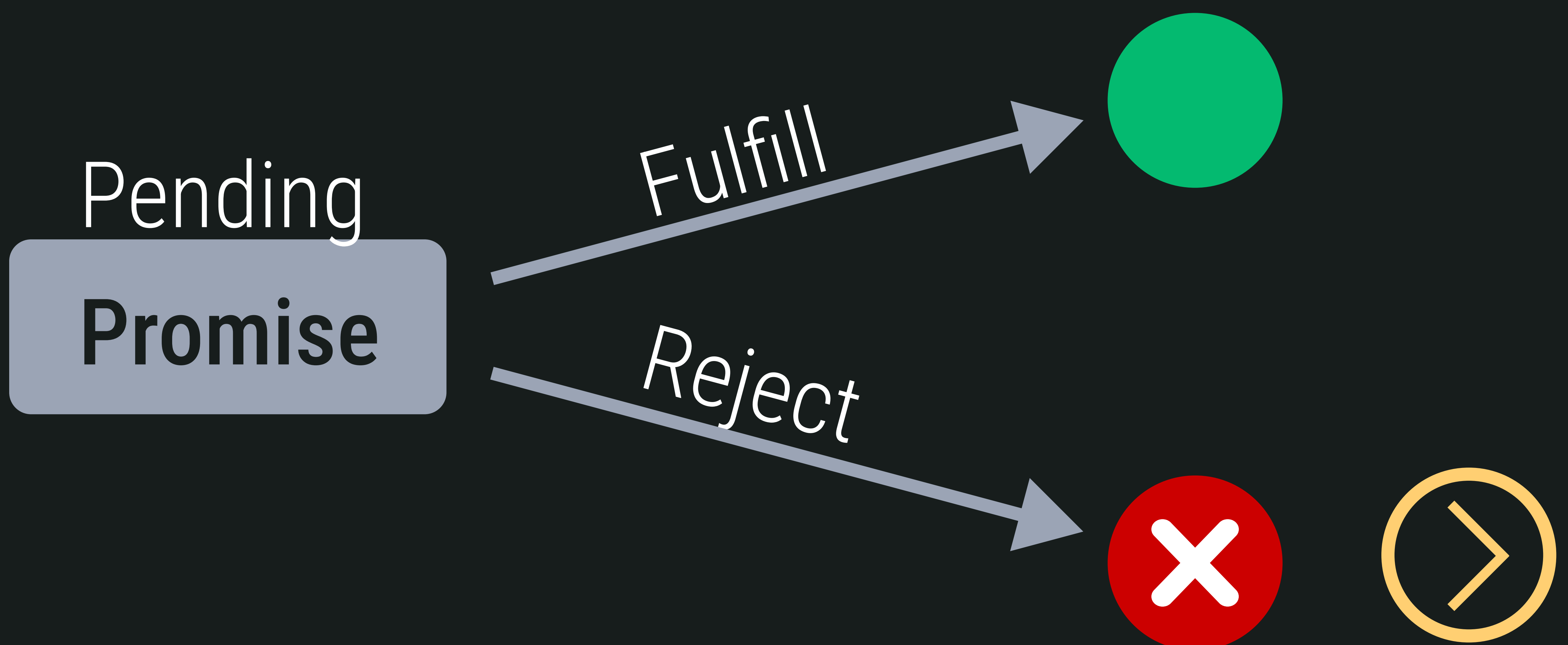
States of Promise

A promise has 3 different states.

Pending: Initial state, neither fulfilled nor rejected.

Fulfilled: Meaning that the operation was completed successfully.

Rejected: Meaning that the operation failed.

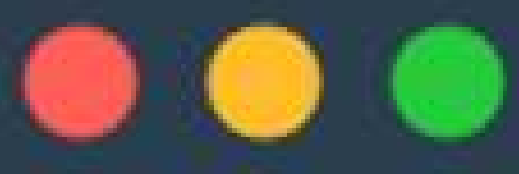


Why Use Promises ?

Think of **Promises** as trusty agreements in your code. When dealing with time consuming tasks like retrieving data from a server, **Promises** free your code from waiting around. They empower your program to keep running while the **asynchronous** task is in progress. Once the task is done, Promises handle **success or error** scenarios gracefully.



Example



```
const deliveryPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const deliveryArrived = Math.random() < 0.5;
    if (deliveryArrived) {
      resolve("Your package has arrived!");
    } else {
      reject("Sorry, there was an issue with your delivery.");
    }
  }, 1000);
});

deliveryPromise
  .then((successMessage) => console.log(successMessage))
  .catch((errorMessage) => console.error(errorMessage));
```

After 1 second. 50% chance of resolving or error. And print message accordingly.



Promise Constructor

Promise(): Creates a new Promise object. The constructor is primarily used to wrap functions that do not already support promises.

new Promise(callback function)

And this callback function has two parameters **resolve & reject** which can be called resolve when promise fulfilled and reject when promise have any error.



Promise Methods

Promise.all(): Promise fulfilled if all of passed promises are resolved and if any of them rejected then this also rejected.

Promise.any(): Promise fulfilled if any of passed promises are resolved and if all of them rejected then this also rejected.

