PART 3

MASYNCTECH
SCHOOL

FULLSTACK WEB DEV

JAVASCRIPT JS

MASYNCTECH MASYNCTECH www.masynctech.com

MASYNCTECH
SCHOOL

JAVASCRIPT ARRAYS OF OBJECTS

JS

ADVANCED

MASYNCTECH
SCHOOL

THE CONCEPT OF OBJECT

REFERENCING IN ARRAYS

JS

ADVANCED

THE CONCEPT OF OBJECT REFERENCES IN ARRAYS

MASYNCTECH SCHOOL

High-Level Explanation

- JavaScript: Objects passed by reference
- Assignment or passing = reference, not copy
- Points to object in memory

Basic Explanation

- Analogy: Object like a rented bike
- You get a "key" (reference), not a new bike
- Friends get a copy of the key, not a new bike
- Changes affect anyone with the key

Deep Dive

- Variable stores a "reference," not object itself
- Variables, functions share same memory location
- Changes affect all references to the object

Best Practices

- Be cautious with functions modifying passed objects; changes impact original.
- Create a "deep copy" to prevent unintended side effects.
- Employ functional programming for immutability, reduce side effects.

Take Note

- "Pass-by-reference" for objects and arrays, not primitives.
- Reassigning a variable to a new object won't change the original object; it updates the reference to a new memory location.

MASYNCTECH
SCHOOL

ARRAY-OBJECT HYBRID

OBJECTS IN ARRAYS

JS

ADVANCED

ARRAY-OBJECT HYBRID (OBJECTS IN ARRAYS)

MASYNCTECH SCHOOL

High-Level Explanation

- Arrays of objects: Elements are objects
- Ideal for lists with multi-property items

Deep Dive

- Combine arrays (order) and objects (key-value pairs)
- Achieve flexible, ordered lists with rich attributes
- Ideal for ordered lists of complex objects, e.g., user data

When to use?

- Objects in arrays useful when:
  - Managing multi-attribute item lists.
  - Preserving item order is essential.
  - Need to access, add, or remove items based on properties.

Basic Explanation

- Analogy: Concert line as an array
- Each person in line as an object
- Object properties: name, seat number, VIP status
- Combining ordered list and rich attributes

Best Practices

- Descriptive property names enhance readability.
- Keep objects homogenous for consistency.
- Utilize built-in array methods (map, filter, find) for manipulation.

Take Note

- Access: Array element first, then property.
- Modifying one object doesn't affect others.
- Array index = order, object properties = attributes.

ARRAY-OBJECT HYBRID (OBJECTS IN ARRAYS)

Syntax

```
const arrayOfObjects = [
  { key1: value1, key2: value2 },
  { key1: value3, key2: value4 },
  ...
];
```

- `const arrayOfObjects = [ ... ];`: Declares an array.

- `{ key1: value1, key2: value2 }, ...`: Objects in the array.

- `key1: value1, key2: value2`: Key-value pairs in each object.

- Useful for managing records with multiple properties.

MASYNCTECH
SCHOOL

ARRAY-OBJECT HYBRID

ARRAYS IN OBJECTS

JS

ADVANCED

ARRAY-OBJECT HYBRID (ARRAYS IN OBJECTS)

MASYNCTECH SCHOOL

High-Level Explanation

- Array-Object Hybrids: Objects containing arrays
- Object properties can be arrays
- Organize and store complex data hierarchically

Basic Explanation

- Analogy: Locker as an object
- Compartments (properties) in the locker
- One compartment with notebooks (an array)
- Objects hold individual items and arrays (stacks) for organization

Best Practices

- Keep arrays in the object focused on a single data type.
- Use arrays for frequent add/remove operations.
- Follow naming conventions, especially with nested data, for clarity.

Deep Dive

- Objects hold properties with any data type, including arrays.
- Arrays in objects group related data, enhancing organization.
- Common in databases, API responses, front-end states.

When to use?

- Array-Object Hybrids useful when:
  - Handling JSON data.
  - Managing state in React and similar frameworks.
  - Categorizing related items under a single entity.

Take Note

- Access: First access object property, then internal array.
- Beware of accidental nested array modifications (references).
- Apply `map`, `filter`, `reduce`, etc., for functional programming on internal arrays.

ARRAY-OBJECT HYBRID (ARRAYS IN OBJECTS)

Syntax

```
const objectName = {
  propertyName1: valueType1,
  propertyName2: valueType2,
  arrayPropertyName: [element1, element2, ...]
};
```

- `const objectName = { ... };`: Declares an object.

- `propertyName1: valueType1, propertyName2: valueType2, ...`: Key-value pairs with various data types.

- `arrayPropertyName: [element1, element2, ...]`: Property for an array, holding elements of any type.

MASYNCTECH
SCHOOL

ITERATE THROUGH ARRAYS USING forEach()

JS

ADVANCED

ITERATE THROUGH ARRAYS USING `FOREACH()`

MASYNCTECH SCHOOL

High-Level Explanation

- `forEach()`: Iterates through array, executes provided function.
- Modern, readable alternative to `for` loops.

Basic Explanation

- Analogy: Basket of fruits as an array.
- Using `forEach()` is like having a helper check each fruit.
- Declarative, focusing on end result, not the process.

Best Practices

- Don't use `forEach()` for early loop exits; use `for`.
- Careful with array modification; `forEach()` changes original.
- `forEach()` skips empty slots in arrays.

Deep Dive

- `forEach()`: Takes callback, runs it for each array item.
- Callback: Current value, index, and array itself.
- Declarative, specifying actions for each element, automated iteration.

When to use?

- `forEach()` suits side-effects on array elements.
- No new array creation needed.
- Useful for order-independent, unbroken loop operations.

Take Note

- `forEach()` returns `undefined`.
- Iterates over existing elements; array length changes aren't considered.

ITERATE THROUGH ARRAYS USING `FOREACH()`

Syntax

```
array.forEach(function callback(currentValue, index, array) {
    // Your code logic here
});
```

- `array`: The array with `forEach()` called on it.
- `callback`: Function for each array element.
  - `currentValue`: Current element.
  - `index`: (Optional) Current element's index.
  - `array`: (Optional) The array itself.

MASYNCTECH
SCHOOL

ITERATE THROUGH ARRAYS USING map()

JS

ADVANCED

ITERATE THROUGH ARRAYS USING `MAP()`

MASYNCTECH SCHOOL

High-Level Explanation

- `map()`: Transforms array elements with a function.
- Returns a new array with transformed elements.

Deep Dive

- `map()`: Takes a callback, applies it to array elements.
- New array contains results of callback on original elements.
- Used for data transformation, preserves original array.

Basic Explanation

- Analogy: `map()` is like photocopying, doubling numbers.
- Original list unchanged; new list with changes.

When to use?

- Use `map()` when:
  - Transforming array elements to a new array.
  - Avoiding modifications to the original array.
  - Chaining with other array methods like `filter()`.

Best Practices

- Avoid `map()` if you won't use the result to save memory.
- Prevent side-effects in callback; no external changes.
- `map()` skips undefined, not `null` values in the array.

Take Note

- `map()` creates a new array; no modification to the original.
- Skips array holes, unlike `forEach()`.

ITERATE THROUGH ARRAYS USING `MAP()`

MASYNCTECH
SCHOOL

Syntax

```
const newArray = originalArray.map(callbackFunction(element, index, array));
```

- `originalArray`: The array with `map()` called on it.
- `newArray`: New array with results.
- `callbackFunction`: Function for each element.
  - `element`: Current element being processed.
  - `index`: Current element's index.
  - `array`: The original array itself.

MASYNCTECH
SCHOOL

ITERATE THROUGH ARRAYS USING filter()

JS

ADVANCED

ITERATE THROUGH ARRAYS USING `FILTER()`

MASYNCTECH SCHOOL

High-Level Explanation

- `filter()`: Creates new array with elements meeting a condition.
- Condition defined in callback function provided to `filter()`.

Basic Explanation

- Imagery: Basket of fruits.
- `filter()`: Selects and keeps only the apples.
- Leaves other fruits untouched.
- Results in a new basket with only apples.

Best Practices

- Use `filter()` for a new array subset.
- Callback function should be pure (no side effects).
- Don't use `filter()` if modifying the original array; it creates a new filtered array.

Deep Dive

- `filter()`: Takes a callback.
- Callback returns `true` or `false`.
- `true`: Element included in new array.
- Filter based on custom conditions without modifying the original array.

When to use?

- Use `filter()` when you need:
  - A new array with specific criteria.
  - Exclusion of certain elements.
  - Preservation of the original array.

Take Note

- `filter()` doesn't change the original array.
- Original array remains as is.
- Skips elements not in resulting array; doesn't delete them.

ITERATE THROUGH ARRAYS USING `FILTER()`

Syntax

```
const newArray = originalArray.filter(callbackFunction);
```

- `originalArray`: The array to be filtered.
- `callbackFunction`: Decides inclusion based on `true` or `false`.
- `newArray`: Contains elements where `callbackFunction` is `true`.

MASYNCTECH
SCHOOL

TRANSFORMATION AND MANIPULATION USING

splice()

JS

ADVANCED

TRANSFORMATION AND MANIPULATION USING `SPLICE()`

MASYNCTECH SCHOOL

High-Level Explanation

- `splice()`: Adds/removes array elements.
- Modifies original array; returns removed elements in a new array.

Basic Explanation

- Imagery: Deck of cards as an array.
- `splice()`: Cut, remove, and insert cards in one go.
- Alters the original deck (array) based on splicing.

Best Practices

- `splice()` modifies the original array; make a copy if needed.
- First argument can be negative (counting from the end).
- Omitting second argument deletes from index to end.

Deep Dive

- `splice()`: Multifunctional - remove, insert, or replace elements.
- First argument: Index to start changes.
- Second argument: Number of elements to remove.
- Optional: Additional arguments for elements to add.

When to use?

- Use `splice()` to:
  - Remove elements.
  - Insert elements at a specific index.
  - Replace elements in an array.

Take Note

- Returns an array of removed elements.
- In-place method, alters the original array.
- Empty array if no changes.
- Array length changes with added/removed elements.

TRANSFORMATION AND MANIPULATION USING `SPLICE()`

MASYNCTECH
SCHOOL

Syntax

```
array.splice(startIndex, deleteCount, element1, element2, ...);
```

- `startIndex`: Index to start making changes.
- `deleteCount`: Number of elements to remove.
- `element1, element2, ...`: Elements to add at `startIndex`.

MASYNCTECH
SCHOOL

TRANSFORMATION AND MANIPULATION USING

concat()

JS

ADVANCED

TRANSFORMATION AND MANIPULATION USING `CONCAT()`

MASYNCTECH SCHOOL

High-Level Explanation

- `concat()` merges arrays into a new one.
- It doesn't alter the originals.
- Useful for combining multiple arrays.

Basic Explanation

- Simplified Explanation:
  - `concat()` is like having two trays of cookies.
  - You get a new empty tray.
  - You put all the cookies from the first and second trays onto the new tray.
  - The original trays still have their cookies; nothing is taken from them.

Best Practices

- Use `concat()` for immutability.
- Consider spread operator for modern codebases.

Deep Dive

- `concat()` takes one or multiple arrays.
- It appends them to the calling array.
- The new array contains shallow copies of elements.
- You can pass individual elements or arrays.

When to use?

- Use `concat()` to merge arrays into one.
- Original arrays remain unchanged.

Take Note

- `concat()` returns a new array.
- It performs a shallow copy, so nested objects are copied by reference.

TRANSFORMATION AND MANIPULATION USING `CONCAT()`

MASYNCTECH
SCHOOL

Syntax

```
const newArray = array1.concat(array2, array3, ..., arrayN);
```

- `array1`: The starting array.
- `array2, array3, ..., arrayN`: Arrays to add to `array1`.

MASYNCTECH
SCHOOL

TRANSFORMATION AND MANIPULATION USING

.assign()

JS

ADVANCED

TRANSFORMATION AND MANIPULATION USING `OBJECT.ASSIGN()`

MASYNCTECH SCHOOL

High-Level Explanation

- `Object.assign()` copies properties from source objects to a target object.
- It modifies the target object.
- It returns the modified target object.

Deep Dive

- `Object.assign()` accepts a target object and source objects.
- It iterates over source object keys.
- Copies source object values into the target object.
- Overwrites target keys if they already exist.

Basic Explanation

- `Object.assign()` is like transferring toys from different boxes to your toy box.
- If a toy is already in your toy box, it gets replaced with a new one of the same kind.

When to use?

- Use `Object.assign()` to copy properties between objects.
- Great for creating new objects with combined properties.

Best Practices

- Start with an empty object `{}` for a new object.
- Watch out for nested structures; it's a shallow copy.

Take Note

- `Object.assign()` performs a shallow copy, doesn't create a new instance.

Syntax

```
const target = {};
Object.assign(target, source1, source2, ..., sourceN);
```

- `array1`: The starting array.
- `array2, array3, ..., arrayN`: Arrays to add to `array1`.

MASYNCTECH
SCHOOL

SEARCH AND FILTER USING find()

JS

ADVANCED

SEARCH AND FILTER USING `FIND()`

MASYNCTECH SCHOOL

High-Level Explanation

- `find()`: Get the first element that matches a condition.

Deep Dive

- `find()` checks each item using a function.
- If the function returns `true`, it stops and returns that item.

Basic Explanation

- `find()` is like searching for the first matching item in a list.
- It stops and returns the first match it finds.

When to use?

- `find()` is great for locating the first item in an array that meets specific criteria.
- It stops as soon as it finds a match.

Best Practices

- The method doesn't change the array.
- It returns `undefined` if no elements match the condition.
- It stops as soon as it finds a match.

Take Note

- Only the first element that satisfies the condition is returned.
- `find()` doesn't mutate the original array.
- The array is not changed in any way.

Syntax

```
const found = array.find(function(element, index, array) {
  // return true or false
});
```

- `element`: The current element being processed.
- `index`: The index of the current element.
- `array`: The array `find()` was called upon.

MASYNCTECH
SCHOOL

SEARCH AND FILTER USING some()

JS

ADVANCED

SEARCH AND FILTER USING `SOME()`

MASYNCTECH SCHOOL

High-Level Explanation

- `some()`: Check if at least one element meets a condition.

Deep Dive

- `some()` checks elements until it finds one that fits the condition.

Basic Explanation

- Imagine a basket of fruits.
- Objective: Find at least one apple.
- Look at each fruit one by one.
- As soon as you find an apple, stop searching and confirm there's an apple.

When to use?

- Use `some()` when:
  - You want to know if at least one element meets a condition.
  - You don't need to identify which one.

Best Practices

- `some()` doesn't change the array.
- It stops on first `true`, returning `true`.

Take Note

- Returns `true` or `false`.
- Doesn't change the original array.

Syntax

```
const result = array.some(function(element, index, array) {
  // return true or false based on condition
});
```

- `element`: The current element being processed.
- `index`: The index of the current element.
- `array`: The array `some()` was called upon.

MASYNCTECH
SCHOOL

SEARCH AND FILTER USING every()

JS

ADVANCED

SEARCH AND FILTER USING `EVERY()`

MASYNCTECH SCHOOL

High-Level Explanation

- `every()` checks if all elements meet a condition.
- Returns `true` or `false`.

Basic Explanation

- Imagine a basket of fruits.
- Objective: Check if all are apples.
- Examine each fruit one by one.
- If you find one non-apple, declare "Not all are apples."

Best Practices

- `every()` stops at the first `false` condition.
- It doesn't modify the original array.

Deep Dive

- `every()` uses a callback on each element.
- Returns `true` if all pass, else `false`.
- Stops at the first failure.

When to use?

- Use `every()` when:
  - You want to confirm that all elements satisfy a condition.
  - You don't need to identify which ones fail.

Take Note

- Returns a boolean (`true` or `false`).
- Doesn't modify the original array.

SEARCH AND FILTER USING `EVERY()`

Syntax

```
const result = array.every(function(element, index, array) {
  // return true or false based on a condition
});
```

- `element`: The current element being processed.
- `index`: The index of the current element.
- `array`: The array `every()` was called upon.

MASYNCTECH
SCHOOL

INTRODUCTION TO

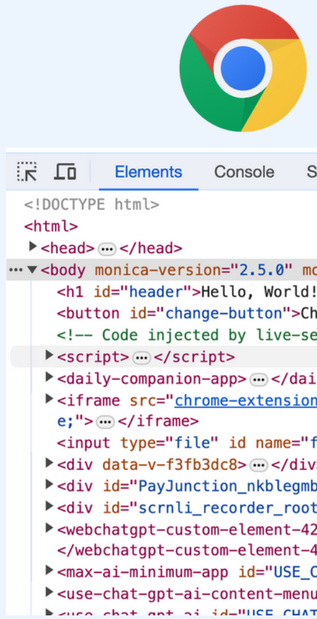
DOM

DOCUMENT OBJECT MODEL

JS

ADVANCED

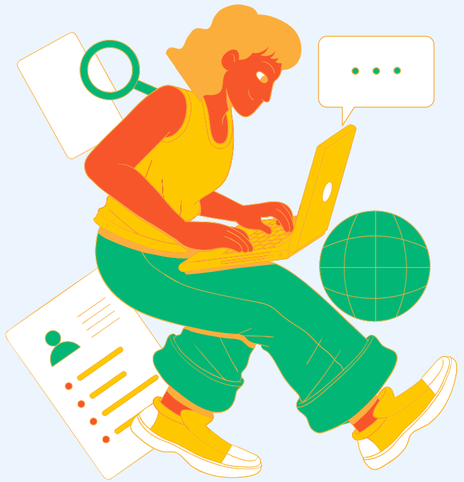
DOM OVERVIEW



DOM

JS

INTERACTING WITH DOM



WEB PAGE

WHAT IS DOM?

MASYNCTECH SCHOOL

High-Level Explanation

- DOM: Programming interface for web documents
- Represents document structure as a tree
- Objects mirror web page elements (HTML, text)
- Includes representation of the browser window

Basic Explanation

- Webpage = family tree; each element = family member.
- DOM = tree diagram showing relationships.
- Facilitates easy navigation and changes on the webpage.

