

JavaScript

# Understanding JavaScript Promises

Handling Asynchronous Operations Like a Pro  
Part 1/3

May 2, 2025



Source Code

# 1. What's the Deal with Promises?

Asynchronous operations are fundamental in modern JavaScript, enabling tasks like fetching data, reading files, or handling user interactions without blocking the main thread. However, managing the flow and potential errors of these operations can be complex. This is where **Promises** provide a robust solution.

## 1.1. Why Promises?

Consider an operation that takes time to complete, such as requesting data from an API. A synchronous approach would halt script execution until the data arrives, leading to unresponsive applications. Promises offer a way to handle such asynchronous tasks effectively.

They act as placeholders for a future value, representing the eventual result of an asynchronous operation. Instead of blocking, the operation is initiated, and the promise object is returned immediately. This allows the rest of the script to continue running.

Promises improve upon older asynchronous patterns like callbacks by providing:


- A clearer, more manageable structure for handling results or errors.
- Better composability for chaining multiple asynchronous operations.
- A standardized way to manage asynchronous flow, reducing complexity (often referred to as "callback hell").

A Promise exists in one of three states:



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

- **Pending:** The initial state; the asynchronous operation has not yet completed.
- **Fulfilled (Resolved):** The operation completed successfully, and the promise holds the resulting value.
- **Rejected:** The operation failed, and the promise holds the reason (typically an error object).

Once a promise transitions from pending to either fulfilled or rejected, it becomes **settled**, and its state and value (or reason) become immutable.

## 2. Creating Your First Promise

Let's make a simple promise. We'll simulate an async task using `setTimeout`.

### How to Run:


- Save the code below as `01_creating_promise.js`.
- Open your terminal and run: `node 01_creating_promise.js`
- Or, paste the code directly into your browser's developer console.

```
1 // --- 01_creating_promise.js ---
2 console.log("Creating a promise...");
3
4 // A promise takes a function (the 'executor') with two
```



**Alejandro Sánchez Yalí**


Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
arguments: resolve and reject.
5 const myFirstPromise = new Promise((resolve, reject) => {
6   console.log("Executor function started (simulating async
   work)...");
7   const success = Math.random() > 0.5; // Simulate success
   or failure randomly
8
9   // Simulate an asynchronous operation (like fetching
   data) using setTimeout
10  setTimeout(() => {
11    if (success) {
12      const data = { message: "Yay! Data fetched
   successfully!" };
13      console.log("Async work finished: Resolving the
   promise.");
14      resolve(data); // If successful, call resolve with
   the result
15    } else {
16      const error = new Error("Oops! Something went
   wrong.");
17      console.log("Async work finished: Rejecting the
   promise.");
```

**Alejandro Sánchez Yalí**

Software Developer | AI &amp; Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
18     reject(error); // If failed, call reject with an
    error
19   }
20 }, 2000); // Simulate a 2-second delay
21 });
22
23 console.log("Promise created. It's now 'pending'.");
24
25 // The mechanism for handling the promise's settlement
    (resolution or rejection) is explored next.
```

When you run this:

- The `new Promise(...)` creates the promise.
- The executor function `(resolve, reject) => {:.}` runs immediately.
- `setTimeout` schedules the success/failure logic to run after 2 seconds.
- The code continues, logging "Promise created...". The promise is **pending**.
- After 2 seconds, `setTimeout`'s callback runs, calling either `resolve` (making the promise fulfilled) or `reject` (making it rejected).

But how do we actually \*use\* the result or handle the error?



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast



[www.asanchezyali.com](http://www.asanchezyali.com)

### 3. Handling Promises: `.then()`, `.catch()`, `.finally()`

Okay, we made a promise. Now, how do we react when it settles (fulfills or rejects)? We use special methods attached to the promise:

- `.then(onFulfilled, onRejected)`: Attaches callbacks for when the promise is fulfilled (first argument) or rejected (second argument - less common, usually use `.catch()`).
- `.catch(onRejected)`: Attaches a callback specifically for when the promise is rejected. It's like a `try...catch` block for promises.
- `.finally(onFinally)`: Attaches a callback that runs *always*, whether the promise was fulfilled or rejected. Great for cleanup tasks (like hiding a loading spinner).

