The Ultimate Web Development Handbook

CodeWithHarry

Introduction to HTML

What is HTML?

HTML stands for **HyperText Markup Language**.
It is the standard language used to create and structure webpages.

Web browsers (like Chrome, Firefox, Safari) read HTML code and display it as websites.

Why Learn HTML?

- It's the **foundation** of all websites.
- You need HTML to build pages before adding CSS (styling) or JavaScript (functionality).
- Knowing HTML helps you understand how websites work behind the scenes.

Basic Terminology

- **Element**: A piece of content in a webpage (like a paragraph, heading, or image).
- **Tag**: Special keywords inside angle brackets like `<p>` or `<h1>` that define elements.
- **Attribute**: Extra information added to tags, like `href` in links or `src` in images.

Basic Example

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My First Page</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <p>This is my first HTML page.</p>
  </body>
</html>
```

Explanation:

- `<!DOCTYPE html>` tells the browser this is an HTML5 document.
- `<html>` is the root of the HTML page.
- `<head>` contains information about the page (not shown on screen).
- `<title>` sets the title seen on the browser tab.
- `<body>` contains everything visible on the webpage.

Key Points

- HTML is made up of **tags**.
- Tags usually come in **pairs**: an opening tag `<p>` and a closing tag `</p>`.
- The content goes **between** the tags.
- Indentation helps make code easier to read, but it's not required.

Basic HTML Structure

What is the Basic Structure of an HTML Page?

Every HTML document follows a basic structure. This structure tells the browser how to read and display the content.

Template of a Basic HTML Page

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <!-- Your content goes here -->
  </body>
</html>
```

Explanation of Each Part

1. `<!DOCTYPE html>`

- Declares that this is an HTML5 document.
- Must be the first line in the file.

2. `<html>...</html>`

- The root element of the page.
- Wraps all the content of your HTML document.

3. `<head>...</head>`

- Contains meta-information **about** the page.
- This can include:
  - The page `<title>`
  - Links to CSS files
  - Meta tags (like keywords, description, etc.)

4. `<title>...</title>`

- Sets the name shown on the browser **tab**.

5. `<body>...</body>`

- Contains everything visible on the page.
- You'll place text, images, links, forms, etc. here.

Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Web Page</title>
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is a simple HTML page with basic structure.</p>
  </body>
</html>
```

Tips

- Always start with `<!DOCTYPE html>`.
- Make sure `<html>`, `<head>`, and `<body>` are properly opened and closed.
- Use indentation to keep your code clean and readable.

Headings and Paragraphs

Headings in HTML

Headings help you organize content into sections.
HTML provides **6 levels** of headings:

- `<h1>` – Main heading (biggest)
- `<h2>` – Subheading
- `<h3>` – Smaller subheading
- `<h4>`, `<h5>`, `<h6>` – Even smaller headings

Example of Headings

```
<h1>This is a Heading 1</h1>
<h2>This is a Heading 2</h2>
<h3>This is a Heading 3</h3>
<h4>This is a Heading 4</h4>
<h5>This is a Heading 5</h5>
<h6>This is a Heading 6</h6>
```


📌 Tip:

- Use **only one** `<h1>` per page (usually for the page title).
- Use headings to **structure your content**, not to make text look big (that's CSS's job).

Paragraphs in HTML

Paragraphs are written using the `<p>` tag.

Example:

```
<p>This is a paragraph. It can contain multiple sentences of text.</p>
```

Notes:

- Browsers automatically add **space** before and after each paragraph.
- You don't need to press Enter manually for new lines. Use a new `<p>` tag instead.

Line Breaks

If you want to break a line **without starting a new paragraph**, use the `<br>` tag.

Example:

```
<p>This is line one.<br>This is line two.</p>
```

> `<br>` is a **self-closing tag**, which means it doesn't need a closing `</br>`.

Complete Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>Headings and Paragraphs</title>
  </head>
  <body>
```

```
    <h1>My Blog</h1>
    <h2>Introduction</h2>
    <p>Welcome to my first HTML blog post!</p>

    <h2>Why I Love Coding</h2>
    <p>Coding lets you build websites, apps, and games.<br>It’s fun and creative!</
p>
  </body>
</html>
```

Formatting Text in HTML

HTML allows you to format your text using different tags. These tags help make your content easier to read and visually appealing.