Best Practices

- Minimize direct DOM manipulations to save CPU.
- Utilize libraries like jQuery or React for ease.
- Opt for a simple DOM structure for better performance.
- Efficient event handling through event delegation.

Deep Dive

- DOM is essential in web development, crucial for JavaScript.
- It bridges web docs and scripts, enabling dynamic manipulation.
- Represents webpages in an object-oriented way.
- It's not a language but a structural representation.
- Allows manipulation with languages like JavaScript.

When to use?

- DOM use cases:
  - Update webpage content/style without refresh.
  - Read/scrape data.
  - Create dynamic, interactive web apps (dashboards, charts).

Take Note

- DOM isn't a language, but a webpage representation.
- DOM manipulations can cause web app performance problems.
- React uses a 'virtual DOM' to alleviate performance issues.

MASYNCTECH
SCHOOL

INTRODUCTION TO DOM

DOM vs JAVASCRIPT

JS

ADVANCED

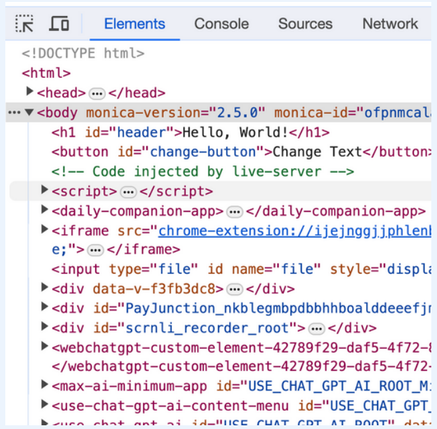
DOM VS JAVASCRIPT

MASYNCTECH SCHOOL

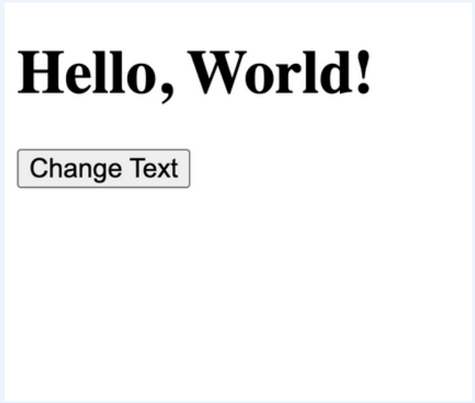
```
<!DOCTYPE html>
<html>
  <head>
    <title>My Webpage
  </head>
  <body>
    <h1 id="header">H
    <button id="chang
  </body>
</html>
```

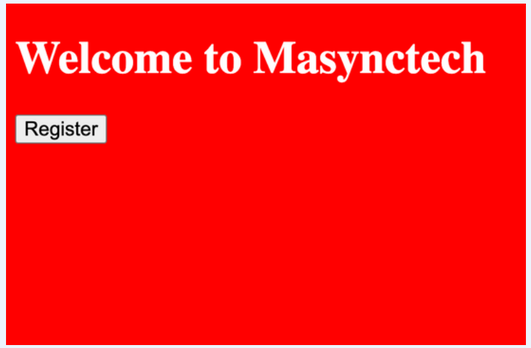
HTML CODE

DOM



WEB PAGE





UPDATED USING DOM

DOM VS JAVASCRIPT

MASYNCTECH SCHOOL

High-Level Explanation

- DOM and JavaScript work together in web development.
- DOM is an interface for web docs' structure and content.
- JavaScript is used to interact with and manipulate the DOM.

Basic Explanation

- DOM: Lego blocks forming a webpage.
- JavaScript: Your hands, manipulates the blocks.
- Use JavaScript to change webpage content.

Best Practices

- **DOM:**
  - Minimize direct manipulations for performance.
  - Efficient event handling with delegation.
- **JavaScript**
  - Maintainable code through modularity.
  - Non-blocking tasks with asynchronous operations.

Deep Dive

- DOM: structured webpage representation, tree structure.
- Standard model for language interaction.
- JavaScript: high-level, interpreted language.
- Controls DOM, enables dynamic changes to content and styling.

When to use?

- DOM: Ideal for web document element access and modification.
- JavaScript: Perfect for interactivity and DOM manipulation.
- Suitable for various programming languages.

Take Note

- DOM works with various languages, not just JavaScript.
- JavaScript extends beyond DOM, covering server tasks, databases, etc.

MASYNCTECH
SCHOOL

INTRODUCTION TO DOM

ANATOMY OF WEBPAGE

JS

ADVANCED

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic Web Page</title>
  <!-- CSS -->
  <style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
    }
    h1 {
      color: blue;
    }
    #infoText {
      color: green;
    }
  </style>
</head>
<body>
  <!-- HTML -->
  <h1 id="mainHeading">Welcome to My Web Page</h1>
  <p id="infoText">This is a simple paragraph.</p>
  <button id="changeTextButton">Change Text</button>
  <img id="sampleImage" src="example.jpg" alt="Example image">

  <!-- JavaScript -->
  <script>
    // Get elements
    const mainHeading = document.getElementById("mainHeading");
    const infoText = document.getElementById("infoText");
    const changeTextButton = document.getElementById("changeTextButton");