Let's handle the promise we created earlier:

#### How to Run:

- Save the code below as `02_then_catch_finally.js`.
- Open your terminal and run: `node 02_then_catch_finally.js`
- Or, paste the code directly into your browser's developer console.




```
1 // --- 02_then_catch_finally.js ---  
2 console.log("Creating a promise...");
```



**Alejandro Sánchez Yalí**


Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

```
3
4 const myDataPromise = new Promise((resolve, reject) => {
5   console.log("Executor: Simulating fetching data...");
6   const success = Math.random() > 0.3; // Higher chance of
      success
7
8   setTimeout(() => {
9     if (success) {
10       const userData = { id: 123, name: "Alex", email:
          "alex@example.com" };
11       console.log("Executor: Data fetched! Resolving...");
12       resolve(userData);
13     } else {
14       const error = new Error("Network Error: Could not
          fetch user data.");
15       console.log("Executor: Failed to fetch data.
          Rejecting...");
16       reject(error);
17     }
18   }, 1500); // Simulate 1.5 seconds delay
19 });
20
```

**Alejandro Sánchez Yalí**

Software Developer | AI &amp; Blockchain Enthusiast


 [www.asanchezyali.com](http://www.asanchezyali.com)



```
21 console.log("Promise created. Waiting for it to
    settle...");
22
23 // --- Handling the Promise ---
24
25 myDataPromise
26   .then((data) => {
27     // Runs ONLY if resolved
28     console.log("\n.then() block executed:");
29     console.log("Received data:", data);
30     console.log('Welcome, ${data.name}!');
31     return data.id; // Can pass data to the next .then()
32   })
33   .catch((error) => {
34     // Runs ONLY if rejected
35     console.error("\n.catch() block executed:");
36     console.error("An error occurred:", error.message);
37   })
38   .finally(() => {
39     // Runs ALWAYS
40     console.log("\n.finally() block executed:");
41     console.log("Promise settled. Cleanup time!");
```

**Alejandro Sánchez Yalí**

Software Developer | AI &amp; Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)



```
42   });  
43  
44   console.log("\nPromise handler attached. Code  
    continues...");
```

Key takeaways:

- Handlers (`.then`, `.catch`, `.finally`) are attached to the promise object to react to its settlement.
- These handlers execute asynchronously when the promise settles, not immediately upon attachment.
- `.then()` receives the resolved value as its argument.
- `.catch()` receives the rejection reason (usually an Error object) as its argument.
- `.finally()` receives no arguments and executes regardless of the outcome.

This forms the foundation of promise-based asynchronous programming. Their true power becomes more apparent when chaining multiple operations, which will be covered subsequently.

## 4. Conclusions about JavaScript Promises

In this first part of our exploration of Promises in JavaScript, we have learned:



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast



[www.asanchezyali.com](http://www.asanchezyali.com)

- **Fundamentals:** Promises are objects that represent the eventual result of an asynchronous operation, allowing for cleaner and more structured code than traditional callbacks.
- **States:** A promise can be in one of three states: pending, fulfilled, or rejected. Once a promise is settled (fulfilled or rejected), its state and value become immutable.
- **Result Handling:** The `.then()`, `.catch()` and `.finally()` methods provide a clear interface for handling both successful results and errors from asynchronous operations.
- **Organized Asynchrony:** Promises keep code readable even when working with operations that don't complete immediately, making it easier to manage asynchronous execution flow.

Promises are fundamental to modern JavaScript development and form the foundation for more advanced features like `async/await`. In part two, we'll explore promise chaining, methods like `Promise.all()` and `Promise.race()`, and advanced patterns that unlock the full potential of asynchronous programming in JavaScript.


## 5. References

- MDN Web Docs. (2025). *Using promises*. [Link](#)
- MDN Web Docs. (2025). *Promise*. [Link](#)



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)

- ECMA International. (2026). *ECMAScript 2026 Language Specification: Promise Objects*. [Link](#)
- javascript.info. (2025). *Promise*. [Link](#)
- Parker, D. (2015). *JavaScript with Promises: Managing Asynchronous Code*. O'Reilly Media.
- This article was translated, edited and written in collaboration with AI. If you find any inconsistencies or have suggestions for improvement, please don't hesitate to open an issue in our GitHub repository at [github](#) or reach out directly.

## 6. Explore My Other Posts

### Enjoyed This Content?

Don't miss my previous post about:


#### **Node.js Streams: Part 2: Types & Advanced Operations**

Learn the fundamentals of Node.js Streams and discover how they can dramatically reduce memory usage when processing large files.



**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 [www.asanchezyali.com](http://www.asanchezyali.com)



Feedback

# Found this helpful?

Save, comment and share

May 2, 2025