Bold Text

Use the `<b>` or `<strong>` tag to make text bold.

```
<p>This is <b>bold</b> text.</p>
<p>This is <strong>important</strong> text.</p>
```

- `<strong>` also means the text is important (for screen readers and SEO).

Italic Text

Use the `<i>` or `<em>` tag to italicize text.

```
<p>This is <i>italic</i> text.</p>
<p>This is <em>emphasized</em> text.</p>
```

- `<em>` gives extra emphasis and has meaning, especially for accessibility.

Underlined Text

Use the `<u>` tag to underline text.

```
<p>This is <u>underlined</u> text.</p>
```

Strikethrough Text

Use the `<s>` or `<del>` tag to show deleted or crossed-out text.

```
<p>This is <s>wrong</s> text.</p>
<p>Old price: <del>$100</del> New price: $80</p>
```

Superscript and Subscript

Use `<sup>` for superscript (above line), `<sub>` for subscript (below line).

```
<p>Water is H<sub>2</sub>O.</p>
<p>E = mc<sup>2</sup></p>
```

Combining Formats

You can combine formatting tags.

```
<p>This is <b><i>bold and italic</i></b> text.</p>
```

Summary of Formatting Tags

Tag	Purpose
`<b>`	Bold (no meaning)
`<strong>`	Bold (important)
`<i>`	Italic (no meaning)
`<em>`	Italic (emphasis)
`<u>`	Underline
`<s>`	Strikethrough
`<del>`	Deleted text
`<sub>`	Subscript
`<sup>`	Superscript

Comments and Whitespace in HTML

HTML Comments

Comments are notes in your HTML code that are ignored by the browser.
They are useful for explaining code or leaving reminders.

Syntax:

```
<!-- This is a comment -->
<p>This is visible content.</p>
<!-- <p>This line will not show on the webpage.</p> -->
```

Comments do not appear on the webpage. They're only visible in the source code.

Whitespace in HTML

Whitespace includes spaces, tabs, and newlines (Enter key). HTML treats multiple spaces as a **single space**.

Example:

```
<p>This        is        spaced.</p>
```

This will be displayed as:

```
This is spaced.
```

If you want to preserve spaces and line breaks, use the `<pre>` tag.

Example:

```
<pre>
This    is     preformatted
   text.
</pre>
```

Tip

- Use comments to explain your HTML when needed.
- Use proper indentation (whitespace) to make your code readable, even though HTML doesn't require it.

Links and Anchor Tags

Creating Links in HTML

HTML uses the `<a>` tag to create links.
The `href` attribute tells the browser **where** the link should go.

Basic Syntax:

```
<a href="https://www.example.com">Visit Example</a>
```

This creates a clickable link that takes you to the specified URL.

Opening Links in a New Tab

Use the `target="_blank"` attribute to open the link in a new tab.

```
<a href="https://www.google.com" target="_blank">Open Google</a>
```

Linking to Other Pages (Internal Links)

You can also link to other pages of your own website.

```
<a href="about.html">About Us</a>
```

Link to a Section on the Same Page

Use the `id` attribute and a hash (`#`) to jump to a section.

```
<a href="#contact">Go to Contact</a>

...

<h2 id="contact">Contact Section</h2>
```

Email and Phone Links

- **Email Link:**

```
<a href="mailto:hello@example.com">Send Email</a>
```

- **Phone Link:**

```
<a href="tel:+1234567890">Call Us</a>
```

Styling Links (Default Behavior)

- **Normal**: Blue and underlined
- **Visited**: Purple
- **Hover**: Changes color when mouse is over it
- **Active**: Red while clicking

These styles can be changed with CSS later.

Images in HTML

Adding Images

Use the `<img>` tag to display images in HTML.
It is a **self-closing tag**, meaning it doesn't need a closing `</img>`.

Basic Syntax:

```
<img src="image.jpg" alt="Description of image">
```

- `src` (source): The path or URL to the image file.
- `alt` (alternative text): Text shown if the image doesn't load, also used by screen readers.

Example with Local Image

```
<img src="my-photo.jpg" alt="My Photo">
```

(Assumes `my-photo.jpg` is in the same folder as your HTML file)

Example with Online Image

```
<img src="https://example.com/image.png" alt="Online Image">
```

Image Size: Width and Height

You can set the size of the image using `width` and `height` attributes.

```
<img src="pic.jpg" alt="Sample" width="200" height="150">
```

You can also use CSS later for better control.

