PART 4

MASYNCTECH
SCHOOL

FULLSTACK WEB DEV

JAVASCRIPT

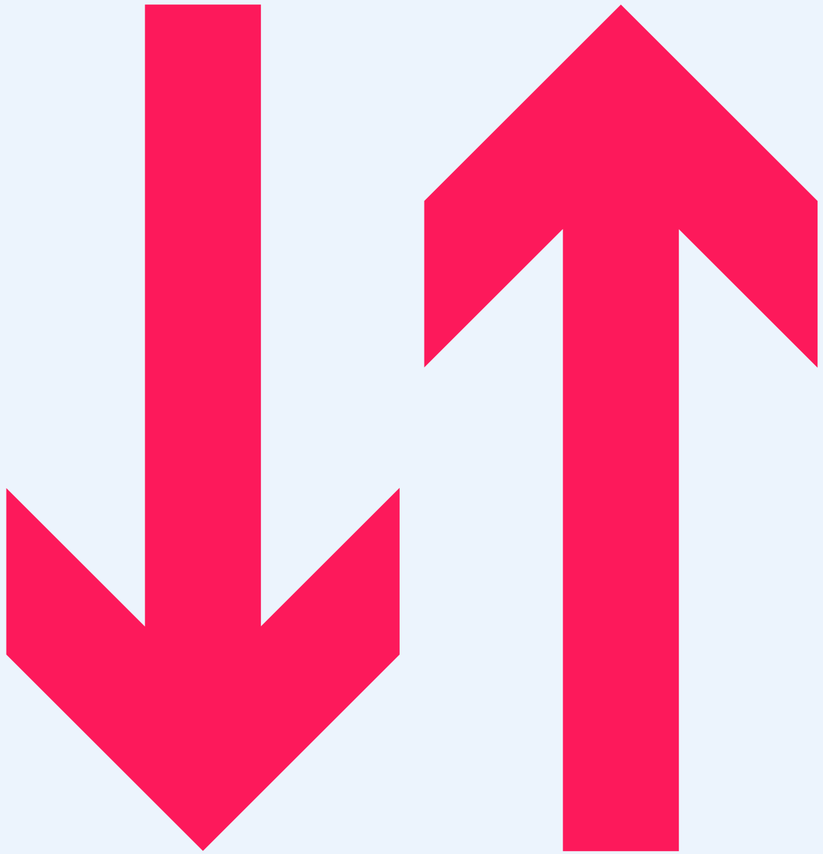
JS

ASYNC PROGRAMMING

MASYNCTECH MASYNCTECH www.masynctech.com

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

WHAT'S IT?

JS

ADVANCED

WHAT IS ASYNCHRONOUS PROGRAMMING?

MASYNCTECH SCHOOL

High-Level Explanation

- Asynchronous programming: Non-blocking operations
- Tasks run independently of main program flow
- Ideal for I/O-bound, network, and long-running tasks

Basic Explanation

- Analogy: Synchronous programming like fast-food queue
- Orders block others, wait to complete
- Asynchronous programming: Order, step aside
- Continue tasks while waiting for others

Best Practices

- Keep MongoDB driver up to date for compatibility, security.
- Prefer officially supported, community-maintained drivers for updates, support.

Deep Dive

- Synchronous programming: Sequential execution
- Each operation waits for the previous one
- Limiting for unpredictable tasks
- Asynchronous decouples initiation from execution
- Allows parallel execution while waiting

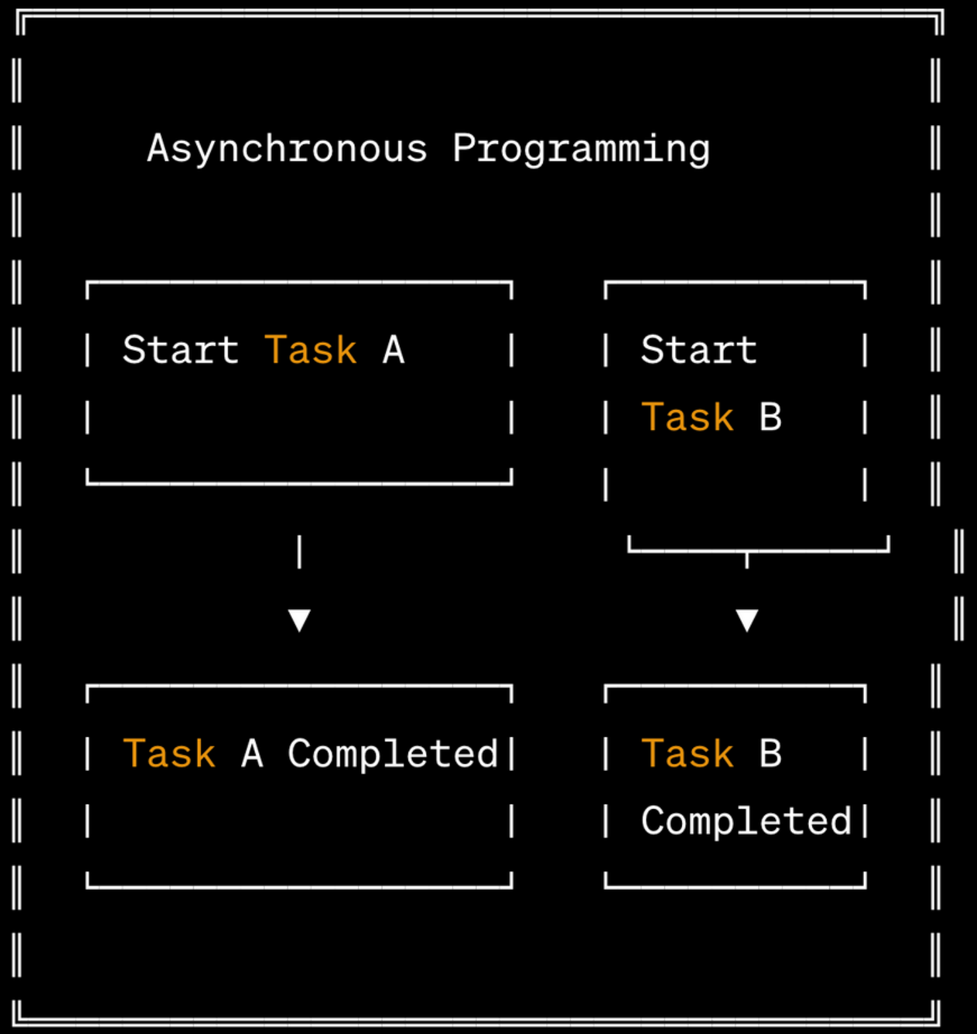
When to use?

- Tasks are time-consuming, no need for sequential completion.
- Building scalable network apps.
- Enhancing UI responsiveness.

Summary

- Asynchronous programming: Independent task execution
- Boosts efficiency, responsiveness
- Ideal for I/O-bound, long tasks
- Best practices like Promises and error handling enhance maintainability.

WHAT IS ASYNCHRONOUS PROGRAMMING?



MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

IT'S IMPORTANCE

JS

ADVANCED

WHY ASYNCHRONOUS PROGRAMMING IS IMPORTANT

MASYNCTECH SCHOOL

High-Level Explanation

- Asynchronous programming is crucial in JavaScript.
- Vital for web browsers and Node.js.
- Non-blocking, event-driven architecture enhances performance, usability.

Basic Explanation

- Analogy: Synchronous app freezes while streaming.
- Annoying, can't do anything else.
- Asynchronous: Stream and browse simultaneously.
- Enhances user experience and multitasking.

Best Practices

- Prefer async/await for readability.
- Stick to one approach (callbacks, Promises, async/await).
- Leverage libraries like Axios, fs.promises.
- Implement robust error handling for catch/rejections.

Deep Dive

- JavaScript is single-threaded.
- Synchronous tasks freeze system with time-consuming ops.
- Asynchronous programming prevents freezing.
- Ensures responsive interfaces, resource management.

When to use?

- JavaScript async crucial for:
  - Web dev: Interactions, APIs, DOM without freezing.
  - Node.js backend: Efficient I/O tasks.
  - Real-time apps: Chat, gaming, collaborative tools.

Summary

- JavaScript relies on asynchronous programming for responsive, non-blocking UI and efficient resource use.
- Critical for I/O-bound and concurrent tasks in frontend and backend.
- Best practices ensure code maintainability, user experience.

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

SYNCHRONOUS VS ASYNCHRONOUS EXECUTION

JS

ADVANCED

SYNCHRONOUS VS. ASYNCHRONOUS EXECUTION

MASYNCTECH SCHOOL

High-Level Explanation

- Synchronous execution: Tasks sequentially, one by one.
- Wait for each to finish before starting next.
- Asynchronous execution: Tasks run in parallel.
- No need to wait for one to finish before starting another.

Deep Dive

- **Synchronous Execution:**
  - Linear, predictable flow.
  - Blocks until current operation finishes.
  - Simple, easy to follow and debug.
  - Can cause performance issues with time-consuming tasks.

- **Asynchronous Execution:**
  - Non-linear, tasks start/complete differently.
  - Doesn't block; allows concurrent tasks.
  - More complex with callbacks, promises, async/await.
  - Ideal for time-consuming or external-dependent operations.

Basic Explanation

- **Synchronous:** Single-window fast-food, wait for food, next person orders. Large orders delay others.

- **Asynchronous:** Multi-window fast-food, order, move aside, next person orders. Collect when ready, no delays.

SYNCHRONOUS VS. ASYNCHRONOUS EXECUTION

MASYNCTECH
SCHOOL

When to use?

- **Synchronous:** Ideal for ordered execution, simplicity. E.g., basic algorithms, calculations, string manipulations.

- **Asynchronous:** Ideal for I/O-bound, concurrent tasks, independent of others. E.g., web dev (API calls, DOM), backend (DB queries).

