

Promises

What is a Promise in JavaScript?

A **Promise** is an object that represents the eventual **completion or failure** of an asynchronous operation.

```
const promise = new Promise((resolve, reject) => {  
  // async operation here  
});
```

javascript

States of promises

A promise has **three states**:

1. **Pending** – Initial state, neither fulfilled nor rejected.
2. **Fulfilled** – Operation completed successfully.
3. **Rejected** – Operation failed.

Creating a simple promise

```
const myPromise = new Promise((resolve, reject) => {  
  const success = true;  
  
  if (success) {  
    resolve("✅ Operation was successful!");  
  } else {  
    reject("❌ Operation failed.");  
  }  
});
```

javascript

```
myPromise  
  .then((message) => {  
    console.log("Then:", message);  
  })  
  .catch((error) => {  
    console.log("Catch:", error);  
  });
```



Practical Analogy

Think of a Promise like ordering food:

- You **place an order** (start async task),
- You get a **token** (the Promise object),
- Later, your order is either **delivered** (fulfilled) or **canceled** (rejected),
- You handle both with **.then** and **.catch**

What is Promise.all() ?

Promise.all() takes an array of promises and **returns a single promise** that:

-  **Resolves** when **all** the input promises resolve
-  **Rejects** immediately if **any one** promise rejects

```
Promise.all([promise1, promise2, promise3])  
  .then(results => {  
    console.log(results); // An array of resolved values  
  })  
  .catch(error => {  
    console.error(error); // First rejection reason  
  });
```

javascript

Example

```
const p1 = Promise.resolve("First");  
const p2 = Promise.resolve("Second");  
const p3 = Promise.resolve("Third");  
  
Promise.all([p1, p2, p3])  
  .then(values => {  
    console.log(values); // ["First", "Second", "Third"]  
  });
```

javascript

Example with rejection

```
const p1 = Promise.resolve("Success");  
const p2 = Promise.reject("Failed");  
const p3 = Promise.resolve("Also Success");  
  
Promise.all([p1, p2, p3])  
  .then(values => {  
    console.log(values);  
  })  
  .catch(error => {
```

javascript

```
console.error("Caught error:", error); // "Caught error: Failed"
});
```

What is Promise.race()

Promise.race() returns a promise that:

- **Resolves or rejects as soon as the *first* promise** in the iterable resolves or rejects.
- The outcome (fulfilled/rejected) is the same as that of the first completed promise.

```
Promise.race([promise1, promise2, promise3])
  .then(result => {
    console.log("Resolved with:", result);
  })
  .catch(error => {
    console.error("Rejected with:", error);
  });
```

javascript

Example

```
const fast = new Promise(resolve => setTimeout(() => resolve("Fast"), 100));
const slow = new Promise(resolve => setTimeout(() => resolve("Slow"), 500));

Promise.race([fast, slow])
  .then(value => console.log(value)); // "Fast"
```

javascript

What is promise.allSettled()

- `Promise.allSettled()` waits for **all promises to settle**, meaning each has either:
 - **Resolved** ✓
 - **Rejected** ✗
- It **never short-circuits**, unlike `Promise.all()` or `Promise.race()`.
- Returns an array of objects describing the **outcome of each promise**.

```
Promise.allSettled([promise1, promise2, promise3])
  .then(results => {
    console.log(results);
  });
```

javascript

Example

```
const p1 = Promise.resolve("✅ Success");
const p2 = Promise.reject("❌ Failed");
const p3 = Promise.resolve("✅ Another success");

Promise.allSettled([p1, p2, p3]).then(results => {
  console.log(results);
});

// output

[
  { status: 'fulfilled', value: '✅ Success' },
  { status: 'rejected', reason: '❌ Failed' },
  { status: 'fulfilled', value: '✅ Another success' }
]
```

javascript

What is promise.any()