Tip

- Always include the `alt` text for accessibility.
- Use proper image sizes to improve page loading speed.
- Avoid stretching images using incorrect width/height ratios.

Lists in HTML

Types of Lists

HTML supports three main types of lists:

1. **Unordered List** – Bulleted list
2. **Ordered List** – Numbered list
3. **Description List** – List of terms and their descriptions

Unordered List

Use the `<ul>` tag for unordered lists. Each item goes inside an `<li>` tag.

```
<ul>
  <li>Apples</li>
  <li>Bananas</li>
  <li>Oranges</li>
</ul>
```

This will display:

- Apples
- Bananas
- Oranges

Ordered List

Use the `<ol>` tag for ordered lists.

```
<ol>
  <li>Wake up</li>
  <li>Brush teeth</li>
  <li>Go to work</li>
</ol>
```

This will display:

1. Wake up
2. Brush teeth
3. Go to work

Description List

Use the `<dl>` tag for description lists. Terms go inside `<dt>`, and descriptions go inside `<dd>`.

```
<dl>
  <dt>HTML</dt>
  <dd>A markup language for creating web pages.</dd>

  <dt>CSS</dt>
  <dd>Used for styling HTML pages.</dd>
</dl>
```

Nesting Lists

You can put one list inside another.

```
<ul>
  <li>Fruits
    <ul>
      <li>Apple</li>
      <li>Mango</li>
    </ul>
  </li>
  <li>Vegetables</li>
</ul>
```

Tip

- Use unordered lists for things without order (like a shopping list).
- Use ordered lists when the **sequence matters**.
- Use description lists for definitions or Q&A-style content.

Tables in HTML

Creating Tables

Use the `<table>` tag to create tables. Inside a table, use:

- `<tr>` for table rows
- `<td>` for table data (cells)
- `<th>` for table headers

Basic Table Example

```
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Alice</td>
    <td>24</td>
  </tr>
  <tr>
    <td>Bob</td>
    <td>30</td>
  </tr>
</table>
```


This will display:

Name	Age
Alice	24
Bob	30

Adding Borders

By default, tables have no border. Use the `border` attribute to add one.

```
<table border="1">
  ...
</table>
```

Table Headings vs Data

- `<th>` is usually bold and centered.
- `<td>` is regular table data.

Spanning Columns and Rows

Use `colspan` and `rowspan` to merge cells.

Column Span Example:

```
<tr>
  <th colspan="2">Employee Details</th>
</tr>
```

Row Span Example:

```
<tr>
  <td rowspan="2">John</td>
  <td>Manager</td>
</tr>
<tr>
  <td>IT Department</td>
</tr>
```

Tip

- Keep your tables organized and easy to read.
- Use headers (`<th>`) for important rows or columns.
- Avoid very complex tables for layout—use CSS for page design.

HTML Forms: Inputs, Labels, and Buttons

What is a Form?

Forms allow users to **input data** and send it to a server.
Use the `<form>` tag to create a form.

```
<form>
  <!-- form elements go here -->
</form>
```

Text Input Field

Use `<input type="text">` to get a single line of text from the user.

```
<form>
  <label for="name">Name:</label>
  <input type="text" id="name" name="name">
</form>
```

- `<label>` is used to describe the input.
- The `for` attribute should match the input's `id`.

Password Field

```
<label for="password">Password:</label>
<input type="password" id="password" name="password">
```

This hides the characters as you type.

Submit Button

Use `type="submit"` to create a button that submits the form.

```
<input type="submit" value="Submit">
```

Placeholder Text

You can show a hint inside the input using the `placeholder` attribute.

```
<input type="text" placeholder="Enter your name">
```

Complete Example

```
<form>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email"
placeholder="you@example.com"><br><br>

  <label for="pass">Password:</label>
  <input type="password" id="pass" name="pass"><br><br>
```

```
  <input type="submit" value="Login">
</form>
```

Tip

- Always label your inputs for better accessibility.
- Use `name` attributes if the form is going to submit data.
- You'll learn more input types in the next lesson.

Form Elements: Radio, Checkbox, Select, Textarea

Radio Buttons

Use radio buttons when users need to **select only one** option from a group.

```
<p>Choose your gender:</p>
<input type="radio" id="male" name="gender" value="male">
<label for="male">Male</label><br>

<input type="radio" id="female" name="gender" value="female">
<label for="female">Female</label>
```

- All radio buttons in a group should have the **same** `name`.
- Only one option can be selected.

