Advanced JavaScript

Motivation

Static vs Live

Static HTML

These items are hard-coded in the HTML:

- Course: Advanced JavaScript
- Topic: Fetch & Async/Await
- Status: Always the same

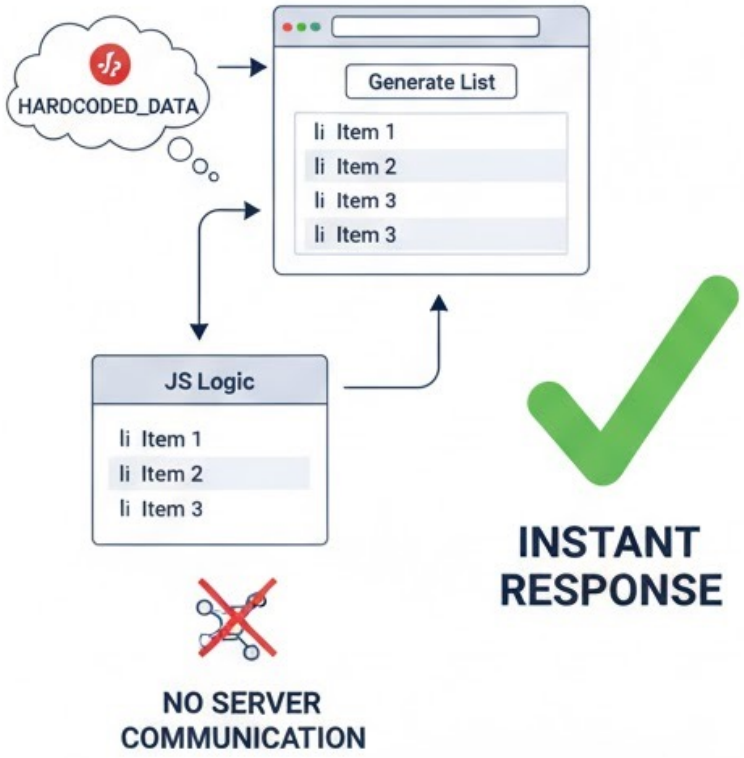
Live Data

Load Data Loaded at 10/19/2025, 10:50:36 PM

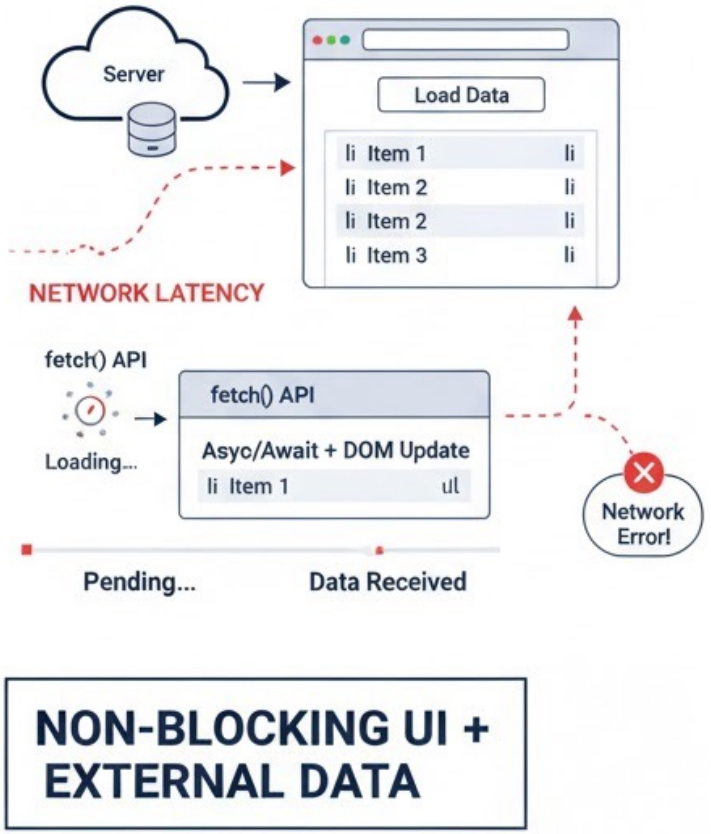
- 1: sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- 2: qui est esse
- 3: ea molestias quasi exercitationem repellat qui ipsa sit aut