Best Practices

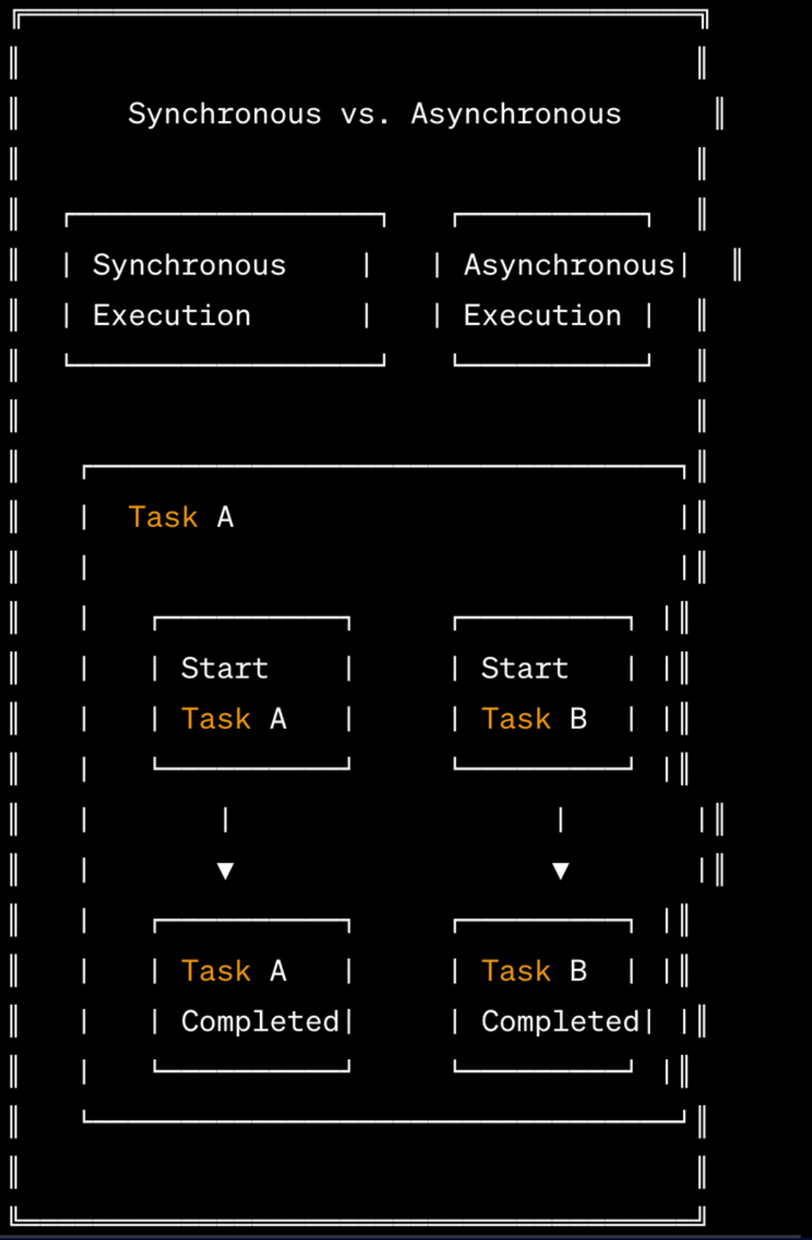
- **Synchronous**
  - Watch for performance bottlenecks.
  - Avoid long-running operations; they can freeze the app.

- **Asynchronous:**
  - Handle errors in async code.
  - Prevent "callback hell" with promises or async/await.
  - Be cautious of race conditions.

Summary

- JavaScript relies on asynchronous programming for responsive, non-blocking UI and efficient resource use.
- Critical for I/O-bound and concurrent tasks in frontend and backend.
- Best practices ensure code maintainability, user experience.

SYNCHRONOUS VS. ASYNCHRONOUS EXECUTION



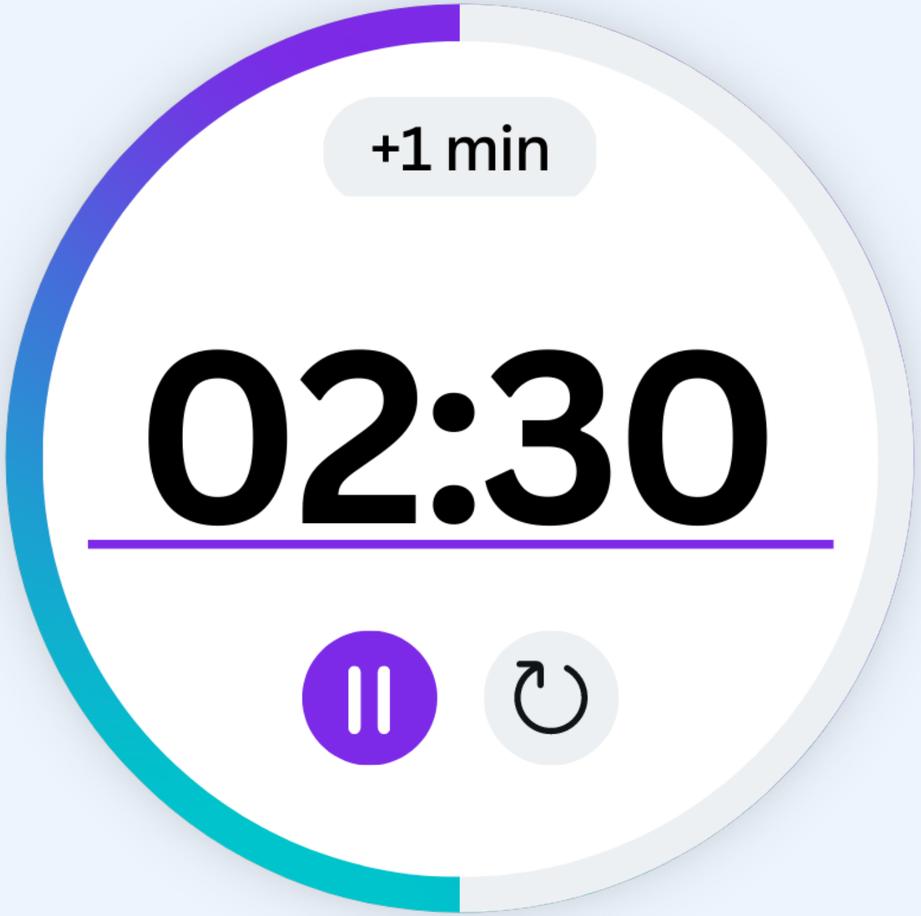
MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

SETTIMEOUT

VS

SETINTERVAL



JS

ADVANCED

SETTIMEOUT AND SETINTERVAL FUNCTIONS

MASYNCTECH SCHOOL

High-Level Explanation

- `setTimeout`: One-time task after delay
- `setInterval`: Recurring task at intervals

Basic Explanation

- setTimeout: Like a one-time alarm reminder
- Rings once and turns off after execution

- **setInterval:** Like a daily repeating alarm
- Rings every day until manually turned off

Best Practices

- Clear intervals/timeouts to prevent issues, memory leaks
- Beware of exact timing reliability due to JS single-threading
- For precision, explore Web Workers or other options

Deep Dive

-setTimeout:
- Schedules callback after specified delay (ms)
- Returns ID for cancellation with `clearTimeout`

- **setInterval:**
- Schedules callback at fixed intervals (ms)
- Returns ID for stopping repetitions with `clearInterval`

When to use?

- **setTimeout:** Ideal for one-time delayed tasks
- Notify after delay, delayed API calls, debouncing input

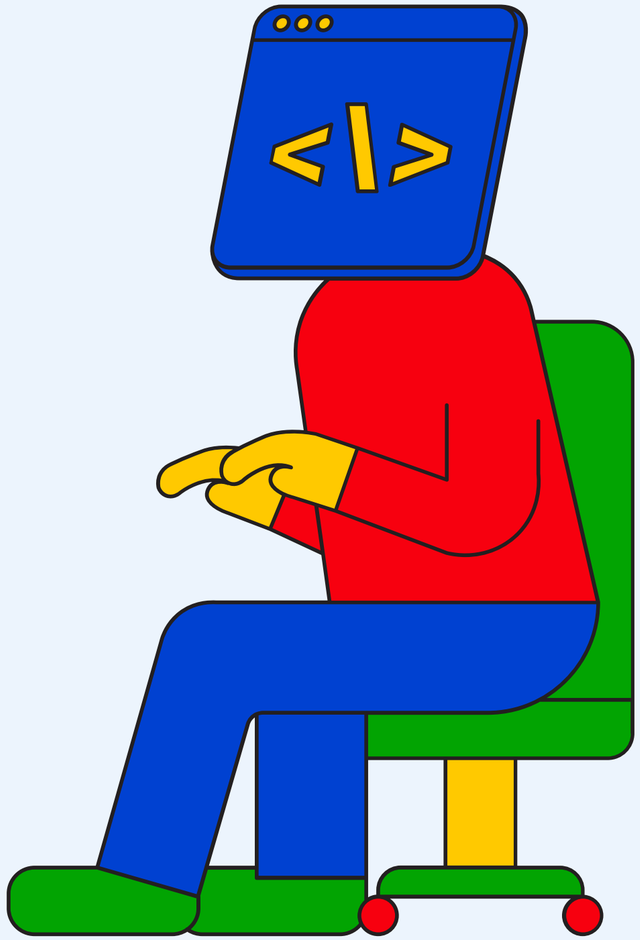
- **setInterva**l: Ideal for recurring tasks at intervals
- Update live clock, poll API, auto-save content regularly

Summary

- `setTimeout` and `setInterval`: Vital for timed operations
- `setTimeout`: For one-time delayed tasks
- `setInterval`: For recurring tasks at intervals
- Proper usage and management ensure efficiency.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

BLOCKING VS NON-BLOCKING CODE

JS

ADVANCED

BLOCKING VS NON-BLOCKING CODE

MASYNCTECH SCHOOL

High-Level Explanation

- Blocking code: Halts program until completion
- Non-blocking code: Allows program to continue without waiting

Basic Explanation

- **Blocking:** Like waiting in line at a slow coffee order
- **Non-blocking**: Like using multiple self-checkout kiosks, tasks proceed independently

Best Practices

- Avoid long-blocking operations in responsive contexts (UI).
- Use non-blocking code for I/O, especially in Node.js (event-driven).
- Handle errors in non-blocking ops (callbacks, promises, async/await).

Deep Dive

- Blocking Code: Operation completes before next one
- Non-blocking Code: Initiates task and moves on without waiting
- Common in JavaScript for I/O operations

When to use?

