◇ Using `.then()` (Without `async/await`)

☑ Code

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

function getData() {
  p.then((res) => console.log(res));
  console.log("JavaScript"); // This runs immediately
}

getData();
```

📤 Output

```
JavaScript
// (After 10 seconds...)
Resolved Value!!
```

🧠 Explanation

- The promise `p` is created and resolves after 10 seconds.
- `getData()` is called:
  - `.then()` attaches a callback to run when the promise resolves.
  - `console.log("JavaScript")` runs **immediately** — JavaScript doesn't wait for the promise.
- After 10 seconds, the promise resolves and logs: `Resolved Value!!`.

◇ Using `async/await`

☑ Code

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

async function handlePromise() {
  const val = await p;
```

```
  console.log("Darshan Vasani");
  console.log(val);
}

handlePromise();
```

📤 Output

```
// (Waits 10 seconds silently...)
Darshan Vasani
Resolved Value!!
```

🧠 Explanation

- The promise `p` resolves after 10 seconds.
- `handlePromise()` is an **async function**.
  - `await p` pauses execution inside the function until the promise resolves.
- After 10 seconds:
  - It logs `"Darshan Vasani"`.
  - Then it logs the resolved value: `"Resolved Value!!"`.

⚖️ Key Differences Between `.then()` and `async/await`

Feature	**`.then()` (Old Way)**	**`async/await` (New Way)**
Syntax	Functional chaining	Looks like synchronous code
Readability	Can be harder with nested `.then()`	More readable and structured
Error Handling	`.catch()` for errors	`try...catch` block
Execution Flow	Continues immediately after `.then()`	Pauses at `await` inside the async function
Output Order (example above)	`"JavaScript"` first, then result	Waits 10s, then `"Darshan Vasani"` + result

---

🚀 Bonus: With Error Handling

☑️ Using `.then().catch()`

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
```

```
    reject("Something went wrong!");
  }, 5000);
});

function getData() {
  p.then((res) => console.log(res))
   .catch((err) => console.error(err));
}

getData();
```

☑ Using `async/await` + `try...catch`

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("Something went wrong!");
  }, 5000);
});

async function handlePromise() {
  try {
    const val = await p;
    console.log(val);
  } catch (err) {
    console.error(err);
  }
}

handlePromise();
```

☑ Code

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

async function handlePromise() {
  console.log("Hello World");

  const val = await p;
  console.log("Darshan Vasani");
  console.log(val);
}

handlePromise();
```

Output

```
Hello World
// (waits 10 seconds...)
Darshan Vasani
Resolved Value!!
```

Explanation (Line-by-Line)

1. **Promise Declaration (`p`)** A promise `p` is created. It will `resolve` with the string `"Resolved Value!!"` after **10 seconds**.
2. `handlePromise()` **Function Call** The `async` function `handlePromise()` is invoked.
3. `console.log("Hello World")` Immediately prints `"Hello World"` to the console. This line executes **synchronously**.
4. `const val = await p;` JavaScript **pauses execution inside** `handlePromise` at this line and waits for `p` to resolve (10 seconds delay).
5. **After 10 Seconds...** Once the promise resolves:
    - `val` is assigned `"Resolved Value!!"`
    - The function resumes execution.
6. `console.log("Darshan Vasani")` **and** `console.log(val)` These lines run after the delay:
    - First, `"Darshan Vasani"` is printed.
    - Then, `"Resolved Value!!"` is printed.

Timeline Summary

Time	Console Output
0 sec	`Hello World`
10 sec	`Darshan Vasani`
10 sec	`Resolved Value!!`

multiple `await` calls on the **same Promise**.

Code with Detailed Logging and Context

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

async function handlePromise() {
  console.log("🚀 Starting async function: handlePromise()");
  console.log("👉 Logging immediately before awaiting the Promise for the first
time.");

  const val1 = await p;
  console.log("☑️ First await completed. Promise has been resolved.");
  console.log("🔁 Logging the first resolved value:");
  console.log("promise 1 time →", val1);

  console.log("🤔 Awaiting the same resolved Promise again — this should be
instant.");
  const val2 = await p;
  console.log("☑️ Second await returned immediately because Promise was already
settled.");
  console.log("🔁 Logging the second resolved value:");
  console.log("promise 2 time →", val2);
}

handlePromise();
```