Motivation

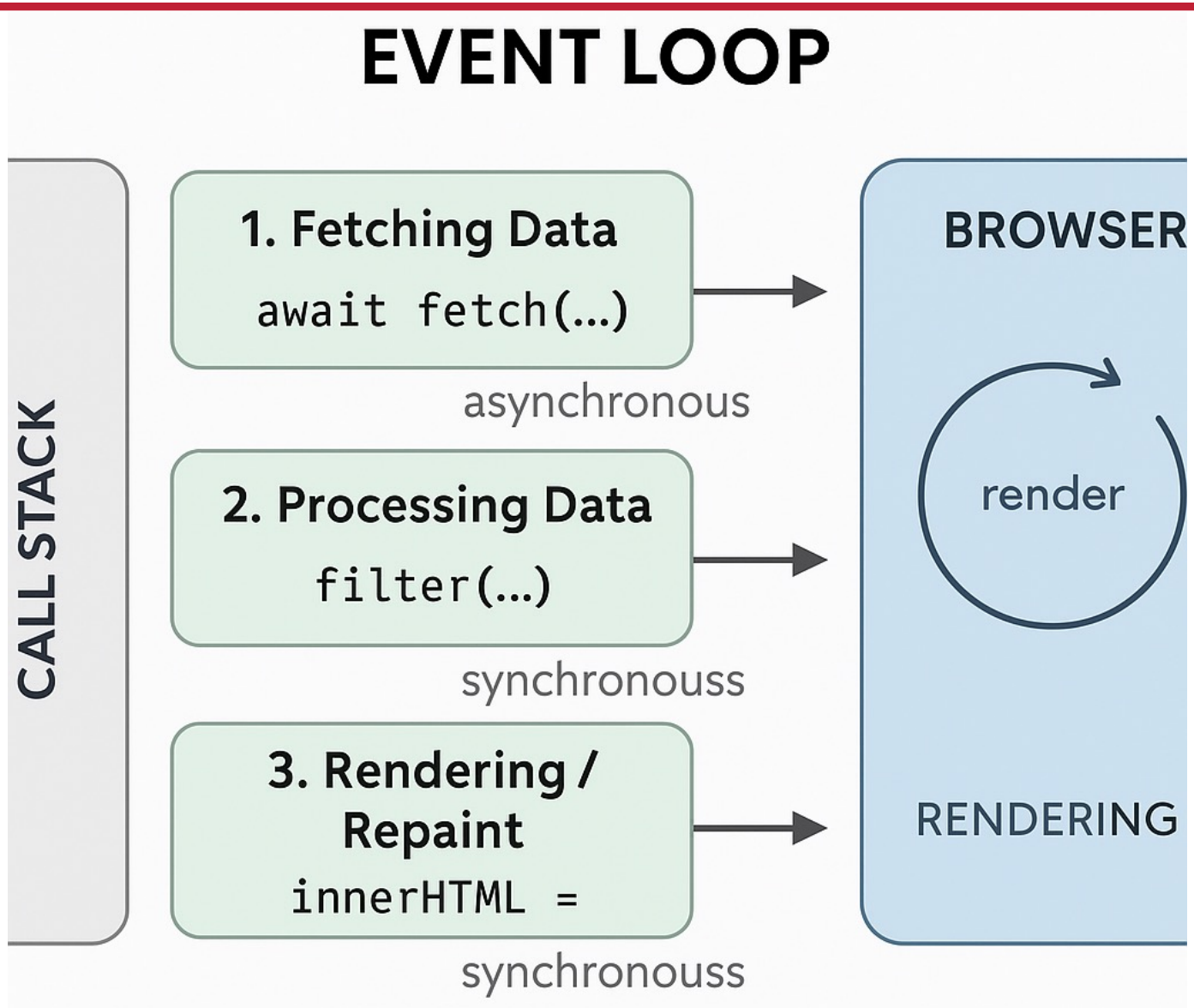
Client-side interactivity



Asynchronous data-driven Applications.



Flow


EVENT LOOP
CALL STACK
1. Fetching Data
await fetch(...)
asynchronous
2. Processing Data
filter(...)
synchronouss
3. Rendering / Repaint
innerHTML =
synchronouss
BROWSER
render
RENDERING

Content

1. Advanced DOM and Form Handling
2. Callbacks and Promises
3. Async/Await
4. Exercises

HUST

hust.edu.vn fb.com/dhbkhn

1. Advanced DOM and Form Handling

Overview of Advanced JavaScript

- Transition:

simple client-side interactivity to handling complex data-driven, asynchronous workflows

- Advanced JavaScript covers how it interacts with:
  - DOM - web structure efficiently.
  - Event – responding to user actions and browser events.
  - Runtime environment– communicating with servers, handling APIs, and managing background tasks.

=> to build interactive, responsive, and scalable web applications.

Accessing Forms and Advanced Fields

- Different input types require different handling
  - Radio: multiple options, only one selected.
  - Multiple Select: users can choose more than one option.
  - Textarea: multiline text input.
- Radio

```
<form id="survey">
  <label><input type="radio" name="gender" value="Male"> Male</label>
  <label><input type="radio" name="gender" value="Female"> Female</label>
</form>

<script>
const gender = document.querySelector('input[name="gender"]:checked')?.value;
console.log("Selected gender:", gender);
</script>
```

Accessing Forms and Advanced Fields

- Multiple Select
  - Array is a built-in class in JavaScript
  - Array.from(): converts HTMLCollection into an array
  - map (o =>o.value): extracts value (string) from each option

```
<select id="skills" multiple>
  <option value="JS">JavaScript</option>
  <option value="Python">Python</option>
  <option value="C++">C++</option>
</select>

<script>
const select = document.getElementById('skills');
const selected = Array.from(select.selectedOptions).map(o => o.value);
console.log(selected);
</script>
```

Accessing Forms and Advanced Fields

- Textarea

```
<textarea id="message">Enter your text...</textarea>

<script>
const text = document.getElementById('message').value;
console.log("Message:", text);

// Set value dynamically
document.getElementById('message').value = "Prefilled content!";
</script>
```

Example

Accessing Advanced Form Fields with JavaScript

Radio Group

- Male
- Female
- Other

Multiple Select

Select your programming skills:

JavaScript
Python
C++
Java

Textarea

Short message:

Xin chào
Chúc bạn đạt kết quả học tập tốt

Show Values

Gender: Not selected

Skills: Python, C++

Message: Xin chào Chúc bạn đạt kết quả học tập tốt

Handling the Submit Event

- By default, when a user submits a form
  - Browser: reloads the page or navigates to another URL
  - This default behavior leaves you out of control of data
  - To handle the form entirely with JavaScript, we use: *event.preventDefault();*

```html
<form id="loginForm">
  <label>Username: <input type="text" id="username" required></label><br>
  <label>Password: <input type="password" id="password" required></label><br>
  <button type="submit">Login</button>
</form>

<script>
document.getElementById('loginForm').addEventListener('submit', (event) => {
  event.preventDefault(); // 🔴 Stop the page from reloading

  const user = document.getElementById('username').value;
  const pass = document.getElementById('password').value;

  console.log("Form submitted:", user, pass);
  alert(`Welcome, ${user}!`);
});
</script>
```

```
[User clicks Submit]
        ↓
Default Browser Action → Page Reload ❌
        ↓
With event.preventDefault() → JS captures data ✅
```

Validation Form – Required Fields

- Ensure all mandatory fields are filled. Combine:
  - HTML5 built-in validation (required) and
  - JavaScript logic

```
<form id="myForm">
  <input id="name" type="text" placeholder="Enter your name" required>
  <button type="submit">Submit</button>
</form>

<script>
document.getElementById("myForm").addEventListener("submit", function(e) {
  e.preventDefault(); // chặn reload
  const name = document.getElementById("name").value.trim();

  if (!name) {
    alert("Please enter your name!");
  } else {
    alert("Form submitted successfully!");
  }
});
</script>
```

Validation Form – Using Basic RegEx

- Regular Expressions (RegEx): patterns
- Use RegEx to validate email, phone number, and numeric formats in form inputs.