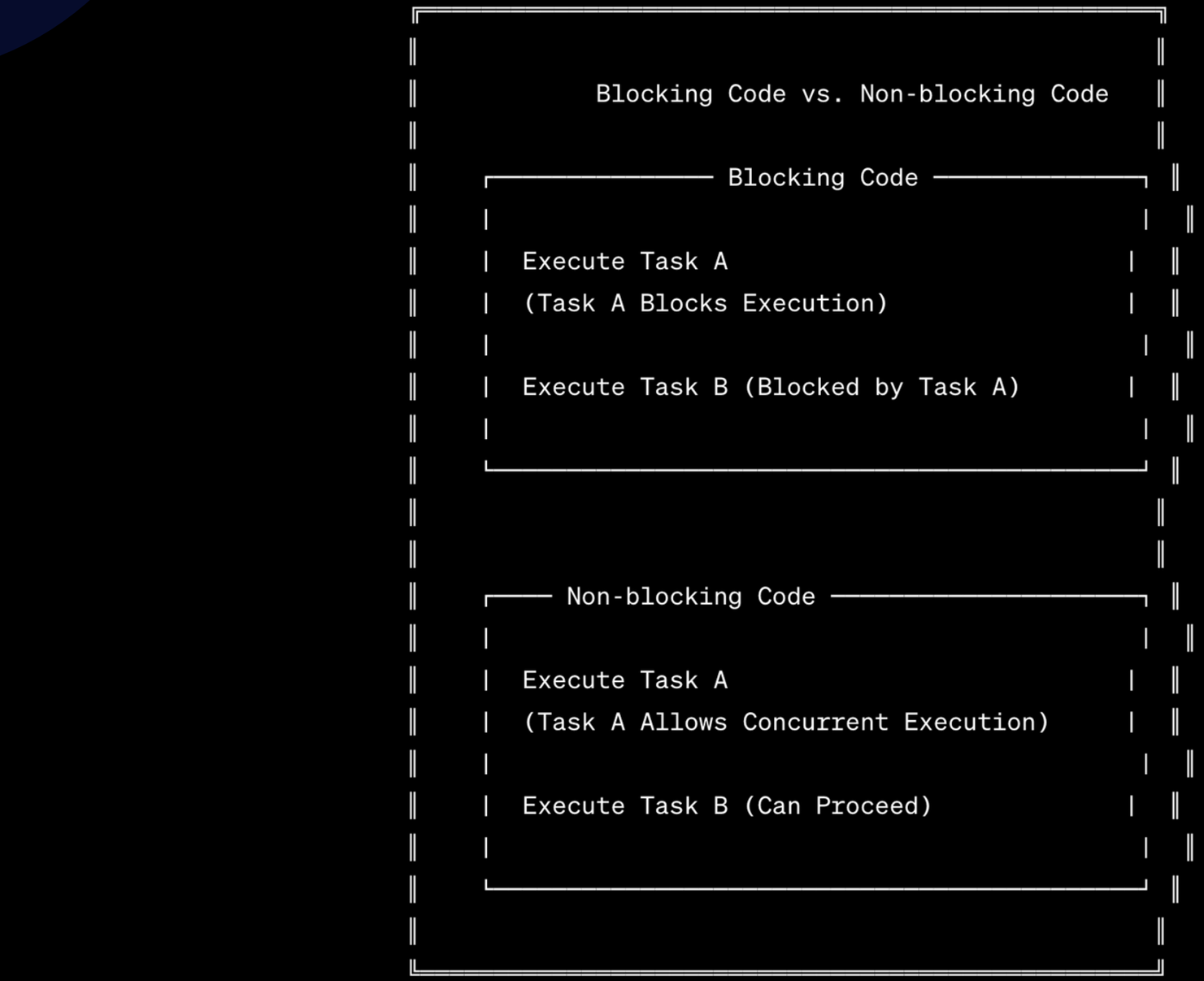
- **Blocking**: Suited for dependent, sequential operations
- Often in scripting tasks, order matters

- **Non-blocking:** Ideal for independent, parallel tasks
- Used in responsive apps, web servers, etc.

Summary

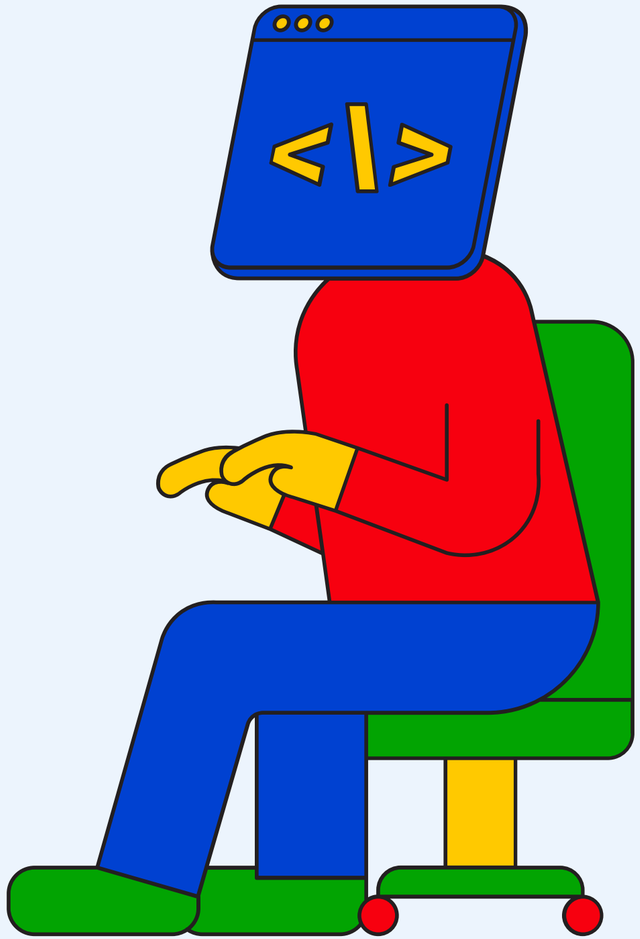
- Blocking and non-blocking code impact execution sequence.
- Blocking waits, non-blocking moves on.
- Proper use ensures efficiency and responsiveness.

BLOCKING VS NON-BLOCKING CODE



MASYNCTECH
SCHOOL

CODE DEMO



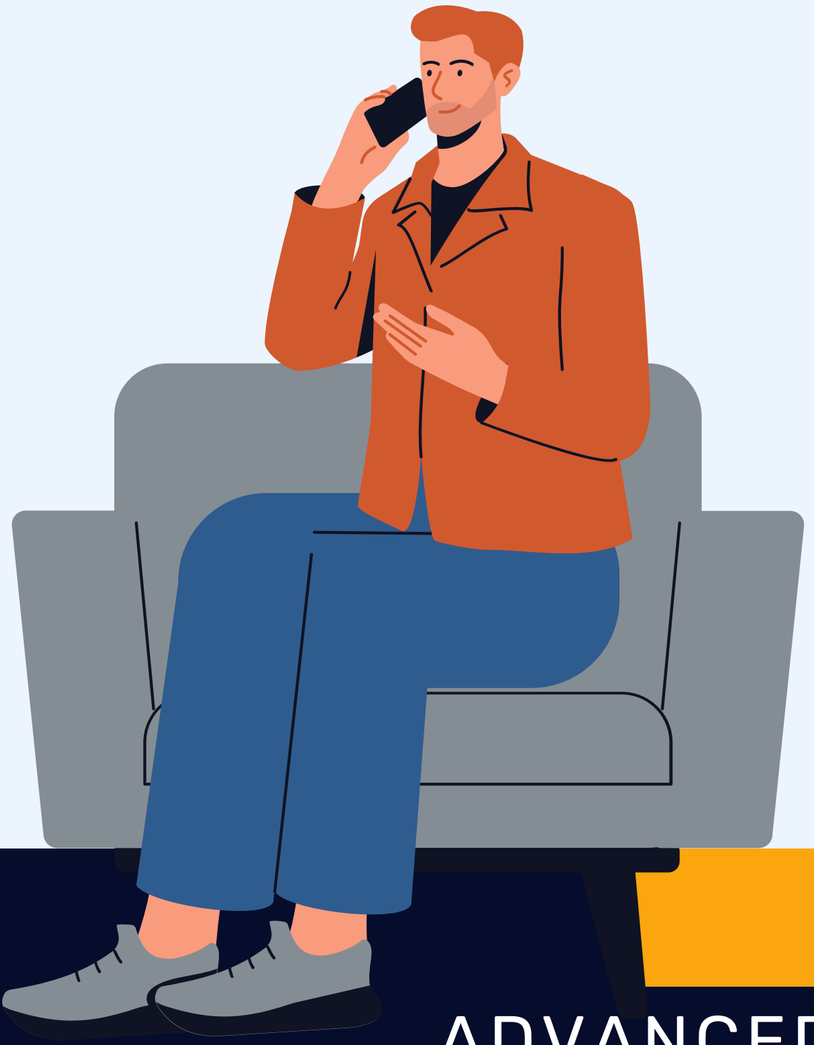
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

CALLBACKS & EVENT LOOP



JS

ADVANCED

MASYNCTECH
SCHOOL

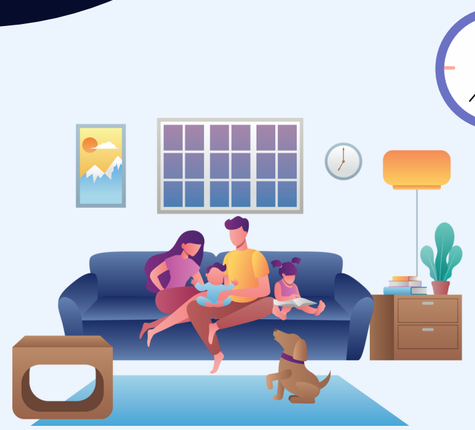
ASYNCHRONOUS PROGRAMMING

UNDERSTANDING CALLBACK FUNCTION

JS

ADVANCED

UNDERSTANDING CALLBACK FUNCTIONS



+233 097533

Callback

UNDERSTANDING CALLBACK FUNCTIONS

MASYNCTECH SCHOOL

High-Level Explanation

- Callback function: Passed as arg, executed after another function
- Fundamental in JavaScript, crucial for async management

Basic Explanation

- Analogy: Pizza delivery callback
- Give phone number (callback) for delivery notification
- They call (execute) when pizza (data) is ready

Best Practices

- Handle errors in callback functions (often first argument).
- Avoid callback hell (nested callbacks); use Promises or `async/await`.
- Be aware of non-linear code flow in callbacks; debug carefully.

Deep Dive

- JavaScript functions: First-class citizens, passed like values
- Pattern: Pass callback into functions for behavior extension
- Callbacks handle async operations, like API responses

When to use?

