📝 JavaScript Detailed Revision Notes

1. Asynchronous Programming in JavaScript

What is Asynchronous Code?

- Allows long-running tasks (like API calls or timers) to run **without blocking** the main thread.
- JS can continue executing other code while waiting for the async task to finish.

Features

Feature	Description
Non-Blocking Execution	Main thread doesn't freeze; async tasks run in the background.
Clean & Concise Code	Promises and async/await avoid nested callbacks ("Callback Hell").
Better Error Handling	Centralized via `.catch()` or `try...catch`.
Easier Debugging	Stack traces are cleaner with Promise chains or async/await.
Improved Performance	Single-threaded JS can handle many concurrent I/O efficiently.

2. Promises in JavaScript

Definition

- A **Promise** represents a **future value** from an asynchronous operation.
- **States of a Promise**

State	Description

Pending	Initial state; async operation ongoing
Fulfilled/Resolved	Operation succeeded; `.then()` handles it
Rejected	Operation failed; `.catch()` handles it

-

Creating a Promise

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Task completed successfully");
        // reject(new Error("Something went wrong"));
    }, 2000);
});
console.log("This runs first!");
```

Handling a Promise

```
myPromise
    .then(result => console.log("Resolved:", result))
    .catch(error => console.error("Rejected:", error))
    .finally(() => console.log("Promise settled"));
```

---

3. Promise Chaining

- **Sequential async tasks** can be linked using `.then()`.

```
const promise1 = new Promise((resolve) => {
    setTimeout(() => resolve("Result 1"), 1000);
});

promise1
    .then(res1 => {
        console.log(res1);
        return new Promise(resolve => setTimeout(() => resolve("Result 2"), 1500));
    })
    .then(res2 => {
        console.log(res2);
        return new Promise(resolve => setTimeout(() => resolve("Result 3"), 1000));
    })
    .then(res3 => console.log(res3))
    .catch(err => console.error(err));
```

Key point: `.catch()` handles errors from **any step** in the chain.

---

4. Async/Await (Modern Syntax)

- Cleaner way to handle Promises as **if synchronous code**.

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

async function runTasks() {
    try {
        const res1 = await wait(1000).then(() => "Task 1 complete");
        console.log(res1);

        const res2 = await wait(1500).then(() => "Task 2 complete");
        console.log(res2);

        const res3 = await wait(1000).then(() => "Task 3 complete");
        console.log(res3);

        console.log("All tasks done!");
    } catch (err) {
        console.error(err);
    }
}
runTasks();
```

Advantages:

- Reads top-to-bottom like synchronous code.
- Simplified error handling.
- Avoids nested `.then()` callbacks.

Async/Await Examples

Sequential Execution

```
async function weatherSequential() {
    let delhi = await new Promise(res => setTimeout(() => res("Delhi
hot"), 1000));
    let hyd = await new Promise(res => setTimeout(() => res("Hyderabad
cool"), 2000));
    let ngp = await new Promise(res => setTimeout(() => res("Nagpur
moderate"), 3000));
    return [delhi, hyd, ngp];
}
```

Parallel Execution using `Promise.all`

```
async function weatherParallel() {
    let delhi = new Promise(res => setTimeout(() => res("Delhi hot"),
1000));
    let hyd = new Promise(res => setTimeout(() => res("Hyderabad
cool"), 2000));
    let ngp = new Promise(res => setTimeout(() => res("Nagpur
moderate"), 3000));

    let results = await Promise.all([delhi, hyd, ngp]);
    return results; // resolves after max time (3s)
}
```

Without await

```
async function weatherNoAwait() {
    let delhi = new Promise(res => setTimeout(() => res("Delhi hot"), 
1000));
    let hyd = new Promise(res => setTimeout(() => res("Hyd cool"), 
2000));
    return [delhi, hyd]; // returns array of Promises
}
```

---

5. Fetch API

- Used to **send requests and get data** asynchronously.

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.error(err));
```

With async/await

```
async function getTodo() {
    try {
        const res = await 
fetch('https://jsonplaceholder.typicode.com/todos/1');
        const data = await res.json();
        console.log(data);
    } catch (err) {
        console.error(err);
    }
}
```

Fetch options

```
fetch(url, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: "Aryan" })
});
```

---

6. Objects & Classes

Objects

```
let person = {
    name: "Aryan",
    age: 22,
    greet: function() { console.log(`Hi, I'm ${this.name}`); }
};
person.greet();
```

Classes

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() { console.log(`Hello, I'm ${this.name}`); }
}

let p1 = new Person("Aryan", 22);
p1.greet();
```

---

7. Closures & Scope

- A **closure** is a function that **remembers its outer scope** even when executed elsewhere.

Example 1: Access parent variable

```
function init() {
    let name = "Mozilla";
    function displayName() {
        console.log(name + " DISPLAYED"); // accesses parent variable
    }
    displayName();
}
init();
```

Example 2: Inner variable overwrites parent

```
function init2() {
    let name = "Mozilla";
    function displayName() {
        let name = "Babar";
        console.log(name + " DISPLAYED"); // inner variable used
    }
    displayName();
}
init2();
```

Example 3: Closure returning a function

```
function makeCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    }
}

let counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Key Takeaways

- Inner function has access to **variables in parent scope**.
- Even after parent finishes, inner function can **remember variables** (closure).

---

8. Scope Types in JS

Scope	Description
Global	Declared outside functions; accessible everywhere
Function / Local	Declared inside function; accessible only inside
Block (let/const)	Declared in `{}` block; block-scoped
Closure	Inner function accessing outer function variables

✅ Quick Recap for Revision

- **Async JS:** Promises, `.then()/.catch()`, chaining, async/await
- **Promise utilities:** `Promise.all`, `Promise.race`, `Promise.allSettled`
- **Fetch API:** GET, POST, async/await usage
- **JSON:** parsing and stringifying
- **Objects & Classes:** properties, methods, constructors
- **Closures & Scopes:** inner functions remembering outer variables, block/function/global scopes