Symbol	Meaning	Example
^	Start of string	^A → must start with “A”
$	End of string	Z$ → must end with “Z”
\d	Any digit (0–9)	\d{3} → exactly 3 digits
[a-zA-Z]	Any letter	[A-Z]+ → 1 or many uppercase letters
.	Any single character	a.c → matches “abc”, “a-c”, etc.
+, *, {n}	Repetition	\d{10} → exactly 10 digits

Validation Form - Realtime

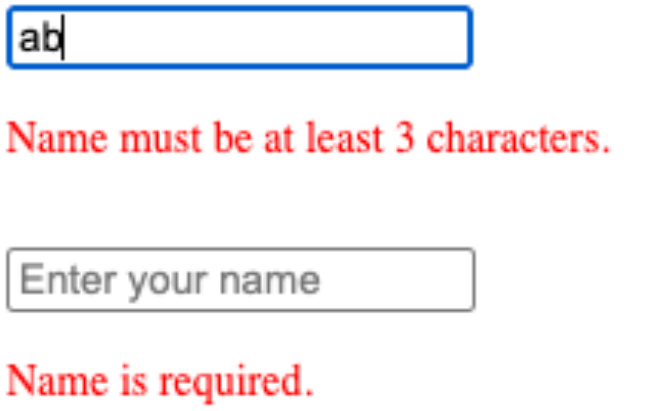
- Show error messages immediately as the user types
- Ideal for live validation like name, password length, etc.
- Approach
  - input event → fires when user types, deletes, pastes
  - change event → fires when the input value changes

```
<input id="name" type="text" placeholder="Enter your name">
<p id="error" style="color: red; font-size: 14px;"></p>

<script>
  const nameInput = document.getElementById("name");
  const errorMsg = document.getElementById("error");

  nameInput.addEventListener("input", () => {
    const value = nameInput.value.trim();

    if (value === "") {
      errorMsg.textContent = "Name is required.";
    } else if (value.length < 3) {
      errorMsg.textContent = "Name must be at least 3 characters.";
    } else {
      errorMsg.textContent = "";
    }
  });
</script>
```



Validation Form - Realtime

- Approach
  - input event → fires when user types, deletes, pastes
  - change event → fires when the input value changes

```
<select id="fruit">
  <option value="">-- Choose one --</option>
  <option>Apple</option>
  <option>Banana</option>
  <option>Mango</option>
</select>

<p id="result" style="color: green; font-weight: bold;"></p>

<script>
  const select = document.getElementById("fruit");
  const result = document.getElementById("result");

  select.addEventListener("change", () => {
    const value = select.value;
    if (value === "") {
      result.textContent = "Please select a fruit.";
      result.style.color = "red";
    } else {
      result.textContent = `You selected: ${value}`;
      result.style.color = "green";
    }
  });
</script>
```

Data Attributes (data-*) – Custom Data Storage

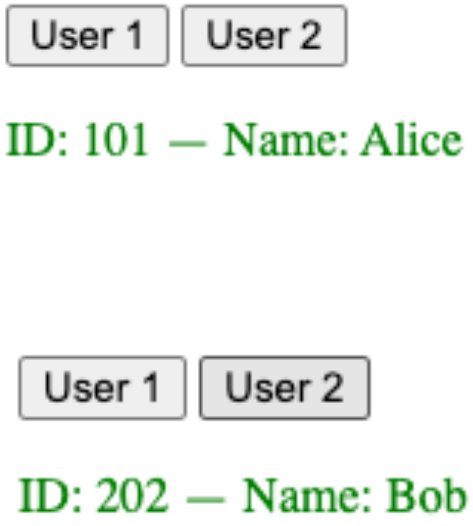
- Use data- to store extra information about an element
  - Small config values
  - Temporary data
- Access in JS through the dataset object
  - data-user-id => userId in JS (automatically converted)
  - data-name => name in JS

```
<button data-user-id="101" data-name="Alice">User 1</button>
<button data-user-id="202" data-name="Bob">User 2</button>

<p id="info" style="color: green;"></p>

<script>
  const buttons = document.querySelectorAll("button");
  const info = document.getElementById("info");

  buttons.forEach(btn => {
    btn.addEventListener("click", () => {
      const id = btn.dataset.userId;  // get data-user-id
      const name = btn.dataset.name;  // get data-name
      info.textContent = `ID: ${id} – Name: ${name}`;
    });
  });
</script>
```



Property classList

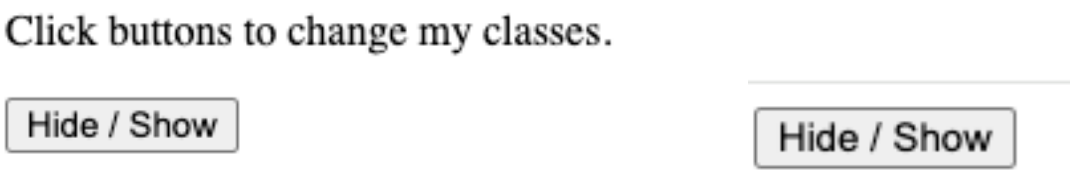
- classList is a DOM property that provides methods to access and manipulate CSS classes on an element

Method	Description	Example
add()	Adds one or more classes	el.classList.add("active")
remove()	Removes one or more classes	el.classList.remove("hidden")
toggle()	Adds if missing, removes if present	el.classList.toggle("dark")
contains()	Checks if a class exists	el.classList.contains("active") → true/false
replace()	Replaces one class with another	el.classList.replace("old", "new")

Property classList

```
<style>
  .hidden { display: none; }
</style>
</head>
<body>

<p id="title">Click buttons to change my classes.</p>
<button id="hide">Hide / Show</button>
<script>
const title = document.getElementById('title');
document.getElementById('hide').addEventListener('click', () => {
  title.classList.toggle('hidden');
});
</script>
```



Event Delegation

- Handle multiple child element events by attaching one event listener to their parent element
- Use: event.target to detect which child was interacted with.

```html
<div id="fruit-list">
  <button>Apple</button>
  <button>Banana</button>
  <button>Mango</button>
</div>
```

```html
<script>
document.querySelectorAll("button").forEach(btn => {
  btn.addEventListener("click", () => alert(btn.textContent));
});
</script>
```

❌ Problem: 3 event listeners → wasteful if 100 buttons.

```html
<script>
document.getElementById("fruit-list").addEventListener("click", (event) => {
  if (event.target.tagName === "BUTTON") {
    alert("You clicked: " + event.target.textContent);
  }
});
</script>
```

DOM Performance Optimization – DocumentFragment

- DocumentFragment helps improve DOM performance:
  - Build multiple elements in memory first,
  - Insert them into the DOM all at once,
  - Avoiding repeated reflow and repaint operations
    - reflow: browser calculates layout (position, sizes) after DOM changes
    - repaint: browser draws visible pixels

Action	Without Fragment	With Fragment
DOM operations	Many	One
Reflow/Repaint	Multiple	Single
Speed	Slower	Faster

DOM Performance Optimization – DocumentFragment

```html
<ul id="list"></ul>

<script>
const ul = document.getElementById("list");
for (let i = 1; i <= 5; i++) {
  const li = document.createElement("li");
  li.textContent = "Item " + i;
  ul.appendChild(li); // Each append triggers reflow/repaint
}
</script>
```