- Callbacks used for:
  - Asynchronous operations (file read, API requests, timeouts)
  - Logic/task separation in functions (array methods)
  - DOM event listeners

Summary

- Callback functions in JS: Dynamic, async code execution.
- Passed as args, executed later or under conditions.
- Provide flexibility, manageability; prioritize clarity, error handling.

Defining a Callback Function

Syntax

```
function callbackFunction(parameter) {
  // Code to be executed
}
```

- callbackFunction: Function passed as callback, contains code to execute.

- parameter: Parameters accepted by callback when invoked.

Passing a Callback Function to Another Function

```
function higherOrderFunction(callback) {
  // Code before calling callback

  callback();

  // Code after calling callback
}

higherOrderFunction(callbackFunction);
```

- higherOrderFunction: Function accepting a callback as parameter, controls async operations.

- callback(): Invokes callback function within higher-order function.

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

CALLBACK HELL

JS

ADVANCED

CALLBACK HELL (CALLBACK PYRAMID) AND ITS ISSUES

MASYNCTECH SCHOOL

High-Level Explanation

- Callback Hell (Pyramid of Doom): Heavily nested callbacks.
- Common in async programming, subsequent ops wait for previous.
- Issues: Poor readability, maintenance, error-prone.

Basic Explanation

- Analogy: Callback Hell like stacking boxes.
- Tasks depend on previous, instructions nested.
- Hard to understand, change; changes affect all.

Best Practices

- Modularization: Break down callbacks into modular functions.
- Error Handling: Implement error handling.
- Promises/Async-Await: Transition for cleaner code.
- Comments: Document flow and purpose.
- Limit Nesting: Keep nesting levels minimal for readability.

Deep Dive

- Callback Hell: Nested, sequential async ops.
- Common in dependent async ops (file reads, API calls, DB queries).
- Issues: Poor readability, understanding, modification, debugging, maintenance, error-prone.

When to use?

- Avoiding Callback Hell is ideal, but sometimes complex async ops.
- Use solutions like Promises, Async/Await, Generators for better control.

Summary

- Callback Hell: Nested callbacks, hard-to-read code
- Common in multiple dependent async operations
- Mitigate with modularization, error handling, Promises/Async-Await, comments, nesting limits

CALLBACK HELL (CALLBACK PYRAMID) AND ITS ISSUES

Syntax

```
performOperation1(data, function(result1) {
  |
  |--- performOperation2(result1, function(result2) {
  |       |
  |       |--- performOperation3(result2, function(result3) {
  |       |       |
  |       |       |--- // More Nested Callbacks...
  |       |       |
  |       |       });
  |       |
  |       });
  |
  });
```

CALLBACK HELL (CALLBACK PYRAMID) AND ITS ISSUES

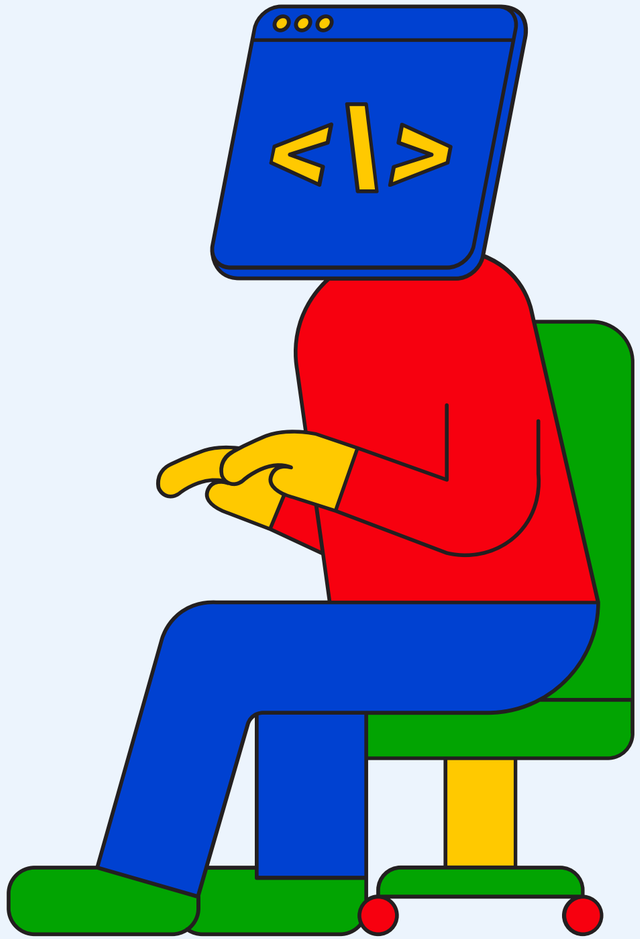
MASYNCTECH
SCHOOL

Syntax

```
fetchUserData(userId, function(userData) {
  |
  |--- fetchUserPosts(userData.id, function(userPosts) {
  |       |
  |       |--- userPosts.forEach(function(post) {
  |       |       |
  |       |       |--- fetchPostComments(post.id, function(comments) {
  |       |       |       |
  |       |       |       |--- // Maybe more nested callbacks for each comment.
  |       |       |       |
  |       |       |       });
  |       |       |
  |       |       });
  |       |
  |       });
  |
  });
```

MASYNCTECH
SCHOOL

CODE DEMO



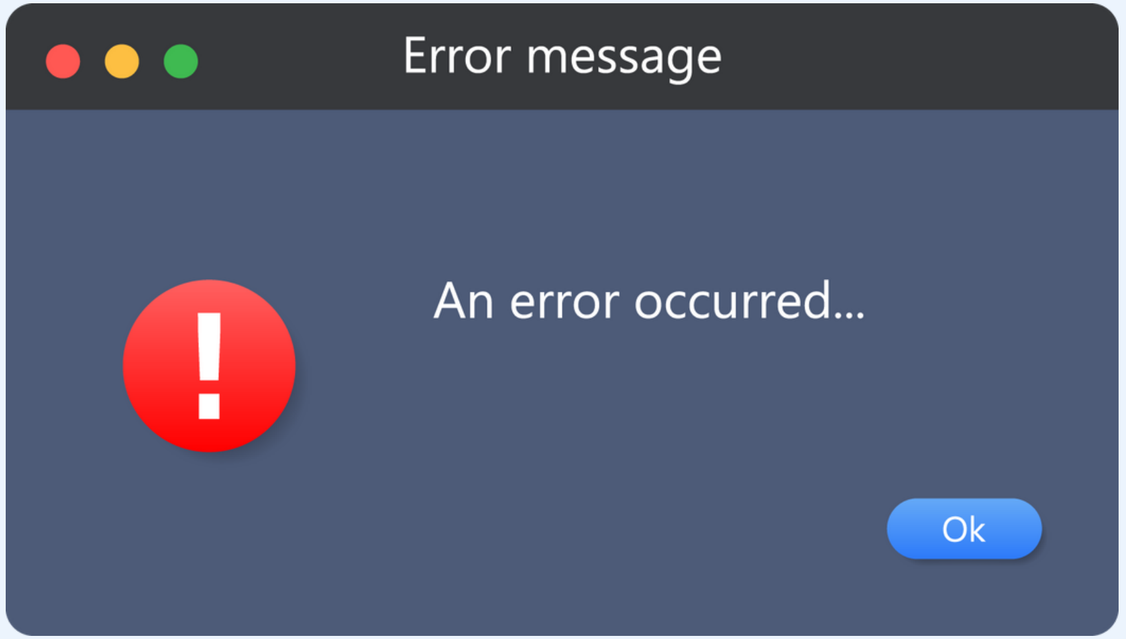
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

HANDLING ERRORS IN CALLBACK



JS

ADVANCED

HANDLING ERRORS IN CALLBACKS

MASYNCTECH SCHOOL

High-Level Explanation

- Robust async JS requires error handling in callbacks.
- Error-first callback style: 1st arg for error, 2nd for result.

Basic Explanation

- Analogy: Error-first callbacks like asking a friend for a task
- Friend calls back, first mentions any problem (error)
- If all good, then shares the task result
- Error known before result

Best Practices

- Check for errors before processing results.
- Propagate errors for proper handling.
- Graceful error handling: Fallbacks, informative messages.
- Minimize excessive callback nesting for simplicity.

Deep Dive

- Effective error management in async callbacks is crucial.
- Error-first callback: Error passed as 1st argument.
- Enables graceful error handling; `null` if no error, result as 2nd arg.

When to use?

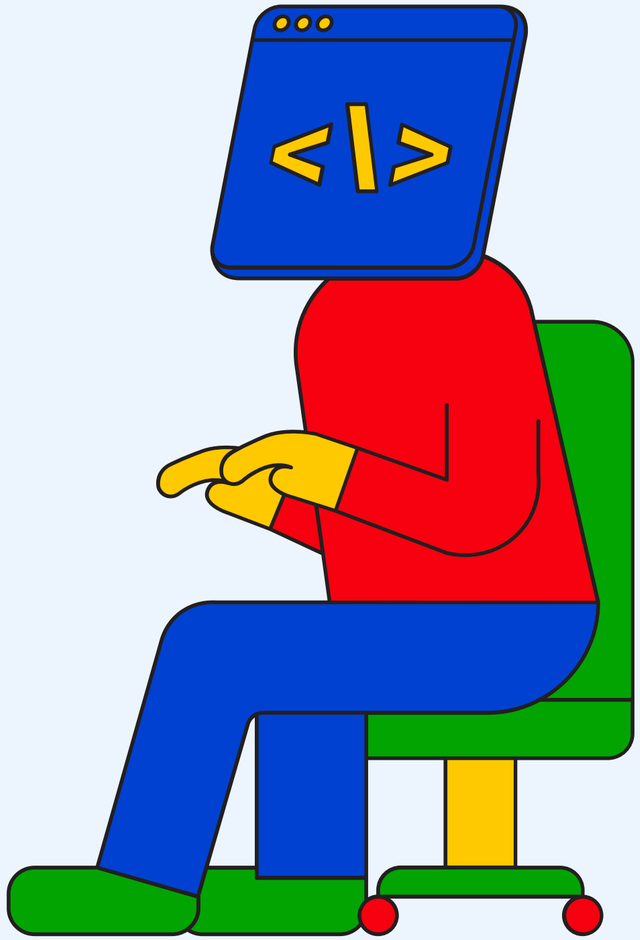
- Error-first pattern valuable for async ops with likely errors.
- Applied in file reading, network requests, database queries.
- Ensures error handling, robust, reliable code.