📤 Output

```
🚀 Starting async function: handlePromise()
👉 Logging immediately before awaiting the Promise for the first time.
// (waits 10 seconds...)
☑️ First await completed. Promise has been resolved.
🔁 Logging the first resolved value:
promise 1 time → Resolved Value!!
🤔 Awaiting the same resolved Promise again — this should be instant.
☑️ Second await returned immediately because Promise was already settled.
🔁 Logging the second resolved value:
promise 2 time → Resolved Value!!
```

🤔 Explanation of Code and Execution Flow

1. Promise Creation

- A **Promise** `p` is created, which will resolve with `"Resolved Value!!"` after **10 seconds**.

2. Calling `handlePromise()`

- The function `handlePromise()` is executed. The first `console.log` statement `"🚀 Starting async function: handlePromise()"` is logged immediately.

3. First `await p`

- The code pauses at the first `await p` for **10 seconds**, as the Promise hasn't been resolved yet.
- Once the Promise resolves, the value `"Resolved Value!!"` is assigned to `val1`.
- `console.log("☑ First await completed. Promise has been resolved.")` and `"promise 1 time → Resolved Value!!"` are printed after the 10-second wait.

4. Second `await p`

- The second `await p` is encountered, but **since the Promise is already resolved**, this `await` resolves immediately without waiting.
- This results in `"☑ Second await returned immediately because Promise was already settled."` and `"promise 2 time → Resolved Value!!"` being printed instantly.

🧠 Detailed `console.log` Statements Explanation

Each `console.log` in the code provides valuable insights into the execution flow:

- 🚀 **Starting async function: handlePromise()** — Marks the start of the function.
- 👉 **Logging immediately before awaiting the Promise for the first time.** — Provides context that the first `await` is about to happen.
- ☑ **First await completed. Promise has been resolved.** — Indicates that the promise has been resolved after the 10-second wait.
- 🔂 **Logging the first resolved value:** — Clarifies that the first value is logged after resolving the Promise.
- 🧠 **Awaiting the same resolved Promise again — this should be instant.** — Explains that the second `await` will resolve immediately since the promise is already settled.
- ☑ **Second await returned immediately because Promise was already settled.** — Provides context that no delay happens this time.
- 🔂 **Logging the second resolved value:** — Logs the second resolved value without delay.

🔄 Key Insights

- **Once a Promise is resolved**, any subsequent `await` on that Promise will return immediately without waiting for the delay.
- **Promises are resolved once** — subsequent calls to `await` on the same Promise will not cause a delay.

This structure helps in understanding **asynchronous behavior** and **Promise resolution**.

⏱ Timeline Summary

Time	Output
0 sec	🚀 `Starting async function: handlePromise()`

Time	Output
0 sec	👉 Logging immediately before awaiting the Promise for the first time.
10 sec	☑️ First await completed. Promise has been resolved.
10 sec	promise 1 time → Resolved Value!!
10.001 sec	🤔 Awaiting the same resolved Promise again — this should be instant.
10.001 sec	☑️ Second await returned immediately because Promise was already settled.
10.001 sec	promise 2 time → Resolved Value!!

JavaScript Multiple Promises (p1 and p2)

☑️ Code with Multiple Promises (p1 and p2)

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 5000); // 5-second delay
});

async function handlePromise() {
  console.log("🚀 Starting async function: handlePromise()");
  console.log("👉 Logging immediately before awaiting the Promise for the first
time.");

  const val1 = await p1;
  console.log("☑️ First await completed. Promise has been resolved.");
  console.log("🔁 Logging the first resolved value:");
  console.log("promise 1 time →", val1);

  console.log("🤔 Awaiting the second Promise (`p2`) — this should have a shorter
delay.");
  const val2 = await p2;
  console.log("☑️ Second await completed. Promise has been resolved.");
  console.log("🔁 Logging the second resolved value:");
  console.log("promise 2 time →", val2);
}