❌ **Problem:** 5 separate DOM insertions → 5 reflows.

```html
<script>
const ul = document.getElementById("list");
const fragment = document.createDocumentFragment();

for (let i = 1; i <= 5; i++) {
  const li = document.createElement("li");
  li.textContent = "Item " + i;
  fragment.appendChild(li); // Added in memory, no reflow
}

ul.appendChild(fragment); // One single reflow when inserted
</script>
```

Timer Management – setTimeout & clearTimeout

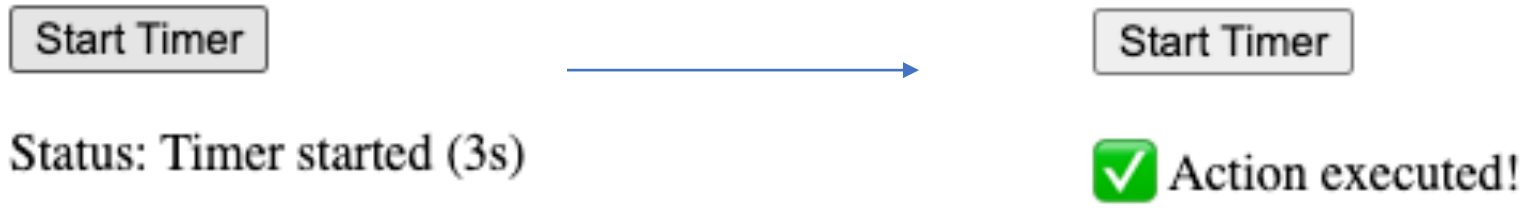
- JavaScript timers let you schedule code to run after a delay,

Function	Purpose
setTimeout(func, delay)	Executes func once after delay ms
clearTimeout(id)	Cancels a timer created by setTimeout
setInterval(func, delay)	Repeats func every delay ms
clearInterval(id)	Cancels a repeating interval

Timer Management – setTimeout & clearTimeout

```html
<button id="start">Start Timer</button>
<p id="status">Status: Waiting...</p>

<script>
let timerId = null;
document.getElementById("start").onclick = () => {
  if (timerId) {
    clearTimeout(timerId);
  }
  document.getElementById("status").textContent = "Status: Timer started (3s)";
  timerId = setTimeout(() => {
    document.getElementById("status").textContent = "✅ Action executed!";
  }, 3000);
};
</script>
```



Debouncing – Limiting Function Call Frequency

- Ensures that a function runs only after the user has stopped triggering event for a certain amount of time

```
<input id="search" placeholder="Enter keyword...">
<p id="result" style="color: green;"></p>

<script>
  function debounce(fn, delay) {
    let timer;
    return function(...args) { //rest parameters
      clearTimeout(timer);
      timer = setTimeout(() => fn.apply(this, args), delay);
    };
  }

  function handleSearch(e) {
    document.getElementById("result").textContent =
      "Searching for: " + e.target.value;
  }

  const input = document.getElementById("search");
  input.addEventListener("input", debounce(handleSearch, 500));
</script>
```

Type to search (debounced 500ms):

Hello World

Searching for: Hello World

Throttling – Limiting Execution Rate

- Throttling is a technique that ensures a function is executed at most once every X milliseconds, even if it's triggered continuously
- Reason
  - Prevents performance issues from too many event calls.
  - Improves smoothness in scroll, resize, or drag
- How It Works
  - When the event fires, the function checks if enough time has passed since the last execution
  - If not, the call is ignored.
  - After the time limit, the function runs again.

Throttling – Limiting Execution Rate

```
<script>
// Throttle function
function throttle(func, delay) {
  let lastCall = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      func.apply(this, args);
      lastCall = now;
    }
  };
}

let counter = 0;
const updateCounter = () => {
  counter++;
  document.getElementById("count").textContent = counter;
};

// Apply throttling: function runs at most once per 1000ms
window.addEventListener("scroll", throttle(updateCounter, 1000));
</script>
```

HUST

hust.edu.vn fb.com/dhbkhn

2. Callbacks and Promises

Asynchronous Programming

- Problem with Synchronous
  - In JavaScript: runs one task at a time.
  - If one operation takes too long => it blocks the rest of the page

```
console.log("Start");
for (let i = 0; i < 1e9; i++) {} // long loop
console.log("End"); // UI freezes until loop finishes
```

- Asynchronous programming
  - Allows non-blocking execution
  - Browser can continue doing other tasks while waiting for time-consuming operations, e.g.,
    - Network operations (fetch, AJAX)
    - File I/O or timers
    - Loop

Synchronous vs. Asynchronous

Scenario	Synchronous Problem	Asynchronous Benefit
API requests	Waiting for server Blocks UI	Fetch runs in background
File I/O or database access	Long read/write delays	User can still interact
Animations	UI stutter	Smooth transitions
User interactions	Delayed response	Immediate feedback