Summary

- Error handling in async JavaScript is crucial.
- Error-first callback: 1st arg error, 2nd result.
- Ensures error awareness, code robustness, and maintainability.
- Best practices: error checks, propagation, minimal nesting.

MASYNCTECH
SCHOOL

CODE DEMO



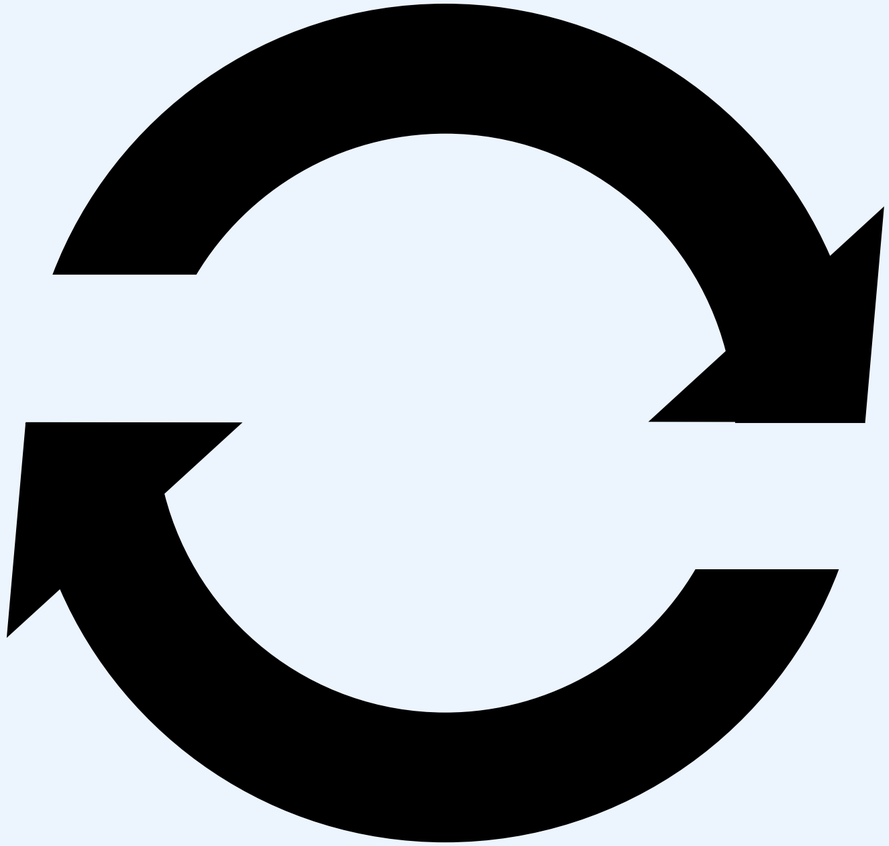
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

EVENT LOOP



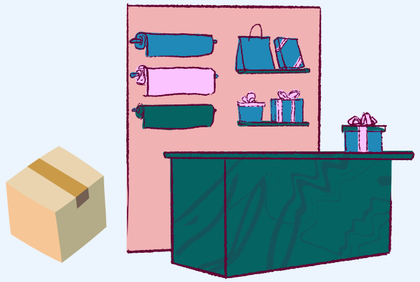
JS

ADVANCED

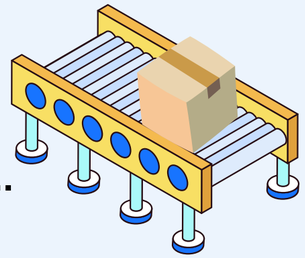
THE JAVASCRIPT EVENT LOOP

MASYNCTECH
SCHOOL

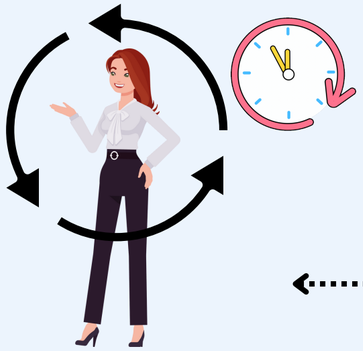
checkout counter(Call Stack)



wrapping station (Web API)



conveyor belt (Callback Queue)



A manager (Event Loop)

THE JAVASCRIPT EVENT LOOP

MASYNCTECH
SCHOOL

High-Level Explanation

- JavaScript Event Loop: Core of async behavior
- Collaborates with Call Stack, Callback Queue, Web APIs
- Orchestrates task execution, keeps JS non-blocking, responsive

Basic Explanation

- Analogy: JavaScript runtime components as a store checkout
- **Call Stack**: Regular items checked out directly.
- **Web APIs:** Special items sent for gift-wrapping (async).
- **Callback Queue:** Wrapped items on conveyor belt (awaiting checkout).
- **Event Loop:** Manager schedules wrapped items once counter is free.

Deep Dive

- JavaScript runtime components:
  - **Call Stack**: Functions executed one at a time, single-threaded.
  - **Web APIs:** Offload async tasks like `setTimeout`.
  - **Callback Queue:** Stores async callback tasks.
  - **Event Loop:** Monitors Call Stack and Callback Queue.
    - Dequeues callbacks to Call Stack when it's empty.

Best Practices

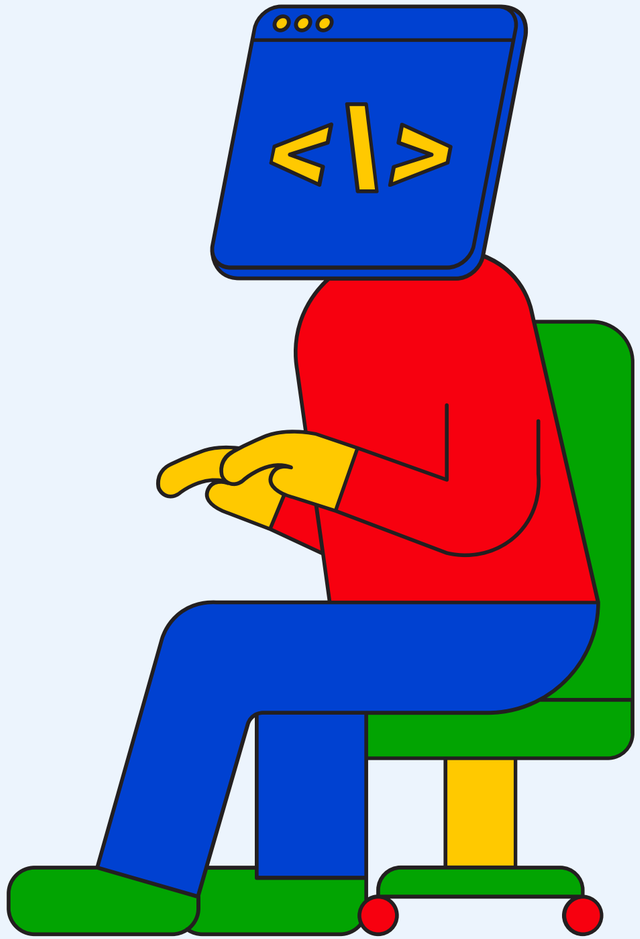
- Avoid long-running tasks to prevent Call Stack blocking.
- For heavy computations, consider Web Workers.
- Handle errors in async code to prevent issues.
- Minimize callback hell; use Promises or async/await for cleaner code.

Summary

- JavaScript runtime uses Call Stack, Web APIs, Callback Queue, Event Loop.
- Manages both sync and async tasks.
- Non-blocking, concurrent handling for smooth user experience.
- Proper coding practices maximize system efficiency.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

PROMISES

JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

INTRODUCTION TO PROMISES

JS

ADVANCED

INTRODUCTION TO PROMISES

initiating a Promise (Creation)

WEB API

waiting period (Pending State)

cancelled (rejected)

Served (fulfilled)

INTRODUCTION TO PROMISES

MASYNCTECH SCHOOL

High-Level Explanation

- JavaScript Promises represent async operation results.
- States: pending, fulfilled, rejected.

Basic Explanation

- Analogy: Promise as ordering a meal at a restaurant
- Order initiation = Promise
- Pending state = Waiting for meal, given a token
- Fulfilled = Meal served, Promise fulfilled
- Rejected = Order canceled, Promise rejected

Best Practices

- Handle errors with `.catch()` to prevent silent failures.
- Return new promises from `then()` for nested promises.
- Minimize promise nesting.
- Chain promises for readability, error handling.
- End promise chain with `catch()` for error handling.

Deep Dive

- Promise: Represents async operation completion or failure.
- Attach callbacks to a returned object.
- Fulfilled (successful) or rejected (failure).
- Returns a promise for the future value, like sync methods.

When to use?

- Promises suitable for async ops like web requests.
- Handle unknown outcomes, non-immediate completion.
- Avoid callback hell, manage async code more comfortably.

Summary

- Promises represent async operation results: pending, fulfilled, rejected.
- Ideal for managing async operations and code.
- Best practices include error handling, avoiding promise nesting.

CREATING PROMISE

Syntax

```
const promise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (/* operation successful */) {
    resolve('Result of the operation');
  } else {
    reject('Error occurred');
  }
});
```

- `new Promise`: Constructor for creating a Promise.
- `resolve`: Function to fulfill Promise with value or another Promise.
- `reject`: Function to reject Promise with a reason (usually an Error).

USING PROMISE

Syntax

```
promise
  .then((result) => {
    // Handle the result of the Promise
    console.log(result);
  })
  .catch((error) => {
    // Handle the error if the Promise is rejected
    console.error(error);
  })
  .finally(() => {
    // Code to be executed regardless of the Promise's fate
  });
```

- `.then()`: Attach callbacks for Promise resolution.
- `.catch()`: Attach callback for Promise rejection.
- `.finally()`: Attach callback for Promise settlement (fulfilled or rejected).

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

CHAINING PROMISES

JS

ADVANCED

CHAINING PROMISES

MASYNCTECH SCHOOL

High-Level Explanation

- Chaining Promises: Sequentially connect promises.
- Each subsequent promise returned from `.then()` or `.catch()` of the previous one.
- Enables handling async tasks in a specific order.

Basic Explanation

- Analogy: Planning a day out with chained activities.
- Each activity as a Promise.
- Chained with `.then()` for sequential execution.

Best Practices

- Return results in `.then()` to keep the chain active.
- Handle errors with a final `.catch()` in the chain.
- Use `.finally()` for code running regardless of outcome.
- Avoid nesting Promises in `.then()`, return them.
- Split complex chains into smaller functions for readability.

