Callback vs Promise:

**1️⃣ Callbacks**

**🔹 What are Callbacks?**

A **callback** is a function passed as an argument to another function. It is executed after the operation is completed.

const fs = require("fs");

fs.readFile("hello.txt", "utf8", (err, data) => {

if (err) {

console.log("Error reading file:", err);

return;

}

console.log("File content:", data);

});

**Problems with Callbacks**

1. **Callback Hell (Pyramid of Doom)** – When multiple nested callbacks are needed, code becomes **hard to read**.
2. **Error Handling is Tricky** – Errors must be handled manually in each callback.
3. **Inversion of Control** – The flow of execution is **controlled by the callback**, not the function itself.

**Promises**

**🔹 What are Promises?**

A **Promise** represents the **eventual completion** (or failure) of an asynchronous operation.  
Instead of using callbacks, it allows chaining .then() and handling errors with .catch().

**✅ Example: Using fs.promises.readFile()**

const fs = require("fs").promises;

fs.readFile("hello.txt", "utf8")

.then((data) => console.log("File content:", data))

.catch((err) => console.log("Error reading file:", err));

Example: Using async/await

const readFileAsync = async () => {

try {

const data = await fs.readFile("hello.txt", "utf8");

console.log("File content:", data);

} catch (err) {

console.log("Error reading file:", err);

}

};

readFileAsync();

**Comparison Table: Callbacks vs Promises**

| **Feature** | **Callbacks** | **Promises** |
| --- | --- | --- |
| **Syntax** | Nested functions | Chained .then(), catch() |
| **Readability** | Hard to read (callback hell) | More readable |
| **Error Handling** | Handled inside each callback | .catch() for all errors |
| **Chaining** | Difficult | Easy using .then() |
| **Inversion of Control** | Yes (callback controls execution) | No (code controls execution) |
| **Debugging** | Harder to trace | Easier, stack trace available |

**Callback Hell vs Promise Chaining**

**🚨 Callback Hell Example**

fs.readFile("file1.txt", "utf8", (err, data1) => {

if (err) return console.error(err);

fs.readFile("file2.txt", "utf8", (err, data2) => {

if (err) return console.error(err);

fs.readFile("file3.txt", "utf8", (err, data3) => {

if (err) return console.error(err);

console.log("All files read!");

});

});

});

Promise Chaining Solution

fs.readFile("file1.txt", "utf8")

.then((data1) => fs.readFile("file2.txt", "utf8"))

.then((data2) => fs.readFile("file3.txt", "utf8"))

.then((data3) => console.log("All files read!"))

.catch((err) => console.error(err));

**When to Use What?**

| **Use Case** | **Callbacks** | **Promises** |
| --- | --- | --- |
| Simple async tasks | ✅ Yes | ✅ Yes |
| Multiple async operations (chained) | ❌ No | ✅ Yes |
| Error handling | ❌ No | ✅ Yes |
| Readability & maintainability | ❌ No | ✅ Yes |

**Event Emitters and Promises** are both used for handling asynchronous operations in Node.js, but they work in **different ways** and serve different use cases. Let’s compare them and understand their relationship.

An **EventEmitter** allows an object to emit named events that other parts of the application can **listen for** and execute callbacks when the event occurs.

const EventEmitter = require('events');

const myEmitter = new EventEmitter();

// Register an event listener

myEmitter.on('dataReceived', (data) => {

console.log(`Received data: ${data}`);

});

// Emit the event

setTimeout(() => {

myEmitter.emit('dataReceived', "Hello from EventEmitter");

}, 2000);

**Key Features of Event Emitters:**

* Can emit multiple events over time.
* Supports **multiple listeners** for the same event.
* Works well for **continuous event-driven programming** (e.g., WebSockets, streams).

**Promises**

A **Promise** represents a value that may be **available now, in the future, or never**. It is a one-time event—either it resolves successfully, or it fails.