Checkboxes

Use checkboxes when users can **select multiple** options.

```
<p>Select your hobbies:</p>
<input type="checkbox" id="reading" name="hobby" value="reading">
<label for="reading">Reading</label><br>

<input type="checkbox" id="sports" name="hobby" value="sports">
<label for="sports">Sports</label><br>
```

```
<input type="checkbox" id="music" name="hobby" value="music">
<label for="music">Music</label>
```

Dropdown List (Select Menu)

Use the `<select>` tag with `<option>` to let users pick one option from a dropdown.

```
<label for="city">Choose a city:</label>
<select id="city" name="city">
  <option value="delhi">Delhi</option>
  <option value="mumbai">Mumbai</option>
  <option value="bangalore">Bangalore</option>
</select>
```

Textarea (Multiline Input)

Use the `<textarea>` tag to let users type multiple lines of text.

```
<label for="message">Your Message:</label><br>
<textarea id="message" name="message" rows="4" cols="30"></textarea>
```

Tip

- Use radio buttons for single choice questions.
- Use checkboxes for multiple selections.
- Use `select` for compact dropdowns.
- Use `textarea` for longer user input like comments or messages.

HTML5 Semantic Tags

Inline vs Block Elements in HTML

In HTML, elements are broadly categorized as **inline** or **block** based on how they behave in the document flow.

Block Elements

- Start on a **new line**.
- Take up the **full width** available.
- Can contain **other block** and **inline** elements.

Common Block Elements:

- `<div>`
- `<p>`
- `<h1>` to `<h6>`
- `<section>`
- `<article>`
- `<ul>`, `<ol>`, `<li>`

Example:

```
<div>
  <h2>This is a heading</h2>
  <p>This is a paragraph inside a div.</p>
</div>
```

Inline Elements

- Do **not start on a new line**.

- Only take up as much **width as necessary**.
- Usually used to style **small portions** of content within block elements.

Common Inline Elements:

- `<span>`
- `<a>`
- `<strong>`, `<em>`
- `<img>`
- `<code>`

Example:

```
<p>This is a <strong>bold</strong> word and <a href="#">this is a link</a>.</p>
```

Summary

Feature	Block Elements	Inline Elements
New Line	Yes	No
Width	Full width	Width of content only
Nesting	Can contain any elements	Only other inline elements

What Are Semantic Tags?

Semantic tags clearly describe the **meaning** of the content they contain.
They help both developers and browsers understand the structure of the page.

Example:
- `<div>` says nothing about its content.
- `<header>` clearly means it's a page or section header.

Common Semantic Tags

Tag	Purpose
`<header>`	Top section of a page or section
`<nav>`	Navigation links
`<main>`	Main content of the page
`<section>`	A standalone section
`<article>`	Self-contained content like a blog
`<aside>`	Sidebar or extra info
`<footer>`	Bottom section of a page or section

Example Usage

```
<!DOCTYPE html>
<html>
  <head>
    <title>Semantic Page</title>
  </head>
  <body>

    <header>
      <h1>My Website</h1>
    </header>

    <nav>
      <a href="#">Home</a> |
      <a href="#">About</a> |
      <a href="#">Contact</a>
    </nav>

    <main>
      <section>
```

```
        <h2>Welcome</h2>
        <p>This is the welcome section.</p>
      </section>

      <article>
        <h2>Blog Post</h2>
        <p>This is a blog post inside an article tag.</p>
      </article>
    </main>

    <aside>
      <p>This is a sidebar with related links.</p>
    </aside>

    <footer>
      <p>Copyright © 2025</p>
    </footer>

  </body>
</html>
```

Tip

- Semantic tags improve accessibility and SEO.
- Use them instead of generic `<div>` and `<span>` wherever possible.

HTML Entities and Special Characters

What Are HTML Entities?

Some characters have **special meaning** in HTML (like `<`, `>`, `&`).
To display these characters on a webpage, you need to use **HTML entities**.

An entity starts with `&` and ends with `;`.