Deep Dive

- Promise chaining: `.then()` handlers invoked sequentially.
- Value returned passed to the next `.then()`.
- New Promise from handler delays next `.then()`.
- Readable, maintainable async code, avoids callback hell.

When to use?

- Chaining Promises valuable for ordered async tasks.
- Useful for tasks dependent on previous outcomes.
- Common pattern for APIs, databases, async ops in JavaScript.

Summary

- Promise chaining: Execute async tasks sequentially.
- Each waits for the previous one to resolve.
- Enhances async code readability, maintainability.
- Ideal for task dependency scenarios.

CHAINING PROMISES

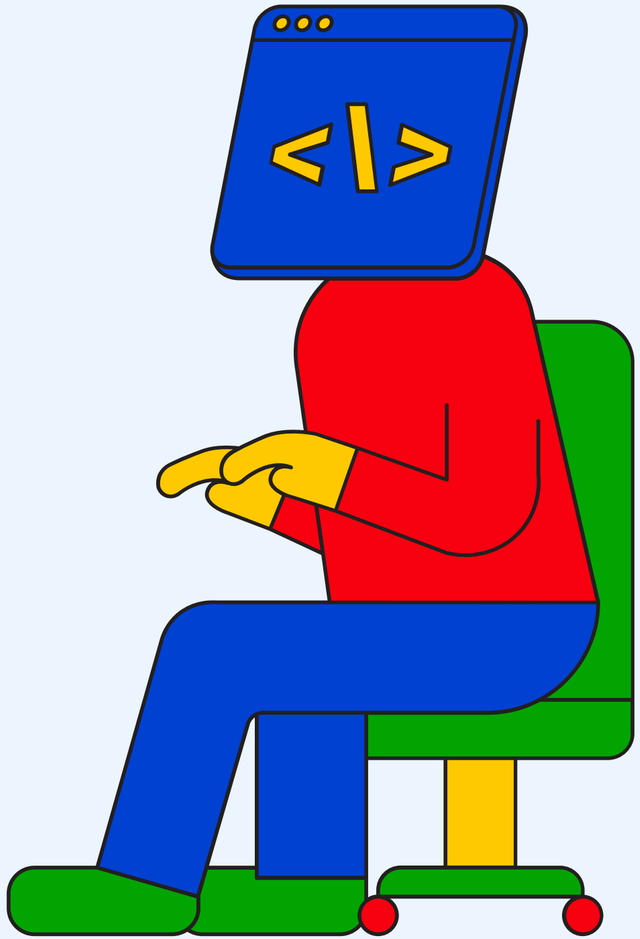
Syntax

```
firstPromise()
  .then((firstResult) => {
    // Handle the result of the first promise
    // Return a value or a new promise
    return secondPromise(firstResult);
  })
  .then((secondResult) => {
    // Handle the result of the second promise
    // Return a value or a new promise
    return thirdPromise(secondResult);
  })
  .catch((error) => {
    // Handle any error that occurred in any of the above promises
    console.error(error);
  })
  .finally(() => {
    // Execute some code, whether the promises were fulfilled or rejected
  });
```

- Placeholder function names: `firstPromise`, `secondPromise`, `thirdPromise`.
- `.then()`: Returns a Promise, handles success and failure.
- `.catch()`: Returns a Promise, handles chain rejection.
- `.finally()`: Returns a Promise, executes regardless of fulfillment or rejection.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

Promise.all()
&
Promise.race()

JS

ADVANCED

PROMISE.ALL AND PROMISE.RACE

MASYNCTECH SCHOOL

High-Level Explanation

- `Promise.all`: Wait for all Promises to resolve.
- `Promise.race`: Wait for the first Promise to resolve or reject.

Basic Explanation

- Analogy: `Promise.all` vs. `Promise.race`
- `Promise.all`: Waiting for friends to arrive at a meetup.
- `Promise.race`: Foot race, first to finish wins, others don't matter.

Best Practices

- `Promise.all`: Handle potential rejections to avoid overall rejection.
- `Promise.race`: Settles quickly but doesn't affect other pending Promises.

Deep Dive

-Promise.all:

- Takes an iterable of Promises.
- Returns a Promise that resolves when all input Promises resolve.
- Returns an array of resolved values in the same order as input.
- Rejects immediately if any input Promise rejects.

- **Promise.race:**
  - Takes an iterable of Promises.
  - Returns a Promise that settles as soon as one input Promise settles.
  - Value or reason from the first settled input Promise.

When to use?

-Promise.all: Use for multiple ops dependent on each other.

- Promise.race: Use for fastest result, e.g., timeouts, resource fetching.

Summary

- `Promise.all`: All Promises must resolve before proceeding.
- `Promise.race`: Any one Promise settling is sufficient.
- Use based on specific task requirements.

Syntax

`Promise.all

```
Promise.all(iterable)
  .then(resultArray => {
    // handle an array of results
  })
  .catch(error => {
    // handle an error
  });
```

- `iterable`: Collection of Promise objects.
- `resultArray`: Array with resolved values in input order.
- `.catch(error => {...})`: Handles rejections from any input Promise.

Syntax

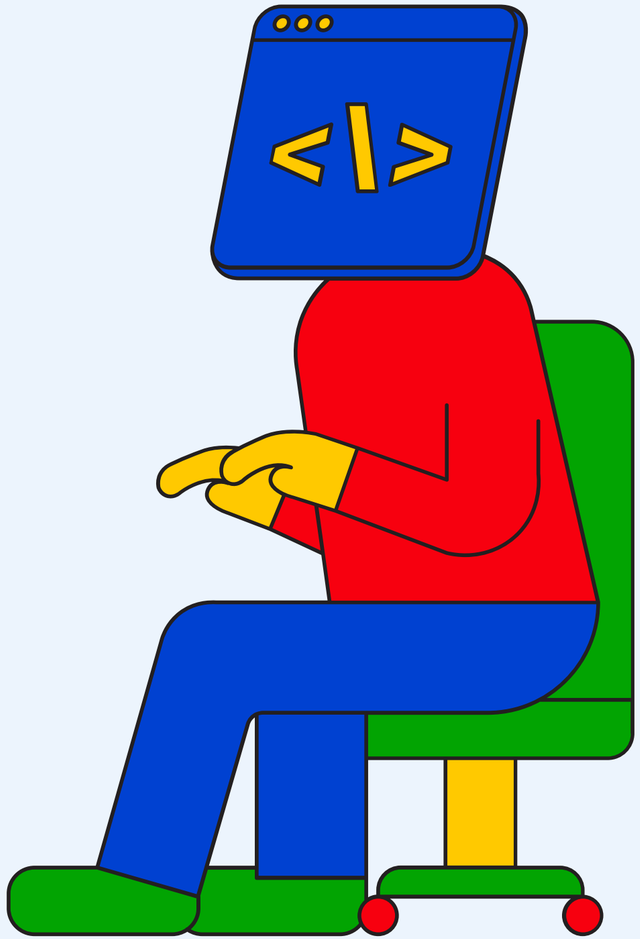
`Promise.all

```
Promise.race(iterable)
  .then(firstResolvedValue => {
    // handle the value of the first resolved Promise
  })
  .catch(error => {
    // handle the error of the first rejected Promise
  });
```

- `iterable`: Collection of Promise objects.
- `firstResolvedValue`: Value from the first resolved Promise.
- `.catch(error => {...})`: Handles rejection from the first rejected Promise.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

ASYNC AWAIT

JS

ADVANCED

INTRODUCTION TO ASYNC/AWAIT

MASYNCTECH SCHOOL

High-Level Explanation

- Modern for async operations.
- Based on Promises.
- `async` for async function.
- `await` pauses until Promise settles.
- Improves code readability, maintainability.

Basic Explanation

- Analogy: Async/Await like cooking
- Boiling pasta is async task
- Chopping veggies done concurrently
- Wait for pasta, do other tasks in coding

Best Practices

- Try/catch for error handling around `await`.
- Avoid `await` in loops for sequentiality.
- `Promise.all` for concurrent `await`.
- `await` inside `async` functions only.

Deep Dive

- Synchronous code runs line by line.
- Async needed for time-consuming tasks.
- Async/Await: Comfortable async code.
- `async` function returns Promise.
- `await` inside `async` for waiting on Promises.
- Code appears synchronous, acts async.
- Syntactic sugar over Promises.

When to use?

- Async/Await for multiple async tasks.
- Sequential or completion handling.
- Examples: API calls, file I/O, database ops.

Summary

- Clean, readable async code in JavaScript.
- Builds on Promises, handles tasks sequentially.
- Simplifies error handling.
- Use for async ops, follow best practices.
- Maintain neat, maintainable code.

MASYNCTECH
SCHOOL

Syntax

async` Function

```
async function functionName() {
  // return promise
}
```

- `async`: Identifies function as asynchronous.
- `functionName()`: Name of the async function.

`await` Expression

```
let result = await functionName();
```

- `let result`: Variable for storing resolved Promise value.
- `await`: Waits for Promise settlement.
- `functionName()`: A Promise-returning function, e.g., API fetch.