handlePromise();
```

👇☑ Expected Behavior

📤 Output

```
🚀 Starting async function: handlePromise()
👉 Logging immediately before awaiting the Promise for the first time.
// (waits 10 seconds...)
☑ First await completed. Promise has been resolved.
🔁 Logging the first resolved value:
promise 1 time → Resolved Value!!
🤔 Awaiting the second Promise (`p2`) — this should have a shorter delay.
// (waits 5 seconds...)
☑ Second await completed. Promise has been resolved.
🔁 Logging the second resolved value:
promise 2 time → Resolved Value!!
```

🤔 Explanation of Code and Execution Flow

1. Promise Creation

- `p1` is a Promise that resolves after **10 seconds**.
- `p2` is a Promise that resolves after **5 seconds**.

2. Calling `handlePromise()`

- The function `handlePromise()` is executed. The first `console.log` statement `"🚀 Starting async function: handlePromise()"` is logged immediately.

3. First `await p1`

- The code pauses at the first `await p1` for **10 seconds** because `p1` has a **10-second delay**.
- Once the Promise resolves, the value `"Resolved Value!!"` is assigned to `val1`.
- `"☑ First await completed. Promise has been resolved."` and `"promise 1 time → Resolved Value!!"` are printed after the 10-second wait.

4. Second `await p2`

- The second `await p2` is encountered and waits for the **5-second delay** of `p2` to resolve.
- Once `p2` resolves, the value `"Resolved Value!!"` is assigned to `val2`.
- `"☑ Second await completed. Promise has been resolved."` and `"promise 2 time → Resolved Value!!"` are printed after the 5-second wait.

🤔 Detailed `console.log` Statements Explanation

Each `console.log` in the code provides valuable insights into the execution flow:

- 🚀 **Starting async function: handlePromise()** — Marks the start of the function.
- 👉 **Logging immediately before awaiting the Promise for the first time.** — Provides context that the first `await` is about to happen.
- ☑️ **First await completed. Promise has been resolved.** — Indicates that the first Promise (`p1`) has been resolved after waiting for 10 seconds.
- 🔁 **Logging the first resolved value:** — Logs the value of `p1` after resolution.
- 🤔 **Awaiting the second Promise (`p2`) — this should have a shorter delay.** — Explains that the second `await` will wait for the 5-second delay of `p2`.
- ☑️ **Second await completed. Promise has been resolved.** — Indicates that the second Promise (`p2`) has been resolved after 5 seconds.
- 🔁 **Logging the second resolved value:** — Logs the value of `p2` after resolution.

🔄 Key Insights

- **Promises with different delays**:
  - `p1` and `p2` have different **resolve times** (10 seconds vs. 5 seconds). The second `await` will still wait for its respective Promise to resolve, causing sequential delays.
  - **Awaiting `p1` first** takes 10 seconds, then **awaiting `p2`** will take an additional 5 seconds.
- **Asynchronous Behavior**:
  - Each `await` pauses execution, waiting for the Promise to resolve before proceeding to the next line.
  - The **second `await` does not wait for the first Promise** to complete; it waits for **its own associated Promise** (`p2`).

⏱️ Timeline Summary

Time	Output
0 sec	`🚀 Starting async function: handlePromise()`
0 sec	`👉 Logging immediately before awaiting the Promise for the first time.`
10 sec	`☑️ First await completed. Promise has been resolved.`
10 sec	`promise 1 time → Resolved Value!!`
10.001 sec	`🤔 Awaiting the second Promise (p2) — this should have a shorter delay.`
15 sec	`☑️ Second await completed. Promise has been resolved.`
15 sec	`promise 2 time → Resolved Value!!`

👇☑️ Real Behavior Great Observation

❓ Ques)

> For those who think:
>
> *"When the code reaches `await p1`, it suspends execution for 10 seconds. After that, it goes to `await p2`, where the timer starts (5 seconds), so the total time should be 15 seconds."*
>
> Is that true?

☑ Answer)

🧐 **No, that's a misunderstanding!** The key thing to understand is:

> ⏰ **The timer for a Promise starts when it's created, not when it's awaited.**

👇 Let's break it down:

📦 Case 1: Promises Declared *Before* Async Function

```
const p1 = new Promise(...setTimeout 10s);
const p2 = new Promise(...setTimeout 5s);

async function handlePromise() {
  await p1;
  await p2;
}
```

- ☑ `p1` and `p2` start **ticking immediately**, even before the function is called.
- ☑ `await p1` suspends execution, but **both promises are running in background**.
- ☑ By the time `await p2` is hit, `p2` may have **already finished** (because its timer started early).
- ⏱ **Total execution time: ~10s**

🚗 Case 2: Promises Created *Inside* Async Function