    // Add button click event
    changeTextButton.addEventListener("click", function() {
      infoText.innerHTML = "The text has been changed!";
    });
  </script>
```

MASYNCTECH SCHOOL

ANATOMY OF A BASIC WEB PAGE.

MASYNCTECH SCHOOL

High-Level Explanation

- Basic web page consists of HTML, CSS, and JavaScript.
- HTML structures content and layout.
- CSS styles design and appearance.
- JavaScript adds interactivity for users.

Basic Explanation

- DOM: Lego blocks forming a webpage.
- JavaScript: Your hands, manipulates the blocks.
- Use JavaScript to change webpage content.

When to use?

- HTML: Define web page structure and content.
- CSS: Style and organize HTML elements.
- JavaScript: Add interactivity, DOM manipulation, async operations.

Deep Dive

- **HTML**: Defines web page structure and content using tags.
- **CSS**: Controls visual design, layout, and styling.
- **JavaScript**: Adds interactivity and dynamic features.

Take Note

- Technologies are intertwined for a complete web experience.
- Separation of concerns (HTML, CSS, JavaScript) for maintainability and scalability.

Best Practices

- HTML:
  - Use semantic elements for SEO and accessibility.
  - Maintain clean indentation and nesting.
- CSS:
  - Prefer classes over inline styles for reusability.
  - Apply responsive design principles.
- JavaScript:
  - Ensure modular and well-commented scripts.
  - Validate user input for security.

MASYNCTECH
SCHOOL

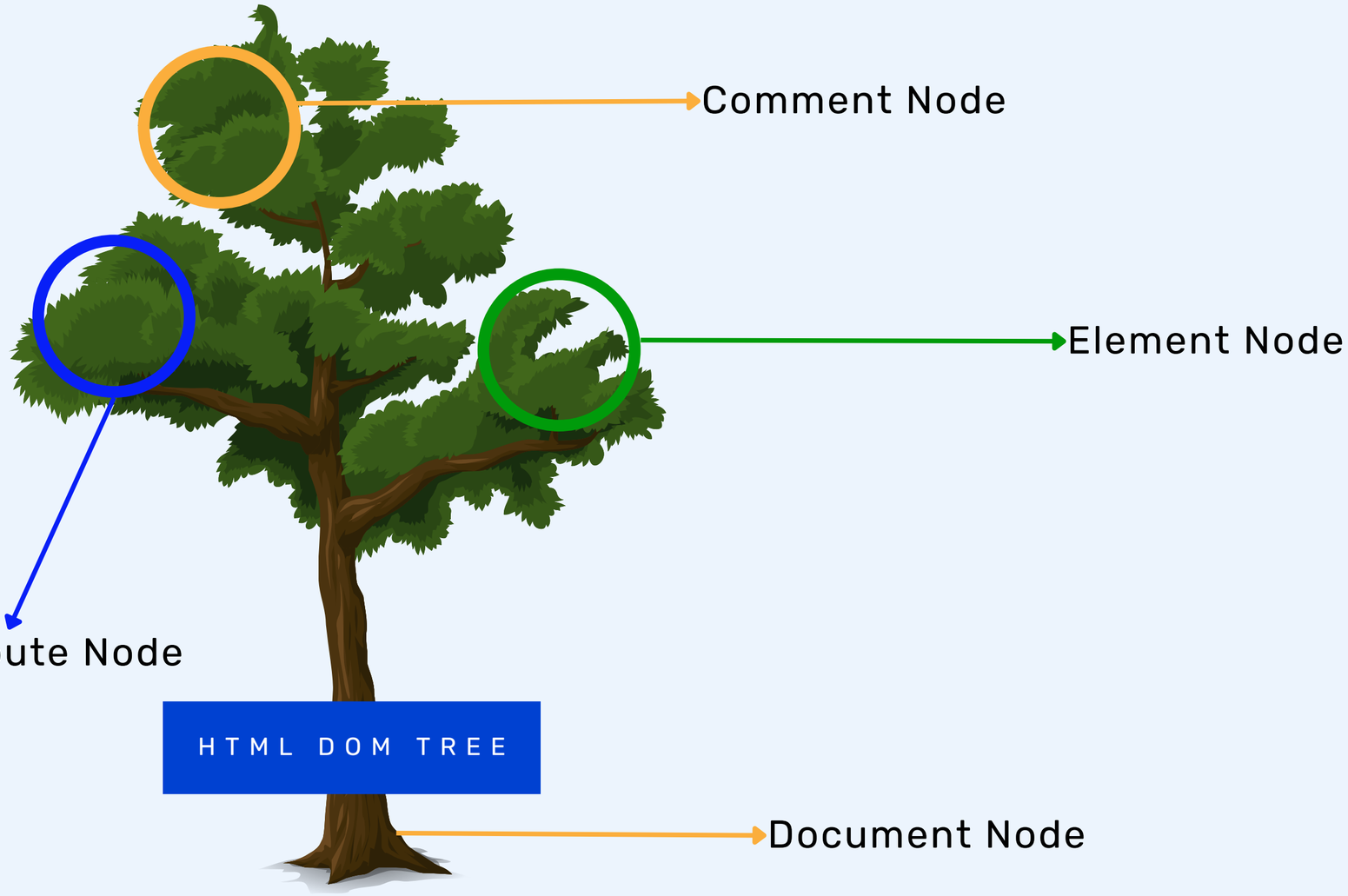
SELECTING DOM ELEMENTS

WHAT'S DOM TREE

JS

ADVANCED

WHAT IS DOM TREE



WHAT IS DOM TREE

MASYNCTECH
SCHOOL

High-Level Explanation

- DOM Tree: Hierarchy of HTML/XML elements and content.
- Connects web pages to JavaScript for scripting.

Basic Explanation

- Webpage as a family tree analogy.
- Grandparent: Root or document.
- Parents: Major divisions (e.g., `head`, `body`).
- Children: Individual elements (paragraphs, images).
- DOM Tree shows family member relationships.
- Changes in the tree reflect on the web page.

Deep Dive

- DOM Tree: Elements, attributes, text as nodes.
- Root node, parent-child structure.
- "Live" model, changes reflect instantly.
- Allows programmatic interaction with web documents.

DOM Tree components:

- Document Node: Represents the entire document.
- Element Nodes: Represent HTML/XML elements.
- Text Nodes: Contain text inside elements.
- Attribute Nodes: Represent element attributes.
- Comment Nodes: Represent document comments.

When to use?

- DOM Tree use cases:
  - Dynamic updates to content, structure, or style.
  - Reading or extracting web page information.
  - Handling user interactions (e.g., clicks, forms).

MASYNCTECH
SCHOOL

Best Practices

- Caution in referencing nodes to prevent errors.
- Minimize direct DOM manipulations for performance.
- Batch changes for efficient DOM updates.
- Prefer modern API methods for clarity and efficiency.

Take Note

- Language-agnostic DOM Tree.
- Nodes include text, attributes, and comments.
- DOM changes impact reflows/repaints, watch frequency.

WHAT IS DOM TREE

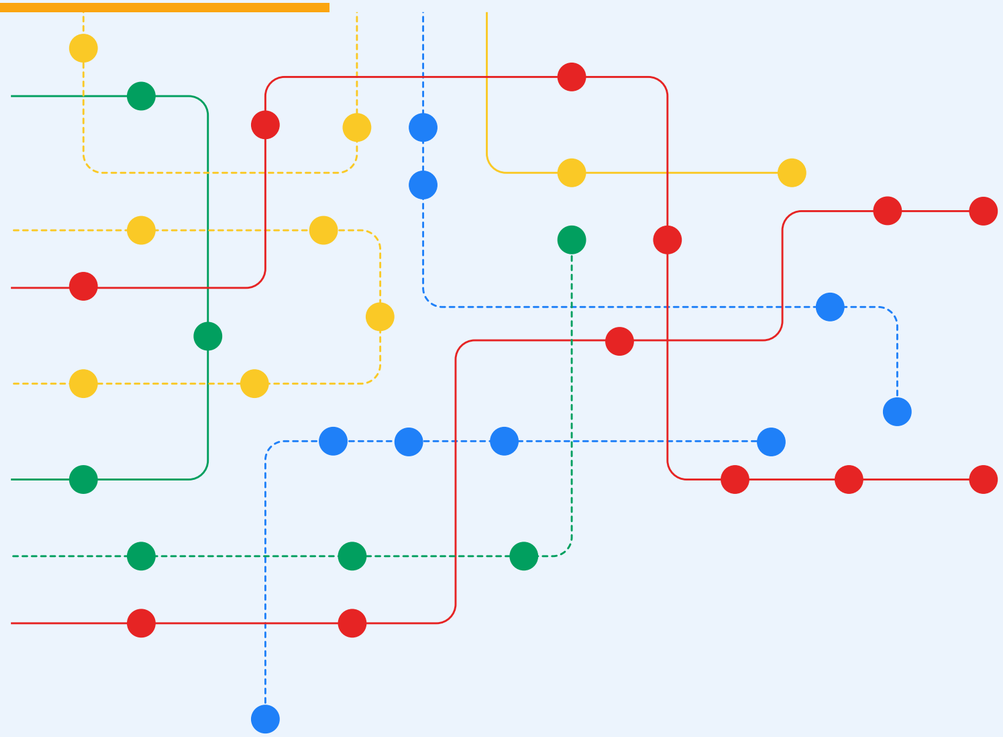
```
├── <html> (Element Node)
|   ├── <head> (Element Node)
|   |   └── <title> (Element Node)
|   |       └── "My Web Page" (Text Node)
|   └── <body> (Element Node)
|       └── <div id="container"> (Element Node)
|           ├── <h1> (Element Node)
|           |   └── "Welcome to My Web Page" (Text Node)
|           ├── <p class="intro"> (Element Node)
|           |   └── "This is an introductory paragraph." (Text Node)
|           ├── <ul id="list"> (Element Node)
|           |   ├── <li> (Element Node)
|           |   |   └── "Item 1" (Text Node)
|           |   ├── <li> (Element Node)
|           |   |   └── "Item 2" (Text Node)
|           |   └── <li> (Element Node)
|           |       └── "Item 3" (Text Node)
```

ANOTHER EXAMPLE

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

DOM NODES



JS

ADVANCED

DOM NODES

MASYNCTECH SCHOOL

High-Level Explanation

- DOM is a programming interface for web docs.
- Structures HTML/XML as object tree (nodes).
- Nodes manipulated via languages like JavaScript.
- Node types: Document, Element, Text, Attribute, Comment.
- Operations: Traverse, manipulate, query, style.

Basic Explanation

- DOM is akin to a house blueprint.
- Document Node is the foundation.
- Element Nodes represent rooms/structures.
- Text Nodes label items.
- Attribute Nodes define specific features.
- Comment Nodes act as side notes.
- JavaScript enables changes like moving furniture.

Deep Dive

- DOM is a live interface for real-time updates.
- Node types (Document, Element, Text, Attribute, Comment) play vital roles.
- Nodes enable traversal, manipulation, querying, styling.
- Unique properties/methods for each node type.

Properties & Methods

-Properties:

- `parentNode`, `nextSibling`, `previousSibling`
- `firstChild`, `lastChild`, `nodeValue`, `nodeName`
- `nodeType`, and more.

- **Methods**
  - `appendChild()`, `removeChild()`, `insertBefore()`
  - `cloneNode()`, `replaceChild()`, and more.

DOM NODES

MASYNCTECH
SCHOOL

Best Practices

- Consider node type for specific properties/methods.
- Utilize the most specific method for performance.
- For complex queries, explore `querySelector` and `querySelectorAll`.

When to use?

- Utilize DOM nodes for:
  - Traversal: Finding specific parts.
  - Manipulation: Adding, deleting, modifying elements.
  - Querying: Searching for elements/attributes.
  - Styling: Applying/changing styles dynamically.

Take Note

- DOM is "live," changes reflect instantly.
- Node types serve distinct roles, have unique properties/methods.

WHAT IS DOM TREE

MASYNCTECH
SCHOOL

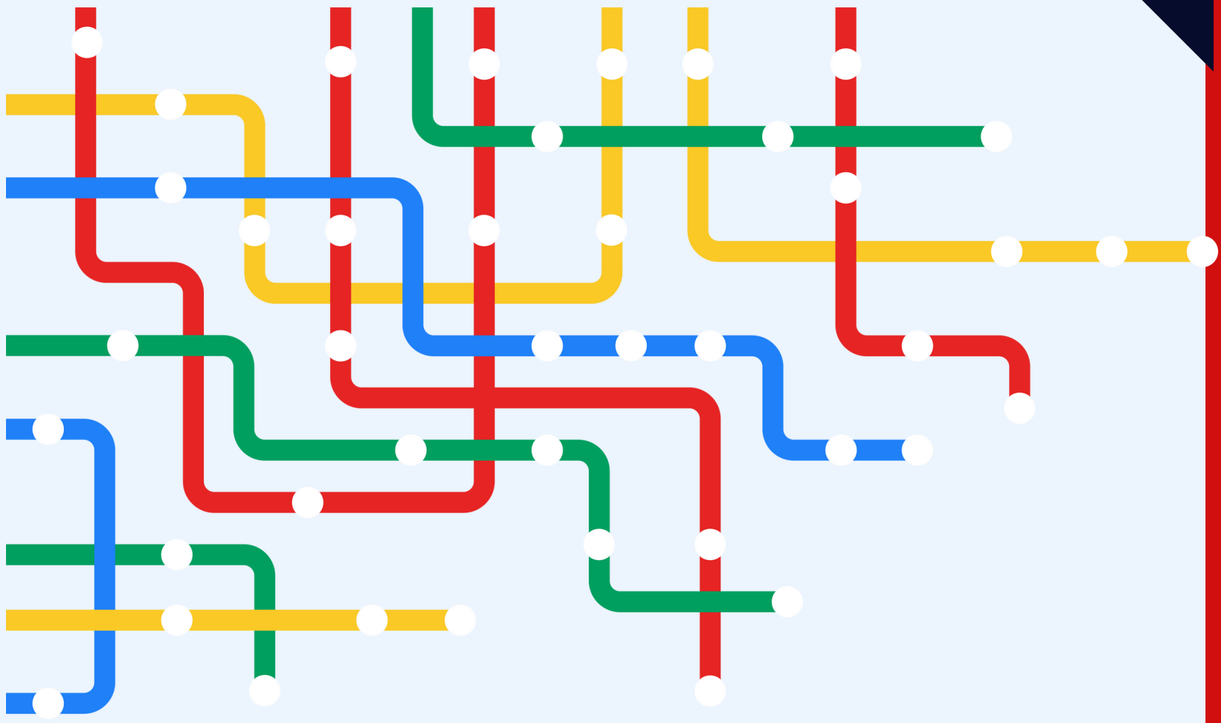
```
├── <html> (Element Node)
|   ├── <head> (Element Node)
|   |   └── <title> (Element Node)
|   |       └── "My Web Page" (Text Node)
|   └── <body> (Element Node)
|       └── <div id="container"> (Element Node)
|           ├── <h1> (Element Node)
|           |   └── "Welcome to My Web Page" (Text Node)
|           ├── <p class="intro"> (Element Node)
|           |   └── "This is an introductory paragraph." (Text Node)
|           ├── <ul id="list"> (Element Node)
|           |   ├── <li> (Element Node)
|           |   |   └── "Item 1" (Text Node)
|           |   ├── <li> (Element Node)
|           |   |   └── "Item 2" (Text Node)
|           |   └── <li> (Element Node)
|           |       └── "Item 3" (Text Node)
```

ANOTHER EXAMPLE

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

ELEMENT NODES



JS

ADVANCED

ELEMENT NODES

MASYNCTECH SCHOOL

High-Level Explanation

- Element nodes represent HTML/XML elements.
- Contain start/end tags, attributes, child nodes.
- Primary building blocks for web docs.
- Enable manipulation of content, structure, style.

Basic Explanation

- Webpage as Lego structure.
- Element nodes like Lego blocks.
- Blocks can have stickers (attributes).
- Connected to smaller blocks (child nodes).
- JavaScript allows block manipulations.

Deep Dive

- Element nodes are objects with properties/methods.
- Part of the DOM hierarchy, have relationships.
- Contain attributes and various child node types.
- Rich API for appending, removing, attributing, and event handling.
- Main interaction point for DOM manipulation.

Properties & Methods

- Accessing Element Nodes: Methods like `getElementById()`, `querySelector()`, etc.
- Modifying Attributes: Methods `getAttribute()`, `setAttribute()`, `removeAttribute()`.
- Changing Content: Properties like `innerHTML` or `innerText`.

When to use?

Element nodes are essential for:

- Modifying element content
- Adding/modifying attributes (e.g., `class`, `id`)
- Inserting new elements into the structure
- Attaching interactive behaviors (click, hover)

MASYNCTECH
SCHOOL

Best Practices

- When manipulating element nodes, be cautious of unintentionally altering or removing child nodes.
- Try to batch your DOM manipulations to reduce reflows and repaints, as these can affect performance.
- Use appropriate methods for tasks: for example, `setAttribute` for modifying attributes and `appendChild` for adding child nodes.

Take Note

- Element nodes ≠ HTML tags; they're DOM objects.
- Include attributes and child nodes.
- Vital for webpage structure/content.
- Offer rich properties/methods for interactions.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

USING

getElementByIld

JS

ADVANCED

USING GETELEMENTBYID

MASYNCTECH
SCHOOL

High-Level Explanation

- `getElementById` method targets HTML element by its unique `id` attribute.

Deep Dive

- `getElementById` in `document` object.
- Searches DOM for matching `id` attribute.
- Returns first matching element as DOM object.
- Enables property/method manipulation.

Basic Explanation

- `getElementById` is like finding a book with an ISBN.
- Uses a unique `id` to locate elements instantly on a webpage.

When to use?

- Use `getElementById` when:
  - Targeting a specific element with a unique `id`.
  - Seeking fast, direct access to a single element.
  - The targeted element is crucial for functionality/interactivity.

Best Practices

- Ensure unique `id` across the document.
- Enhance code readability with descriptive `id` names.
- Check for `null` before manipulating; it's returned if element not found.

Take Note

- `getElementById` is case-sensitive.
- If multiple elements share the same `id`, it returns the first encountered.
- Only returns a single element, even if multiple have the same `id`.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

BROWSER WINDOW OBJECT

JS

ADVANCED

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

USING

getElementsByClassName

JS

ADVANCED

USING GETELEMENTSBYCLASSNAME

MASYNCTECH
SCHOOL

High-Level Explanation

- `getElementsByClassName` selects elements by class.
- Returns an HTMLCollection, a live list.
- Enables interaction with multiple elements.

Basic Explanation

- `getElementsByTagName` is like selecting folders.
- Retrieves all items matching specified tag.
- Similar to selecting all 'Photos' folders on a computer.

Best Practices

- HTMLCollection is live; DOM changes affect it.
