JavaScript Asynchronous

1. What is Asynchronous Programming?

Synchronous code: runs **line by line**. Each line waits for the previous to finish.

```
 console.log("Start");
console.log("End");
```
 Output:

```
 Start
End
```

> **Asynchronous code:** allows some operations to **run in the background**, so other code can continue executing **without waiting**.
>
> Example: Fetching data from the internet.

2. Why Do We Need Asynchronous Programming?

- Some tasks take time: API requests, reading files, timers.
- Blocking the main thread would make the app **slow or unresponsive**.

Example:

```
console.log("Start");

setTimeout(() => {
  console.log("Waited 2 seconds");
}, 2000);

console.log("End");
```

Output:

```
Start
End
Waited 2 seconds
```

Notice how `End` prints **before** the `setTimeout` callback — that's asynchronous behavior.

3. Callbacks

- **Definition:** A callback is a **function passed as an argument** to another function, which is called **later**.

Example:

```
function greet(name, callback) {
  console.log("Hello " + name);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Celia", sayGoodbye);
```

Output:

```
Hello Celia
Goodbye!
```

Problem with Callbacks

- Can lead to **"callback hell"** if nested too deeply.

```
doSomething(() => {
  doSomethingElse(() => {
    doAnotherThing(() => {
      console.log("Done!");
    });
  });
});
```

4. Promises

- **Definition:** A Promise is an object representing a **future value** (success or failure).
- Promises have **3 states**:
    1. **Pending** – initial state
    2. **Fulfilled** – operation completed successfully
    3. **Rejected** – operation failed

Creating a Promise:

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;

  if (success) {
    resolve("Operation successful!");
  } else {
    reject("Operation failed!");
  }
});
```

Using a Promise:

```
myPromise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.log(error);
  });
```

5. Chaining Promises

- Promises allow **cleaner chaining** instead of nested callbacks.

```
new Promise((resolve, reject) => {
  resolve(5);
```

```
})
.then(result => {
  console.log(result); // 5
  return result * 2;
})
.then(result => {
  console.log(result); // 10
})
.catch(error => {
  console.error(error);
});
```

6. Async/Await

- **Async/await** is syntax sugar on top of Promises.
- Makes asynchronous code **look synchronous**, easier to read.

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

async function asyncFunction() {
  console.log("Start waiting...");
  await wait(2000);
  console.log("2 seconds passed!");
}

asyncFunction();
```

Key points:

- `async` function always **returns a Promise**.
- `await` can **pause execution** inside an async function until the Promise resolves.

7. Fetch API Example (Real Use Case)

```
// Using Promises
```

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

// Using Async/Await
async function fetchTodo() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchTodo();
```