**Key Features of Promises:**

* Handles **one-time** async operations.
* Guarantees only **one resolution** (either .then() or .catch()).
* Can be easily chained for **sequential execution**.

**Differences Between Event Emitters and Promises**

| **Feature** | **Event Emitters** | **Promises** |
| --- | --- | --- |
| **Type of Operation** | Multiple asynchronous events over time | A **single** asynchronous operation |
| **Number of Executions** | Can emit multiple times | Resolves **only once** |
| **Listeners** | Multiple listeners (.on()) | Single .then() for resolved value |
| **Error Handling** | Needs error event handling | .catch() handles errors |
| **Best Use Case** | Handling repeated events (e.g., WebSockets, Streams) | Handling a single async operation (e.g., API request) |

**Conclusion**

* **Use Event Emitters** when you need to handle **multiple events** over time.
* **Use Promises** when handling a **single async operation**.
* **They can work together**, where **a Promise can listen for an event** before resolving.

💡 **Rule of Thumb:**

* **"If it happens once, use a Promise. If it happens multiple times, use an EventEmitter."** 🚀

First off, you pretty much never want to write code that is a mix of callbacks and promises for async operations. If you're moving to promises or introducing some promises, then you probably want to refactor the callbacks in that same section of code into promises. For the appropriate types of operations, there are so many advantages of promises over plain callbacks that it is well worth the effort to convert when already working in an area of code.

**Promises are great for:**

* Monitoring synchronous operations
* That need to notify only once (usually completion or error)
* Coordinating or managing multiple asynchronous operations such as sequencing or branching async operations or managing multiple operations in flight at the same time
* Propagating errors from nested or deeply nested async operations
* Getting code ready for the use of async/await (or using it now with a transpiler)
* Operations that fit the Promise model where there are only three states: pending, fulfilled and rejected and where the state transitions from pending => fulfilled or from pending => rejected can then not change (a single one-way transition).
* Dynamically linking or chaining asynchronous operations (such as do these two async operations, examine the result, then decide which other async operations to do based on the intermediate result)
* Managing a mix of asynchronous and synchronous operations
* Automatically catching and propagating upwards any exceptions that occur in async completion callbacks (in plain callbacks these exceptions are sometimes silently hidden).

**Plain callbacks are good for things that promises cannot do:**

* Synchronous notifications (such as the callback for Array.prototype.map())
* Notifications that may occur more than once (and thus need to call the callback more than once). Promises are one-shot devices and cannot be used for repeat notifications.
* Situations that cannot be mapped into the pending, fulfilled, rejected one-way state model.

And, I'd also add EventEmitter to the mix.

**EventEmitters are great for:**

* Publish/subscribe type notifications
* An interface with an event model, particular when events can occur more than once (like streams)
* Loose couplings when 3rd party code wants to participate or monitor something without any more of an API than an eventEmitter. No API to design. Just make an eventEmitter public and define some events and the data that goes with them.

**Notes about converting plain callback code to Promises**

If your callbacks fit the node calling convention with the callback passed as the last argument and called like this callback(err, result), then you somewhat automatically wrap the parent function in a promise with util.promisify() in node.js or if using the [Bluebird promise library](http://bluebirdjs.com/docs/api-reference.html), with [Promise.promisify()](http://bluebirdjs.com/docs/api/promise.promisify.html).

With Bluebird, you can even promisify an entire module (that uses async callbacks in the node.js calling convention) at once such as:

const Promise = require('bluebird');

const fs = Promise.promisifyAll(require('fs'));

fs.writeFileAsync("file.txt", data).then(() => {

// done here

}).catch(err => {

// error here

});

**In node.js version 8+**

There is now util.promisify() which will convert an async function that uses the node.js async calling convention to a function that returns a promise.

Example from [the doc:](https://nodejs.org/api/util.html#util_util_promisify_original)

const util = require('util');

const fs = require('fs');

const stat = util.promisify(fs.stat);

// usage of promisified function

stat('.').then((stats) => {

// Do something with `stats`

}).catch((error) => {

// Handle the error.

});