- Be cautious when iterating; changes can impact loops.

Deep Dive

- Versatile for actions on multiple elements.
- Returned HTMLCollection is live (updates with DOM).
- Case-insensitive, works on `document` or elements.

When to use?

- Use `getElementsByTagName` when:
  - Styling/manipulating multiple elements of the same type.
  - Creating a collection for iteration.
  - Needing a live collection reflecting DOM changes.

Take Note

- HTMLCollection is zero-indexed, accessed with `[0]`, `[1]`, etc.
- `getElementsByTagName` is case-insensitive.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

USING

querySelector

JS

ADVANCED

USING QUERYSELECTOR

MASYNCTECH SCHOOL

High-Level Explanation

- `querySelector` selects elements using CSS selectors.
- Returns the first matching element or `null` if none.

Basic Explanation

- `querySelector` is like an advanced Twitter search.
- Enables specific criteria to find elements.
- Similar to searching tweets by user, hashtag, date.

Best Practices

- Correct CSS selector syntax for `querySelector`.
- Use when sure only one element is needed.
- Avoid in loops or high-performance scenarios due to computational cost.

Deep Dive

- `querySelector` uses various CSS selectors.
- Versatile: Classes, IDs, pseudo-classes, attributes, combos.
- Adaptable, mirrors CSS targeting syntax.

When to use?

- Use `querySelector` for complex or combined criteria.
- Ideal for selecting the first matching element.

Take Note

- `querySelector` is case-sensitive.
- Returns the first element among multiple matches.
- Searches depth-first in the DOM tree.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

USING

querySelectorAll

JS

ADVANCED

USING QUERYSELECTORALL

MASYNCTECH SCHOOL

High-Level Explanation

- `querySelectorAll` selects multiple elements with CSS selectors.
- Returns a NodeList with all matching elements.

Basic Explanation

- `querySelector` finds the first post with a hashtag.
- `querySelectorAll` finds all posts with the same hashtag.

Best Practices

- Check NodeList length for expected elements.
- Re-run `querySelectorAll` if DOM changes.
- Convert NodeList to an array for extra methods.

Deep Dive

- `querySelectorAll` is flexible, similar to `querySelector`.
- Complex CSS selectors and combinations work.
- Differs in returning all matches, not just the first.
- NodeList is static, doesn't update with DOM changes.

When to use?

- Use `querySelectorAll` for operations on multiple elements.
- Ideal for tasks like styling or DOM manipulation with element collections.

Take Note

- `querySelectorAll` is not live, doesn't auto-update.
- Returns a NodeList, not an array.
- Case-sensitive matching.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

Child VS Children Nodes

JS

ADVANCED

CHILD VS CHILDREN NODES

MASYNCTECH SCHOOL

High-Level Explanation

- "Child Nodes" are all direct descendants.
- "Children Nodes" specifically mean element nodes.
- `.childNodes` includes all node types.
- `.children` fetches only element nodes.

Basic Explanation

- `.childNodes` includes everyone in the family.
- `.children` focuses only on human family members.
- Like taking a full family portrait vs. a human-centric one.

Best Practices

- Both `.childNodes` and `.children` are "live."
- Watch out for whitespace creating text nodes in `.childNodes`.
- Check length before accessing items to avoid errors.

Deep Dive

- "Child Nodes" include all DOM descendants.
- "Children Nodes" are only element nodes.
- `.childNodes` gives full structure, `.children` simplifies.
- `.childNodes` returns a NodeList, `.children` returns HTMLCollection.

When to use?

- Use `.childNodes` for a full picture, including text/comments.
- Useful for understanding/manipulating the entire DOM.
- Use `.children` for element nodes, e.g., changes to visible page parts.

Take Note

- `.children` returns an HTMLCollection, not a NodeList.
- HTMLCollection and NodeList have different methods/properties.
- `.childNodes` includes all node types, not just element nodes.

MASYNCTECH
SCHOOL

SELECTING DOM ELEMENTS

USING

getElementByName

FOR FORM ELEMENTS

JS

ADVANCED

GETELEMENTBYNAME (FOR FORM ELEMENTS)

MASYNCTECH SCHOOL

High-Level Explanation

- `getElementsByName` selects elements by `name` attribute.
- Commonly used with form elements but not limited to them.
- Returns a `NodeList` with matching elements.

Basic Explanation

- `getElementsByName` is like finding health potions in a game.
- Helps locate elements by their specified name.
- Returns a list of all matching elements.

Best Practices

- Check the `NodeList` length to confirm elements found.
- Loop through the `NodeList` for individual changes.
- Consider modern query methods for complex queries.

Deep Dive

- `getElementsByName` is in the `document` object.
- Differs from `getElementById` as it returns multiple elements.
- Results in a `NodeList` (array-like but not live).
- Changes after NodeList generation don't update it.

When to use?

- Use `getElementsByName` for:
  - Multiple form elements with the same `name`.
  - Grouping and programmatically manipulating elements.

Take Note

- `getElementsByName` is case-sensitive.
- Returns elements in document order.
- Found in the `document` object, not on individual elements.

MASYNCTECH
SCHOOL

ACCESSING

PARENT & CHILDREN ELEMENTS

JS

ADVANCED

ACCESSING PARENT AND CHILD ELEMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Web development frequently involves DOM manipulation.
- JavaScript provides tools for accessing parent and child elements.
- Properties like `parentNode` and `parentElement` help move up the DOM tree.
- Methods like `firstChild`, `lastChild`, `firstElementChild`, and `lastElementChild` aid in navigating downwards.

Basic Explanation

- Group photo: You & mom = parent & child.
- Siblings: Immediate neighbors in DOM.
- Identify: Locate adjacent elements in the picture.

Best Practices

- Check for `null` before node access.
- Specify node type: element or any.
- Use appropriate properties accordingly.

Deep Dive

- DOM interaction involves nodes' relatives.
- Access parent: `parentNode`, `parentElement`.
- Note: Differences between `parentNode` and `parentElement`.
- Get children: `firstChild`, `lastChild`.
- For elements: `firstElementChild`, `lastElementChild`.

When to use?

- Upward navigation: `parentNode`, `parentElement`.
- Downward navigation: `firstChild`, `lastChild`, `firstElementChild`, `lastElementChild`.

Take Note

- `parentNode` vs. `parentElement` differences.
- `firstChild`, `lastChild` may return text/comments.
- For elements only: `firstElementChild`, `lastElementChild`.

MASYNCTECH
SCHOOL

USING

DATA ATTRIBUTES

JS

ADVANCED

USING DATA ATTRIBUTES

MASYNCTECH SCHOOL

High-Level Explanation

- Data attributes in HTML for extra info.
- Access and change with JS: `.getAttribute()`, `.setAttribute()`, `dataset` property.

Basic Explanation

- Party name tags: Like data attributes.
- QR codes hold extra info, like data attributes.
- QR codes keep name tags tidy, like HTML elements.
- JavaScript "scans" data attributes for extra info.

Best Practices

- camelCase for `data-*` with `dataset`.
- Avoid if standard HTML attributes suffice.
- Data attributes ≠ database or localStorage.

Deep Dive

- Data attributes store custom data in HTML.
- Start with "data-" followed by a name.
- Avoid non-semantic practices like extra classes or IDs.
- Used for data not fitting into existing attributes.
- In JavaScript: `getAttribute('data-*')`, `setAttribute('data-*', value)`.
- `element.dataset` as DOMStringMap for all data attributes.

When to use?

- Use data attributes for storing metadata.
- Track state (selected/disabled).
- Dynamic elements with extra info not in HTML attributes.

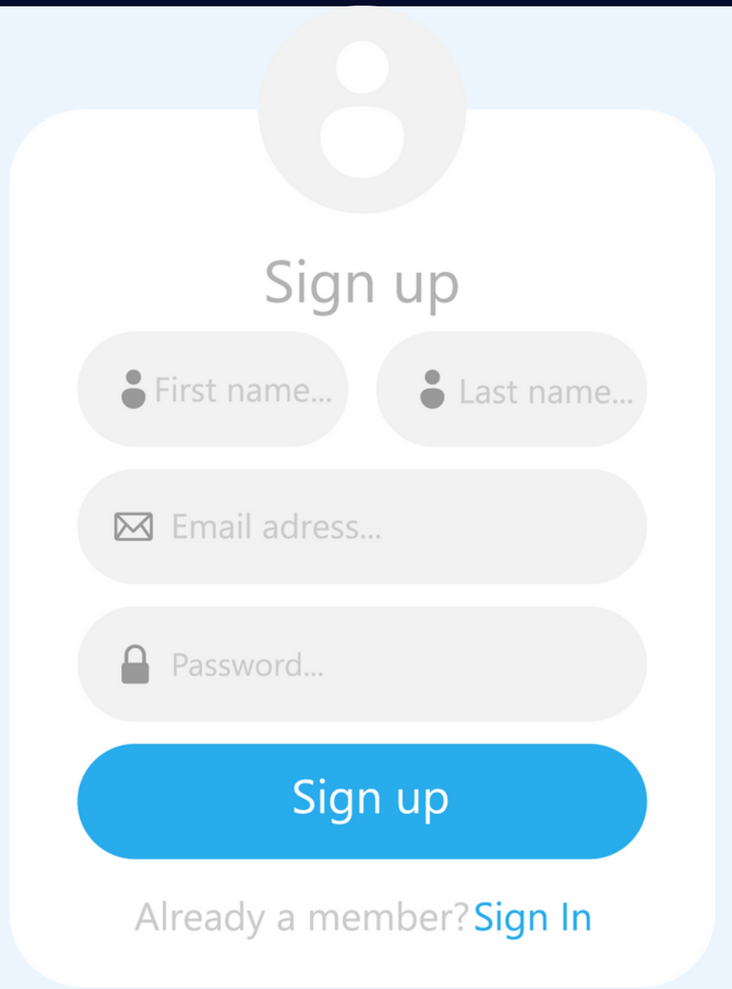
Take Note

- `dataset` for `data-*` attributes only.
- JavaScript changes in `dataset` are session-specific.

MASYNCTECH
SCHOOL

SELECTING

FORM ELEMENTS



JS

ADVANCED

SELECTING FORM ELEMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Select form elements in JavaScript.
- Methods: `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`.
- Also, `querySelector()` and `querySelectorAll()`.

Basic Explanation

- Supermarket analogy for web form selection.
- Specific: Unique ID elements.
- General: Group-related elements, e.g., checkboxes.
- JavaScript provides tools for both scenarios.

Best Practices

- Check if selector returns an element to avoid errors (null).
- `getElementById()`: Faster, simpler for ID selection.
- `querySelectorAll()`: Returns NodeList, not full array methods.

Deep Dive

- JS selects HTML form elements.
- Methods: `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`.
- `querySelector()` and `querySelectorAll()` for advanced querying.

When to use?

- `getElementById()` for unique elements.
- `querySelector()` for complex or first match.
- `querySelectorAll()`, `getElementsByClassName()`, or `getElementsByTagName()` for multiple elements.

Take Note

- `getElementById()`: Case-sensitive.
- `getElementsByClassName()`, `getElementsByTagName()`: Case-insensitive.
- `querySelectorAll()`: Static NodeList, no automatic DOM updates.

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

JS

ADVANCED

TRAVERSING OVERVIEW

MASYNCTECH SCHOOL

High-Level Explanation

- Traversing in JavaScript DOM: Navigation through HTML structure.
- Essential for locating, accessing, and manipulating nodes.
- Involves moving through the DOM tree hierarchy.

Basic Explanation

- Family tree analogy for DOM traversal.
- Seeking relatives like grandparents or cousins.
- Follow lines up, down, or sideways.
- Web page elements form a 'family tree.'
- Traversing means moving between family members.
- Goal: Finding the needed element.

Take Note

- Traversal impact on complex DOMs.
- Watch for browser compatibility.
- Beware of incorrect traversing causing bugs.

Deep Dive

- Traversing in DOM like a family tree search.
- Finding specific elements in the hierarchy.
- Elements can be parents, children, or siblings.
- Techniques for moving up and down hierarchy.
- Use properties like `parentNode`, `nextSibling`.
- Methods like `querySelector` and `getElementsByTagName`.