Example

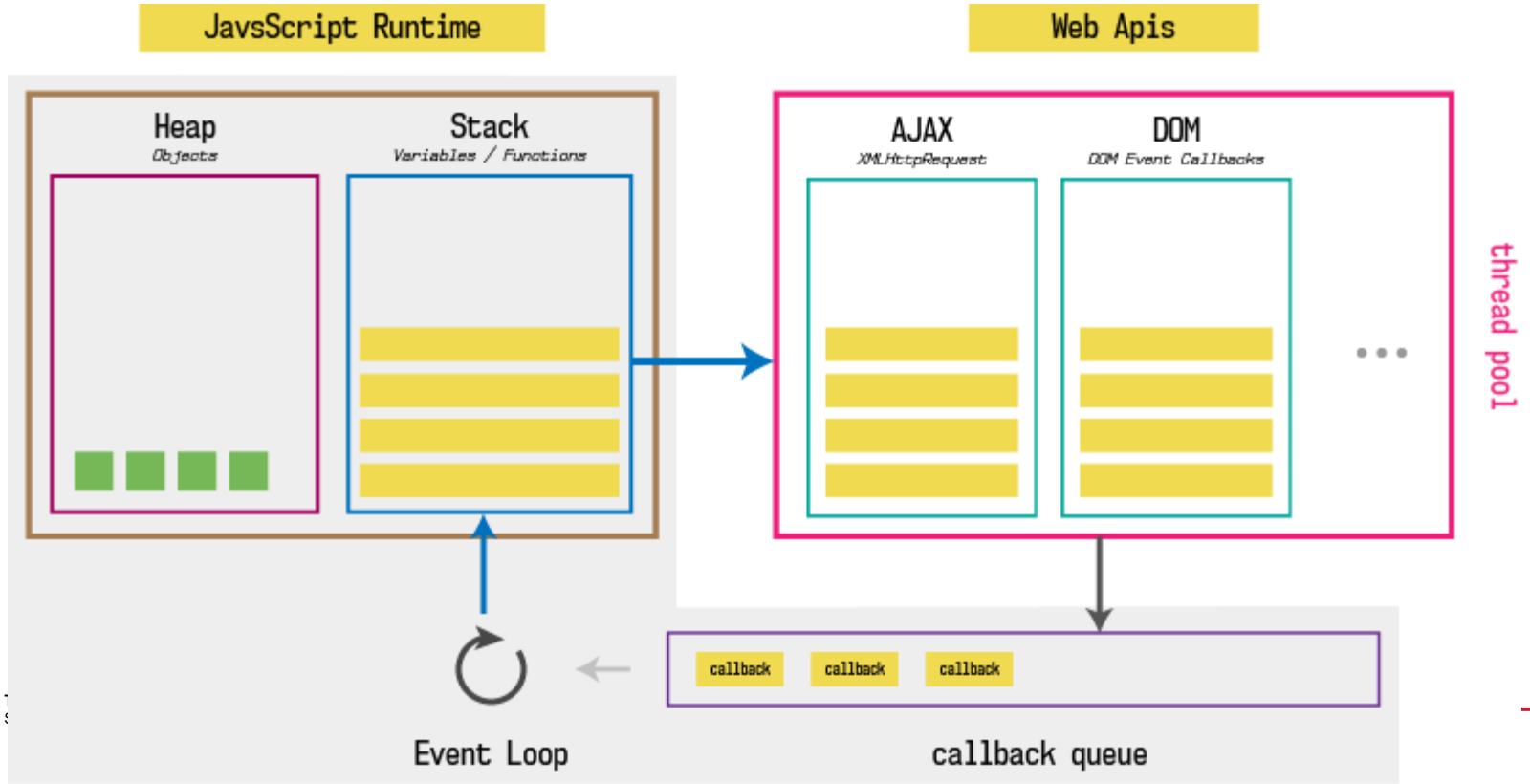
```
<body>
    <div>
        <button onclick="chayBlocking()">1. Chạy Hàm BLOCKING</button>
        <button onclick="chayNonBlocking()">2. Chạy Hàm NON-BLOCKING</button>
        <p id="trangThai">Trạng thái: Đang chờ lệnh...</p>
    </div>

    <script>
        const elm = document.getElementById('trangThai');
        function chayBlocking() {
            elm.textContent = "Trạng thái: Bắt đầu quá trình BLOCKING...";
            alert("Đây là alert(). Trình duyệt đã bị BLOCK! Nhấn OK để tiếp tục.");
            elm.textContent = "Trạng thái: Quá trình BLOCKING đã HOÀN THÀNH.";
        }

        function chayNonBlocking() {
            elm.textContent = "Trạng thái: Bắt đầu quá trình NON-BLOCKING (sau 2 giây).";
            setTimeout(() => {
                elm.textContent = "Trạng thái: Quá trình NON-BLOCKING đã HOÀN THÀNH.";
            }, 2000);
        }
    </script>
</body>
```

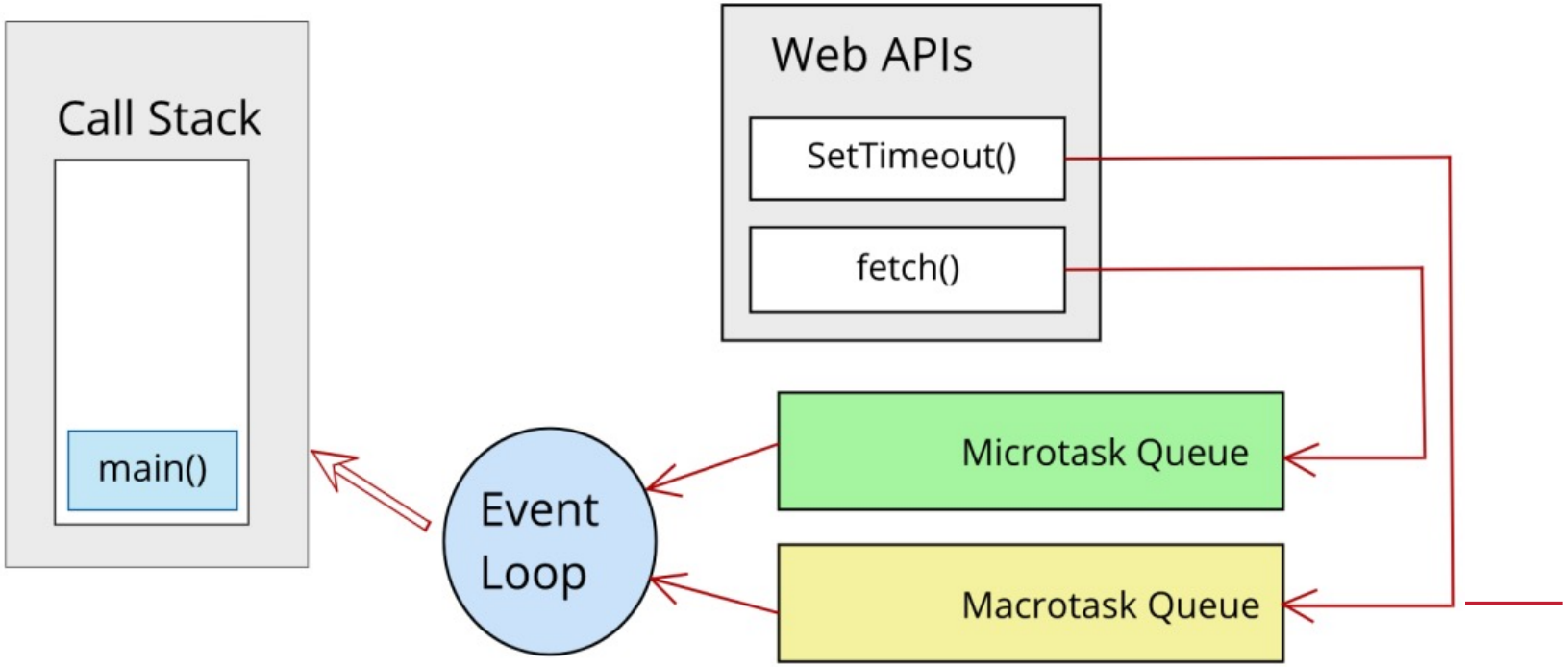
JavaScript - Single Thread

- **Event Loop** enables **asynchronous** operations to be handled
  - Is the Call Stack empty?
  - If yes, move the next task from the Callback Queue to the stack and execute it.
  - WebAPIs: handle async operations (setTimeout, fetch, etc)



Microtask vs Macrotask

- Event Loop = Microtask Queue + Macrotask Queue
  - Microtask Queue → high priority (Promises, async/await)
  - Macrotask Queue → normal priority (setTimeout, setInterval, DOM events)
- After tasks in the Call Stack finishes, Event Loop runs all Microtasks first, then moves to Macrotasks.

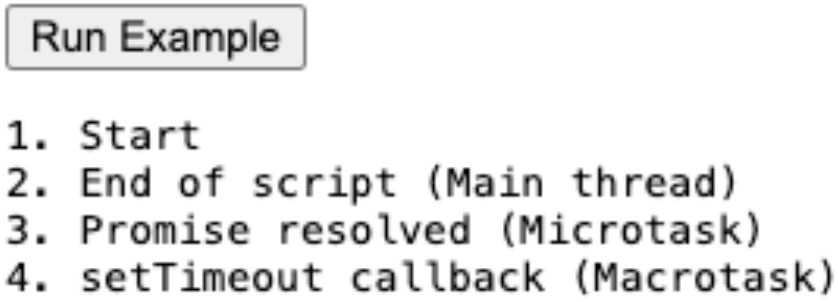


Microtask vs Macrotask

```html
<body>
  <button id="run">Run Example</button>
  <pre id="log"></pre>

  <script>
    const log = (msg) => {
      document.getElementById("log").textContent += msg + "\n";
    };

    document.getElementById("run").onclick = () => {
      log("1. Start");
      setTimeout(() => log("4. setTimeout callback (Macrotask)"), 0);
      Promise.resolve().then(() => log("3. Promise resolved (Microtask)"));
      log("2. End of script (Main thread)");
    };
  </script>
</body>
```



Summary

Component	Belongs To	Examples	Role in Asynchronous Execution
JavaScript Engine	JS Language	Call Stack, Heap	Executes JavaScript code synchronously. No real async capability by itself.
JavaScript Syntax Features	JS Language	Promise, async/await	Provides a language-level abstraction to manage asynchronous results.
Runtime Environment	Browser or Node.js	setTimeout, fetch,	Performs actual asynchronous operations
Event Loop	Runtime Mechanism	Event Loop cycle	Constantly checks if the Call Stack is empty,
Web APIs / libuv APIs	Runtime Environment	Timers, DOM Events	Manage asynchronous system calls.
Code	User (You)	Using fetch(), .then(), await	Uses the language syntax + runtime APIs to compose asynchronous programs.

Callback Function

- A callback is a function passed as an argument to another function, usually **after** a task completes.
- It allows asynchronous actions, keeping JavaScript non-blocking.

```
function doTask(callback) {
  console.log("Doing something...");
  callback(); // invoke the callback when done
}

function afterTask() {
  console.log("Task completed!");
}

doTask(afterTask);
```

```
Doing something..
Task completed!
```

Simulating Asynchronous Behavior with Callback

```
<body>
  <h3>Simulating Asynchronous Callback – Named Function</h3>
  <button id="loadBtn">Load Data</button>
  <p id="status">Status: Waiting...</p>

  <script>
    function getData(callback) {
      document.getElementById("status").textContent = "Loading data...";
      setTimeout(() => {
        const data = { name: "Alice", age: 25 };
        callback(data);
      }, 2000);
    }
    function handleData(result) {
      document.getElementById(
        "status"
      ).textContent = `Data loaded: ${result.name}, ${result.age} years old`;
    }
    document.getElementById("loadBtn").addEventListener("click", function () {
      getData(handleData);
    });
  </script>
</body>
```

Error-first Callback Pattern

- A convention used in Node.js
- The first argument of a callback function is reserved for error handling.

```
function callback(error, data) {
  if (error) {
    // handle the error
  } else {
    // process the data
  }
}
```

Error-first Callback Pattern

```
function readFile(callback) {
  document.getElementById("output").textContent = "Reading file...\n";
  setTimeout(() => {
    const success = Math.random() > 0.5; // random success/failure
    if (success) {
      callback(null, "File content: Hello world!");
    } else {
      callback("File not found", null);
    }
  }, 1000);
}
function handleResult(err, data) {
  const log = document.getElementById("output");
  if (err) {
    log.textContent += "Error: " + err + "\n";
  } else {
    log.textContent += "Success: " + data + "\n";
  }
}
document.getElementById("run").addEventListener("click", () => {
  readFile(handleResult);
});
```

Callback Hell (Pyramid of Doom)

- Callback Hell refers to deeply nested callback functions
- Make code hard to read, maintain, and debug.

```
function getUser(id, callback) {
  setTimeout(() => callback(null, { id, name: "Alice" }), 500);
}

function getPosts(user, callback) {
  setTimeout(() => callback(null, ["Post1", "Post2"]), 500);
}

function getComments(post, callback) {
  setTimeout(() => callback(null, ["Nice!", "Great!"]), 500);
}

// ❌ Callback Hell: nested structure
getUser(1, function(err, user) {
  if (err) return console.error(err);
  getPosts(user, function(err, posts) {
    if (err) return console.error(err);
    getComments(posts[0], function(err, comments) {
      if (err) return console.error(err);
      console.log("✅ Done:", user.name, posts[0], comments);
    });
  });
});
```

Promise

- A Promise is a built-in JavaScript object that represents the eventual completion (or failure) of an asynchronous operation — and its resulting value.