```
async function handlePromise() {
  const p1 = new Promise(...setTimeout 10s);
  const val1 = await p1;

  const p2 = new Promise(...setTimeout 5s);
  const val2 = await p2;
}
```

- ❌ `p1` starts only when function hits it.
- ❌ `p2` starts **only after** `await p1` completes.
- ⏱ Total time = 10s + 5s = **15s**

🎯 Visual Timeline Comparison

☑️ Case 1 (Parallel)

```
Time: 0s  → p1 starts, p2 starts
Time: 10s → p1 resolves, p2 already resolved
Total:     ☑️ 10s
```

❌ Case 2 (Sequential)

```
Time: 0s  → p1 starts
Time: 10s → p1 resolves
Time: 10s → p2 starts
Time: 15s → p2 resolves
Total:     ❌ 15s
```

🧠 Final Takeaway:

> 🕰️ **Declaring a Promise = starting its timer.**
>
> 🔘 Awaiting a Promise = waiting for its result (but it doesn't start the timer).

🧠 Key Learning: `await ≠ start timer`

> **Declaring a Promise starts the timer, not the `await`.** `await` just pauses execution until the promise resolves.
>
> 🧠 So if both promises are declared upfront, they run **in parallel**, and total time = time of the slower one.

❓ Why does this **seem like** it's behaving *concurrently*, even though the `await`s are **sequential**?

Because both Promises `p1` and `p2` are **created before** the `async` function starts. Let's break it down:

🧠 Here's what's *actually* happening:

☑️ Step 1: Promise Initialization

```
const p1 = new Promise(...) // starts the 10s timer immediately
const p2 = new Promise(...) // starts the 5s timer immediately
```

➡ These Promises **start running their timers at the same time**, before `handlePromise()` even starts executing. That means:

- `p1` will resolve **10 seconds later**
- `p2` will resolve **5 seconds later**, regardless of when you `await` them

---

🩹 Execution Timeline (Real Behavior)

Time	What Happens
0s	`p1` and `p2` are both created & timers start
0s	`handlePromise()` starts
0s	`await p1` → waits for remaining ~10s
10s	`p1` resolves
10s	now `await p2` → `p2` has already resolved after 5s, so this resolves **instantly** (no additional wait)
10s	function completes

So in effect, **you only wait 10 seconds total**, not 15 — because `p2` was *already resolved* while you were waiting on `p1`.

📌 Key Insight

> ❗ `await` pauses execution of the async function — but **it doesn't delay the start of the Promise**. Promises start running as soon as they are **constructed**.

☑ Want true sequential behavior (total wait = 15s)?

If you want `p2` to **start only after** `p1` completes, then you must create it *after* the first `await`:

```
async function handlePromise() {
  const val1 = await new Promise((resolve) => {
    setTimeout(() => resolve("Resolved p1"), 10000);
  });

  const val2 = await new Promise((resolve) => {
    setTimeout(() => resolve("Resolved p2"), 5000);
  });

  console.log("val1:", val1);
  console.log("val2:", val2);
}
```

This way, the second Promise doesn't even *exist* until the first one finishes.

Summary

- Your code is **sequentially awaiting** promises, but the Promises themselves are **concurrently executing** from the beginning.
- To get sequential behavior in **execution *and* start time**, you need to **construct the Promises after the awaits**, not before.

Perfect! Here's a **fully formatted**, emoji-rich, **comprehensive markdown version** covering:

- Both cases with code
- Sample output with timestamps
- Execution timelines
- Behavior summary
- Real-world analogy
- Key takeaways

Understanding `await` Behavior: Parallel vs Sequential Promises in JavaScript

---

Case 1: Promises Created *Before* the Async Function (Parallel Execution)

Code

```
const p1 = new Promise((resolve) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10s delay
});

const p2 = new Promise((resolve) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 5000); // 5s delay
});

async function handlePromise() {
  const startTime = Date.now();

  console.log(`[${new Date().toISOString()}] Starting async function:
handlePromise()`);
  console.log(`[${new Date().toISOString()}] Logging immediately before
awaiting the Promise for the first time.`);

  const val1 = await p1;
  console.log(`[${new Date().toISOString()}] First await completed. Promise
```