- Method choice depends on task and DOM complexity.

When to use?

- Traversing benefits:
  - Access dynamically generated elements.
  - Navigate complex, nested web apps.
  - Create reusable components independent of HTML structure.

Best Practices

- Specificity speeds up traversal.
- Cache elements for reuse.
- Opt for modern methods (e.g., `querySelector`).
- Minimize excessive DOM manipulation.
- Efficiency is key to avoid performance problems.

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING parentNode()

JS

ADVANCED

USING `PARENTNODE` IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- `parentNode` in DOM traversal.
- Move from child to immediate parent.
- Useful for relative positioning.

Basic Explanation

- `parentNode` in forest analogy.
- Baby tree to mother tree.
- Look directly up for parent element.

Best Practices

- Check for `null` before using `parentNode`.
- Cache parent node for repeated use.
- Understand the DOM tree structure to avoid unnecessary traversal.

Deep Dive

- Nodes in hierarchical relationships.
- `parentNode` accesses immediate parent.
- Trace back up hierarchy.
- Useful for dynamic content and nested structures.

When to use?

- Use `parentNode` when:
  - Manipulating a child's parent.
  - Navigating from deeply nested elements.
  - Code needs to be generic and independent of IDs or classes.

Take Note

- `parentNode` returns `null` for root or removed nodes.
- Includes all node types (e.g., text or comments).

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING childNodes

JS

ADVANCED

USING CHILDNODES IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- - `childNodes` property: Returns NodeList of child nodes
- - Includes text, comment nodes

- Useful for traversing DOM tree, accessing parent's children

Basic Explanation

- - Analogy: Toy box for `childNodes`
- - Get everything when you dump it out

- Includes toys (elements) and not-so-good stuff (text, comments)

Best Practices

- Type checking important for specific element nodes.
- Watch out for whitespace treated as text nodes.
- `children` property preferred for element nodes.

Deep Dive

- - `childNodes` for raw, low-level DOM traversal
- - Access immediate children
- - Includes element, text, comment nodes

- Requires care to avoid unexpected results

When to use?

- `childNodes` usage:
  - Loop through all nodes, regardless of type.
  - Working with XML where text nodes matter.
  - Need fine-grained DOM structure control.

Take Note

- `childNodes` returns NodeList, array-like but lacks all array methods.
- NodeList is live, updates real-time with DOM changes.

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING

previousSibling()

JS

ADVANCED

USING `PREVIOUSSIBLING` IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- `previousSibling` navigates sideways in DOM.
- Selects immediately preceding sibling node.
- Useful for accessing related elements with known parent.

Basic Explanation

- `previousSibling` akin to kids in an ice cream line.
- Identifies the node just before the current one.
- Works within the same parent element.

Best Practices

- Check node type when using `previousSibling`.
- Handle `null` for elements lacking previous siblings.
- Use `previousSibling` carefully for performance optimization.

Deep Dive

- `previousSibling` returns various node types.
- Granularity offers power and complexity.
- May require extra checks for intended node type.

When to use?

- Use `previousSibling` when:
  - Linear navigation through elements.
  - Accessing immediate neighbor of a node.
  - Prioritizing performance by avoiding extra queries.

Take Note

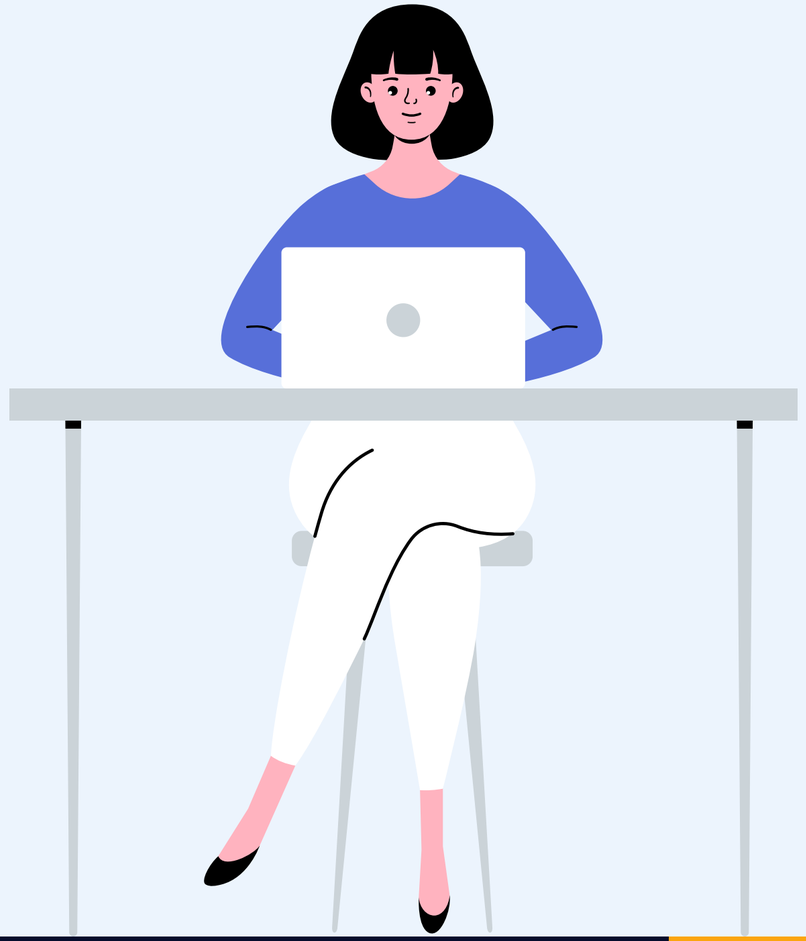
- `previousSibling` can return `null` for first child.
- Returns various node types, including text and comments.

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING

nextSibling()



JS

ADVANCED

USING NEXTSIBLING` IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- `nextSibling` accesses succeeding sibling.
- Allows horizontal traversal within same parent.
- Navigates to elements sharing the same parent.

Basic Explanation

- `nextSibling` like a bookshelf with books.
- Finds next node with the same parent.
- Works within the DOM tree.

Best Practices

- Confirm node type when using `nextSibling`.
- Prepare for `null` if the last child has no next sibling.
- Optimize performance by minimizing reflows and repaints with `nextSibling`.

Deep Dive

- `nextSibling` moves in opposite direction of `previousSibling`.
- Returns various node types, including text and comments.
- Additional checks often needed for intended node type.

When to use?

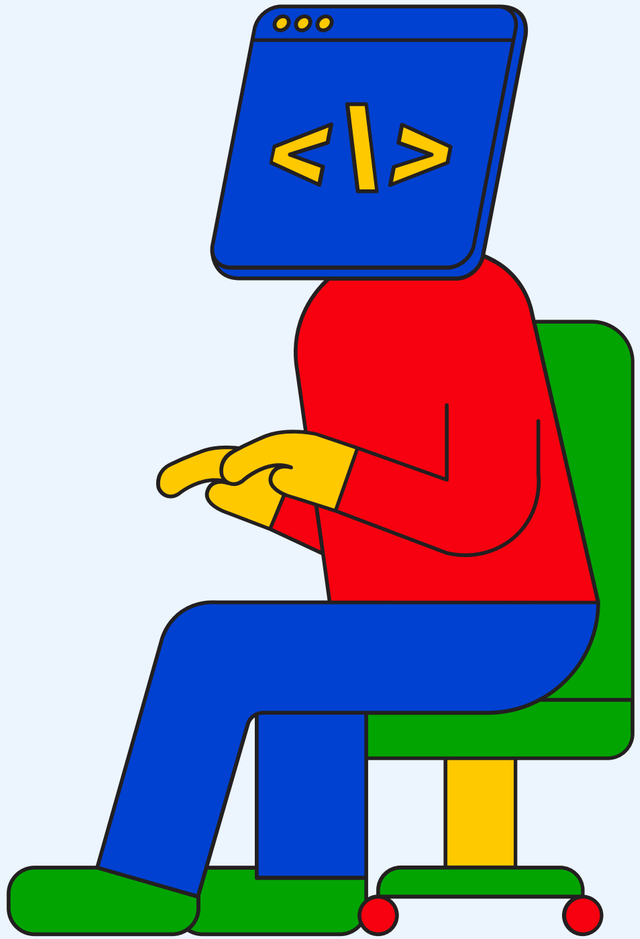
- Utilize `nextSibling` when:
  - Accessing immediate next sibling of a node.
  - Traversing parent node's children sequentially.
  - Enhancing performance by minimizing extra queries.

Take Note

- `nextSibling` may return `null` without a next sibling.
- Potential for text or comment nodes; check node type.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING

previousElementSibling()

JS

ADVANCED

USING USING PREVIOUSELEMENTSIBLING` IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- `previousElementSibling` accesses preceding sibling.
- Considers only element nodes, skips text/comments.
- Efficient for horizontal DOM traversal.

Basic Explanation

- In a queue, the person in front is your "previous sibling."
- `previousElementSibling` finds the immediate preceding node.
- Similar to identifying the person right in front.

Best Practices

- Perform null checks for `previousElementSibling`.
- No need for node type checks; it filters non-element nodes.

Deep Dive

- `previousElementSibling` gets preceding element.
- Ignores text and comment nodes.
- Returns closest previous element or `null`.

When to use?

- Employ `previousElementSibling` when:
  - Navigating backward from a known element.
  - Focusing solely on element nodes, ignoring text/comments.
  - Aiming for efficient element traversal, reducing DOM queries.

Take Note

- `previousElementSibling` returns `null` for the first child.
- Ignores non-element nodes, simpler than `previousSibling`.

MASYNCTECH
SCHOOL

TRANSVERSING THE DOM

USING

nextElementSibling()

JS

ADVANCED

USING USING NEXTELEMENTSIBLING` IN DOM TRAVERSAL

MASYNCTECH SCHOOL

High-Level Explanation

- `nextElementSibling` returns following sibling element.
- Returns `null` if no such element exists.

Basic Explanation

- `nextElementSibling` like hitting "Next" on a playlist.
- Reveals the next element in the sequence.
- Returns `null` at the end, like the last song.

Best Practices

- Check for `null` to prevent errors with `nextElementSibling`.
- Consider bi-directional traversal with `previousElementSibling`.
- Use IDs or classes for critical navigation alongside `nextElementSibling`.

Deep Dive

- `nextElementSibling`: Sibling-to-sibling DOM navigation.
- Ignores non-element nodes (text, comments).
- Returns only actual HTML elements.

When to use?

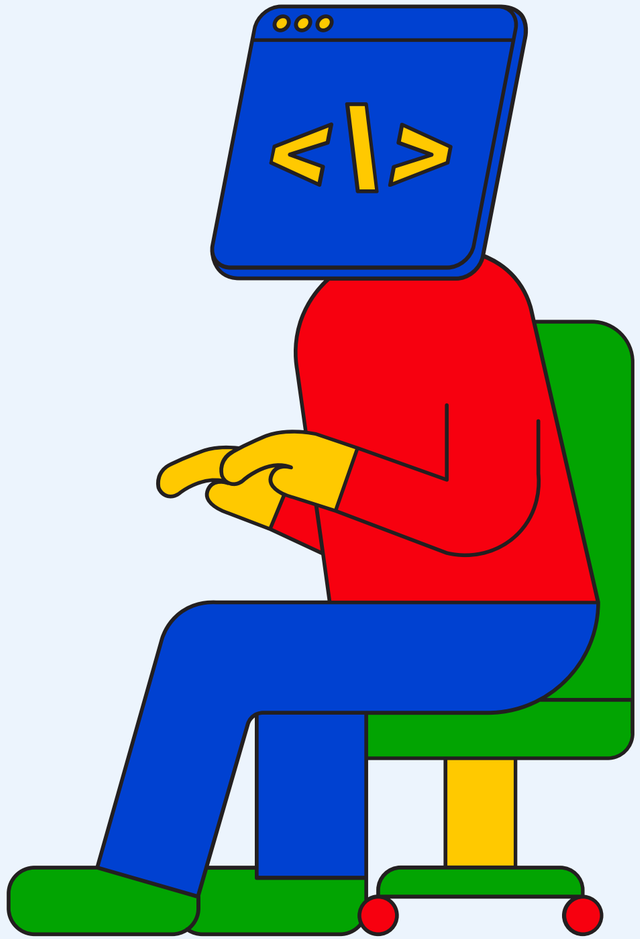
- Use `nextElementSibling` when:
  - Iterating over a list to find what comes next.
  - Traversing menus for dynamic styles/classes.
  - Guiding users through interactive forms.
  - Managing slideshows or carousels to find the next slide.

Take Note

- Returns elements, ignores text and non-element nodes.
- Read-only; cannot be manually set.
- Generally well-supported but check compatibility.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM MANIPULATION

JS

ADVANCED

MASYNCTECH
SCHOOL

DOM MANIPULATION

INNERHTML
TEXTCONTENT
INNERTEXT

JS

ADVANCED

USING TEXTCONTENT, INNERHTML & INNERTEXT

MASYNCTECH SCHOOL

High-Level Explanation

- `textContent`: Accesses plain text in an element and its children.
- `innerHTML`: Accesses HTML content within an element.
- `innerText`: Accesses visible text in an element.

Basic Explanation

- `textContent`: Gets plain text, like reading a book's story.
- `innerHTML`: Gets text, images, fonts, colors, like a styled book.
- `innerText`: Gets only visible words, like reading open book pages.

Best Practices

- `textContent`: Safely handles plain text, no XSS risk.
- `innerHTML`: Caution needed for dynamic HTML, XSS risk.
- `innerText`: Triggers reflow, impacts performance.

Deep Dive

- `textContent`: Returns raw text, ignores HTML tags, and is faster.
- `innerHTML`: Includes HTML tags, can expose to XSS if misused.
- `innerText`: Considers styling and doesn't show hidden CSS text.

When to use?

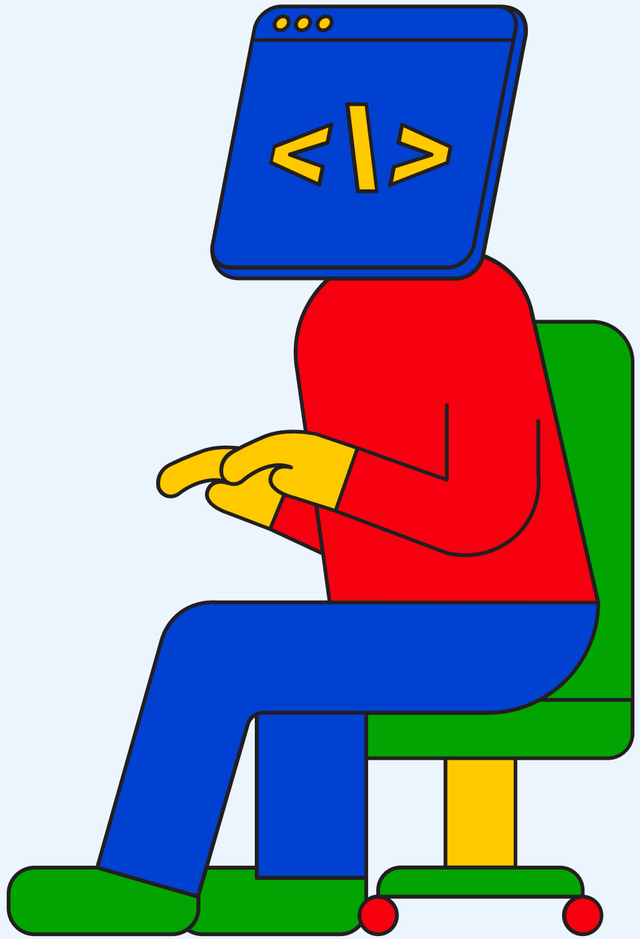
- `textContent`: For quick text access, no HTML structure.
- `innerHTML`: Embedded HTML content handling.
- `innerText`: Visible text with styling.

Take Note

- `textContent`: Fastest and safest for text handling.
- `innerHTML`: Handles HTML, but watch for security.
- `innerText`: Slower, considers visibility.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM
MANIPULATION
setAttribute()

JS

ADVANCED

DOM MANIPULATION USING USING SETATTRIBUTE

MASYNCTECH SCHOOL

High-Level Explanation

- `setAttribute`: Sets an attribute's value on an element.

Basic Explanation

- `setAttribute` analogy: Like changing shirt color in a game.
- It modifies attributes in HTML elements.
- Enables customization, e.g., shirt color from red to blue.

Best Practices

- `setAttribute` use cases:
- Works for HTML and custom data attributes (`data-*`).
- Caution with Boolean attributes; properties often better.

Deep Dive

- Method updates or adds attributes in HTML elements.
- Takes 2 arguments: attribute name and new value.
- Updates existing or creates new attributes.

When to use?

- `setAttribute` usage:
- Modify attribute values programmatically.
- Add new attributes to HTML elements.
- Useful for dynamic attribute manipulation.

Take Note

- `setAttribute` behavior:
- Overwrites existing attribute values.
- To modify, retrieve existing value and set a new one.

MASYNCTECH
SCHOOL

DOM MANIPULATION

getAttribute()

JS

ADVANCED

DOM MANIPULATION USING USING GETATTRIBUTE

MASYNCTECH SCHOOL

High-Level Explanation

- `getAttribute`: Retrieves specified attribute value from an HTML element.

