6. Promise async await

Monday, 22 December 2025 2:00 PM

Great question 👍
Promises and async/await are **core JavaScript concepts**, especially important when you come from a **Java / backend background** and move into **React / Next.js**.
I'll explain this **step by step**, starting simple and then going deeper.

1️⃣ Why Promises exist (the problem they solve)

JavaScript is **single-threaded**.
Some tasks take time:

- API calls (fetch user data)
- Reading files
- Database calls
- Timers (setTimeout)

We **don't want to block** the app while waiting.
👉 Promises represent **a value that will be available later**.

2️⃣ What is a Promise (simple definition)

Promise = a container for a future result

> "I promise I'll give you the result later — either success or failure."

Java analogy:

```
Future<User> userFuture;
```

JavaScript analogy:

```
Promise<User>
```

3️⃣ Promise states (VERY IMPORTANT)

A Promise can be in **only one** of these states:

State	Meaning
pending	Still running
fulfilled	Success (resolved)
rejected	Failed (error)

Once fulfilled or rejected → **state is final** ❌

4️⃣ Creating a Promise

```
const promise = new Promise((resolve, reject) => {
 const success = true;
if (success) {
   resolve("Data received");
 } else {
   reject("Something went wrong");
 }
});
```

- resolve(value) → success
- reject(error) → failure

5️⃣ Consuming a Promise (then, catch, finally)

```
promise
 .then(result => {
   console.log(result); // success
 })
 .catch(error => {
   console.error(error); // failure
 })
 .finally(() => {
   console.log("Done");
 });
```

Flow:

Promise → then() → catch() → finally()

6️⃣ Promise chaining (very common)

```
fetchUser()
  .then(user => fetchOrders(user.id))
  .then(orders => fetchOrderDetails(orders[0].id))
  .then(details => console.log(details))
  .catch(err => console.error(err));
```

Each .then():

- returns a **new Promise**
- receives previous result

7️⃣ Promise Hell (why async/await was introduced)

```
doTask1()
  .then(res1 => {
    return doTask2(res1)
      .then(res2 => {
        return doTask3(res2);
      });
  });
```

Hard to read 😵

8️⃣ async / await (cleaner way)

async/await is **just syntax sugar over Promises**.

Same example rewritten:

```
async function process() {
  try {
    const res1 = await doTask1();
    const res2 = await doTask2(res1);
    const res3 = await doTask3(res2);
    console.log(res3);
  } catch (err) {
    console.error(err);
  }
}
```

Rules:

- async function **always returns a Promise**
- await pauses execution **until promise resolves**
- Use try/catch for errors

9️⃣ async vs Promise.then (when to use what)

Promise.then	async/await
Functional style	Imperative style
Harder to read	Easier to read
Useful for chaining	Best for logic
Older style	Modern standard

👉 **Use async/await in most cases**

🔟 Different types / patterns of Promises

1️⃣ Promise.resolve()

```
Promise.resolve(10).then(console.log);
```

2️⃣ Promise.reject()

```
Promise.reject("Error").catch(console.error);
```

3️⃣ Promise.all() (parallel execution)

Runs all promises **together**

❌ Fails if **any one fails**

```
const result = await Promise.all([
  fetchUser(),
  fetchOrders(),
  fetchProducts()
]);
```

4️⃣ Promise.allSettled() (wait for all results)

✔ Doesn’t fail if one fails

```
const results = await Promise.allSettled([
  fetchUser(),
  fetchOrders()
]);
```

Returns:

```
[
  { status: "fulfilled", value: ... },
  { status: "rejected", reason: ... }
]
```

5️⃣ Promise.race() (first response wins)

```
await Promise.race([
  fetchFastAPI(),
  fetchSlowAPI()
]);
```

6️⃣ Promise.any() (first success wins)

```
await Promise.any([
  Promise.reject("fail"),
  Promise.resolve("success")
]);
```

1️⃣1️⃣ Error handling (important)

❌ Wrong

```
await fetchData();
```

✅ Correct

```
try {
  await fetchData();
} catch (e) {
  console.error(e);
}
```

Or with .catch():

```
fetchData().catch(console.error);
```

1️⃣2️⃣ Real React / Next.js example

```
useEffect(() => {
  async function loadData() {
    try {
      const res = await fetch("/api/users");
      const data = await res.json();
      setUsers(data);
    } catch (e) {
      console.error(e);
    }
  }
loadData();
}, []);
```

1️⃣3️⃣ Java comparison (easy mapping)

Java	**JavaScript**
Thread	Event loop
Future	Promise
try/catch	try/catch
Blocking	Non-blocking

🧠 Mental model to remember

Promise = “I will give you data later”
then = “When it’s ready”
catch = “If it fails”
async/await = “Write async code like sync code”

If you want, next I can:

- Explain **event loop & microtasks**
- Show **Promise interview questions**
- Compare **callback vs promise vs async**
- Show **common mistakes developers make**

Just tell me 🙂

Perfect 👍
These are **very common interview areas**, so I’ll structure this clearly and practically.

