📚 Async Await in JavaScript – *Summary and Key Notes*

1. Introduction

- **Async/Await** is a crucial JavaScript concept every developer must know.
- It's **used daily** in coding and heavily asked in **interviews**.
- This session covers **what async and await are**, **how they work internally**, **error handling**, **real-life examples**, and **interview tips**.

---

2. What is `async`?

- `async` is a **keyword** used before a function to **make it an async function**.
- An **async function always returns a Promise**, *either*:
  - **Directly** if it returns a Promise.
  - **Automatically wrapped** inside a Promise if it returns a normal value (string, number, boolean, etc.).

Example:

javascript
CopyEdit
```
async function getData() {
  return "Namaste";  // Automatically wrapped in
Promise.resolve("Namaste")
}
```

- `getData()` returns a **Promise**.

3. What is `await`?

- `await` is a **keyword** that can **only be used inside an async function**.
- It is placed **in front of a Promise** to pause the function execution **until the Promise resolves**.

Example:

javascript
CopyEdit

```javascript
async function handlePromise() {
  const result = await somePromise;
  console.log(result);
}
```

- **Execution Suspends** at the `await` line until `somePromise` resolves.

---

4. Difference Between Then-Catch vs Async-Await

Then-Catch	Async-Await
JavaScript **doesn't wait**, moves to the next line.	**Pauses execution** at `await` until promise resolves.
Callback-based chaining.	**Cleaner, more readable** syntax.
Time management is manual.	Automatic wait, making async code look like sync code.

Important Note:

- **JS Engine is not really "waiting"** — it suspends the function execution and continues handling other events.
 (Call stack remains free!)

---

5. Deep Dive: How Async/Await Actually Works Internally

- When an `await` is encountered:
  - Execution **suspends** at that line.
  - **Call stack** is emptied to handle other operations (no blocking).
  - Once the Promise resolves, the function **resumes** from where it was suspended.

👉 *Even though it feels synchronous, under the hood it's still asynchronous.*

---

6. Real-World Example: Fetch API with Async/Await

javascript
CopyEdit
```
async function fetchUser() {
  try {
    const response = await
fetch('https://api.github.com/users/akshaymarch7');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data', error);
  }
}
fetchUser();
```

- **fetch()** returns a Promise.
- **response.json()** also returns a Promise.

---

7. Error Handling in Async/Await

- **Use try-catch block** inside async functions to catch errors.
- Alternatively, **use `.catch()`** outside by handling the returned promise.

Try-Catch Example:

javascript
CopyEdit
```
async function fetchData() {
  try {
    const data = await fetch('invalid-url');
  } catch (error) {
    console.error(error);
  }
}
```

---

8. Interview Preparation Tips

When asked about async-await:

- Clearly explain:
  - `async` always returns a Promise.
  - `await` is used **only inside async functions**.
  - It helps make asynchronous code **easier to read and maintain**.
- **Optional Deep Dive**: Explain how function execution **suspends** internally, but **JS engine never blocks** the call stack.

---

9. Async-Await vs Then-Catch: Which is Better?

- **Async/Await** is **preferred**:

    - Cleaner and easier to read.
    - No messy `.then().then().catch()`.
- But **under the hood**, async-await **is just syntactic sugar** over Promises and `.then()`.

🎯 Final Takeaways

- Async functions always return a Promise.
- Await pauses the async function without blocking the call stack.
- Prefer using async/await over traditional Promise chains for cleaner code.
- Understand the **internal execution suspension model** to explain clearly in interviews.

📝 Async Await Quick Interview Notes

Basics

1. `async` makes a function always return a **Promise**.
2. If a function returns a value, `async` **wraps it into a Promise** automatically.
3. `await` can **only** be used inside an `async` function.

Working

4. `await` **pauses** function execution until the Promise resolves.

5. During `await`, **JS Engine does not block** the call stack — it **suspends** the function execution.

6. After Promise resolves, function **resumes from where it was paused**.

Comparison

7. **Promise.then**: No waiting, code continues execution.

8. **Async/Await**: Looks synchronous, but internally uses Promises.

9. **Async/Await is syntactic sugar** over `.then()/.catch()`.

Real World Usage

10. Use `await fetch()` to make API calls cleanly.

11. After `fetch`, use `await response.json()` to parse the result.

12. Always handle errors using **try-catch** inside async functions.

Error Handling

13. Place `await` calls inside **try-catch** block to catch errors.

14. Alternatively, `.catch()` can be used outside async function (handlePromise().catch()).

Behind the Scenes

15. JavaScript **does not freeze** while awaiting.

16. **Call stack is free** while waiting for async operations.
17. **Execution suspends**, not JavaScript itself.

Bonus (Interview Extra Points)

18. If you `await` multiple Promises, **they are awaited sequentially** unless you run them concurrently with `Promise.all()`.
19. Async/Await makes code **more readable**, especially in **complex promise chains**.
20. **Best practice**: Use Async/Await for new code unless you have a specific reason to use `.then()` chaining.

⚡ Pro Tip for Interviews

When asked:

- Explain with a **basic async/await function**.
- Mention **internal suspension**, **non-blocking nature**, and **cleaner syntax**.
- Optionally explain how **fetch + await** works.