Basic Explanation

- `getAttribute` analogy:
- Like using a cheat sheet in a game.
- Check it for specific details, like your character's armor type.

Best Practices

- Use `getAttribute` for HTML and `data-*` attributes.
- Boolean attributes may return empty string as `true`.

Deep Dive

- `getAttribute` method:
- Accesses current attribute value.
- Accepts attribute name as argument.
- Returns value as string, or `null` if attribute doesn't exist.

When to use?

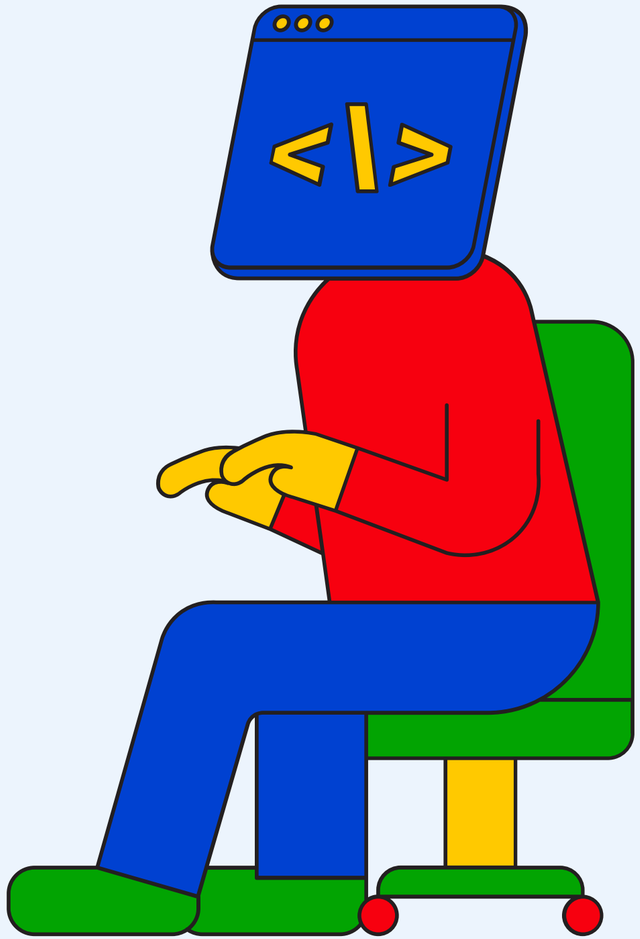
- `getAttribute` usage:
- To read current HTML attribute value.
- Useful for decision-making or code manipulations.

Take Note

- `getAttribute` retrieves the initial attribute value.
- It doesn't reflect property changes.
- Useful for accessing initial values.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM MANIPULATION

createElement()

appendChild()

JS

ADVANCED

USING CREATEELEMENT & APPENDCHILD()

MASYNCTECH SCHOOL

High-Level Explanation

- `createElement` creates empty HTML elements.
- `appendChild()` inserts elements into the DOM.
- Essential for dynamic HTML element creation.
- `appendChild()` adds elements as last children.

Basic Explanation

- `createElement` is like getting a new Lego piece.
- Piece isn't part of the structure yet.
- `appendChild()` is attaching the Lego piece.
- It becomes a permanent part of the layout.

Best Practices

- Append elements semantically in HTML.
- Use `textContent` to avoid script injection.
- Batch DOM operations for better performance.
- Consider accessibility when creating elements.

Deep Dive

- `createElement` constructs an HTML element.
- Doesn't append to DOM, stays virtual.
- `appendChild()` adds to DOM, last child.
- Connects elements to specified parent.

When to use?

- Dynamic list item addition
- Safe insertion of user-generated content
- Creating single-page applications (SPAs)
- Form validation with immediate feedback

Take Note

- `createElement` creates but doesn't add to DOM.
- `appendChild()` can relocate elements.
- Watch out for inefficient DOM changes.

MASYNCTECH
SCHOOL

DOM MANIPULATION

insertBefore()

JS

ADVANCED

USING INSERTBEFORE

MASYNCTECH SCHOOL

High-Level Explanation

- `insertBefore` inserts a node before an existing child.
- Provides precise control over insertion position.
- Useful for specifying element placement.

Basic Explanation

- `insertBefore` is like placing books on a shelf.
- Allows precise placement, like before a favorite book.
- `appendChild()` puts the book at the end.

Best Practices

- Maintain semantic sense in HTML structure.
- Batch DOM operations for performance.
- Verify conditions to avoid unintended placements.

Deep Dive

- `insertBefore` offers nuanced element positioning.
- Requires two arguments: new node and existing child.
- If second argument is `null`, it's like `appendChild()`.

When to use?

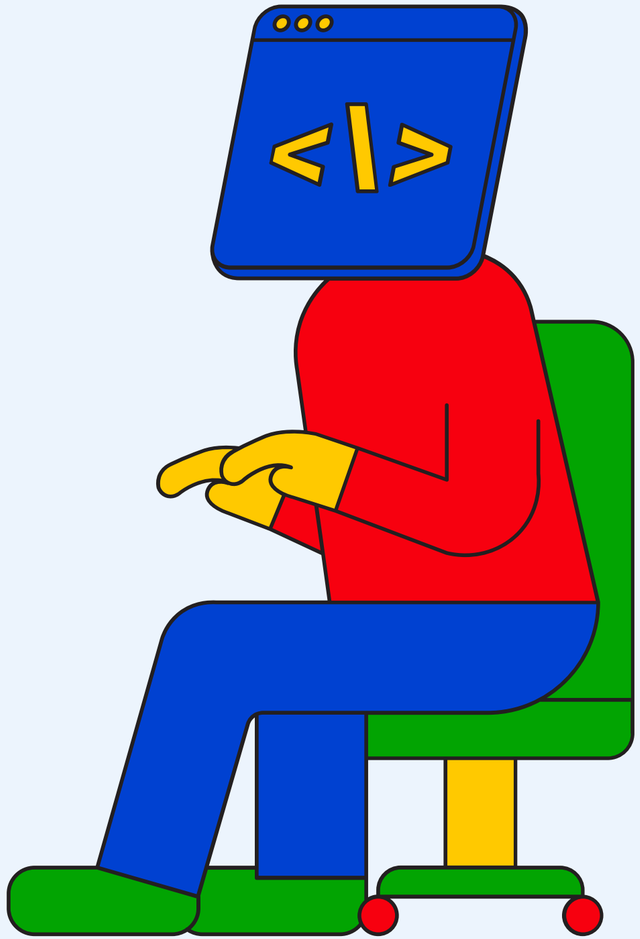
- Insert into a sorted list without reordering.
- Add ads within an article.
- Change element order based on user actions.
- Paginate content at specific positions.

Take Note

- Second argument must be a child or `null`.
- `insertBefore` relocates an existing node.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM MANIPULATION

REPLACING ELEMENTS

JS

ADVANCED

REPLACING ELEMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- `replaceChild` swaps an existing child node.
- Replaces one element with another.
- Alters the DOM structure.

Basic Explanation

- `replaceChild` swaps elements like changing photos.
- Replaces the old element with a new one.
- Simulates updating elements on a webpage.

Best Practices

- Prepare the new element fully.
- Minimize layout recalculations.
- Don't break existing JavaScript references.
- Watch out for event handler issues.

Deep Dive

- `replaceChild` on a parent node.
- Arguments: new and existing nodes.
- Replaces existing with new node.
- Atomic update of the DOM.

When to use?

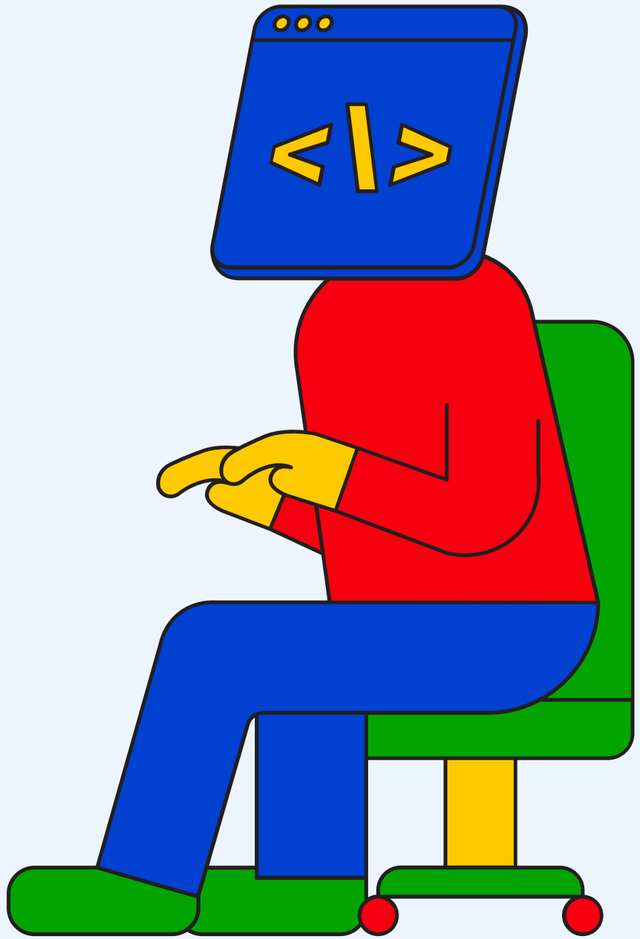
- Update UI components selectively.
- Swap elements based on interactions or updates.
- Create tabbed interfaces with dynamic content.

Take Note

- Replace direct child nodes only.
- Removed child can be reused elsewhere.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM
MANIPULATION

removeChild()

JS

ADVANCED

USING REMOVECHILD

MASYNCTECH SCHOOL

High-Level Explanation

- `removeChild` removes a child element.
- Eliminates the targeted element and its content.
- Alters the DOM structure.

Basic Explanation

- `removeChild` is like removing a chocolate.
- Eliminates unwanted items, makes room for desired ones.
- Instantaneous change in the web structure.

Best Practices

- Check for attached event listeners.
- Beware of live collection updates.
- Ensure the child is part of the parent.

Deep Dive

- `removeChild` on parent with child argument.
- Returns removed node for potential re-insertion.
- Instantaneous removal from the DOM.

When to use?

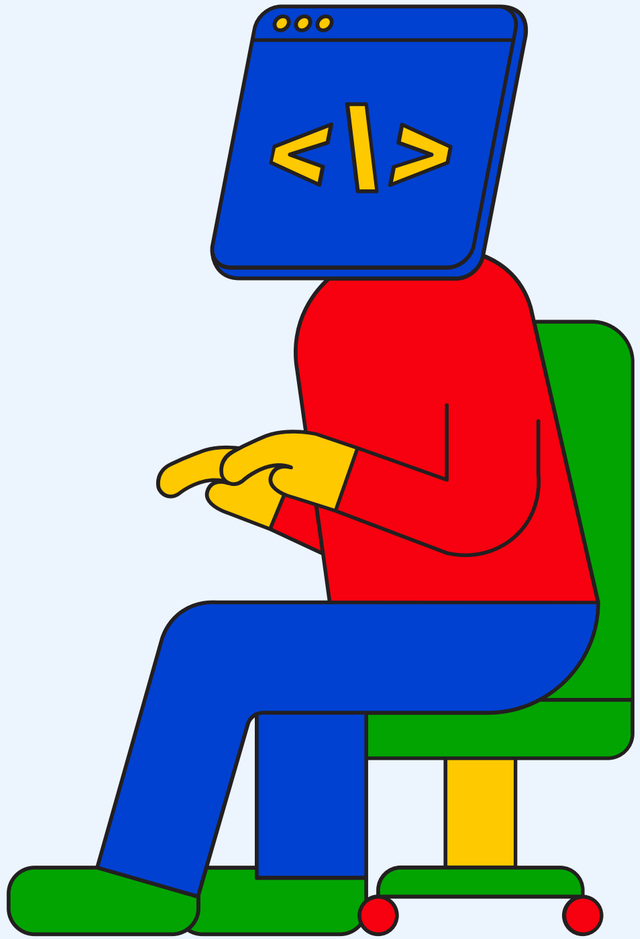
- Remove outdated content from web pages.
- Implement delete buttons for items.
- Optimize performance by cleaning up DOM.

Take Note

- Removed node exists in memory, can be reinserted.
- Removing non-direct child causes DOM exception.

MASYNCTECH
SCHOOL

CODE DEMO



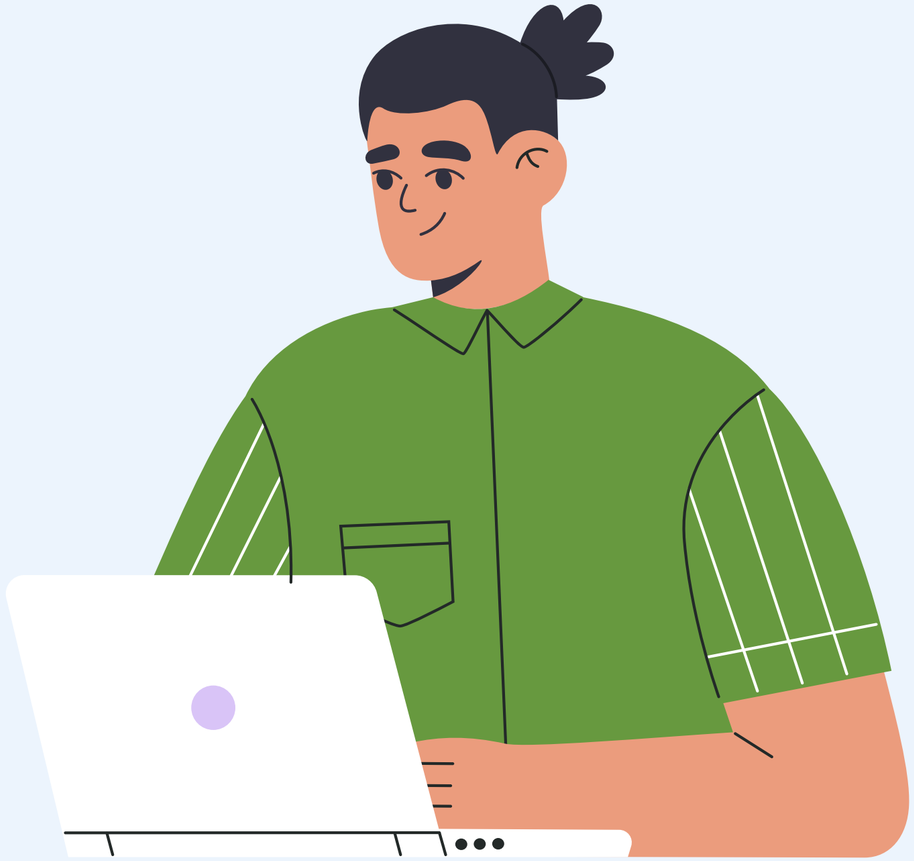
JS

ADVANCED

MASYNCTECH
SCHOOL

DOM MANIPULATION

classList



JS

ADVANCED

USING CLASSLIST

MASYNCTECH SCHOOL

High-Level Explanation

- `classList` manipulates HTML element classes.
- Add, remove, toggle, and check class existence.

Basic Explanation

- Wardrobe as an HTML element.
- Classes represent clothing types.
- `classList` for adding, removing, switching.

Best Practices

- Use descriptive class names.
- Check class existence before removal.
- Consider class sequence for CSS logic.

Deep Dive

- `classList` returns a live `DOMTokenList`.
- Offers methods like `add()`, `remove()`, `toggle()`, and `contains()`.
- Provides intuitive class list manipulation.

When to use?

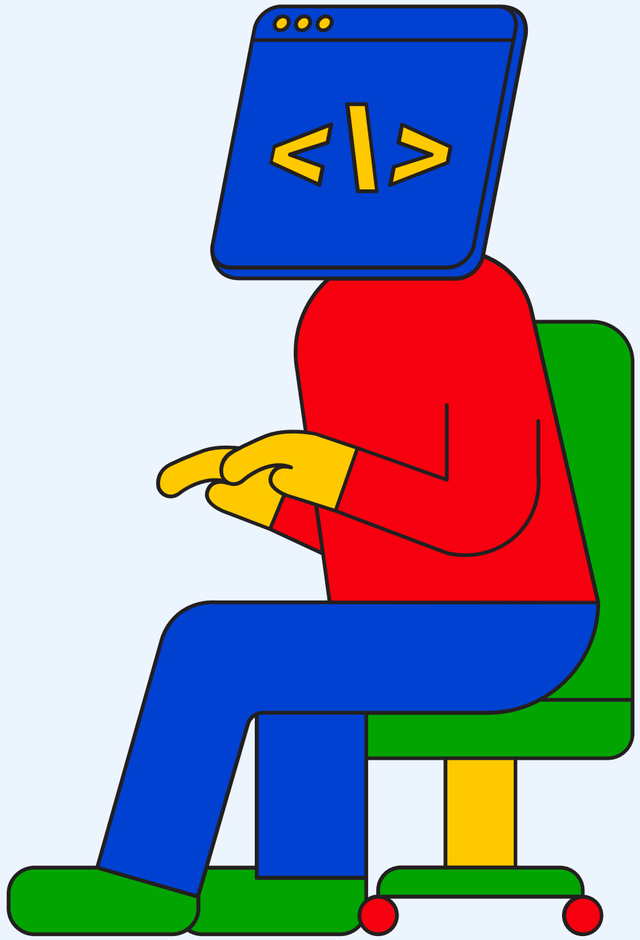
- Dynamic styling with user interactions.
- Toggles for UI elements like buttons.
- Filtering in lists or grids.

Take Note

- `classList` is a live collection.
- Automatically updates with class attribute changes.
- Chain methods for compact code.

MASYNCTECH
SCHOOL

CODE DEMO

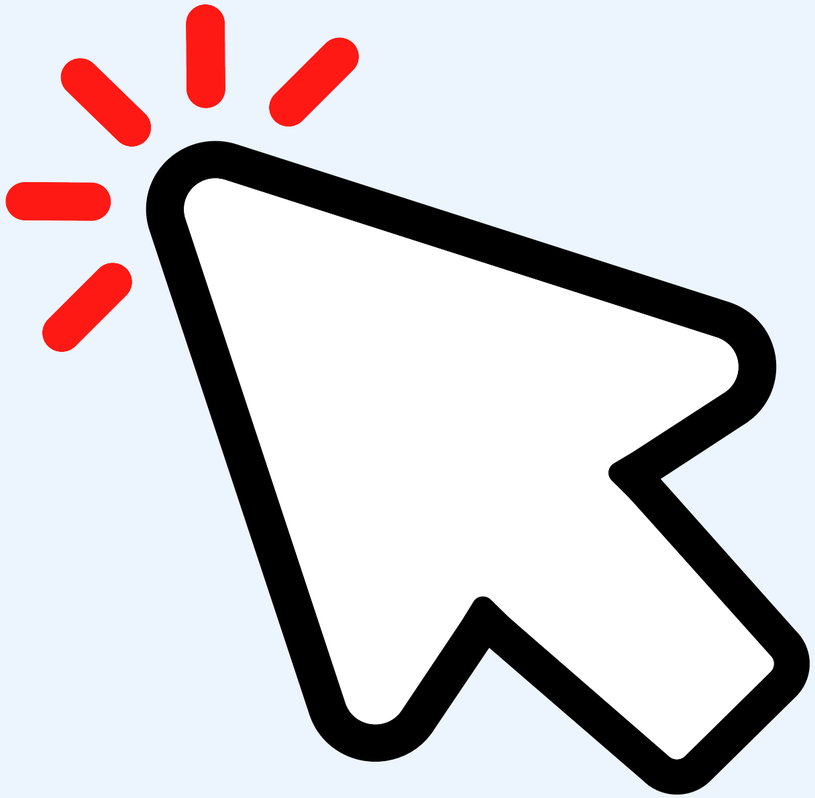


JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

WHAT ARE EVENTS

JS

ADVANCED

WHAT ARE DOM EVENTS?

MASYNCTECH SCHOOL

High-Level Explanation

- DOM events: Browser interactions or occurrences
- User/system-triggered
- Examples: button click, window resize, form submit

Basic Explanation

- Analogy: DOM events as buttons in a game
- Click, scroll, etc., like pressing buttons
- Instruct website/app to perform actions

Best Practices

- Event delegation: Single listener for multiple child elements.
- Event propagation awareness: Understand bubbling.
- Remove unused event listeners to prevent memory leaks.

Deep Dive

- DOM events crucial for interactive web apps
- Generated on user/system interactions
- Simple (mouse-over) to complex (data fetch)
- Enable dynamic, interactive experiences

When to use?

- Use DOM events to:
  - Enhance interaction (e.g., toggle menus)
  - Validate data (e.g., form field on blur)
  - Load dynamic content (e.g., scroll-triggered fetch)

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

COMMON EVENTS

TYPES

JS

ADVANCED

COMMON EVENT TYPES

MASYNCTECH SCHOOL

High-Level Explanation

- JavaScript DOM events: Numerous types
- Mouse events, e.g., `click`
- Form events, e.g., `submit`
- Linked to specific interactions or triggers

Mouse Events

- `click`: Triggered on mouse click.
- `dblclick`: Triggered on double-click.
- `mouseover`: Fires on mouse enter.
- `mouseout`: Fires on mouse exit.

Keyboard Events

- `keydown`: Triggered on key press down.
- `keyup`: Triggered on key release.
- `keypress`: Triggered on key press and release.

Windows Events

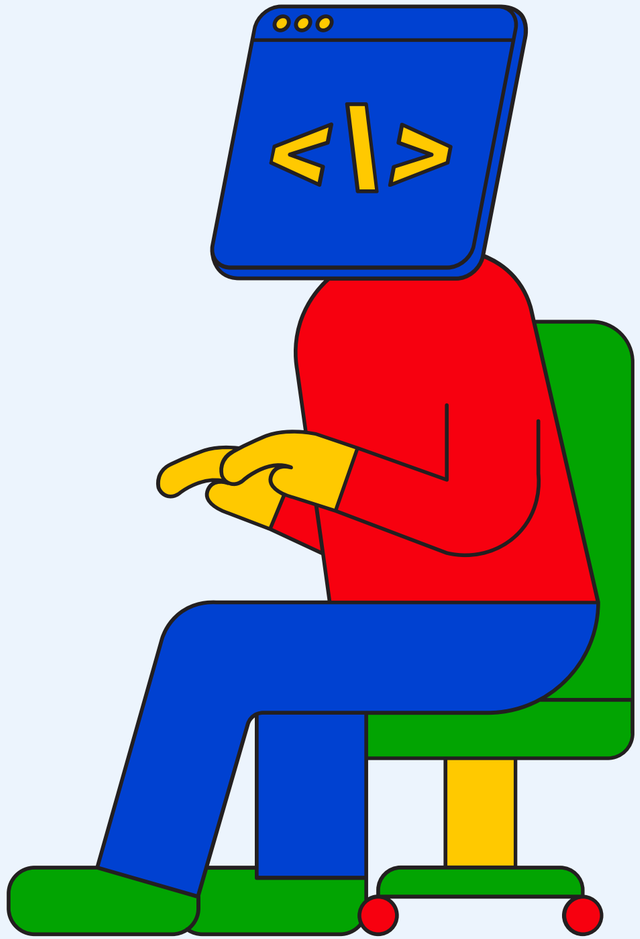
- `resize`: Triggered on browser window resize.
- `scroll`: Triggered on viewport scrolling.

Forms Events

- `submit`: Triggered on form submission.
- `focus`: Triggered on element focus.
- `blur`: Triggered on element loss of focus.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

INLINE VS TRADITIONAL EVENTS HANDLERS

JS

ADVANCED

INLINE EVENT VS TRADITIONAL EVENT HANDLERS

MASYNCTECH SCHOOL

High-Level Explanation

- Inline event handlers: JavaScript code in HTML
- Traditional event handlers: Separate JS file or script block

Deep Dive

- Inline Event Handlers:
  - JavaScript in HTML tags (`onclick`, `onsubmit`)
  - Tight coupling of HTML and JS

- Traditional Event Handlers:
  - Separates JavaScript from HTML
  - `element.addEventListener` promotes separation of concerns

Basic Explanation

- Analogy: Inline vs. Traditional event handlers
- Inline: Like a direct speed dial, quick but less flexible
- Traditional: Like a separate app, more setup but easier changes

When to use?

- Inline Event Handlers:
  - Quick prototyping, limited control over JS
  - HTML generated by inflexible systems

- Traditional Event Handlers:
  - Larger, complex apps
  - Multiple functions for one event
  - Advanced features like event delegation

INLINE EVENT VS TRADITIONAL EVENT HANDLERS

MASYNCTECH SCHOOL

When to use?

- Inline Event Handlers:
  - Quick prototyping, limited control over JS
  - HTML generated by inflexible systems

- Traditional Event Handlers:
  - Larger, complex apps
  - Multiple functions for one event
  - Advanced features like event delegation

Best Practices

- Inline Event Handlers:
  - Avoid for maintainability
  - Keep them simple; use separate functions

- Traditional Event Handlers:
  - Prefer for flexibility and maintainability
  - Utilize event delegation for multiple similar elements

Take Note

- Inline event handlers: No event delegation support.
- Traditional event handlers: Easier event object manipulation.
- Traditional handlers: Simplify event unbinding, preventing memory leaks.

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

addEventListener()

METHOD

JS

ADVANCED

`ADDEVENTLISTENER` METHOD