1️⃣ Promise Interview Questions (with answers)

Q1. What is a Promise?

Answer:
A Promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Q2. What are the states of a Promise?

Answer:

- pending
- fulfilled
- rejected

Once fulfilled/rejected → state cannot change.

Q3. Is Promise synchronous or asynchronous?

Answer:
Creating a Promise is **synchronous**, but .then() callbacks run **asynchronously** (microtask queue).

Q4. Difference between .then() and await?

Answer:

- Both handle resolved Promises
- await is syntactic sugar over .then()

- await is syntactic sugar over .then()
- await pauses execution inside async function

Q5. What happens if you await a non-Promise?

Answer:
It gets automatically wrapped in Promise.resolve(value).

```
await 5; // works
```

Q6. Difference between Promise.all and Promise.allSettled?

Answer:

Promise.all	**Promise.allSettled**
Fails fast	Never fails
Stops on first reject	Waits for all

Q7. Can we cancel a Promise?

Answer:
❌ No, Promises cannot be cancelled directly.
✔ Use AbortController or flags.

Q8. Does async function always return a Promise?

Answer:
✔ Yes — even if you return a normal value.

Q9. What happens if you don't handle a rejected Promise?

Answer:
You get **UnhandledPromiseRejection** (runtime error).

Q10. Difference between Promise.any and Promise.race?

Answer:

- race → first settled (success or failure)
- any → first fulfilled only

2️⃣ Callback vs Promise vs Async/Await

🔷 Callback

```
getUser(id, (err, user) => {
  if (err) return handle(err);
getOrders(user.id, (err, orders) => {
    if (err) return handle(err);
console.log(orders);
  });
});
```

❌ Problems

- Callback hell
- Hard error handling
- Inversion of control

🔷 Promise

```
getUser(id)
  .then(user => getOrders(user.id))
  .then(orders => console.log(orders))
  .catch(err => console.error(err));
```

✅ Improvements

- Chainable
- Centralized error handling

- Centralized error handling
- More readable

🔷 Async / Await (BEST)

```
async function loadData() {
  try {
    const user = await getUser(id);
    const orders = await getOrders(user.id);
    console.log(orders);
  } catch (err) {
    console.error(err);
  }
}
```

✅ Advantages

- Looks synchronous
- Easy debugging
- Clean error handling

📊 Comparison Table

Feature	Callback	Promise	Async/Await
Readability	❌	👍	⭐⭐⭐
Error handling	❌	👍	⭐⭐⭐
Debugging	❌	😐	⭐⭐⭐
Nesting	❌	😐	❌
Modern use	❌	👍	⭐⭐⭐

👉 **Interview line:**

Async/await is built on top of Promises; it doesn’t replace them.

3️⃣ Common Mistakes Developers Make 🚨

❌ 1. Forgetting await

```
const data = fetchData(); // Promise, not value ❌
```

✅ Fix

```
const data = await fetchData();
```

❌ 2. Using await inside forEach

```
arr.forEach(async item => {
  await process(item); // ❌ does not wait
});
```

✅ Fix

```
for (const item of arr) {
  await process(item);
}
```

or parallel:

```
await Promise.all(arr.map(process));
```

❌ 3. Not handling errors

```
await fetchData(); // ❌ crash on reject
```

✅ Fix

```
try {
  await fetchData();
} catch (e) {
} catch (e) {
  console.error(e);
}
```

❌ 4. Mixing .then() with await

```
await fetchData().then(res => res.json()); // ❌
```

✅ Fix

```
const res = await fetchData();
const data = await res.json();
```

❌ 5. Sequential calls instead of parallel

```
const a = await api1();
const b = await api2();
```

✅ Fix (parallel)

```
const [a, b] = await Promise.all([api1(), api2()]);
```

❌ 6. Assuming Promise.all always returns results

```
await Promise.all([api1(), api2()]); // ❌ one fails → all fail
```

✅ Fix

```
await Promise.allSettled([...]);
```

❌ 7. Returning inside then incorrectly

```
getData().then(data => {
  process(data); // ❌ missing return
});
```

✅ Fix

```
return getData().then(data => process(data));
```

❌ 8. Using async without need

```
async function sum(a, b) {
  return a + b; // unnecessary async ❌
}
```

❌ 9. Thinking async code is multithreaded

👉 JavaScript is **single-threaded**
Async ≠ parallel threads (unless Web Workers)

❌ 10. Forgetting return in async function

```
async function getData() {
  fetch(url); // ❌ returns undefined
}
```

✅ Fix

```
return fetch(url);
```

🧠 Interview-ready summary

Callbacks → Promises → Async/Await is the evolution
Async/await improves readability but relies on Promises
Always handle errors and use parallel execution when possible

If you want next, I can:

- Explain **event loop & microtask queue (very common interview)**