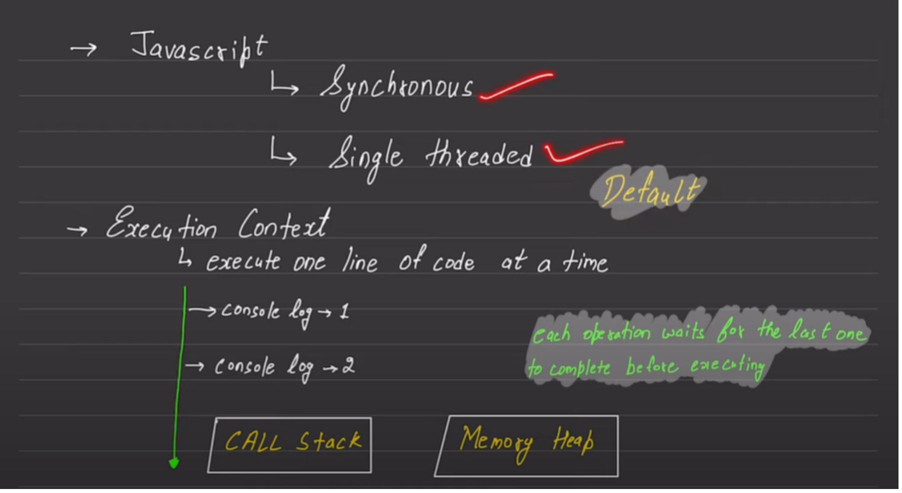
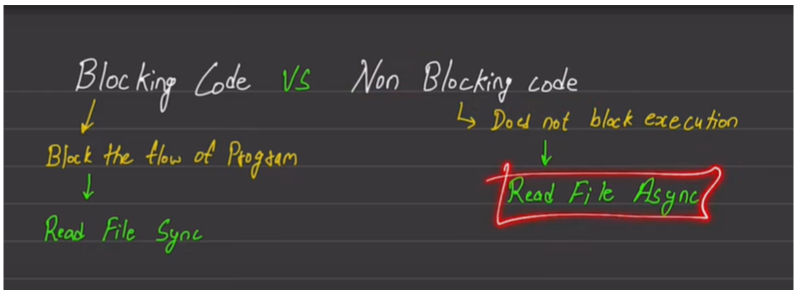
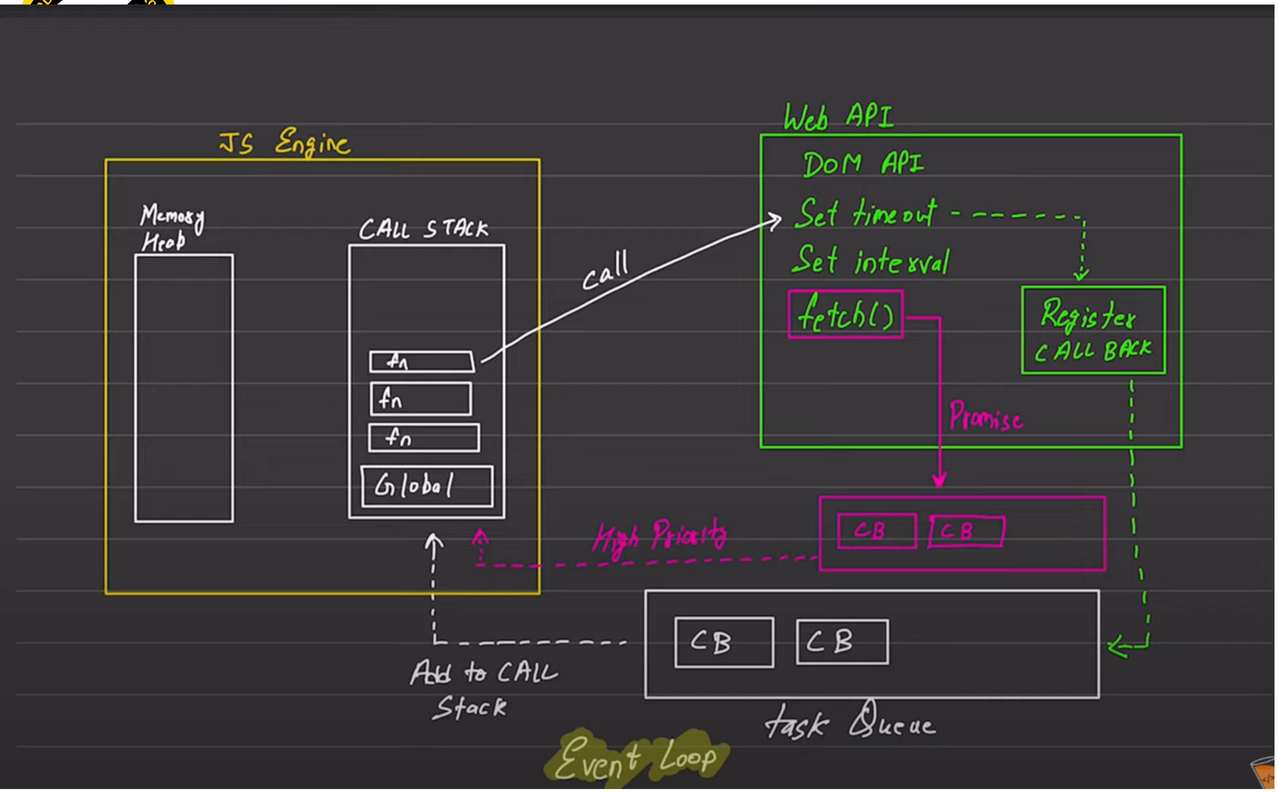