Promise.race(): This returned promise settles with the eventual state of the first promise that settles.



Promise Methods

Promise.resolve(): Returns a Promise object that is resolved with the given value.
If you don't know if a value is a promise or not, **Promise.resolve(value)** it instead and work with the return value as a promise

Promise.reject(): Returns a new Promise object that is rejected with the given reason.



Promise Chaining

Promise chaining is a technique that allows us to execute **multiple asynchronous** operations in a sequence. Instead of nesting callbacks within callbacks (**callback hell**), you can chain **.then()** methods to a **Promise**. This creates a linear and readable flow of async tasks.

Example

```
fetchUserData()  
  .then(something_1)  
  .then(something_2)  
  .catch(error)
```



Pros

Error Handling: Promises centralize error handling using `.catch()`, making it simpler to manage errors across asynchronous tasks.

Chaining: Promises can be chained using `.then()` for sequential operations, making complex async flows easier to manage.

Readability: Promises provide a clear structure that's easy to follow, avoiding callback hell and improving code readability.



Pros

Parallel Execution: Multiple Promises can be executed in parallel, enhancing performance by not blocking the main thread.

State Management: Promises have clear states (pending, fulfilled, rejected), making it easy to track the status of asynchronous tasks.



Cons

Still Complex: Although Promises improve over callbacks, they can still get complex when dealing with intricate async flows.

Promise Hell: Chaining multiple `.then()` calls can lead to "promise hell," a variation of callback hell when not properly managed.

Learning Curve: Understanding Promises might require a learning curve, especially for developers new to asynchronous programming.



WEB WEB WEB
DEVELOPMENT DEVELOP

JavaScript

Asynchronous JS

Async & Await

Gagan Saini

Async & Await

They simplify the syntax for handling promises and make your asynchronous code more readable and elegant.

Async: The keyword is used before a function to indicate that it contains asynchronous operations.

Await: This is used inside an async function to pause execution until the promise is resolved, making code appear sequential.



Example



```
async function fetchUserData() {  
  try {  
    const response = await fetch(someAPIURL);  
    const user = await response.json();  
    console.log('User Data:', user);  
  } catch (error) {  
    console.error('Error:', error);  
  }  
}  
  
fetchUserData();
```

try and **catch** blocks are used to handle errors gracefully in **async** functions, just like with regular try-catch blocks.



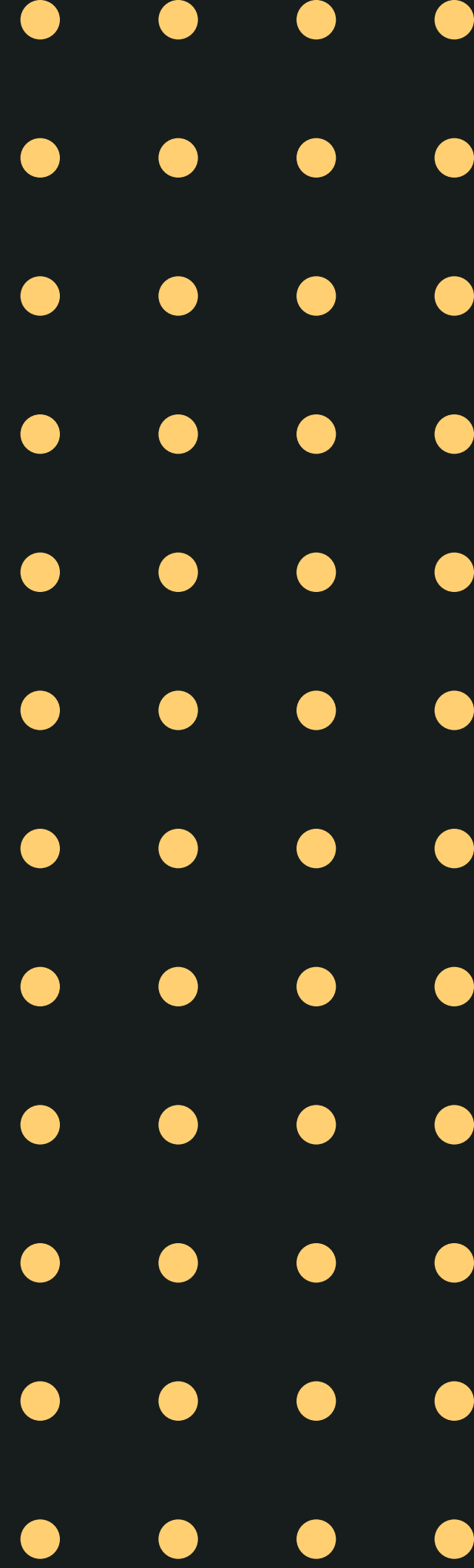
Benefits of Using.

Readability: `async` and `await` simplify asynchronous code, making it resemble synchronous code, which is easier to understand.

Error Handling: Error handling becomes more straightforward and intuitive using `try` and `catch`.

Sequential Logic: They allow you to write asynchronous code in a more sequential manner, enhancing the logical flow.





Repost this to help others!



@Gagan Saini

WEB DEVELOPER