Common HTML Entities

Character	Entity Code	Description
<	`<`	Less than
>	`>`	Greater than
&	`&`	Ampersand
"	`&quot;`	Double quote
'	`&apos;`	Single quote
©	`&copy;`	Copyright symbol
®	`&reg;`	Registered symbol
₹	`&#8377;`	Indian Rupee sign
→	`&rarr;`	Right arrow

Example

```
<p>5 &lt; 10</p>
<p>Use &amp; to join strings</p>
<p>Price: &#8377;499</p>
```

This will display as:

5 < 10 Use & to join strings Price: ₹499

Non-Breaking Space

Use ` ` to add extra space that the browser won't collapse.

```
<p>Hello   World</p>
```

Tip

- Use entities when you want to **show special characters** as text.
- HTML automatically converts most symbols when needed, but using entities ensures correct display.

Audio and Video Embedding in HTML

Embedding Audio

Use the `<audio>` tag to add sound or music to your webpage.

Basic Example:

```
<audio controls>
  <source src="audio.mp3" type="audio/mpeg">
  Your browser does not support the audio element.
</audio>
```

- `controls` adds play, pause, and volume controls.
- The `<source>` tag specifies the audio file and type.

Audio Formats

Format	MIME Type
MP3	`audio/mpeg`
OGG	`audio/ogg`
WAV	`audio/wav`

To support all browsers, you can include multiple sources:

```
<audio controls>
  <source src="audio.mp3" type="audio/mpeg">
```

```
  <source src="audio.ogg" type="audio/ogg">
</audio>
```

Embedding Video

Use the `<video>` tag to add videos to your page.

Basic Example:

```
<video width="320" height="240" controls>
  <source src="video.mp4" type="video/mp4">
  Your browser does not support the video tag.
</video>
```

- `width` and `height` control the video size.
- `controls` adds playback controls.

Video Formats

Format	MIME Type
MP4	`video/mp4`
WebM	`video/webm`
OGG	`video/ogg`

You can include multiple sources to ensure browser compatibility:

```
<video controls>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
</video>
```

Tip

- Always provide `controls` so users can interact with media.
- Use multiple formats for broader browser support.
- Include fallback text for unsupported browsers.

IFrames and Embedding Content

What is an IFrame?

An `<iframe>` (inline frame) is used to embed another webpage or external content inside your HTML page.

Basic Syntax

```
<iframe src="https://example.com" width="600" height="400"></iframe>
```

- `src` specifies the URL of the page to embed.
- `width` and `height` set the size of the frame.

Example: Embed a Website

```
<iframe src="https://www.wikipedia.org" width="800" height="500"></iframe>
```

Example: Embed a YouTube Video

YouTube provides embed code for each video.

```
<iframe width="560" height="315"
  src="https://www.youtube.com/embed/dQw4w9WgXcQ"
```

```
  frameborder="0"
  allowfullscreen>
</iframe>
```

Attributes

Attribute	Description
`src`	URL of the page or content
`width , height`	Size of the iframe
`frameborder`	Border of the frame (0 = none)
`allowfullscreen`	Allows video to go full screen
`loading="lazy"`	Delays loading until iframe is visible

Security Note

Some websites may **block iframe embedding** for security reasons using headers like `X-Frame-Options` .

Tip

- Use `<iframe>` to embed maps, videos, forms, and external tools.
- Always set appropriate width and height for better layout control.

Using Meta Tags and SEO Basics

What Are Meta Tags?

Meta tags provide **information about the webpage** to browsers and search engines.
They go inside the `<head>` section and do not appear on the page itself.

Common Meta Tags

1. Charset

```
<meta charset="UTF-8">
```

- Defines the character encoding.
- UTF-8 covers most characters in all languages.

---

2. Viewport (Mobile Responsiveness)

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- Makes your website mobile-friendly.
- Tells the browser to match the screen's width.

---

3. Page Description

```
<meta name="description" content="Learn HTML from scratch with simple examples.">
```

- Summarizes the page content.
- Often shown in search engine results.

---

4. Keywords (Less important today)

```
<meta name="keywords" content="HTML, web development, coding">
```

- List of keywords related to your page.
- Not heavily used by modern search engines.

---

5. Author

```
<meta name="author" content="Your Name">
```

- Specifies the name of the content creator.

---

6. Refresh / Redirect (Optional)

```
<meta http-equiv="refresh" content="5; url=https://example.com">
```

- Redirects the page after 5 seconds.

SEO Basics

- Use meaningful page titles with the `<title>` tag.

- Include a clear meta description.
- Structure content using headings (`<h1>` , `<h2>` , etc.).
- Use semantic tags to describe content.
- Make sure the page loads fast and works well on mobile.