```
has been resolved.`);
  console.log(`[${new Date().toISOString()}] 🔁 Logging the first resolved
value:`);
  console.log(`[${new Date().toISOString()}] promise 1 time (${Date.now() -
startTime}ms) →`, val1);

  console.log(`[${new Date().toISOString()}] 🧐 Awaiting the second Promise
(\`p2\`) — this should have a shorter delay.`);
  const val2 = await p2;
  console.log(`[${new Date().toISOString()}] ☑️ Second await completed. Promise
has been resolved.`);
  console.log(`[${new Date().toISOString()}] 🔁 Logging the second resolved
value:`);
  console.log(`[${new Date().toISOString()}] promise 2 time (${Date.now() -
startTime}ms) →`, val2);
}

handlePromise();
```

🖨️ Sample Output

```
[2025-05-05T18:42:22.914Z] 🚀 Starting async function: handlePromise()
[2025-05-05T18:42:22.920Z] 👉 Logging immediately before awaiting the Promise for
the first time.
[2025-05-05T18:42:32.918Z] ☑️ First await completed. Promise has been resolved.
[2025-05-05T18:42:32.918Z] 🔁 Logging the first resolved value:
[2025-05-05T18:42:32.919Z] promise 1 time (10005ms) → Resolved Value!!
[2025-05-05T18:42:32.919Z] 🧐 Awaiting the second Promise (`p2`) — this should
have a shorter delay.
[2025-05-05T18:42:32.919Z] ☑️ Second await completed. Promise has been resolved.
[2025-05-05T18:42:32.919Z] 🔁 Logging the second resolved value:
[2025-05-05T18:42:32.919Z] promise 2 time (10005ms) → Resolved Value!!
```

📊 Output Timeline

⏱️ Time	📍 Event
0ms	Function starts, both `p1` and `p2` already running
~10,000ms	`p1` resolves
~10,000ms	`p2` already resolved too
☑️ Total Time	~10s (parallel execution)

❌ Case 2: Promises Created *Inside* the Async Function (🚗 Sequential Execution)

📜 Code

```
async function handlePromiseSequentially() {
  const startTime = Date.now();

  console.log(`[${new Date().toISOString()}] 🚀 Starting async function:
handlePromiseSequentially()`);

  console.log(`[${new Date().toISOString()}] ⏳ Creating and awaiting p1 (10s
delay)`);
  const val1 = await new Promise((resolve) => {
    setTimeout(() => resolve("Resolved Value!!"), 10000);
  });

  console.log(`[${new Date().toISOString()}] ☑️ p1 resolved`);
  console.log(`[${new Date().toISOString()}] 🪄 value →`, val1);
  console.log(`[${new Date().toISOString()}] Time so far: ${Date.now() -
startTime}ms`);

  console.log(`[${new Date().toISOString()}] ⏳ Creating and awaiting p2 (5s
delay)`);
  const val2 = await new Promise((resolve) => {
    setTimeout(() => resolve("Resolved Value!!"), 5000);
  });

  console.log(`[${new Date().toISOString()}] ☑️ p2 resolved`);
  console.log(`[${new Date().toISOString()}] 🪄 value →`, val2);
  console.log(`[${new Date().toISOString()}] 🧮 Total time: ${Date.now() -
startTime}ms`);
}