→ Javascript
↳ Synchronous
↳ Single threaded
Default
→ Execution Context
↳ execute one line of code at a time
→ console log → 1
→ console log → 2
each operation waits for the last one to complete before executing
CALL Stack
Memory Heap


Blocking Code VS Non Blocking code
Block the flow of Program
Read File Sync
↳ Does not block execution
Read File Async



JS Engine
Memory Heap
CALL STACK
fn
fn
fn
Global
call
Web API
DOM API
Set timeout
Set interval
fetch()
Register CALL BACK
Promise
CB
CB
High Priority
CB
CB
Add to CALL Stack
task Queue
Event Loop

Key Concepts

1. **Memory Management**:
   - **Stack**: Used for managing function calls. It follows the **LIFO** (Last In, First Out) principle.
   - **Heap**: Used for storing objects, arrays, and other dynamically allocated data.
2. **Execution Context**:
   - **Global Context**: Created when the program starts. It contains global variables and functions.
   - **Function Context**: Created whenever a function is called. It includes:
     - Local variables.
     - `this` binding.
     - Reference to the outer environm... (if needed).
3. **Call Stack**:
   - The stack where function calls are tracked. Each function call adds a "stack frame" (a record of that function's execution).
4. **Task Handling**:
   - **Microtask Queue**: For high-priority asynchronous tasks (e.g., resolved Promises).
   - **Callback/Task Queue**: For lower-priority asynchronous tasks (e.g., `setTimeout`).
5. **Event Loop**:
   - A loop that continuously checks if there's any task in the **call stack**, **microtask queue**, or **callback queue** to execute.

CODE EXECUTION AND CREATION:

2. Steps in Code Execution:

a. Creation Phase:

- When a script is loaded, the JavaScript engine first creates the **Global Execution Context (GEC)**.
- Memory is allocated for variables and functions:
  - Variables are initialized with `undefined`.
  - Functions are stored in memory as a whole.

b. Execution Phase:

- Code is executed line by line.
- Values are assigned to variables, and functions are invoked.
- When a function is called, a **Function Execution Context (FEC)** is created and pushed onto the call stack.

4. Asynchronous Operations:

JavaScript is single-threaded but handles asynchronous tasks using:

1. **Web APIs**:
   - Browsers provide APIs like `setTimeout`, `fetch`, etc., which operate outside the main thread.
2. **Event Loop**:
   - Ensures that the call stack is empty before pushing callback functions from the **Task Queue** or **Microtask Queue**.
   - **Microtask Queue** (e.g., promises) is prioritized over the **Task Queue** (e.g., `setTimeout`).

1. Call Stack (Execution Stack)

The **call stack** is a stack data structure that stores functions that need to be executed. It follows the Last-In-First-Out (LIFO) principle.

- When a function is called, it is added to the **top** of the call stack.
- When the function finishes execution, it is **popped off** the stack.

2. Callback Queue

The **callback queue** (or task queue) is where **asynchronous callbacks** (like event handlers, `setTimeout`, `setInterval`, or Promises) are placed once they are ready to be executed.