MASYNCTECH SCHOOL

High-Level Explanation

- `addEventListener`: JS function for event handling
- Attach event handlers to DOM elements
- Listen for event types, execute functions

Basic Explanation

- Analogy: `addEventListener` as programming a remote button
- Remote control (DOM element)
- Program button (event) to change channel (execute function)

Best Practices

- Prefer anonymous or named functions over inline code.
- Check for browser compatibility despite wide support.
- Remove unused event listeners with `removeEventListener` to save resources.

Deep Dive

- `addEventListener` is flexible and powerful.
- Separates HTML and JS, multiple event handlers.
- Three main arguments: event type, function, options or capture.

When to use?

- Attaching multiple handlers to the same element.
- Separating HTML and JavaScript for clarity.
- Utilizing advanced event features (capturing, propagation control).

Take Note

- `addEventListener` doesn't overwrite existing handlers.
- `this` in handler refers to the attached DOM element.
- Named function needed for removal; anonymous can't be removed.

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

removeEventListener()

METHOD

JS

ADVANCED

REMOVING EVENT LISTENERS WITH `REMOVEEVENTLISTENER`

MASYNCTECH SCHOOL

High-Level Explanation

- `removeEventListener`: Detach event listener
- Undo `addEventListener`, prevent event triggering

Basic Explanation

- Analogy: `removeEventListener` as unprogramming a remote button.
- Previously changed channel; now, pressing does nothing.

Best Practices

- Prefer named functions for easy removal.
- Caution with `removeEventListener` in conditions.
- Parameters must match `addEventListener`, including options and useCapture flags.

Deep Dive

- `addEventListener` is flexible and powerful.
- Separates HTML and JS, multiple event handlers.
- Three main arguments: event type, function, options or capture.

When to use?

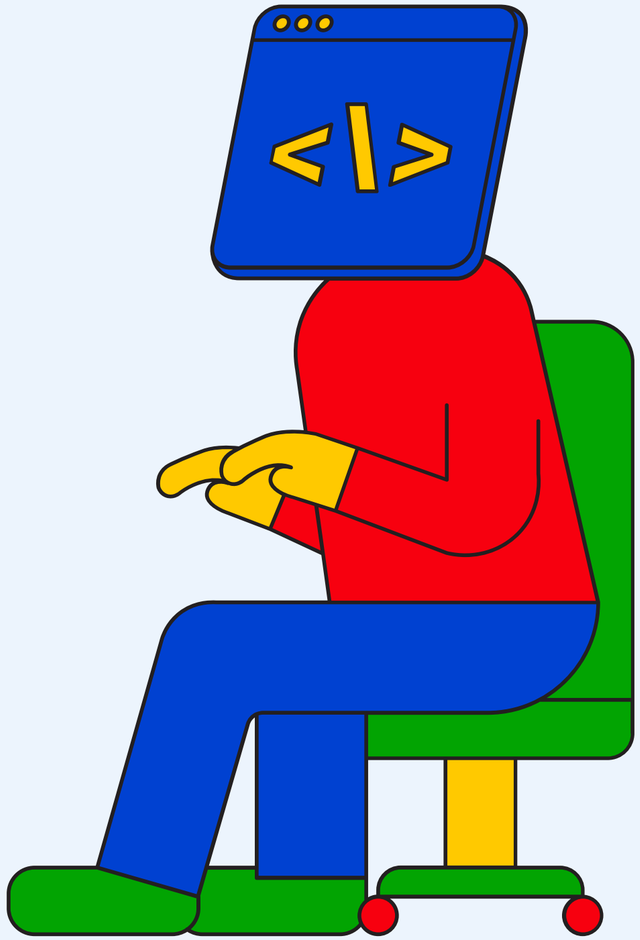
- To prevent an event from executing its function.
- For dynamically adding/removing elements and events.
- To temporarily disable specific functionality.

Take Note

- `removeEventListener` requires the same function as `addEventListener`.
- Binding with `.bind()` requires reference to the bound function for removal.

MASYNCTECH
SCHOOL

CODE DEMO



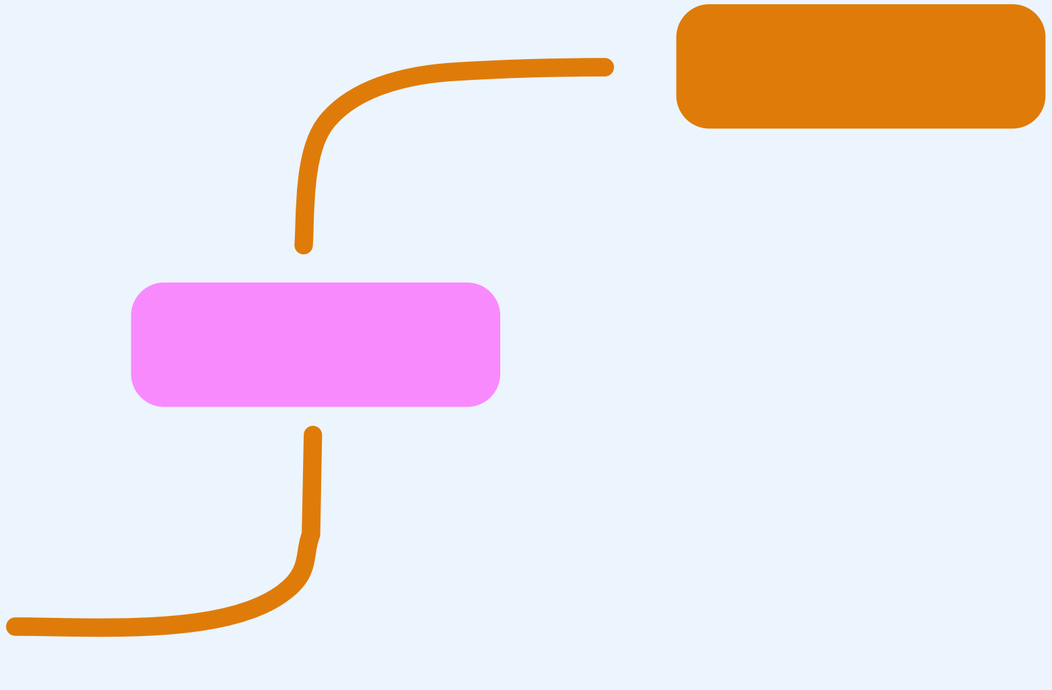
JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

UNDERSTANDING THE EVENT FLOW



JS

ADVANCED

UNDERSTANDING THE EVENT FLOW

MASYNCTECH SCHOOL

High-Level Explanation

- Event flow: Capturing, target, and bubbling phases
- Describes event propagation through DOM

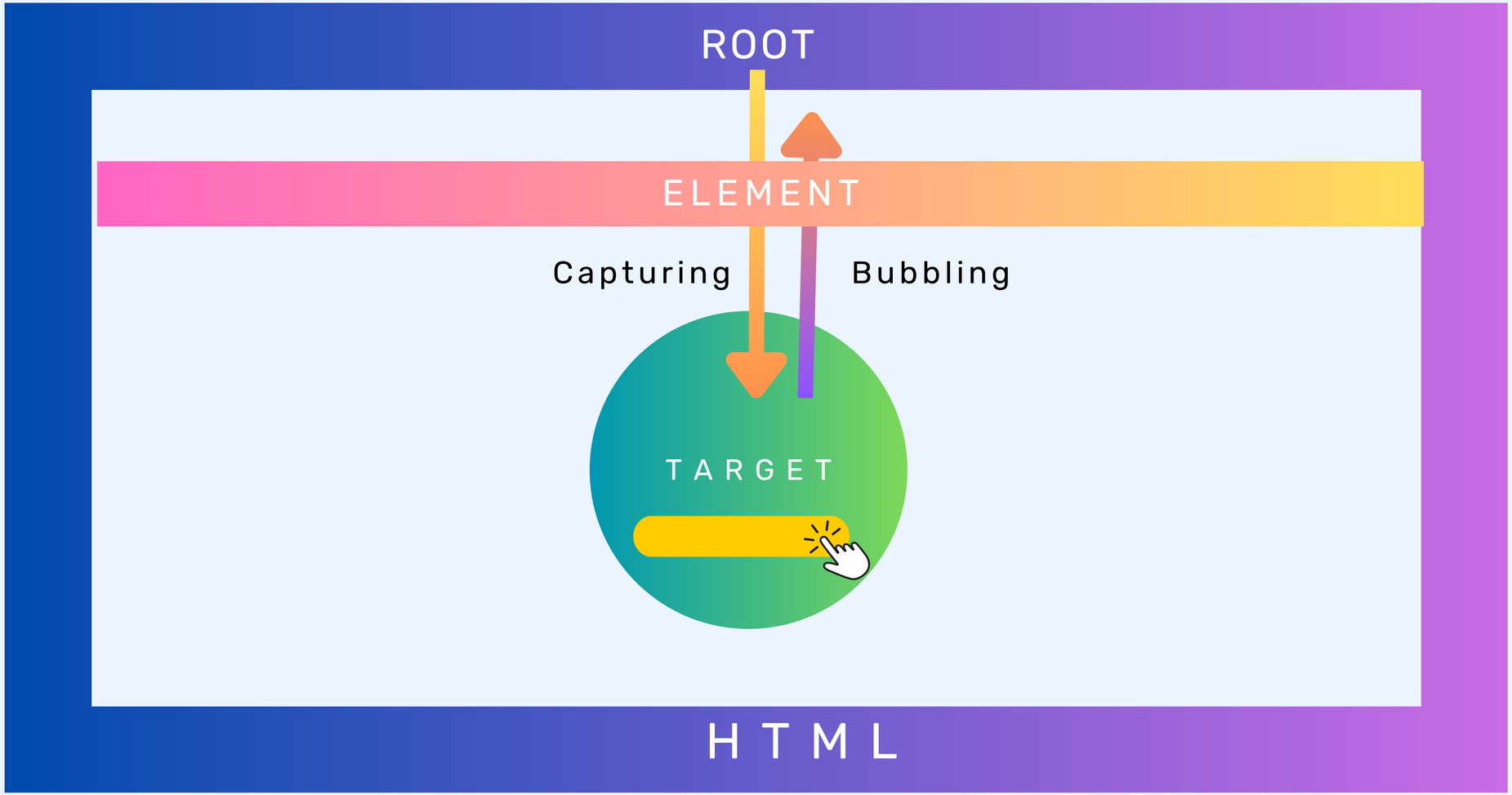
Basic Explanation

- Analogy: Event flow as a family gathering rumor
- Capturing: Oldest to target
- Target: Confirmation/denial
- Bubbling: Target to oldest family members

Deep Dive

- Event flow affects ancestors and children.
- Event flow phases:
  - Capturing: Root to target
  - Target: At target
  - Bubbling: Target to root

EVENT FLOW



UNDERSTANDING THE EVENT FLOW

MASYNCTECH SCHOOL

High-Level Explanation

- Event flow: Capturing, target, and bubbling phases
- Describes event propagation through DOM

Basic Explanation

- Analogy: Event flow as a family gathering rumor
- Capturing: Oldest to target
- Target: Confirmation/denial
- Bubbling: Target to oldest family members

Best Practices

- Caution with stopping propagation; unintended effects possible.
- Event delegation improves performance but needs careful implementation.
- `addEventListener` can specify phase with `capture` parameter.

Deep Dive

- Event flow affects ancestors and children.
- Event flow phases:
  - Capturing: Root to target
  - Target: At target
  - Bubbling: Target to root

When to use?

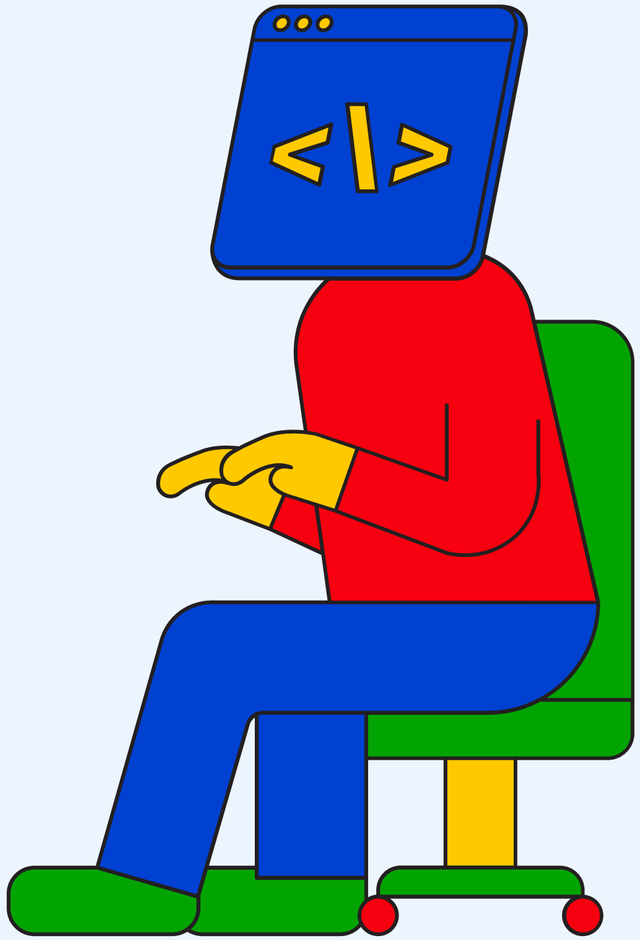
- Event flow understanding important for nested elements.
- Control event propagation, vital in event delegation.
- Parent element listens for child-triggered events.

Take Note

- Events must have `bubbles: true` to bubble.
- Not all events have capturing phases; some start at target.
- `capture` parameter is optional, default is `false` (bubbling phase).

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

event.stopPropagation()

METHOD

JS

ADVANCED

USING `EVENT.STOPPROPAGATION()`

MASYNCTECH SCHOOL

High-Level Explanation

- `event.stopPropagation()`: Prevent event propagation
- Stops capturing or bubbling phases in event flow

Basic Explanation

- Analogy: `event.stopPropagation()` as "Telephone" game.
- Message stops with current element.
- Event doesn't continue to parent/child elements.

Best Practices

- Use `event.stopPropagation()` judiciously; it can cause unexpected issues.
- Comment code to clarify why propagation is stopped for other developers.

Deep Dive

- Event flow: Capture, target, bubble phases.
- `event.stopPropagation()`: Stops flow in current phase.
- Parent/child elements not notified when stopped.

When to use?

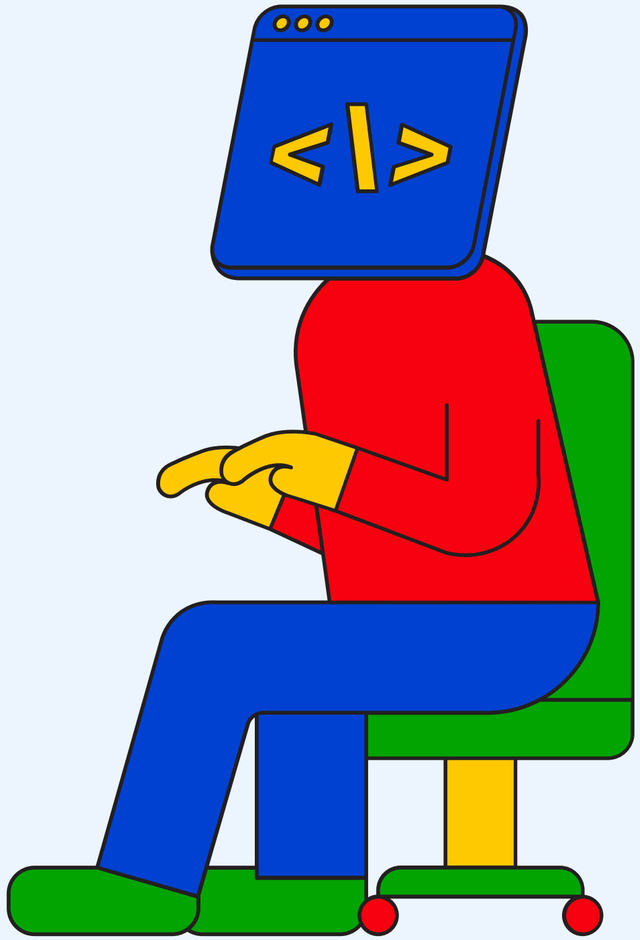
- Use `event.stopPropagation()` to prevent parent reactions to child events.
- Valuable in complex UIs with diverse element roles.

Take Note

- Stops propagation but not other handlers on the same element.
- Only halts event movement, doesn't affect element behavior.

MASYNCTECH
SCHOOL

CODE DEMO



JS

ADVANCED

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

EVENT DELEGATION

JS

ADVANCED

EVENT DELEGATION

MASYNCTECH SCHOOL

High-Level Explanation

- Event delegation: Delegate handling to a parent element for multiple children.

Deep Dive

- Event delegation: Attach one listener to a common parent.
- Parent listens for event and identifies which child was clicked.
- Memory-efficient and simplifies code maintenance.

Basic Explanation

- Analogy: Event delegation as a house party with a trash bin.
- One trash bin for all cups (common parent for multiple elements).
- Minimizes effort for handling common event.

When to use?

- Many child elements needing similar event handling.
- Dynamic addition/removal of child elements, avoiding listener management.

Best Practices

- Identify child element with `event.target`.
- Prefer events that bubble for delegation to work.
- Careful with stopping event propagation; can affect delegation.

Take Note

- Use `event.target` to identify event source.
- Not suitable for non-bubbling events.

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

MULTIPLE EVENTS HANDLERS

JS

ADVANCED

USING MULTIPLE EVENT LISTENERS ON AN ELEMENT

MASYNCTECH SCHOOL

High-Level Explanation

- Multiple event listeners for diverse interactions.
- Handles various types or aspects of one interaction.

Basic Explanation

- Smartphone screen: Tap, swipe, pinch = multiple listeners.
- Different interactions on the same element (screen).

Best Practices

- `addEventListener` for modularity.
- Careful with `event.stopPropagation()`.
- Remove unused listeners to free resources.

Deep Dive

- Multiple listeners for complex interactions.
- Separate handlers for clicks, movements, keyboard.
- Different stages of an interaction (e.g., drag-and-drop).

When to use?

- Element responds to multiple interactions.
- Break down complex behavior into manageable parts.

Take Note

- Listeners execute in order if all are capturing/bubbling.
- Removing one won't affect others on the same element.

MASYNCTECH
SCHOOL

DOM EVENT HANDLING

CUSTOM EVENTS

JS

ADVANCED

CREATING AND TRIGGERING CUSTOM EVENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Create custom events for specific situations.
- Useful when standard DOM events don't fit.
- Dispatch events with `new Event()` and `element.dispatchEvent()`.

Basic Explanation

- Analogy: Custom events as secret handshakes in code
- Unique communication between code parts
- Beyond standard events

Best Practices

- Descriptive event names for clarity.
- Minimal data in `detail` attribute.
- Namespace custom events for library/framework to prevent conflicts.

Deep Dive

- Use the `detail` property to carry additional data.
- Promote modularity and maintainability in applications.
- Facilitate communication between components without tight coupling.

When to use?

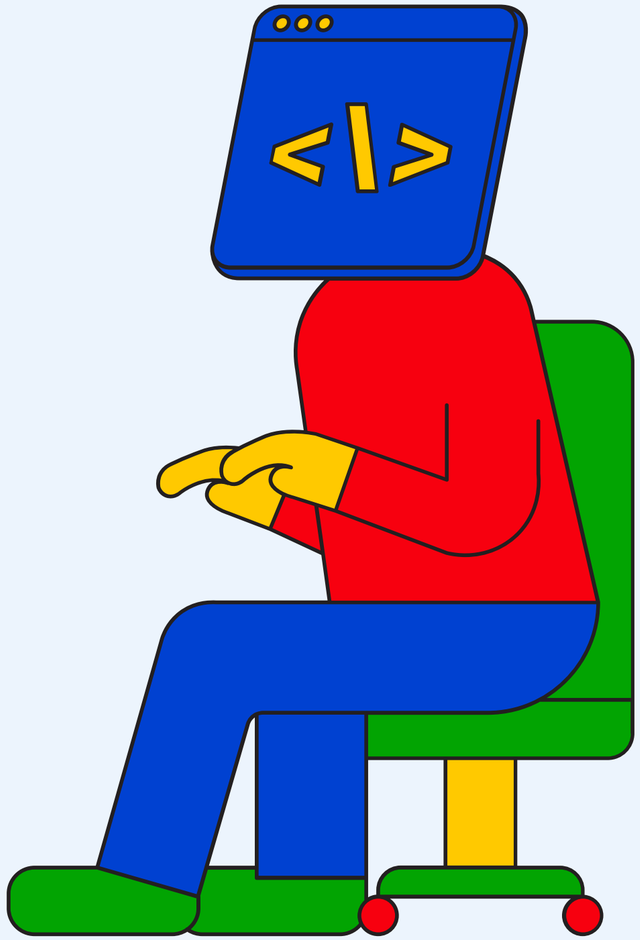
- Use custom events when standard DOM events fall short.
- Ideal for reusable components sharing state or info.

Take Note

- Custom events need `bubbles` set to `true` to propagate.
- Use `cancelable` property to specify if event is cancelable.

MASYNCTECH
SCHOOL

CODE DEMO

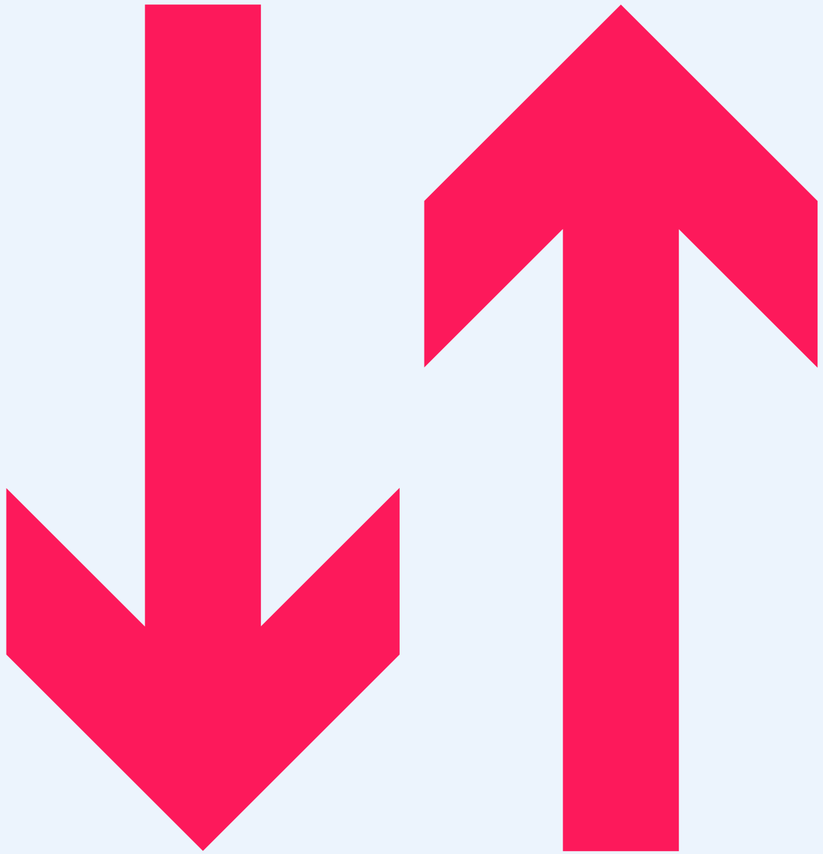


JS

ADVANCED

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

- Prefer Promises or async/await for clarity
- Properly handle errors with `catch` or `.catch()`
- Prevent "callback hell" with modularization
- Utilize async libraries/frameworks when available

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.

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.

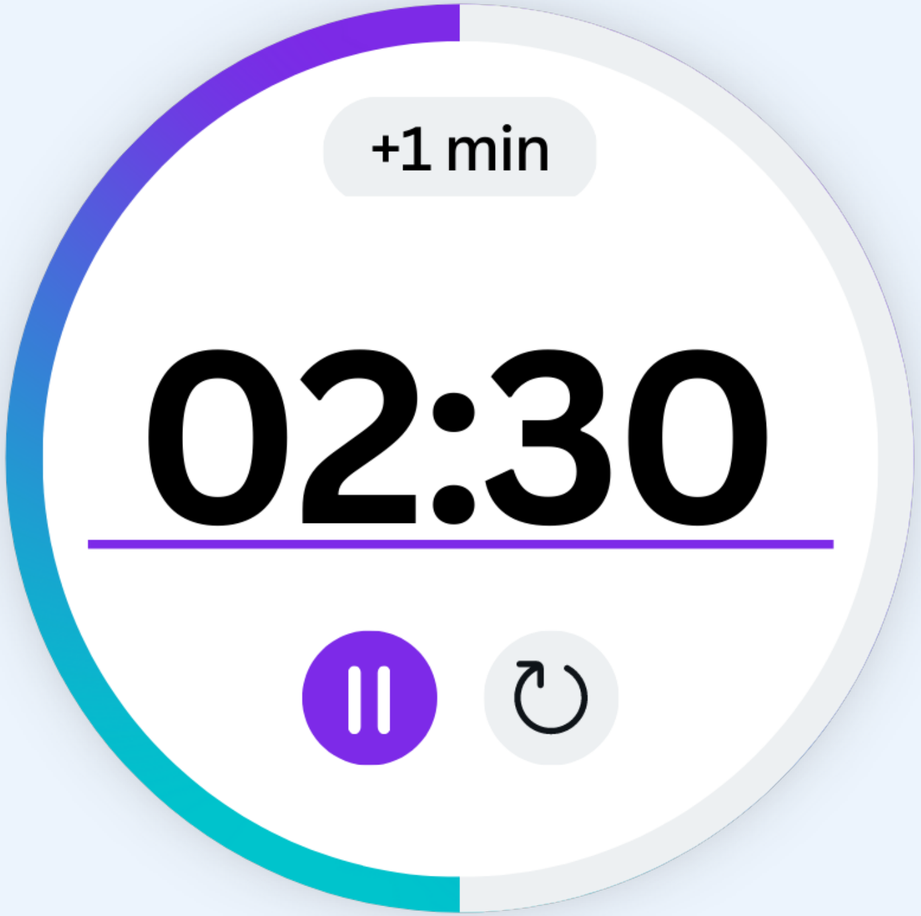
MASYNCTECH
SCHOOL

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

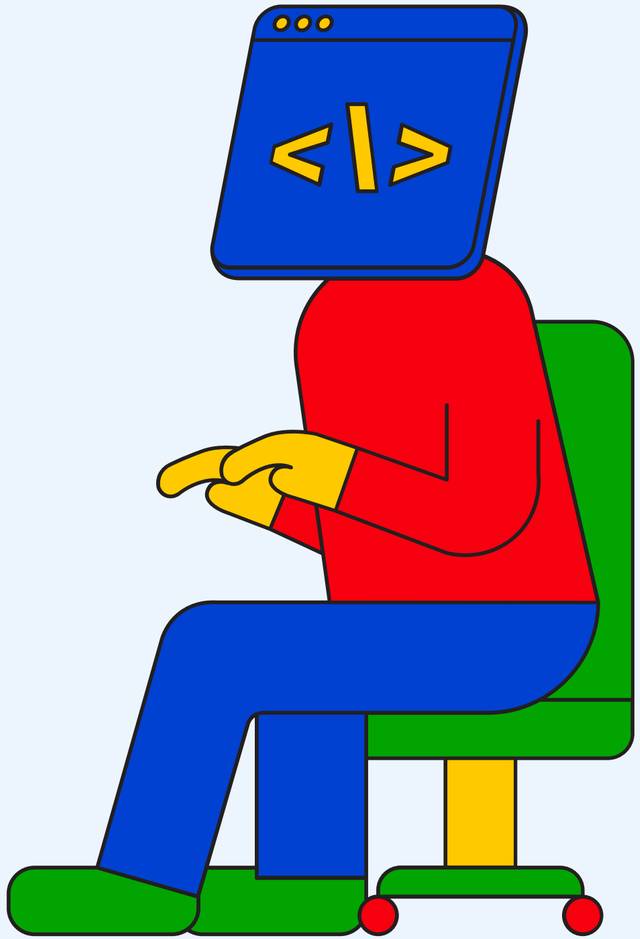
- **setInterva**l: Ideal for recurring tasks at intervals
- Update live clock, poll API, auto-save content regularly

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
NODE-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

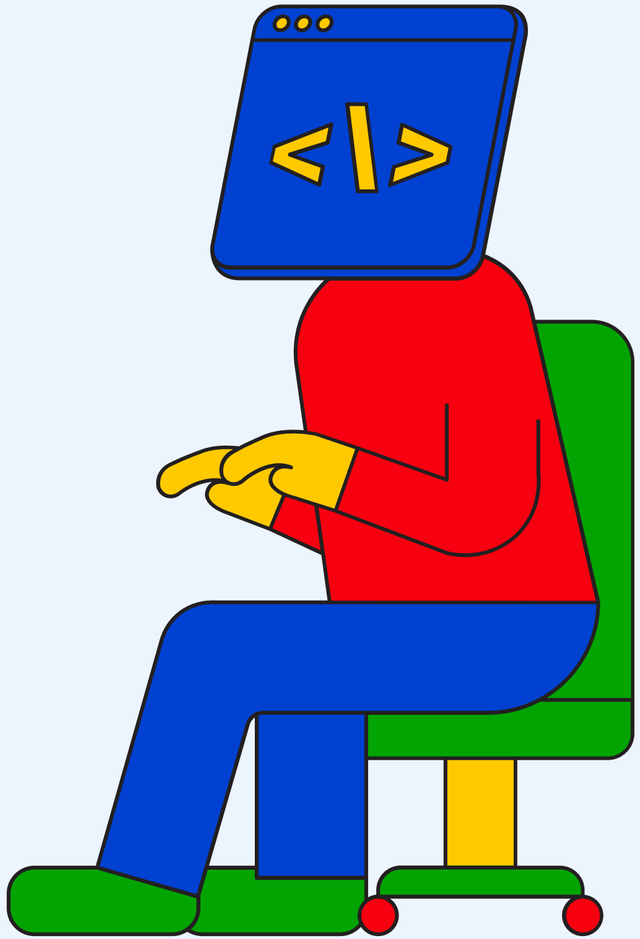
- **Non-blocking:** Ideal for independent, parallel tasks
- Used in responsive apps, web servers, etc.

Summary

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

MASYNCTECH
SCHOOL

CODE DEMO



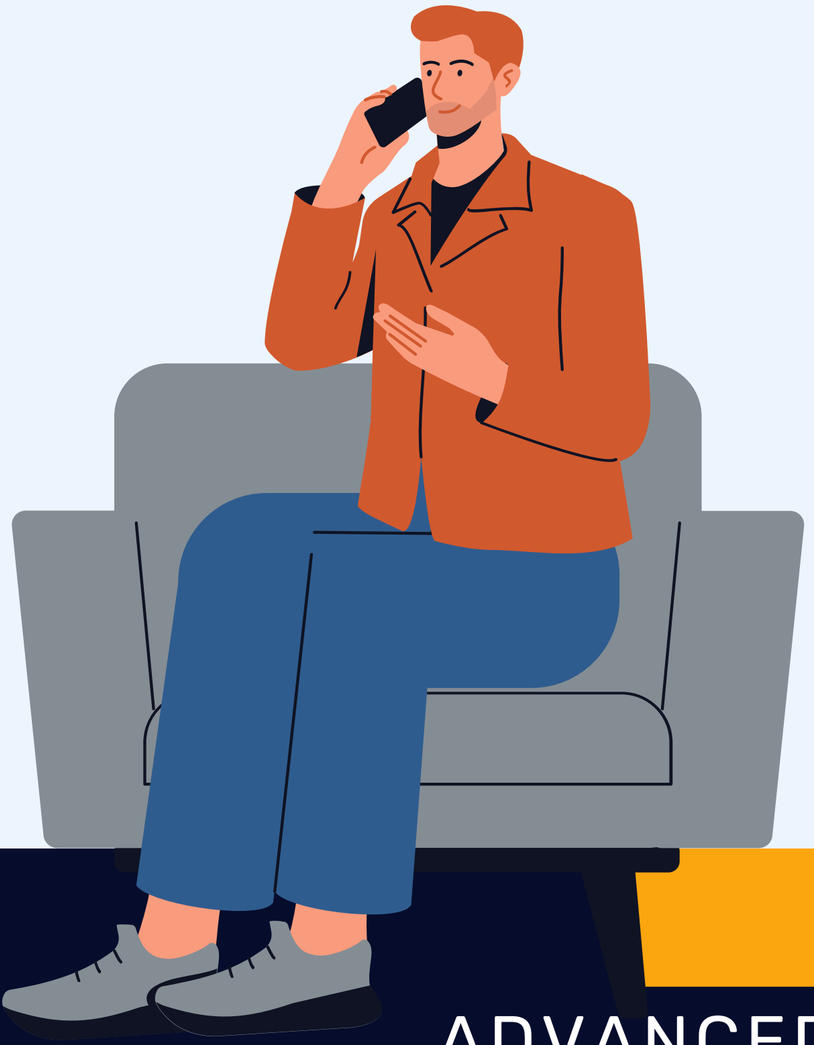
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

CALLBACKS & EVENT LOOP



JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

UNDERSTANDING CALLBACK FUNCTION

JS

ADVANCED

UNDERSTANDING CALLBACK FUNCTIONS

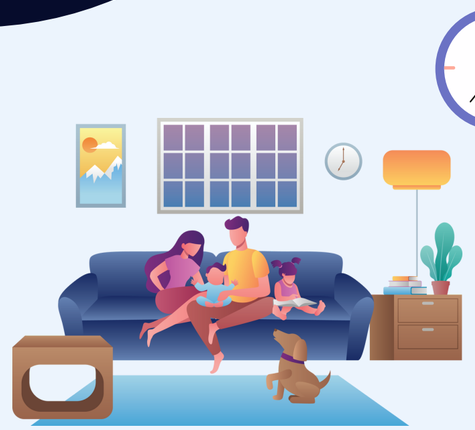
MASYNCTECH
SCHOOL

+233 097533

20 Minutes

UNDERSTANDING CALLBACK FUNCTIONS

MASYNCTECH
SCHOOL



+233 097533

Callback

20 Minutes

MASYNCTECH SCHOOL

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

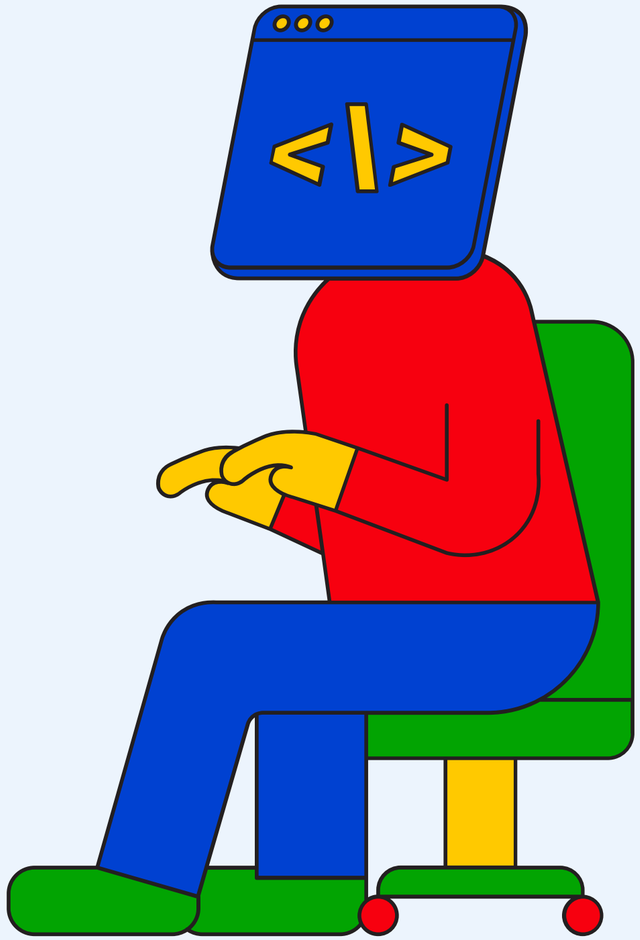
```
function higherOrderFunction(callback) {
  // Code before calling callback