handlePromiseSequentially();
```

🖨️ Sample Output

```
[2025-05-05T18:50:10.000Z] 🚀 Starting async function:
handlePromiseSequentially()
[2025-05-05T18:50:10.001Z] ⏳ Creating and awaiting p1 (10s delay)
[2025-05-05T18:50:20.002Z] ☑️ p1 resolved
[2025-05-05T18:50:20.002Z] 🪄 value → Resolved Value!!
[2025-05-05T18:50:20.002Z] Time so far: 10001ms
[2025-05-05T18:50:20.002Z] ⏳ Creating and awaiting p2 (5s delay)
[2025-05-05T18:50:25.003Z] ☑️ p2 resolved
[2025-05-05T18:50:25.003Z] 🪄 value → Resolved Value!!
[2025-05-05T18:50:25.003Z] 🧮 Total time: 15003ms
```

📊 Output Timeline

⏱️ Time	📍 Event
0ms	Function starts
0ms	`p1` created and awaited (10s)
~10,000ms	`p1` resolved
~10,000ms	`p2` created and awaited (5s)
~15,000ms	`p2` resolved
☑ Total Time	~15s (sequential execution)

🧠 Behavior Summary Table

🔍 Criteria	☑ Case 1 (Parallel)	❌ Case 2 (Sequential)
When Promises Start	Immediately when declared	Only when encountered in code
Execution	Concurrent	One after the other
`await` impact	Waits on already running task	Triggers then waits
Total Time	~10s	~15s
Real-world Analogy	Cooking rice + curry together	Cook rice, then cook curry

---

💡 Real-World Analogy

> 🍛 You're making dinner.
>
> - **Parallel (Case 1)**: Start cooking **rice** and **curry** at the same time. Curry finishes in 5 mins, rice in 10 — total: **10 mins**.
> - **Sequential (Case 2)**: Cook **rice** (10 mins), then **start** cooking curry (5 mins) — total: **15 mins**.

📌 Key Takeaways

☑ Create promises **early** if you want **parallel execution**. ❌ Avoid creating promises **inside** `await` **chains** unless **order matters**.

> 📊 Optimizing `await` behavior is crucial for performance in async-heavy JavaScript apps.

Here's a **comprehensive breakdown** of your provided code with:

- ☑ Code explanation (line-by-line)
- 🕰️ Execution behavior summary
- 📊 Timestamped Output Table (based on real-time simulation)
- 🧠 Key Learning

☑ Code Breakdown & Explanation

```
// 🧱 Step 1: Define two promises
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 5000); // resolves in 5 seconds
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // resolves in 10 seconds
});

// 🚀 Step 2: Async function execution starts here
async function handlePromise() {
  console.log("🚀 Starting async function: handlePromise()");
  console.log("👉 Logging immediately before awaiting the Promise for the first
time.");

  // ⏳ Awaiting p1 (5 sec delay)
  const val1 = await p1;
  console.log("☑ First await completed. Promise has been resolved.");
  console.log("🔁 Logging the first resolved value:");
  console.log("promise 1 time →", val1);

  // ⏳ Awaiting p2 (but its 10-sec timer already started earlier!)
  console.log("🤔 Awaiting the second Promise (`p2`) — this should have a shorter
delay.");
  const val2 = await p2;
  console.log("☑ Second await completed. Promise has been resolved.");
  console.log("🔁 Logging the second resolved value:");
  console.log("promise 2 time →", val2);
}

handlePromise();
```

☑ Reversed Reorder Promise Version

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 5000); // 5-second delay
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
```

```
  }, 10000); // 10-second delay
});

async function handlePromise() {
  const startTime = Date.now(); // Capture start time

  console.log(`[${new Date().toISOString()}] 🚀 Starting async function:
handlePromise()`);
  console.log(`[+${Date.now() - startTime}ms] 👉 Logging immediately before
awaiting the Promise for the first time.`);

  const val1 = await p1;
  console.log(`[+${Date.now() - startTime}ms] ☑️ First await completed. Promise
has been resolved.`);
  console.log(`[+${Date.now() - startTime}ms] 🔁 Logging the first resolved
value:`);
  console.log(`[+${Date.now() - startTime}ms] promise 1 time →`, val1);

  console.log(`[+${Date.now() - startTime}ms] 🤔 Awaiting the second Promise (p2)
— this should have a shorter delay.`);
  const val2 = await p2;
  console.log(`[+${Date.now() - startTime}ms] ☑️ Second await completed. Promise
has been resolved.`);
  console.log(`[+${Date.now() - startTime}ms] 🔁 Logging the second resolved
value:`);
  console.log(`[+${Date.now() - startTime}ms] promise 2 time →`, val2);
}

handlePromise();
```

📊 Simulated Output Table (with Elapsed Time + UTC Timestamp)

⏱️ Elapsed Time	🕒 Timestamp (UTC)	📝 Log Output
+0ms	2025-05-05T19:00:00.000Z	🚀 Starting async function: handlePromise()
+2ms	2025-05-05T19:00:00.002Z	👉 Logging immediately before awaiting the Promise for the first time.
+5002ms	2025-05-05T19:00:05.002Z	☑️ First await completed. Promise has been resolved.
+5003ms	2025-05-05T19:00:05.003Z	🔁 Logging the first resolved value:
+5003ms	2025-05-05T19:00:05.003Z	promise 1 time → Resolved Value!!