- Callbacks are pushed into the queue when their corresponding events (like user input or timers) are triggered.
- The event loop processes the callback queue and moves callbacks to the call stack when it is empty.

3. Event Loop

The **event loop** is a continuous process that checks whether the call stack is empty. It then moves tasks from the callback queue to the call stack for execution. This process enables JavaScript to handle asynchronous operations without blocking the main thread.

How the Event Loop Works:

1. **Call Stack**: The event loop starts by checking if there are any functions in the call stack.
2. **Callback Queue**: If the call stack is empty, the event loop moves a callback from the callback queue to the call stack.
3. **Execution**: The callback from the queue is then executed, and the event loop continues.

4. Microtask Queue

The **microtask queue** is used for promises and other microtasks (e.g., `Promise.then()` or `async/await`). Microtasks have a higher priority than regular tasks in the callback queue.

- When a promise is resolved, its `.then()` callback is placed in the **microtask queue**.
- The event loop first checks and executes all microtasks before moving on to the regular callback queue.

DIFFERECE IN WORKING OF MICROTASK QUEUE(HIGH PRIORITY Q) ANDCALL BACK

Click to BUY NOW! PDF-XChange Editor www.pdf-xchange.com

1. Microtask Queue

- **Purpose:** The microtask queue is primarily used for tasks like **Promises**, `Promise.then()`, and `async/await`. These are often referred to as **microtasks**.
- **Execution Priority:** Tasks in the microtask queue have **higher priority** than those in the callback queue.
- **Processing Timing:**
  - After every **synchronous task** has been completed, the event loop first checks the microtask queue.
  - The event loop will execute **all** microtasks in the microtask queue before moving on to the callback queue.
  - This ensures that any promise resolution or rejection is handled as soon as possible, even before the browser has a chance to render or execute timer-based tasks.
- **Examples:**
  - `Promise.resolve().then(...)`
  - `Promise.catch(...)`
  - `async/await` operations

2. Callback Queue

- **Purpose:** The callback queue is used for tasks like `setTimeout`, `setInterval`, `event listeners`, and other asynchronous tasks (excluding promises).
- **Execution Priority:** Tasks in the callback queue have **lower priority** compared to the microtask queue.
- **Processing Timing:**
  - The event loop will process the callback queue only after the microtask queue has been emptied.
  - Even if there are multiple tasks in the callback queue, only one task will be picked and executed at a time after all microtasks are done.
- **Examples:**
  - `setTimeout()`
  - `setInterval()`
  - Event listeners (like `click`, `keydown`, etc.)

call bask ya task q tb hi chlta hai jb call stack ne saare synchronus task complete kar liye ho ya call stack khali ho
sath hi micro task ya high priority me promise ke .then . catch hote hai j timely check krte rhte h ki unke promise , ya aysnc tasks , fetch api complete hue ya nhi kyunki fetch api to agr uska promise reolsve hona mtlb wo complete kar chuki ho apna time yani agr timeout tha to wo exexution liye ready rhega ab
even chahe call back queu me bhut saari padi ho agr microtask wali ki koi call back hai jo complte ho chuki hai to wo call back q se phle execute ho jati hai

agr call back me sari complete ho chuki hai to wo use let hai
jo usme phle aayi thi kyunki queue hai pr complte honi chahiye
call back queue

Yes, both the **microtask queue** and the **callback queue** only execute when the **call stack is empty**, but with an important difference in their execution order and priority.

1. Microtask Queue

- **Execution Timing**: The **microtask queue** is processed immediately after the **call stack** is empty, but **before** the event loop moves on to process the **callback queue**.
- This ensures that microtasks (like promise callbacks) are executed as soon as possible after the synchronous code completes, but before any I/O events or timers (like `setTimeout`) are handled.

2. Callback Queue

- **Execution Timing**: The **callback queue** is processed only after the **microtask queue** is emptied. The event loop first processes all microtasks, then moves on to the callback queue when the call stack is empty.

Event Loop Flow in Detail:

1. **Call Stack**: The event loop continuously checks if the call stack is empty.
   - If the stack is not empty, it continues processing the synchronous code.
   - If the stack is empty, it then moves to the next step.
2. **Microtask Queue**: After the call stack is empty, the event loop will **first** process all the tasks in the **microtask queue** (if any) before proceeding to the callback queue.
   - This guarantees that promises and `async/await` handlers are executed as soon as possible.
3. **Callback Queue**: Only after the microtask queue is empty does the event loop start processing tasks in the **callback queue**, such as those from `setTimeout`, `setInterval`, or event listeners.