  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

CODE DEMO



JS

ADVANCED

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...
  |       |       |
  |       |       });
  |       |
  |       });
  |
  });
```

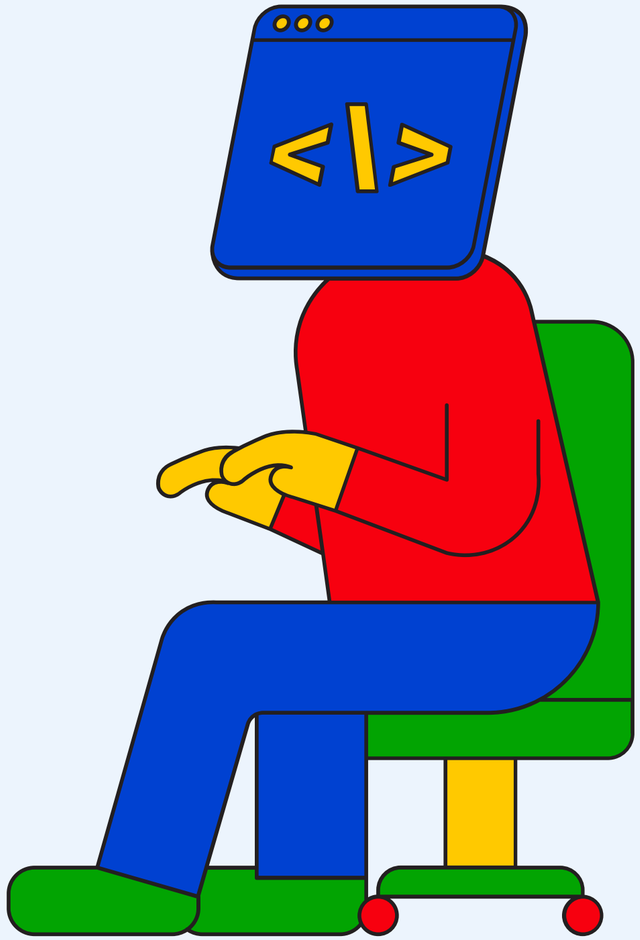
MASYNCTECH SCHOOL

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



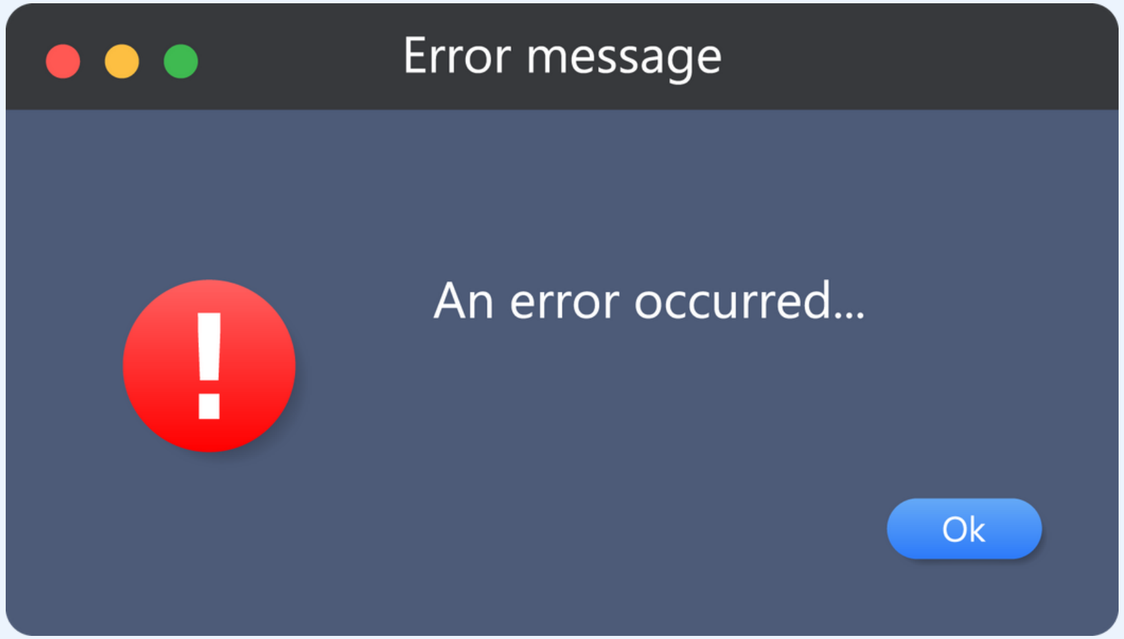
JS

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?

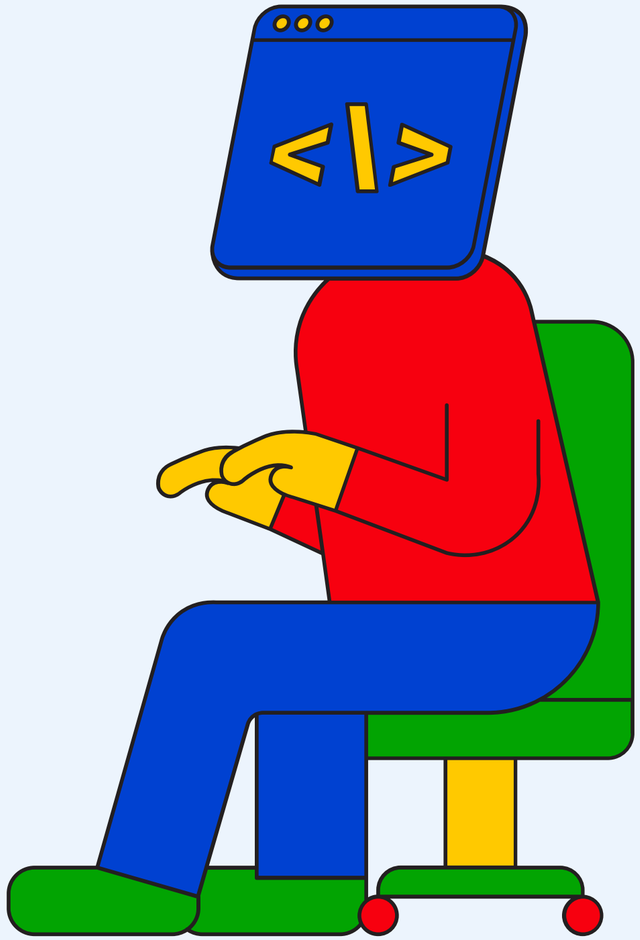
- Error-first pattern valuable for async ops with likely errors.
- Applied in file reading, network requests, database queries.
- Ensures error handling, robust, reliable code.

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



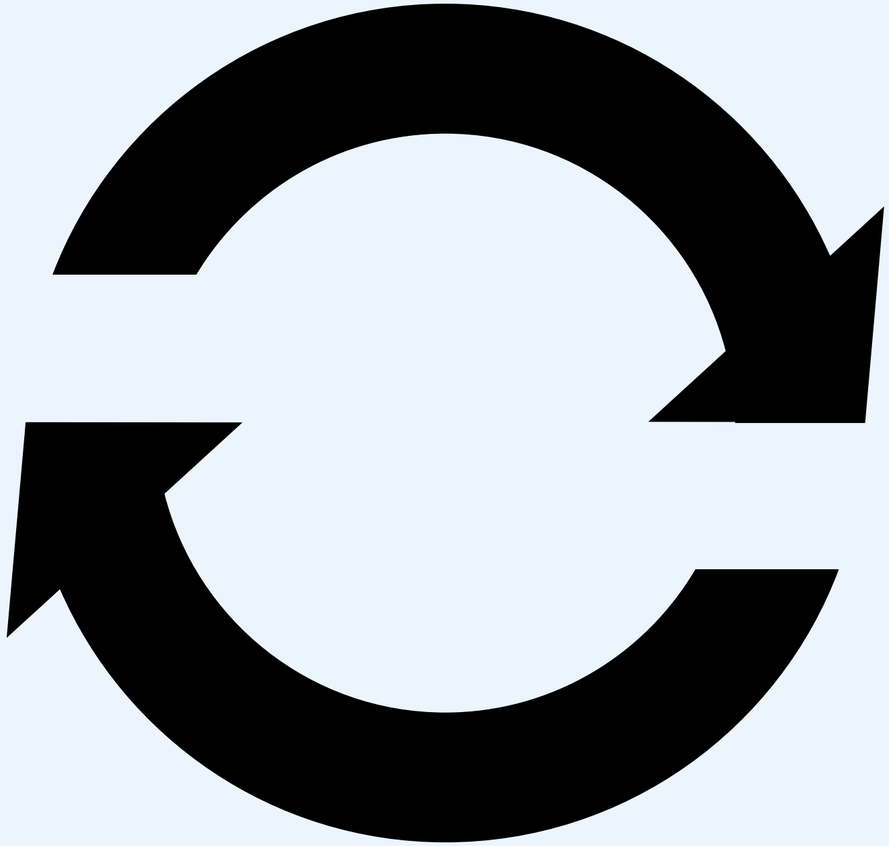
JS

ADVANCED

MASYNCTECH
SCHOOL

ASYNCHRONOUS PROGRAMMING

EVENT LOOP

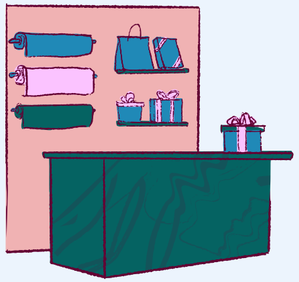


JS

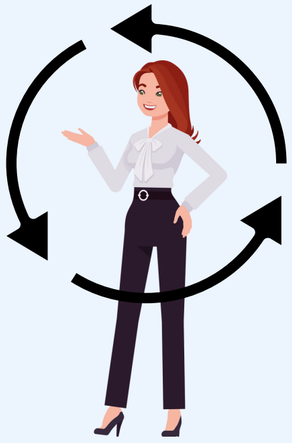
ADVANCED

THE JAVASCRIPT EVENT LOOP

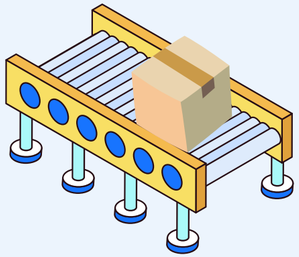
checkout counter(Call Stack)



wrapping station (Web API)



A manager (Event Loop)



conveyor belt (Callback Queue)

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

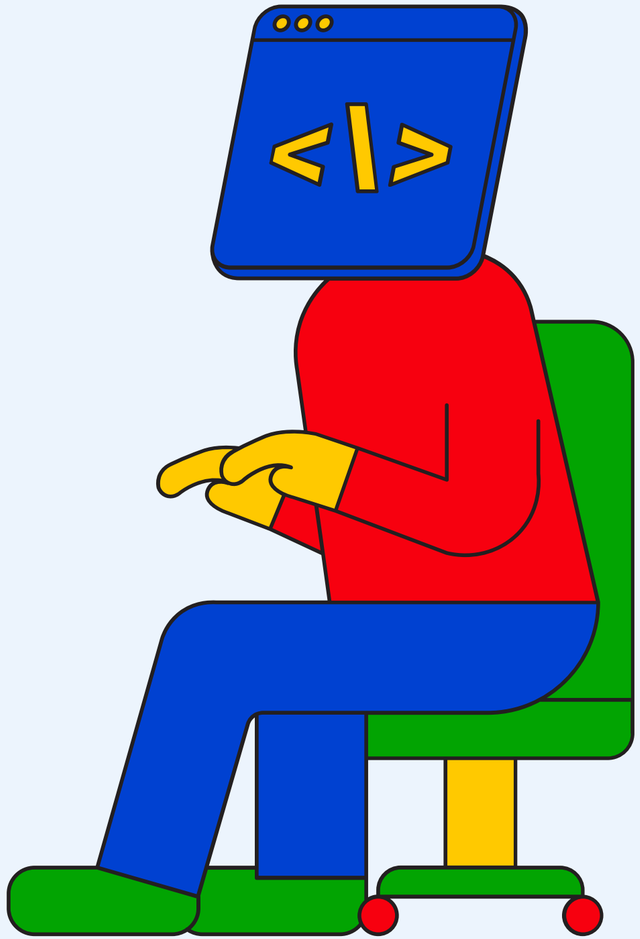
- Avoid long-running tasks to prevent Call Stack blocking.
- For heavy computations, consider Web Workers.
- Handle errors in async code to prevent issues.
- Minimize callback hell; use Promises or async/await for cleaner code.

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