⏱️ Elapsed Time	🕒 Timestamp (UTC)	📝 Log Output
+5004ms	2025-05-05T19:00:05.004Z	🤔 Awaiting the second Promise (p2) — this should have a shorter delay.
+10004ms	2025-05-05T19:00:10.004Z	☑️ Second await completed. Promise has been resolved.
+10005ms	2025-05-05T19:00:10.005Z	🔁 Logging the second resolved value:
+10005ms	2025-05-05T19:00:10.005Z	promise 2 time → Resolved Value!!

🤔 Behavior Summary

Aspect	Details
🟣 When timers started	Immediately when `p1` and `p2` were declared
🔄 Await blocks code	`await p1` paused execution for 5 seconds, then `await p2` ran
⏳ p2 already ticking	So when we awaited `p2`, **5 seconds** had already passed
🕰️ Total function time	Only **10 seconds**, **not 15 seconds**
🔍 What if p2 declared later?	If declared *after* p1 resolves, total time would be **15 seconds**

☑️ Takeaway (in plain words)

> `await` **pauses** your async function, but it doesn't delay the start of a Promise's timer. Declaring the Promise triggers the timer **right away** — regardless of when you `await` it.

Execution Flow

The execution flow of the provided code demonstrates how asynchronous JavaScript handles promises and `await` statements. Here's a step-by-step breakdown:

1. **Promise Initialization**:
    - Both `p1` and `p2` are created immediately when the script runs.
    - Their timers (`setTimeout`) start counting down concurrently.
2. **Calling the Async Function**:
    - The `handlePromise()` function is invoked.
    - Initial `console.log` statements inside the function execute synchronously.
3. **First `await` (`p1`)**:

- The function pauses at `await p1` and waits for `p1` to resolve.
- Meanwhile, the timer for `p2` continues running in the background.

4. **Resuming After `p1`**:
   - After 10 seconds, `p1` resolves, and the function resumes execution.
   - The resolved value of `p1` is logged.
5. **Second `await` (`p2`)**:
   - The function encounters `await p2`.
   - Since `p2` started its timer earlier and resolves after 5 seconds, it has already resolved by this point.
   - The function resumes immediately without additional delay.
6. **Completion**:
   - The resolved value of `p2` is logged.
   - The function completes execution.

Key Points:

- Promises start their timers when they are created, not when they are awaited.
- `await` pauses the function execution but does not block the entire script.
- If multiple promises are created upfront, their timers run concurrently, optimizing execution time.

☑ Code: Promise Timing and Async/Await Behavior

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 10000); // 10-second delay
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Resolved Value!!");
  }, 5000); // 5-second delay
});

async function handlePromise() {
  console.log("🚀 Starting async function: handlePromise()");
  console.log("👉 Logging immediately before awaiting the Promise for the first
time.");

  const val1 = await p1;
  console.log("☑ First await completed. Promise has been resolved.");
  console.log("🔁 Logging the first resolved value:");
  console.log("promise 1 time →", val1);

  console.log("🤔 Awaiting the second Promise (`p2`) — this should have a shorter
```

```
delay.");
  const val2 = await p2;
  console.log("✅ Second await completed. Promise has been resolved.");
  console.log("🔁 Logging the second resolved value:");
  console.log("promise 2 time →", val2);
}

handlePromise();
console.log("👋 This log appears immediately after calling handlePromise() but
before the first await is resolved.");
```

🧠 Updated Execution Flow with Call Stack and Event Loop

⏱️ Timeline Breakdown:

Time	Event
0ms	`p1` and `p2` are created with `setTimeout()` running in background.
1ms	`handlePromise()` is called.
2ms	`console.log("🚀 Starting...")` and logs inside async function run.
3ms	`console.log("👉 Logging immediately...")` runs.
4ms	First `await p1` suspends function execution.
5ms	`console.log("👋 This log appears...")` runs immediately after `handlePromise()` call.
10000ms	`p1` resolves after 10s. Execution resumes after first `await`.
10001ms	Logs the first resolved value from `p1`.
10002ms	Now `await p2` starts.
15000ms	`p2` resolves after 5s, finishing by the time `await p2` is encountered.
15001ms	Logs the second resolved value from `p2`.

📦 Call Stack & Event Loop Summary :

1. **Call Stack**: Starts `handlePromise()` → hits `await p1` → suspends function execution.
2. **Timer Phase** (in Web APIs): `setTimeout`s for both promises start counting down concurrently.
3. **Callback Queue**: After 10 seconds, `p1` resolves and goes into the microtask queue.
4. **Re-entry**: Async function resumes with `val1`, then hits `await p2`.
5. Since `p2` started 5 seconds ago, it resolves **immediately** after the first `await` completes.
6. Logs all values and exits the function.

📤 Output:

```
🚀 Starting async function: handlePromise()
👉 Logging immediately before awaiting the Promise for the first time.
🤚 This log appears immediately after calling handlePromise() but before the
first await is resolved.
☑️ First await completed. Promise has been resolved.
🔁 Logging the first resolved value:
promise 1 time → Resolved Value!!
🥱 Awaiting the second Promise (`p2`) — this should have a shorter delay.
☑️ Second await completed. Promise has been resolved.
🔁 Logging the second resolved value:
promise 2 time → Resolved Value!!
```

🔎 Key Learning Points:

- **Timers start when promises are created**: `p1` starts the 10-second timer as soon as it's created, and `p2` starts the 5-second timer at the same time.
- `await` **does not delay promise execution**: The promises are already running in the background when the function hits the `await` statements.
- **Async behavior**: `await` suspends the async function but doesn't block the entire execution. Other code continues, like logging `"🤚 This log appears immediately..."`.

❓ Does `await` always return a Promise?

☑️ `await` **always returns the resolved value of a Promise**, **but it can be used with non-Promises too**.
Here's what happens in both cases:

◇ Case 1: `await` with a **Promise**

```js
const p = new Promise((resolve) => setTimeout(() => resolve("Resolved!"), 1000));

const result = await p;  // Waits for 1 second
console.log(result);     // Output: "Resolved!"
```

- `await` pauses execution until the Promise resolves.
- It returns the **resolved value** of the Promise.

◇ Case 2: `await` with a **non-Promise (e.g., a string or number)**

```js
const result = await 42;
console.log(result); // Output: 42
```

- `await 42` is treated like `await Promise.resolve(42)`.

- So even if it's not a Promise, `await` **wraps it into a resolved Promise internally**.

🟤 Rejection case:

```
await Promise.reject("Error!"); // throws immediately
```

- This will throw and should be wrapped in `try...catch`.

☑️ Summary:

What You Await	How `await` Behaves
A `Promise`	Waits for it, returns resolved value or throws.
A non-Promise value	Converts to `Promise.resolve(value)`.
A rejected Promise	Throws the rejection error.

🔍 How `async/await` is Syntactic Sugar over `.then()` and `.catch()`

☑️ The Core Idea:

`async/await` doesn't add new functionality to JavaScript — it's **just cleaner syntax** built on top of Promises.

> Think of `async/await` as a more readable and structured way to **write promise chains**.

🔁 Example using `.then()` and `.catch()`

```
function getData() {
  return fetch('https://api.github.com/users/dpvasani')
    .then((response) => response.json())
    .then((data) => {
      console.log("User Data:", data);
    })
    .catch((error) => {
      console.error("❌ Error fetching data:", error);
    });
}
```

🧠 Same logic using `async/await`

```
async function getData() {
  try {
```

```
    const response = await fetch('https://api.github.com/users/dpvasani');
    const data = await response.json();
    console.log("User Data:", data);
  } catch (error) {
    console.error("❌ Error fetching data:", error);
  }
}
```

🧬 Under the Hood:

- `await` **waits** for a Promise to resolve and **extracts the resolved value**.
- It's equivalent to chaining `.then()` to get the resolved value.
- `try...catch` is used instead of `.catch()` for cleaner error handling.

📓 Table Summary:

Feature	`.then().catch()`	`async/await`
Syntax Style	Functional chaining	Imperative (step-by-step)
Error Handling	`.catch()`	`try...catch`
Readability	Harder with multiple chains	Easier and cleaner, looks synchronous
Underlying Mechanism	Uses Promises	Uses Promises

⚙️ Internally:

```
async function foo() {
  const data = await fetch(...);
}
```

Is internally equivalent to:

```
function foo() {
  return fetch(...).then(data => {
    return data;
  });
}
```

☑️ TL;DR:

- `async/await` **does not replace** Promises — it **uses them**.

- It's **syntactic sugar** for chaining `.then()` and `.catch()` — more readable and easier to follow, especially with multiple async operations.