Example Head Section

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="Simple HTML tutorial for beginners.">
  <meta name="author" content="John Doe">
  <title>Learn HTML</title>
</head>
```

Tip

- Good meta tags improve search visibility and user experience.
- Always include the viewport meta tag for mobile responsiveness.

Internal vs External Links

Internal Links

Internal links connect one page of your website to another.
They help users **navigate** within your site.

Example:

```
<a href="about.html">About Us</a>
```

This opens the `about.html` page located in the same folder.

Linking to a Section on the Same Page

Use `#id` to jump to a specific section.

```
<a href="#contact">Go to Contact Section</a>

...

<h2 id="contact">Contact Us</h2>
```


External Links

External links take the user to a **different website**.

Example:

```
<a href="https://www.google.com">Visit Google</a>
```

Open External Links in a New Tab

Use the `target="_blank"` attribute.

```
<a href="https://www.example.com" target="_blank">Open Example</a>
```

Adding `rel="noopener noreferrer"` for Security

When using `target="_blank"`, it's recommended to add `rel="noopener noreferrer"` to prevent security risks.

```
<a href="https://external.com" target="_blank" rel="noopener noreferrer">
  Visit External Site
</a>
```

Summary

Link Type	Example
Internal Link	`href="about.html"`
Section Jump	`href="#section-id"`
External Link	`href="https://example.com"`
New Tab + Safe	`target="_blank" rel="noopener"`

Tip

- Use internal links to connect your content and improve navigation.
- Use external links to reference useful outside resources.

Best Practices for Writing Clean HTML

1. Use Proper Indentation

Indent nested elements for better readability.

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

2. Always Close Your Tags

Even if some tags are optional, it's best to close them properly.

```
<p>This is correct.</p>
```

3. Use Meaningful Tag Structure

Use semantic tags like `<header>`, `<main>`, `<footer>` instead of relying only on `<div>`.

4. Include `alt` Text for Images

This improves accessibility and helps screen readers understand image content.

```
<img src="photo.jpg" alt="A smiling person">
```

---

5. Use Lowercase for Tags and Attributes

HTML is not case-sensitive, but using lowercase is the standard.

```
<!-- Good -->
<input type="text">

<!-- Avoid -->
<INPUT TYPE="TEXT">
```

---

6. Organize Your Code

Keep your HTML structured by grouping related elements together. Use comments to separate sections.

```
<!-- Navigation -->
<nav>...</nav>

<!-- Main Content -->
<main>...</main>
```


---

7. Don’t Use Inline Styles (if possible)

Avoid putting CSS styles directly into HTML tags. Use external CSS files instead.

```
<!-- Avoid -->
<p style="color: red;">Red text</p>

<!-- Prefer -->
<p class="red-text">Red text</p>
```

---

8. Validate Your HTML

Use tools like W3C HTML Validator to check for errors in your code.

---

9. Keep File Names Simple and Clear

Use lowercase letters, dashes instead of spaces, and meaningful names.

```
✔ about-us.html
✘ About Us!.html
```

---

10. Comment Your Code (When Needed)

Use comments to explain complex sections or to label page areas.

```
<!-- Contact Form -->
<form>...</form>
```

Tip

Clean HTML is easier to read, debug, maintain, and scale as your website grows.

Introduction to CSS

CSS (Cascading Style Sheets) is used to style and layout web pages — including colors, fonts, spacing, and positioning of elements. While HTML gives structure to a web page, CSS makes it look beautiful and usable.

Why CSS?

Without CSS, all websites would look plain, like unstyled documents. CSS helps you:

- Change colors and fonts
- Add spacing and layout
- Make responsive designs for mobile
- Animate and transition between states
- Separation of concerns: HTML for structure, CSS for style

How CSS Works with HTML

CSS can be applied to HTML in **three main ways**:

1. Inline CSS

CSS written inside an HTML tag using the `style` attribute.

```
<p style="color: blue; font-size: 18px;">This is a blue paragraph.</p>
```

2. Internal CSS

CSS written inside a `<style>` tag within the `<head>` section of the HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      p {
        color: green;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p>This is a green bold paragraph.</p>
  </body>
</html>
```

3. External CSS (Best Practice)

CSS written in a separate file and linked to the HTML file. This is the **most recommended** method for real-world projects.