MASYNCTECH
SCHOOL

Syntax

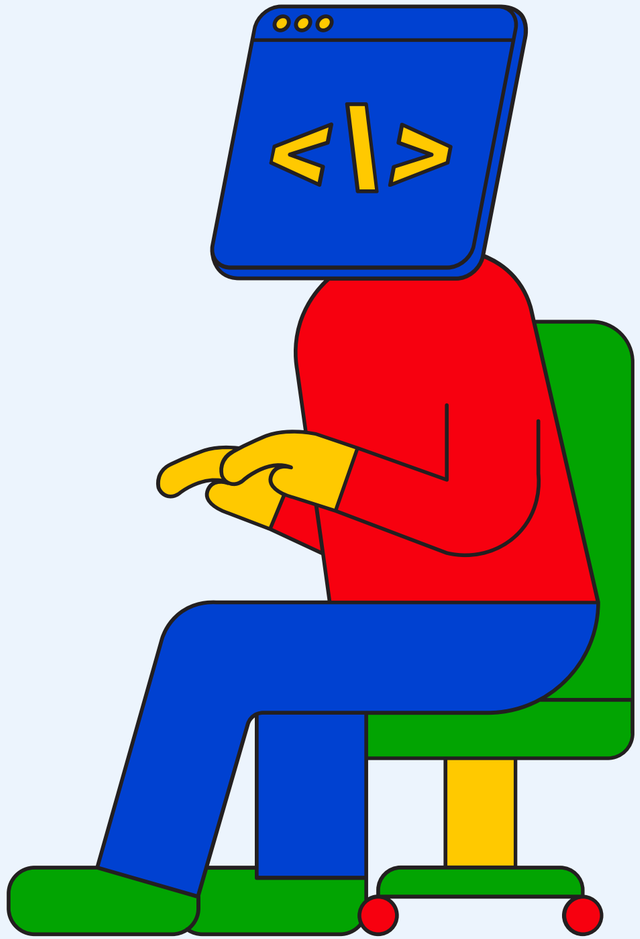
async` Function

```
try {
  let result = await someAsyncFunction();
  // Handle result
} catch (error) {
  // Handle error
}
```

- `try`: Contains async operations.
- `catch (error)`: Handles errors from `try` block.- `functionName()`: Name of the async function.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

ASYNC AWAIT ERROR HANDLING

JS

ADVANCED

ERROR HANDLING WITH ASYNC/AWAIT

MASYNCTECH SCHOOL

High-Level Explanation

- Error handling in Async/Await: Vital for graceful async ops
- Use `try-catch` in `async` functions
- Capture and handle errors to prevent crashes, issues

Basic Explanation

- Analogy: Vending machine behavior like error handling
- Payment failure doesn't freeze or confuse.
- Machine informs about error, suggests actions.
- User-friendly behavior mirrors Async/Await error handling.

Best Practices

- Wrap `await` in `try-catch`.
- Specify error messages for debugging, user feedback.
- Avoid mixing Promise and Async/Await error handling.
- Handle errors; don't ignore, consider logging.
- Use `finally` for cleanup, regardless of async outcome.

Deep Dive

- Asynchronous programming can lead to various issues.
- API downtimes, network failures, data format errors can cause problems.
- Unhandled promise rejections can crash or destabilize apps.
- Using `try-catch` with Async/Await:
  - Traps errors
  - Manages errors effectively
  - Offers alternative flows, user messages
- Enhances code robustness, ensures consistent user experience.

When to use?

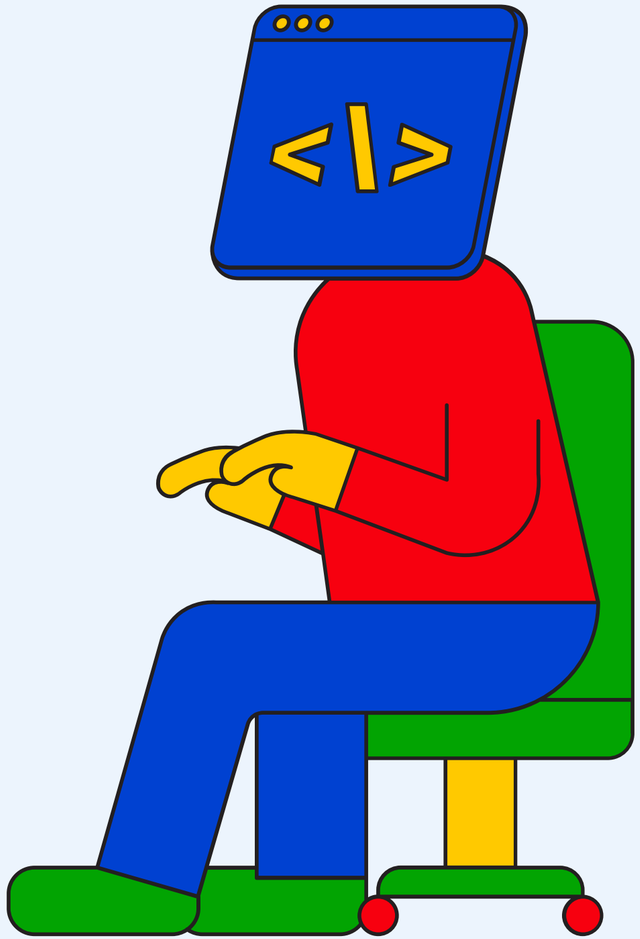
- Error handling in Async/Await crucial for:
  - API requests (server down, incorrect data)
  - Reading files (nonexistent, permission issues)
  - User input (unexpected format, out-of-range)

Summary

- Async/Await error handling ensures graceful async issue management.
- `try-catch` in `async` functions creates robust apps.
- Consistent, user-friendly experiences in error scenarios.
- Error handling isn't just capturing; it guides flow, informs users.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

MAKING HTTP REQUEST AJAX

JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

AJAX

OVERVIEW

JS

ADVANCED

AJAX OVERVIEW

MASYNCTECH SCHOOL

High-Level Explanation

- Ajax: Asynchronous JavaScript and XML
- Technique for dynamic, interactive web apps
- Send/receive data from server without full page reload

Deep Dive

- Traditional pages required full reload for new content.
- Ajax revolutionized by enabling partial data exchange.
- Components: XHR, JavaScript, HTML/CSS for data presentation.
- Data Formats: JSON now more popular than XML for interchange.

Basic Explanation

- Analogy: Ajax as a messenger
- Avoiding your (webpage) trips to the main office (server)
- The messenger (Ajax) fetches data, delivers it, and you continue working

When to use?

- Dynamic loading (e.g., infinite scrolling).
- Form submissions without page reload.
- Periodic updates (e.g., live sports scores).
- Fetching product details without refreshing.

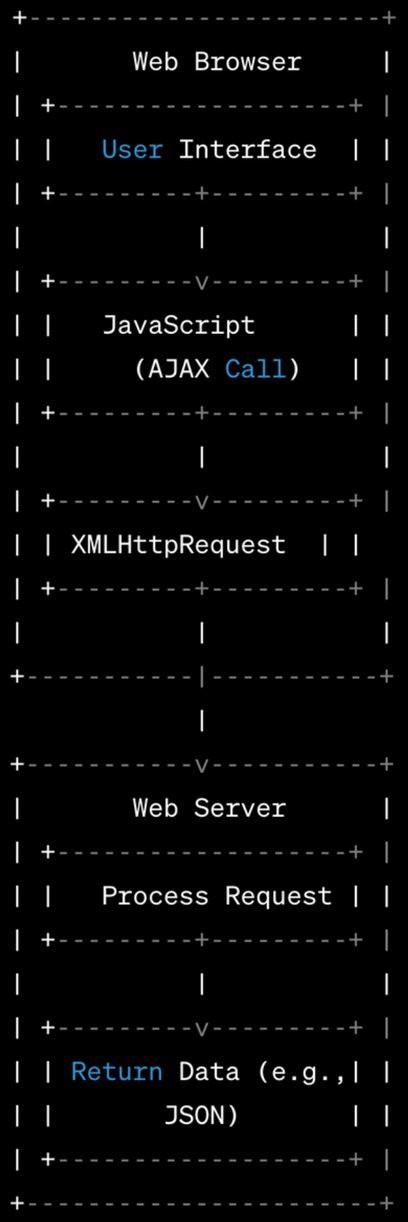
Best Practices

- Handle potential errors in requests.
- Give user feedback (e.g., loading spinner).
- Limit request frequency for performance.
- Prefer JSON for ease and efficiency.

Summary

- Ajax: Fetch/send data without full page reload.
- Uses XMLHttpRequest, often works with JSON.
- Vital for dynamic, responsive web apps.
- Best practices enhance user experience and interaction.

AJAX OVERVIEW



MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

MAKING HTTP REQUEST

XMLHTTPREQUEST

JS

ADVANCED

MAKING HTTP REQUESTS WITH XMLHTTPREQUEST

MASYNCTECH SCHOOL

High-Level Explanation

- XMLHttpRequest (XHR): Browser object for HTTP requests.
- Key for AJAX, fetch data without full page refresh.
- Send/receive HTTP requests and responses.

Basic Explanation

- Analogy: XMLHttpRequest is like texting a friend.
- No need to restart (refresh) the whole site.
- Just "text" the server for info or data exchange.

Best Practices

- Handle potential errors gracefully.
- Use `readystatechange` event to track request state.
- Avoid synchronous requests; prefer asynchronous.
- Consider modern alternatives like Fetch or Axios.

Deep Dive

- XMLHttpRequest introduced in the early 2000s.
- Revolutionary, made web dynamic and responsive.
- Process involves object creation, config, event listeners, sending.
- Data received in various formats: text, JSON, XML, etc.

When to use?

- Fetching data post-page load (e.g., feed updates).
- Sending form data without page reload.
- Polling: regular server data checks.

Summary

- XMLHttpRequest: Allows web apps to communicate with servers.
- Key player in AJAX revolution for dynamic web.
- Dated but important for web dev history.
- Coexists with modern alternatives like Fetch API.

MASYNCTECH
SCHOOL

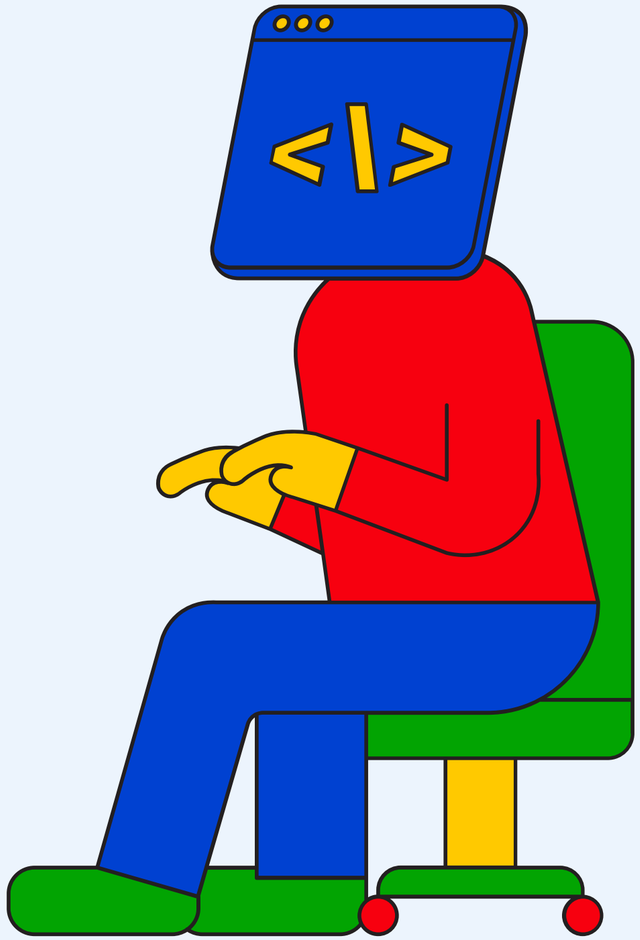
Syntax

```
// Step 1: Create a new instance of XMLHttpRequest
let xhr = new XMLHttpRequest();

// Step 2: Configure the request
xhr.open('GET', 'https://api.example.com/data', true);

// Step 3: Attach event listeners
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    // Handle the response here
    let response = JSON.parse(xhr.responseText);
  }
};

// Step 4: Send the request
xhr.send();
```

1. Initialize XMLHttpRequest with `let xhr = new XMLHttpRequest();`.
2. Configure request details with `xhr.open()`.
3. Set up an event listener for state changes with `xhr.onreadystatechange`.
4. Check if the request is complete and successful.
5. Parse JSON response with `JSON.parse(xhr.responseText)`.
6. Send the request with `xhr.send()`.