State	Description	Transition
Pending	Initial state; operation not completed yet	→ Fulfilled or Rejected
Fulfilled	Operation completed successfully	.then() executes
Rejected	Operation failed	.catch() executes

Promise

- A **Promise** is created using the constructor:

new Promise(executor)

- The **executor** is a function that takes two arguments:

function executor(resolve, reject) { ... }

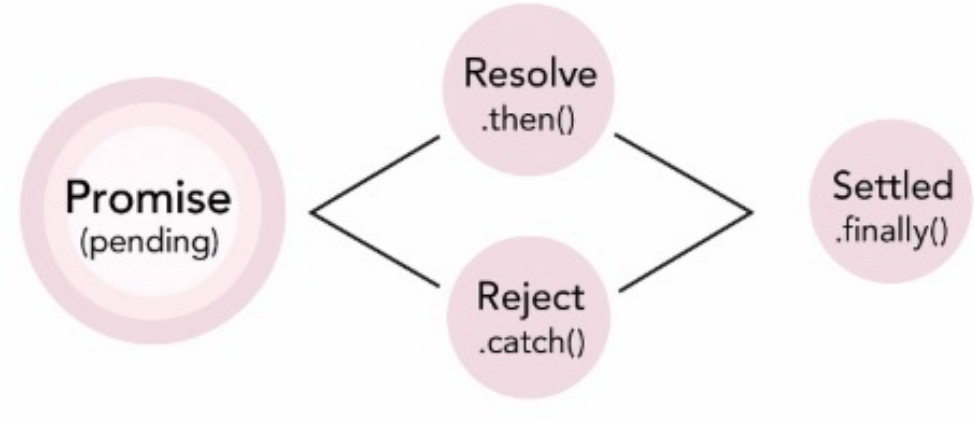
```
let promise = new Promise(function (resolve, reject) {
    if (/* success */) {
        resolve("Success message");
    } else {
        reject("Error message");
    }
});
```

Promise

```
document.getElementById("run").addEventListener("click", function() {
  log("⏳ Starting async operation...");

  let promise = new Promise(function(resolve, reject) {
    let ok = Math.random() > 0.5;
    setTimeout(function() {
      if (ok) resolve("✅ Operation successful!");
      else reject("❌ Operation failed!")·
    }, 1500);
  });

  promise
    .then(function(result) {
      log(result);
    })
    .catch(function(error) {
      log(error);
    })
    .finally(function() {
      log("🏁 Promise settled");
    });
});
```



Promise Chaining

- Promise chaining allows you to run multiple asynchronous operations in sequence
  - each *.then()* receives the result of the previous one.
  - each *.then()* returns a new Promise

```
doTask1()
  .then(doTask2)
  .then(doTask3)
  .then(finalResult)
  .catch(handleError);
```

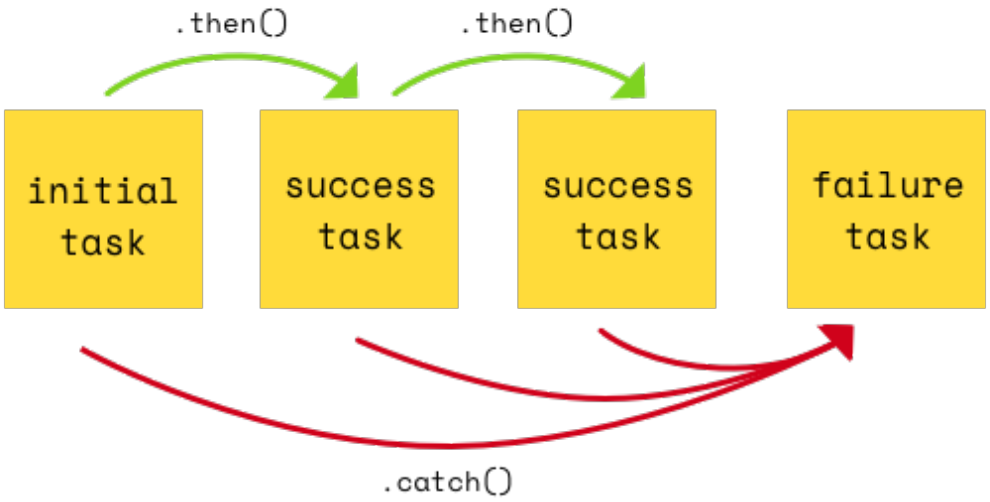
Promise Chaining

```
step1()
  .then(function (result1) {
    log("→ Result from Step 1: " + result1);
    return step2(result1);
  })
  .then(function (result2) {
    log("→ Result from Step 2: " + result2);
    return step3(result2);
  })
  .then(function (finalResult) {
    log("🎉 Final Result: " + finalResult);
  })
  .catch(function (error) {
    log("❌ Error: " + error);
  })
  .finally(function () {
    log("🏁 Chain completed (fulfilled or rejected).");
  });
```

Promise - Handling Errors Globally

- .catch() method handles all errors that occurs in the entire Promise chain.

```
promise
  .then(function(result) { /* success */ })
  .then(function(next) { /* another step */ })
  .catch(function(error) { /* handle error */ });
```



Promise - Handling Errors Globally

```
step1()
  .then(function(result1) {
    return step2(result1);
  })
  .then(function(result2) {
    return step3(result2);
  })
  .then(function(finalResult) {
    log("🎉 Success: " + finalResult);
  })
  .catch(function(error) {
    log("❗ Caught error: " + error);
  })
  .finally(function() {
    log("🏁 Process finished (success or failure).");
  });
```

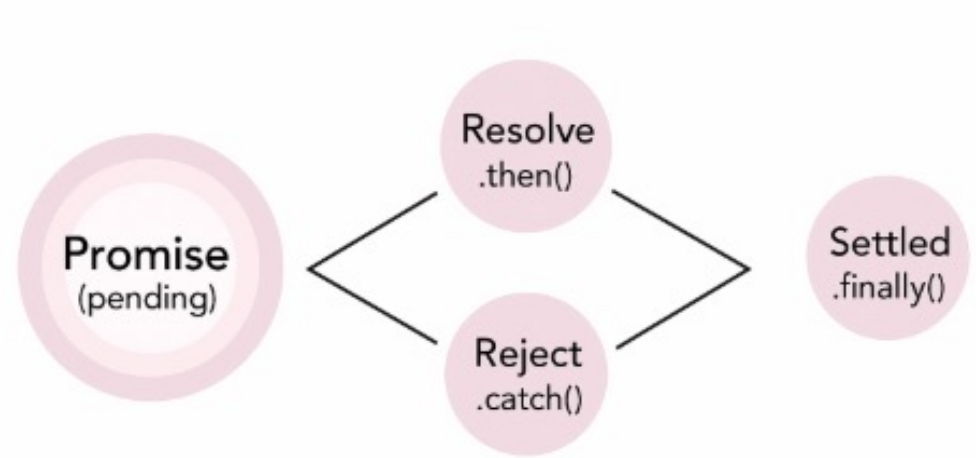
```
⏳ Starting Promise chain...

✅ Step 1: Connected to server
⚙️ Step 2: Processing data...
❗ Caught error: ❌ Step 2 failed: Processing error
🏁 Process finished (success or failure).
```

Promise - finally()

- .finally() method is called after a Promise settles, no matter whether the result was fulfilled or rejected.

```
promise
  .then(function(result) { /* handle success */ })
  .catch(function(error) { /* handle error */ })
  .finally(function() { /* cleanup */ });
```



Promise all

- Promise.all() runs multiple Promises at the same time and waits until all of them are fulfilled or one fails.