- `Promise.any()` returns the **first fulfilled** promise. ✅
- It **ignores all rejected** promises. ❌
- If **all promises reject**, it throws an `AggregateError`.

```
Promise.any([promise1, promise2, promise3])
  .then(result => {
    console.log(result); // first resolved value
  })
  .catch(err => {
    console.log(err); // if all promises reject
  });
```

javascript

Example

```
const p1 = Promise.reject("❌ Error 1");
const p2 = Promise.resolve("✅ Success from p2");
const p3 = Promise.resolve("✅ Success from p3");

Promise.any([p1, p2, p3]).then(result => {
  console.log(result);
}).catch(error => {
  console.log("All promises failed");
});
```

javascript

```
// output
✅ Success from p2
```

Promise Combinators Comparison

Method	When it Resolves	When it Rejects	Notes
Promise.all()	When all promises resolve	If any promise rejects	Fails fast; returns array of results
Promise.allSettled()	When all promises settle (resolved or rejected)	Never rejects	Returns array of { status, value/reason }
Promise.race()	When the first promise settles (resolve or reject)	Same	Returns value or error of the first settled promise
Promise.any()	When the first promise resolves	If all promises reject (AggregateError)	Ignores rejections unless all fail

Promise Chaining

Promise chaining allows you to perform multiple asynchronous operations sequentially.

Each `.then()` returns a new promise, which can be chained to the next operation. This helps to manage sequences of asynchronous operations.

```
fetchData()
  .then(result => processData(result))
  .then(processedData => displayData(processedData))
  .catch(error => handleError(error));
```

javascript

Explanation:

1. The first `then()` processes the result of `fetchData()`.
2. The second `then()` processes the output of the previous `.then()`.
3. If any of the promises fails, the error will be caught in the `catch()` block at the end.

Example

```
function getUserData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve("User data fetched"), 1000);
  });
}
```

javascript

```
function processUserData(data) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(data + " and processed"), 1000);
  });
}

getUserData()
  .then(data => {
    console.log(data);
    return processUserData(data);
  })
  .then(processedData => {
    console.log(processedData);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

In this example:

1. The `getUserData()` function returns a promise.
2. The result is processed by `processUserData()`.
3. If any step fails, the error is caught by `catch()`.

Async / Await

Async/Await is a modern way to handle asynchronous code, making it easier to work with promises and asynchronous operations. It allows you to write asynchronous code in a way that looks and behaves like synchronous code, improving readability and reducing complexity.

- `async`: A function marked with the `async` keyword always returns a promise.
- `await`: The `await` keyword can only be used inside an `async` function and it pauses the execution of the function until the promise is resolved.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}
```

javascript

Example

```
function wait(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
async function run() {  
  console.log("Start");  
  await wait(1000);  
  console.log("1 second later");  
}  
  
run();
```

javascript

Concurrent Async Operations

When working with async/await, sometimes you may need to execute multiple asynchronous operations concurrently. Instead of awaiting each operation one by one, you can run them simultaneously and wait for all of them to finish.

Using Promise.all for Concurrent Async Operations

Promise.all() is used when you want to execute multiple promises concurrently and wait for all of them to resolve (or any of them to reject).

```
async function fetchData() {  
  try {  
    const userPromise = fetch('https://api.example.com/user');  
    const postsPromise = fetch('https://api.example.com/posts');  
    const commentsPromise = fetch('https://api.example.com/comments');  
  
    const [user, posts, comments] = await Promise.all([userPromise, postsPromise, commentsPromise]);  
  
    const userData = await user.json();  
    const postsData = await posts.json();  
    const commentsData = await comments.json();  
  
    console.log(userData, postsData, commentsData);  
  } catch (error) {  
    console.log("Error: " + error.message);  
  }  
}  
  
fetchData();
```

javascript

Key Points:

1. Promise.all(): Accepts an array of promises and returns a new promise that resolves when all input promises are resolved. If any promise is rejected, the entire Promise.all() call will reject.

2. Concurrency: The promises are run concurrently, which is faster than waiting for each one to finish before starting the next.
3. Destructuring: You can use array destructuring to access the results from each promise once all of them resolve.