MASYNCTECH
SCHOOL

CODE DEMO



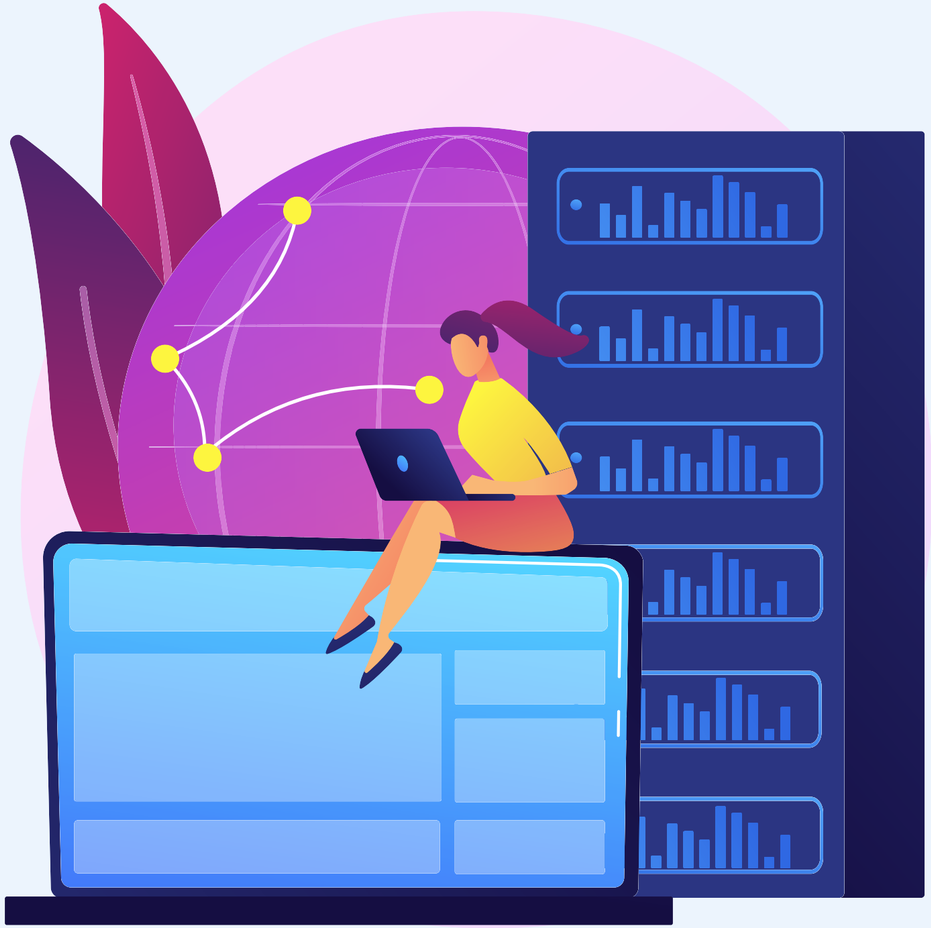
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

MAKING HTTP REQUEST FETCH API



JS

ADVANCED

MAKING HTTP REQUESTS WITH THE FETCH API

MASYNCTECH SCHOOL

High-Level Explanation

- Fetch API: Modern alternative to XMLHttpRequest.
- Uses Promises for readability and manageability.
- Native to JavaScript, widely supported in modern browsers.

Basic Explanation

- Analogy: Fetch is like ordering a pizza online.
- Hitting "Order" is like using Fetch in JavaScript.
- You request data, get a promise, and use it once it arrives.

Best Practices

- Handle errors with `.catch()` or `try/catch`.
- Customize requests with options object.
- Check `ok` property for request success.
- Be mindful of CORS for cross-origin requests.

Deep Dive

- Fetch API introduced in ECMAScript 2015 (ES6).
- Cleaner and more powerful network requests.
- Promise-based for asynchronous handling.
- Compatible with `async/await` for synchronous-like code.

When to use?

- Fetching data in modern web apps.
- API calls in single-page apps (SPA).
- CRUD operations, managing requests and responses.

Summary

- Fetch API: Modern, promise-based network requests.
- Cleaner and more powerful than XMLHttpRequest.
- Widely used in modern web development.

Syntax

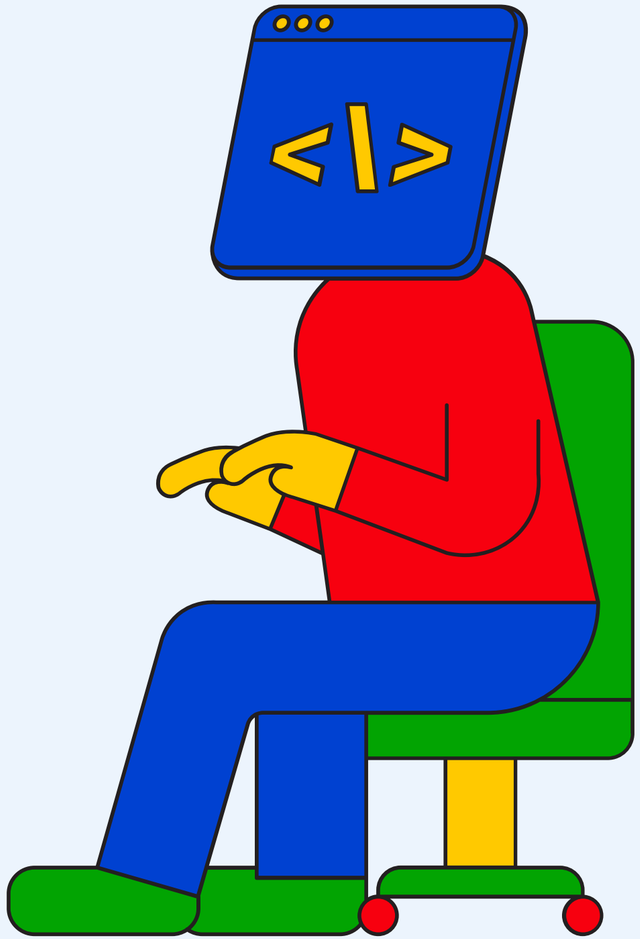
```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

With Async Await

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

fetchData();
```

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

MAKING HTTP REQUEST AXIOS CLIENT

JS

ADVANCED

MAKING HTTP REQUESTS WITH THE AXIOS API

MASYNCTECH SCHOOL

High-Level Explanation

- Axios: JavaScript library for HTTP requests.
- Must be installed separately.
- Promises-based, offers cleaner, powerful client-server communication API.

Deep Dive

- Axios: Popular in front-end and back-end development.
- Offers extensive features: request/response intercepting, data transformation, error handling.
- Works seamlessly in Node.js, versatile for various JavaScript environments.

Basic Explanation

- Analogy: Axios as a super-smart remote control.
- Controls multiple aspects, versatile.
- Easy to use, works everywhere, advanced settings.
- Axios for web developers, communicates with servers, packed with features.

When to use?

- For HTTP requests in both browser and Node.js.
- When advanced features like request cancellation or transformation are needed.
- To create cleaner asynchronous code with a promise-based API.

Best Practices

- Handle errors with `catch` or `try/catch`.
- Employ interceptors for central request/response handling.
- Import Axios from trusted sources.
- Leverage Axios instances for reusable configurations.

Summary

- Axios: Powerful promise-based HTTP client.
- Works across environments, flexible.
- Advanced features for efficient, manageable communication.

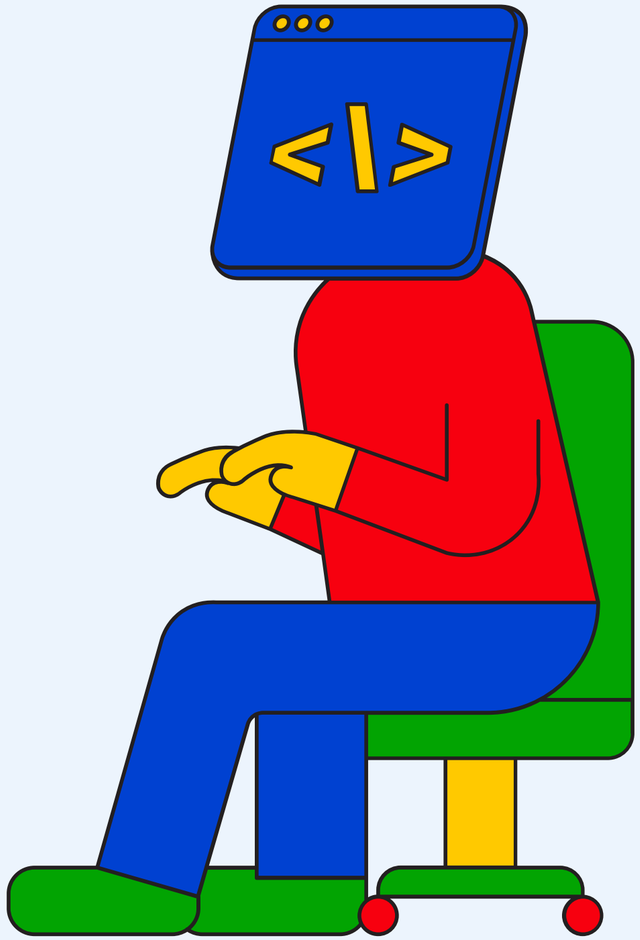
MASYNCTECH
SCHOOL

Syntax

```
axios.get('https://api.example.com/data')
  .then(response => {
    // handle response
  })
  .catch(error => {
    // handle error
  });
```

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED