JavaScript Asynchronous Programming — Advanced Master Guide (Document 2)

This document is the **advanced continuation** of the Async Basics document. It focuses on: - Promises (deep explanation) - Promise error handling - All Promise helper methods - Async/Await in detail - Converting callback → promise → async/await - Real-time async patterns - Task → Explanation → Code → Output format

---

1. What is a Promise? (In Detail)

A Promise represents a value that will be available **now, later, or never**.

States

- **pending** – still running
- **fulfilled** – success (resolved)
- **rejected** – failed

Basic Syntax

```
const promise = new Promise((resolve, reject) => {
  // do work
  if (/* success */) resolve(value);
  else reject(error);
});
```

---

2. Promise — Beginner to Advanced Examples

Example 1: Simple Promise (No setTimeout)

Task

Check if a number is even or odd using a promise.

Code

```
function isEven(num) {
  return new Promise((resolve, reject) => {
    if (typeof num !== "number") reject("Invalid number");
    else resolve(num % 2 === 0);
  });
}
```

```
isEven(5)
  .then(result => console.log("Is Even?", result))
  .catch(err => console.log("Error:", err));
```

Output

```
Is Even? false
```

Example 2: Promise with Delay (Simulating API)

Task

Return a fake user after a delay.

```
function getUser() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ id: 1, name: "Alice" });
    }, 1000);
  });
}

g etUser().then(user => console.log(user));
```

Output

```
{ id: 1, name: 'Alice' }
```

---

3. Error Handling in Promises

Task

Safely divide numbers.

```
function divide(a, b) {
  return new Promise((resolve, reject) => {
    if (b === 0) reject("Cannot divide by zero");
    else resolve(a / b);
  });
}
```

```
divide(10, 0)
  .then(res => console.log(res))
  .catch(err => console.log("Error:", err))
  .finally(() => console.log("Operation complete"));
```

Output

```
Error: Cannot divide by zero
Operation complete
```

---

4. All Promise Helper Methods (Explained)

4.1 Promise.resolve(value)

```
Promise.resolve(50).then(console.log);
```

Output: `50`

4.2 Promise.reject(error)

```
Promise.reject("Wrong").catch(console.log);
```

Output: `Wrong`

4.3 Promise.all([...]) — Parallel Execution

Waits for **all** to finish.

```
Promise.all([
  Promise.resolve(1),
  Promise.resolve(2),
  Promise.resolve(3)
]).then(console.log);
```

Output: `[1,2,3]`

4.4 Promise.race([...]) — First Result Wins

```
const p1 = new Promise(res => setTimeout(() => res("A"), 100));
const p2 = new Promise(res => setTimeout(() => res("B"), 50));
Promise.race([p1, p2]).then(console.log);
```

Output: `B`

4.5 Promise.allSettled([...]) — Success + Failure Reports

```
Promise.allSettled([
  Promise.resolve("OK"),
  Promise.reject("Fail")
]).then(console.log);
```

4.6 Promise.any([...]) — First SUCCESS Wins

```
Promise.any([
  Promise.reject("Bad"),
  Promise.resolve("Success")
]).then(console.log);
```

Output: `Success`

---

5. async / await — Detailed

What is async?

Adds a **promise-returning context** to a function.

What is await?

Pauses execution inside an async function until a Promise resolves.

Example 1: Converting Promise `.then` → async/await

```
function getNumber() {
  return Promise.resolve(20);
}

async function show() {
  const n = await getNumber();
  console.log(n);
}
```

```
show();
```

Example 2: async/await with try/catch

```
function dividePromise(a, b) {
  return new Promise((resolve, reject) => {
    if (b === 0) reject("Divide error");
    else resolve(a / b);
  });
}

async function safeDivide() {
  try {
    const result = await dividePromise(10, 0);
    console.log(result);
  } catch (err) {
    console.log("Error:", err);
  }
}

safeDivide();
```

---

6. Full Transformations (Callback → Promise → Async/Await)

Example Transformation 1 — Basic

Callback Version

```
function task(cb) {
  cb("done");
}
```

Promise Version

```
function task() {
  return Promise.resolve("done");
}
```

async/await Version

```
async function run() {
  const r = await task();
  console.log(r);
}
```

Example Transformation 2 — Step-by-Step Workflow

Callback Hell

```
step1(function(a) {
  step2(function(b) {
    step3(function(c) {
      console.log(a, b, c);
    });
  });
});
```

Promise Chain

```
step1()
  .then(a => step2())
  .then(b => step3())
  .then(c => console.log(c));
```

async/await

```
async function flow() {
  const a = await step1();
  const b = await step2();
  const c = await step3();
  console.log(a, b, c);
}
```

---

7. Real-Time Async Patterns

Pattern 1 — Fetch user → fetch orders

```
async function getUserDetails() {
  const user = await getUser();
```

```
  const orders = await getOrders(user.id);
  return { user, orders };
}
```

Pattern 2 — Parallel API Calls

```
const [products, categories] = await Promise.all([
  getProducts(),
  getCategories()
]);
```

Pattern 3 — Retry Logic (Manual)

```
async function retry(fn, retries = 3) {
  for (let i = 0; i < retries; i++) {
    try {
      return await fn();
    } catch (err) {
      if (i === retries - 1) throw err;
    }
  }
}
```

---

8. Visual Diagrams (Advanced)

Diagram A — Promise Lifecycle

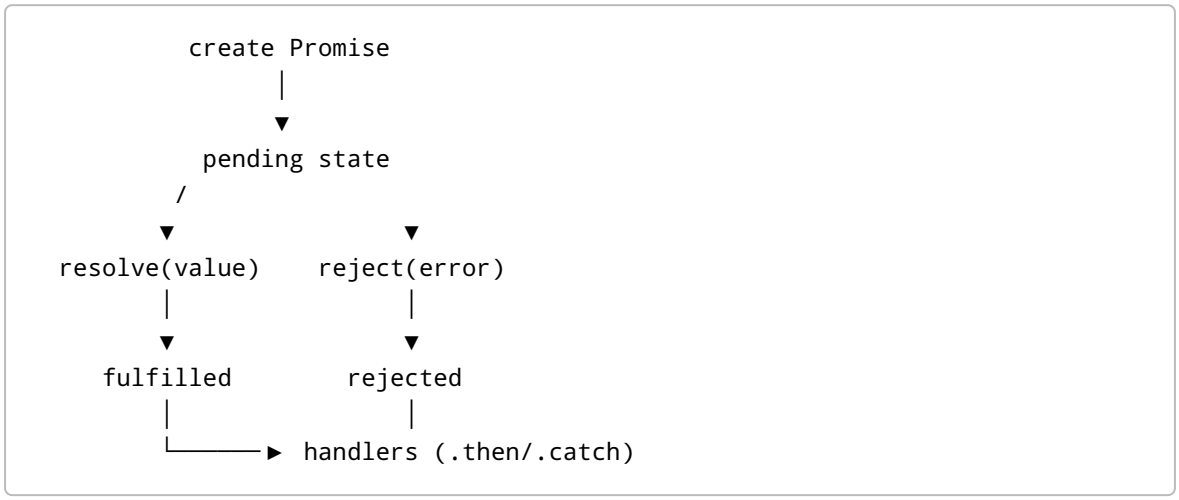


Diagram B — async/await Flow

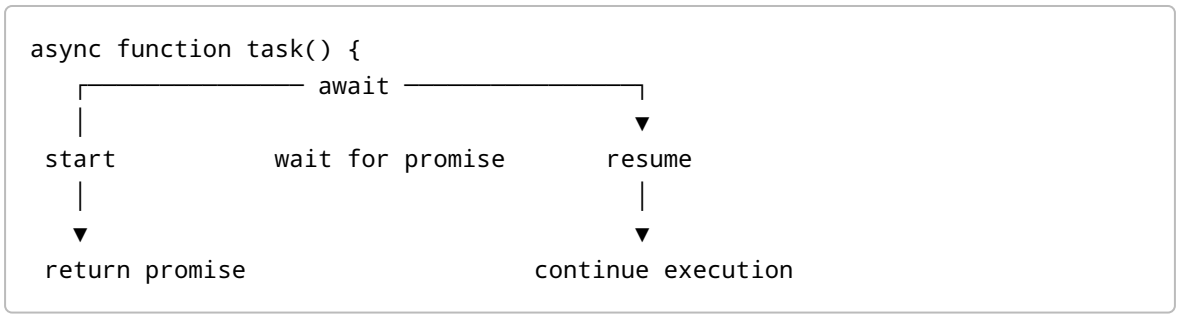
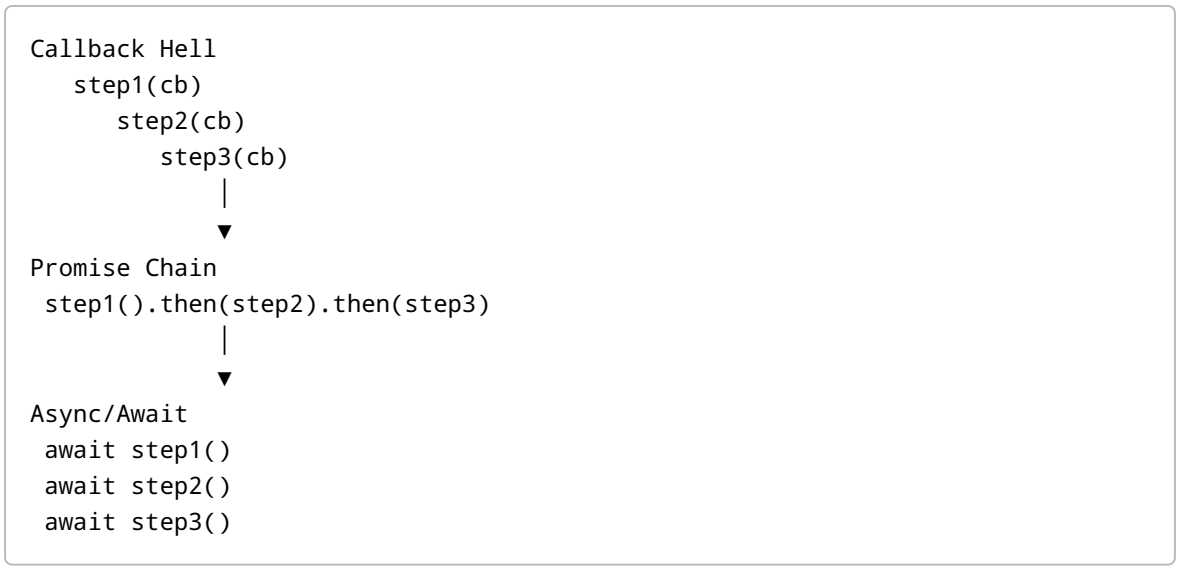


Diagram C — Promise Queue vs Callback Queue

Diagram D — Callback → Promise → Async/Await



---

10. Interview Questions (Promises & async/await)

1. What is a Promise?

A Promise represents a value that will be available now or in the future.

2. What are the states of a Promise?

Pending, fulfilled, rejected.

3. What is the difference between then() and catch()?

`then` handles success; `catch` handles errors.

4. Does async function always return a Promise?

Yes.

5. What is the benefit of async/await over Promises?

Cleaner, more readable syntax.

6. How do you handle errors in async/await?

Using try/catch.

7. What is Promise.all used for?

Execute multiple async tasks in parallel.

8. What is Promise.race used for?

Return the first completed Promise.

9. What is Promise.any?

Returns first fulfilled Promise. Ignores failures.

10. What is Promise.allSettled?

Returns status for all promises regardless of success/failure.

11. Why do we need promises?

To avoid callback hell and handle async operations cleanly.

12. What is the microtask queue?

Queue where Promise callbacks run.

13. Why is async/await preferred?

Readability + synchronous-like flow + easy error handling.

14. Can we mix async/await and then()?

Yes, but avoid unless needed.

15. What happens if you forget await?

The function returns a pending Promise instead of its result.

End of Document