The **microtask queue** contains the **callback functions** of promises, specifically the ones associated with `.then()`, `.catch()`, and `.finally()` methods. These are the callback functions that are executed when a promise is either resolved or rejected.

Code:

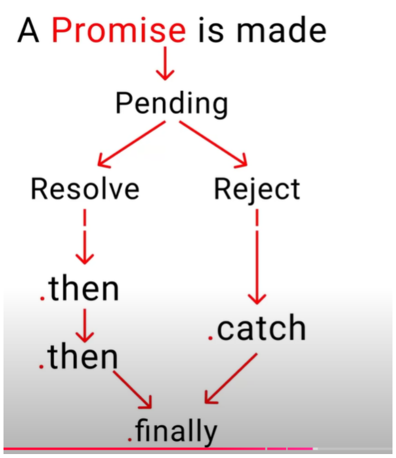
```javascript
function placeOrder(oderno) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Ge");
            if (shouldResolve) {
                resolve("Data fetched successfully!");
            } else {
                reject("Failed to fetch data.");
            }
        }, 3000); // Simulates a 3-second delay
    });
}
```

Execution Flow:

1. **Global Execution Context:**
   - The JavaScript engine starts executing the global code, creating the global execution context.
   - `placeOrder` **function** is defined in the global scope, and when it's called, it will return a **Promise**.
2. **Calling** `placeOrder`**:**
   - When the function `placeOrder(oderno)` is invoked, it triggers the creation of a **Promise** object.
   - A **new execution context** for the function `placeOrder` is created and placed on the **call stack**.
3. **Inside the** `placeOrder` **Function (Execution Context):**
   - A **new Promise** is created, and a **callback function** is passed to the Promise constructor (which takes two parameters: `resolve` and `reject`).
   - The callback function will be executed asynchronously, meaning it will not execute immediately but after 3 seconds.
   - The `setTimeout` function is called, which also gets added to the **Web APIs** environment (not the call stack). It will execute after a 3-second delay.
4. **Web API / Asynchronous Execution:**
   - After the `setTimeout` is set, JavaScript continues executing the remaining code synchronously.
   - Once the 3-second delay passes, the `setTimeout` callback is moved to the **callback queue** (a task queue).
5. **Callback Queue:**
   - The **callback function** inside the `setTimeout` (which logs "Ge" and resolves or rejects the promise) is now in the **callback queue**.
   - The **event loop** is constantly monitoring the call stack and callback queue. If the call stack is empty, the event loop will move the first function in the callback queue to the call stack for execution.
6. **Execution of the** `setTimeout` **Callback:**
   - After 3 seconds, the `setTimeout` callback function is executed, and the following happens:
     - **"Ge"** is logged to the console.
     - Based on the value of `shouldResolve` (which seems to be missing in the code, but assumed to be a global variable), either:
       - If `shouldResolve` is `true`, the `resolve` function is called, which fulfills the promise with the value `"Data fetched successfully!"`.
       - If `shouldResolve` is `false`, the `reject` function is called, and the promise is rejected with the value `"Failed to fetch data."`.
7. **Microtask Queue:**
   - After the promise is either resolved or rejected, the corresponding `.then` or `.catch` handlers will be added to the **microtask queue** (if these handlers exist).
   - The **event loop** will now check the microtask queue after the current call stack is cleared and process any microtasks (promise resolutions or rejections) in the queue.
8. **Final Execution:**
   - Once the promise is resolved or rejected, the appropriate action will be taken (for example, logging "Data fetched successfully!" or "Failed to fetch data.") based on the handlers for the promise.

Summary:

- The JavaScript execution starts in the global execution context and defines the `placeOrder` function.
- When `placeOrder` is invoked, a new promise is created with an asynchronous operation (`setTimeout`).
- The promise callback (setTimeout) will execute after 3 seconds and log "Ge" to the console.
- Based on the value of `shouldResolve`, the promise is either resolved or rejected.
- The event loop checks the callback queue and moves the promise resolution or rejection to the microtask queue, which is processed after the call stack is empty.



5. Role of the Event Loop

The event loop ensures the following:

1. Complete all **synchronous tasks** in the call stack.
2. Process tasks in the **microtasks queue** (e.g., resolved promises).
3. Move to the **macrotasks queue** (e.g., `setTimeout` callbacks).

8. Comparison: Async/Await vs Promises

Feature	Promises	Async/Await
Syntax	`.then()` chains	Looks synchronous
Error Handling	`.catch()`	`try-catch`
Readability	Complex with chaining	Cleaner, more readable
Parallel Execution	`Promise.all`	`Promise.all` + `await`

Understand Via Examples:

Function Creation:

Example:

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

greet("Alice");
```

Memory Creation (Compilation Phase):

1. **Global Execution Context (GEC):**
   - The JavaScript engine allocates memory for global variables and functions.
   - The function `greet` is stored in memory as an object with its **code** and **scope chain**.

Execution Flow (Execution Phase):

1. **Global Execution Context:**
   - The code is executed line by line.
   - When the `greet` function is declared, it is stored in memory, but not executed.

2. **Function Call (`greet("Alice")`):**
   - The call to `greet("Alice")` creates a new **Function Execution Context (FEC)** for the `greet` function.
   - The argument `name = "Alice"` is passed into the function.

3. **Inside the `greet` Function:**
   - The **local execution context** of `greet` is created, with `name` being `"Alice"`.
   - The `console.log("Hello, " + name)` statement is executed, and **"Hello, Alice"** is printed.
4. **End of Function Execution:**
   - After `greet()` finishes execution, the **Function Execution Context (FEC)** for `greet` is popped off the call stack.

```javascript
let x;
const myName = "Ashish";
console.log("1 global");

function outerFunction() {
    const outerVar = "Outer";
    console.log("2 in outer");

    function innerFunction() {
        console.log("Accessing:", outerVar);
        console.log("3 in inner");
    }

    console.log("4 in outer ");
    return innerFunction;
}

console.log("5 global");
const myFunc = outerFunction();
console.log(myFunc);

myFunc();

console.log("6 global");
```

Step-by-Step Execution:

Step 1: Global Execution Context

When JavaScript starts executing the code, it enters the **Global Execution Context**:

- **Global Context Creation:**
  - `let x;` creates a variable `x` in the **global memory** (initially set to `undefined`).
  - `const myName = "Ashish";` creates a constant `myName` and assigns it the value `"Ashish"`.
  - The first `console.log("1 global")` is executed and logs `"1 global"`.
- **Function Declaration** `outerFunction` is hoisted, so it's available in memory.
  - This doesn't execute it yet, it simply stores a reference to the function.

Global Execution Context (in memory):

- **Stack**: [Global Context]
- **Heap**:
  - `x` (undefined)
  - `myName` ("Ashish")
  - `outerFunction` (reference to the function)

After this, the **execution moves to the next line**, which is `console.log("5 global")`, logging `"5`

Step 2: Calling `outerFunction()`

When you execute `const myFunc = outerFunction();`:

1. **New Execution Context for `outerFunction`** is created:
   - The `outerFunction` is invoked, and its **execution context** is pushed onto the **call stack**.
   - A new local memory is created for `outerFunction`:
     - `outerVar` is created and assigned the value `"Outer"`.
   - The **first** `console.log("2 in outer")` is executed and logs `"2 in outer"`.
   - The `innerFunction` is defined inside `outerFunction`, so it's hoisted, and a reference to it is stored in memory.

Memory during `outerFunction()` execution:

Memory during `outerFunction()` execution:

- **Stack**: [Global Context → `outerFunction` Context]
- **Heap**:
  - `x` (undefined)
  - `myName` ("Ashish")
  - `outerFunction` (reference to the function)
  - `outerVar` ("Outer")
  - `innerFunction` (reference to the function)

2. **Logging** `"4 in outer"` after the second `console.log("4 in outer")` is executed.
3. **Return** `innerFunction`:
   - `outerFunction()` completes execution and returns the `innerFunction` reference. This reference is stored in the variable `myFunc` in the **global memory**.

Stack after `outerFunction()` completes:

- **Stack**: [Global Context]
- **Heap**:
  - `x` (undefined)
  - `myName` ("Ashish")
  - `outerFunction` (reference to the function)
  - `myFunc` (reference to `innerFunction`)

After returning `innerFunction`, the `outerFunction` **context is popped off** the call stack.

Step 3: Logging the Value of `myFunc`

Next, `console.log(myFunc)` is executed, and it logs the function definition of `innerFunction` since `myFunc` holds the reference to `innerFunction`.

Output at this point:

bash Copy code

```bash
[Function: innerFunction]
```

Step 4: Calling `myFunc()`

Now, `myFunc()` is called, which invokes `innerFunction`:

1. **New Execution Context for `innerFunction`** is created:
   - A new execution context for `innerFunction` is pushed onto the **call stack**.
   - `innerFunction` has access to the **lexical environment** of its **outer function** (`outerFunction`), so it can access `outerVar`.
   - The **first** `console.log("Accessing:", outerVar)` logs `"Accessing: Outer"`.
   - The **second** `console.log("3 in inner")` logs `"3 in inner"`.

Stack during `innerFunction()` execution:

- **Stack**: [Global Context → `innerFunction` Context]
- **Heap**:
  - `x` (undefined)
  - `myName` ("Ashish")
  - `outerFunction` (reference)
  - `myFunc` (reference to `innerFunction`)
  - `outerVar` ("Outer")
  - `innerFunction` (reference to the function)

After `innerFunction` finishes, its context is **popped from the call stack**.

Step 5: Final Global Log

Finally, the **last** `console.log("6 global")` is executed and logs `"6 global"`.

Final Memory State:

- **Stack**: [Global Context]
- **Heap**:
  - `x` (undefined)
  - `myName` ("Ashish")
  - `outerFunction` (reference to the function)
  - `myFunc` (reference to `innerFunction`)

Execution Flow Summary:

1. The **global context** is created, variables and function declarations are hoisted.
2. `outerFunction` is invoked, creating a new execution context, defining `outerVar` and `innerFunction`, and logging `"2 in outer"`.
3. `outerFunction` completes, returning the reference to `innerFunction`, which is stored in `myFunc`.
4. `myFunc()` is called, invoking `innerFunction`, which logs `"Accessing: Outer"` and `"3 in inner"`.
5. The program logs `"6 global"` at the end.

Output:

```bash
1 global
5 global
2 in outer
4 in outer
[Function: innerFunction]
Accessing: Outer
3 in inner
6 global
```


Explanation:

1. `1 global` is logged from the first `console.log` in the global scope.
2. `5 global` is logged after the global `console.log` before calling `outerFunction()`.
3. When `outerFunction()` is invoked:
   - `2 in outer` is logged.
   - `4 in outer` is logged after the inner function is defined but before it is returned.
4. `[Function: innerFunction]` is logged because `myFunc` stores the reference to `innerFunction`.
5. `Accessing: Outer` and `3 in inner` are logged when `myFunc()` is invoked, and it runs the `innerFunction` that has access to `outerVar`.
6. Finally, `6 global` is logged.

SetTimeOut :

```
console.log("Hello 1")
// console.log("Hello 1")


setTimeout(()=>{
    console.log("Hello 2")
} ,1000);

console.log("Hello 3")
```

```
/* Immediate Execution:

console.log("Hello 1") prints "Hello 1".
console.log("Hello 3") prints "Hello 3".
setTimeout:

setTimeout schedules the callback to print "Hello 2" after 1000ms.
The callback is placed in the callback queue.
After 1000ms:

The callback moves to the call stack and "Hello 2" is printed. */
/*

OutPut:
Hello 1
Hello 3
Hello 2
 */
```

ASYNC AWAIT

Key Points:

- `await` pauses the execution of the `async` function until the promise resolves.
- If you have multiple `await` statements in an `async` function, they are executed sequentially, meaning the next `await` doesn't execute until the current one resolves.
- The function will still **return a promise** immediately, and the code execution will continue **non-blocking** outside of the function.

```
async function fetchDataFromServer1() {
    return new Promise(resolve => setTimeout(() => resolve("Data1"), 1000));  // Simulates a
}

async function fetchDataFromServer2() {
    return new Promise(resolve => setTimeout(() => resolve("Data2"), 500));   // Simulates a
}

async function fetchDataFromServer3() {
    return new Promise(resolve => setTimeout(() => resolve("Data3"), 1500));  // Simulates a
}

async function exampleFunction() {
    console.log("Start");

    const result1 = await fetchDataFromServer1();  // Await #1
    console.log("Fetched data 1:", result1);

    const result2 = await fetchDataFromServer2();  // Await #2
    console.log("Fetched data 2:", result2);
```

```
    const result3 = await fetchDataFromServer3();  // Await #3
    console.log("Fetched data 3:", result3);

    console.log("End");
}

exampleFunction();
```

Behavior:

1. **Execution Flow:**
   - When `exampleFunction()` is called, it starts executing.
   - It logs `"Start"`.
   - It reaches the first `await` and waits for the `fetchDataFromServer1()` function to resolve. During this wait, the event loop can execute other tasks, but **the function itself pauses**.
2. **Waiting for the First `await`:**
   - Once `fetchDataFromServer1()` resolves (e.g., after an API call or any other promise), `result1` is assigned, and it logs `"Fetched data 1: [result1]"`.

3. **Next Await**:
   - The function continues to the second `await`. It will not move to `await fetchDataFromServer2()` until the first one has finished.
   - It waits for `fetchDataFromServer2()` to resolve, and after that, it logs `"Fetched data 2: [result2]"`.
4. **Subsequent Awaits**:
   - The same process happens with the third `await`, and the function continues until all the `await` statements are resolved.

continues until all the `await` statements are resolved.

5. Finally, `"End"` is logged when all promises have resolved.

Visualizing the Flow:

plaintext

```
Start
(await fetchDataFromServer1) -> Waiting...
(FetchData1 resolves) -> Fetched data 1: result1
(await fetchDataFromServer2) -> Waiting...
(FetchData2 resolves) -> Fetched data 2: result2
(await fetchDataFromServer3) -> Waiting...
(FetchData3 resolves) -> Fetched data 3: result3
End
```

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

WHEN TO USE AWAIT

The `await` keyword in JavaScript is used within an `async` function to pause the execution of the function until a **Promise** is resolved or rejected. You should use `await` in situations where you need to **wait for asynchronous operations to complete** before proceeding with further code execution. It makes asynchronous code easier to write and read by avoiding callback chains or `.then()` syntax.

2. **To simplify the code and avoid nested callbacks or `.then()` chains:** Without `await`, you would need to use `.then()` to handle the Promise resolution, which could lead to nested callback functions. `await` makes the code look more like synchronous code, improving readability.

   Example with `.then()`:

   javascript

   ```javascript
   fetch('https://api.example.com/data')
       .then(response => response.json())
       .then(data => console.log(data))
       .catch(error => console.error(error));
   ```

The same code with `await`:

javascript

```javascript
async function getData() {
    try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

getData();
```

1. Sequential Execution with `async/await`

In **sequential execution**, each asynchronous task is executed one after another. The next task waits for the previous one to complete before it starts.

2. **Sequential Execution:**
   - When you use multiple `await` expressions one after another, JavaScript waits for each Promise to resolve before moving on to the next one.
   - This means each asynchronous task happens **one after the other**, and the total time will be the sum of all individual task times.

javascript

```javascript
async function sequential() {
    console.log('Sequential Start');
    const res1 = await fetchData1(); // Waits for fetchData1
    const res2 = await fetchData2(); // Then waits for fetchData2
    console.log('Sequential End');
}

function fetchData1() {
    return new Promise((resolve) => setTimeout(() => resolve('Data 1'),
}

function fetchData2() {
    return new Promise((resolve) => setTimeout(() => resolve('Data 2'),
}

sequential(); // Takes 2 seconds
```

Output (Sequential):

plaintext

```plaintext
Sequential Start
Data 1
Data 2
Sequential End
```

3. **Parallel Execution:**
   - With parallel execution, you can start multiple asynchronous operations at once without waiting for them to complete.
   - You then `await` them all using `Promise.all()` to wait for all Promises to resolve simultaneously.
   - This makes the operations run **concurrently**, significantly reducing the overall time if the operations are independent.

Example:

javascript