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

const p1 = fetch("https://jsonplaceholder.typicode.com/users/1");
const p2 = fetch("https://jsonplaceholder.typicode.com/users/2");
const p3 = fetch("https://jsonplaceholder.typicode.com/users/3");

Promise.all([p1, p2, p3])
  .then(() => console.log("All done"))
  .catch(err => console.error(err));

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

```
Start
End
All done
```

When a Promise fails, do other Promises continue to run?

Callback vs Promise

- Both Callbacks and Promises are used to handle asynchronous operations in JavaScript.
- Promises make async flow easier to read, chain, and handle errors.

Feature	Callback	Promise
Definition	A function passed into another function to be executed later.	An object representing the eventual result of an async operation.
Syntax	Function nesting	Chainable using .then() and .catch().
Readability	Can lead to “Callback Hell.”	Cleaner and more sequential flow.
Used	Older style (before ES6).	Standard for async code in modern JS

HUST

hust.edu.vn fb.com/dhbkhn

3. Async/Await

Limitations of Promise Chaining

- .then() chains can still become hard to read and maintain — especially when tasks depend on previous results.

```
fetchUser()
  .then(function(user) {
    return fetchPosts(user.id);
  })
  .then(function(posts) {
    return fetchComments(posts[0].id);
  })
  .then(function(comments) {
    console.log("✅ Done:", comments);
  })
  .catch(function(error) {
    console.error("❌ Error:", error);
  });
```

```
async function showComments() {
  try {
    const user = await fetchUser();
    const posts = await fetchPosts(user.id);
    const comments = await fetchComments(posts[0].id);
    console.log("✅ Done:", comments);
  } catch (error) {
    console.error("❌ Error:", error);
  }
}
```

async/await: Introduction

- async/await are modern JavaScript keywords (introduced in ES8 / ES2017)
- asynchronous code look and behave like synchronous code, while still being non-blocking
- async function
  - Declares a function that always returns a Promise.
  - Inside it, you can use the await keyword.
- await
  - Pauses the execution until the Promise resolves (or rejects).
  - Returns the resolved value.

async/await: Introduction

```
async function functionName() {
  await somePromise;
}
```

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

async function run() {
  console.log("Start");
  await wait(1000);
  console.log("After 1 second");
}

run();
```

async Keyword

- async keyword is used to declare an asynchronous function.
- Every async function always returns a Promise, no matter what the function explicitly returns.

```
async function myFunction() {
  // some code
  return "Hello";
}
```

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

Both functions return a Promise that resolves with "Hello".

await Keyword

- await keyword is used inside an async function to pause execution until a Promise is settled

```
let result = await somePromise;
```

```
function fetchData() {
  return new Promise(resolve =>
    setTimeout(() => resolve("📦 Data received"), 1000)
  );
}

async function load() {
  console.log("Before await");
  const data = await fetchData();
  console.log("After await:", data);
}

load();
console.log("This log runs first!");
```

```
Before await
This log runs first!
After await: 📦 Data received
```

await (ES2022)

- From ES2022, we can use await in top-level inside ES modules

Before ES2022

```
(async () => {
  const res = await fetch("/data.json");
  const data = await res.json();
  console.log(data);
})();
```

From ES2022

```
// Only works in ES modules
const response = await fetch("/data.json");
const data = await response.json();
console.log(data);
```

```
<script type="module">
  try {
    console.log("⏳ Fetching data...");
    const res = await fetch("https://jsonplaceholder.typicode.com/users/1");
    const user = await res.json();
    console.log("✅ User loaded:", user.name);
  } catch (err) {
    console.error("❌ Fetch failed:", err);
  }
</script>
```

Callback - Promise - Async/Await

Aspect	Callback	Promise	Async/Await
Concept	Pass a function as an argument	Represents a future value	Looks and behaves like synchronous.
Syntax	Nested callbacks	.then() calls	Cleaner syntax with async and await
Error	Handled manually inside callbacks.	Handled with .catch() method.	Handled with try...catch block.
Readability	Hard to read and maintain.	Easier to read than nested callbacks.	Most readable, looks.
Advantage	Simple for one-time asynchronous logic.	Better structure and chaining.	Clean, synchronous -like syntax

Discussion

Advanced DOM and Forms

1. Why do we need to use event.preventDefault() when handling a form's submit event?
2. What's the difference between innerHTML and textContent?
3. What is Event Delegation and what is one advantage of it?
4. What's the difference between Debouncing and Throttling?

Discussion

Callback and Promise

1. What does it mean that JavaScript is “single-threaded”?
2. Explain how the **Event Loop** works in simple terms.
3. What are states of a Promise?
4. What’s the difference between .then() and .catch() in Promises?

Discussion

Async and Await

1. What does an async function always return?
2. How does the await keyword work?
3. What is “Top-Level Await”?
4. Compare Callback, Promise, and Async/Await.

Exercises

- Ex1. Real-Time Form Validation
- Goal: DOM access, form events, and dynamic validation.
- Task:
  - Create a form with inputs: name, email, and age.
  - When typing:
    - Show a red message if the field is empty.
    - Validate email format using a regular expression.
    - Display “All good!” only when all fields are valid.

Exercises

- Ex2. Handling Multiple Select Values
- Goal: DOM access, form events, and dynamic validation.
- Task
  - When the user clicks the Show Selected Skills button
  - Read all selected values.
  - Display them inside an element
  - If no option is selected, display “No skill selected.”

Exercises

- Ex3. Simulated Data Loading (Callback)
- Goal: Understand asynchronous behavior using callbacks.
- Task:
  - Write a function loadData(callback) that
  - Waits 2 seconds (via setTimeout).
  - Returns fake data (e.g., {name: "Alice"}).
  - Then call it and print the data using the callback.

Exercises

- Ex 4 — Promise Chain Practice
- Goal: Convert callback logic to Promises.
- Task:
  - Create three functions returning Promises:
  - fetchUser() → resolves {id: 1, name: "Alice"}
  - fetchPosts(userId) → resolves an array of posts
  - displayPosts(posts) → logs post titles Chain them with .then() calls.

- Ex 5 — Async/Await
- Goal: Convert Promises to async/await