```
async function parallel() {
    console.log('Parallel Start');
    const p1 = fetchData1();  // Starts fetchData1
    const p2 = fetchData2();  // Starts fetchData2
    const [res1, res2] = await Promise.all([p1, p2]); //
    console.log('Parallel End');
}

parallel(); // Takes 1 second
```

Output (Parallel):

plaintext

```
Parallel Start
Data 1
Data 2
Parallel End
```


Summary:

- **Sequential (`await` one after another):**
  - Slower, as each task has to finish before the next one starts.
  - Ideal when the tasks depend on each other (e.g., when the result of one task is needed for the next).
- **Parallel (`Promise.all` to await multiple promises):**
  - Faster, as tasks run concurrently.
  - Best when the tasks are independent, and their execution doesn't depend on the result of one another.

1. How They Work

Promises:

- Promises represent a value that may be available now, or in the future, or never.
- A Promise has three states: **Pending**, **Resolved (Fulfilled)**, or **Rejected**.
- It uses `.then()` and `.catch()` for chaining operations and handling success or errors, respectively.

`async/await`:

- Built on top of Promises to make asynchronous code easier to write and read.
- `await` pauses the execution of the async function until the Promise is resolved or rejected.
- It is syntactic sugar over Promises, allowing for a more synchronous-looking code.

```
async function fetchDataAsync() {
    try {
        const data = await new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve("Data fetched successfully");
            }, 1000);
        });
        console.log(data);

        const next = "Next task";
        console.log(next);
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchDataAsync();
```

Using Promise

javascript

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data fetched successfully");
        }, 1000);
    });
}

// Using .then() and .catch()
fetchData()
    .then(data => {
        console.log(data);
        return "Next task";
    })
    .then(next => {
        console.log(next);
    })
    .catch(error => {
        console.error("Error:", error);
    });
```

2. When to Use `async/await` vs Promises

Use Promises when:

1. **Chaining multiple independent async tasks:**
   - Promises handle task chains well when tasks don't depend on each other.
   - Example: Fetching two unrelated resources in parallel:

javascript   Copy code

```
const promise1 = fetchData1();
const promise2 = fetchData2();

Promise.all([promise1, promise2])
    .then(([data1, data2]) => {
        console.log(data1, data2);
    })
    .catch(err => console.error("Error:", err));
```

2. **Using Promise combinators:**
   - When using methods like `Promise.all`, `Promise.race`, or `Promise.allSettled`, Promises are more natural.
   - Example: Fetch multiple APIs, but proceed with the first one that resolves:

javascript Copy code

```javascript
Promise.race([fetchData1(), fetchData2()])
    .then(firstResolved => console.log(firstResolved))
    .catch(err => console.error(err));
```

3. **Compatibility with older code:**
   - If the codebase already uses Promises, sticking to `.then()`/`.catch()` ensures consistency.

Use `async/await` when:

1. **Improving readability for dependent async tasks:**
   - When you have tasks that need to execute in sequence, `async/await` makes the code look more like synchronous code, improving clarity.
   - Example: Fetch data, process it, and then save it:

javascript Copy code

```javascript
async function processData() {
    const data = await fetchData();
    const processedData = process(data);
    await save(processedData);
    console.log("Data saved!");
}
```

2. **Error handling with try-catch:**
   - `async/await` uses standard `try-catch` blocks for error handling, which is cleaner than using `.catch()` multiple times.
   - Example:

javascript Copy code

```javascript
async function fetchWithErrorHandling() {
    try {
        const data = await fetchData();
        console.log(data);
    } catch (error) {
        console.error("Error:", error);
    }
}
```

3. **Avoiding deeply nested Promise chains:**
   - When dealing with long chains of Promises, `async/await` can help avoid "callback hell"-like situations.

```javascript
// With Promises (deep nesting)
fetchData()
    .then(data => process(data))
    .then(processed => save(processed))
    .then(() => console.log("Data saved!"))
    .catch(error => console.error("Error:", error));

// With async/await
async function processData() {
    try {
        const data = await fetchData();
        const processedData = await process(data);
        await save(processedData);
        console.log("Data saved!");
    } catch (error) {
        console.error("Error:", error);
    }
}
```

3. Comparison Table

Feature	Promises	Async/Await
Code readability	Can get messy with long chains (`.then()`)	Looks cleaner and synchronous-like
Sequential operations	Harder to manage with chaining	Easier to manage
Error handling	`.catch()` for each chain or at the end	Standard `try-catch` block
Parallel execution	Natural with `Promise.all`	Can be done with `Promise.all` too
Syntax	Requires chaining methods	Uses standard `async` functions
Learning curve	Easy to understand	Slightly more complex but intuitive
Callback hell	Possible with deeply nested chains	Avoided completely

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com

PDF-XChange Editor
Click to BUY NOW!
www.pdf-xchange.com