`style.css`

```
h1 {
  color: darkred;
  text-align: center;
}
```

`index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>Welcome to CSS</h1>
```

```
  </body>
</html>
```

The “Cascading” in CSS

If there are multiple rules targeting the same element, CSS uses the **cascade** to decide which rule to apply. This depends on:

- **Specificity** (How specific the selector is)
- **Order of appearance**
- **Importance (`!important`)**

Example:

```
<p style="color: red;">This will be red because inline CSS wins.</p>
```

Anatomy of a CSS Rule

```
selector {
  property: value;
}
```

Example:

```
p {
  color: black;
  font-size: 16px;
}
```

- `p` → Selector (targets all `<p>` elements)
- `color` , `font-size` → Properties
- `black` , `16px` → Values

What You’ll Learn in CSS

As we move forward, you’ll learn how to:

- Style text, backgrounds, and borders
- Control layout with Flexbox and Grid
- Make your website responsive and mobile-friendly
- Animate elements and transitions
- Write modern, maintainable CSS

CSS Syntax and Selectors

To apply styles to HTML elements, you need to understand the basic **syntax of CSS** and how to **select elements** on the page.

CSS Syntax

Every CSS rule consists of a **selector** and a **declaration block**.

```
selector {
  property: value;
}
```

Example:

```
h1 {
  color: navy;
  font-size: 32px;
}
```

- `h1` is the **selector**
- `color` and `font-size` are **properties**
- `navy` and `32px` are the **values**
- The curly braces `{}` contain the **declaration block**
- Each declaration ends with a semicolon `;`

Types of Selectors

1. Element Selector

Selects all elements of a specific type.

```
p {
  color: gray;
}
```

This targets all `<p>` elements.

---

2. Class Selector

Selects elements with a specific class.

HTML:

```
<p class="highlight">This is important.</p>
```

CSS:

```
.highlight {
  background-color: yellow;
}
```

Use a period `.` before the class name.

---

3. ID Selector

Selects a single element with a unique ID.

HTML:

```html
<h1 id="main-heading">Welcome</h1>
```

CSS:

```css
#main-heading {
  font-family: Arial, sans-serif;
}
```

Use a hash `#` before the ID name.

---

4. Universal Selector

Applies styles to **all elements** on the page.

```css
* {
  margin: 0;
  padding: 0;
}
```

This is commonly used for resetting default styles.

---

5. Grouping Selectors

Apply the same styles to multiple selectors at once.

```css
h1, h2, h3 {
  color: darkblue;
}
```

This avoids repetition.

---

6. Descendant Selector

Targets elements nested inside other elements.

HTML:

```
<div>
  <p>This is a paragraph inside a div.</p>
</div>
```

CSS:

```
div p {
  font-style: italic;
}
```

Only `<p>` tags inside `<div>` will be affected.

---

7. Combining Class and Element Selectors

You can be more specific by combining them.

```
p.note {
  color: teal;
}
```

This targets only `<p>` elements with the class `note`.

Summary

- CSS selectors help you choose **which** HTML elements to style.
- Use `.` for classes, `#` for IDs, and tag names for element selectors.
- Combine and group selectors for powerful control.

Colors in CSS

Colors play a major role in the visual appearance of a website. In CSS, you can apply colors to text, backgrounds, borders, and other elements using different formats.

Ways to Define Colors

CSS supports several formats for defining colors:

---

1. Named Colors

CSS has a set of predefined color names like `red`, `blue`, `green`, `black`, etc.

```
h1 {
  color: red;
}
```

---

2. HEX Codes

A hexadecimal value represents a color using a six-digit code.

```
body {
  background-color: #f0f0f0;
}
```

- `#000000` → black
- `#ffffff` → white

- `#ff0000` → red

You can also use shorthand if all pairs are the same:

```
#fff  /* same as #ffffff */
```

---

3. RGB (Red, Green, Blue)

You can define a color using the RGB color model.

```
p {
  color: rgb(255, 0, 0);
}
```

- Values range from `0` to `255`
- `rgb(0, 0, 0)` → black
- `rgb(255, 255, 255)` → white

---

4. RGBA (RGB + Alpha)

Adds **opacity** to RGB using the alpha channel (0 = fully transparent, 1 = fully opaque).

```
div {
  background-color: rgba(0, 0, 0, 0.5);
}
```

This creates a semi-transparent black background.

---

5. HSL (Hue, Saturation, Lightness)

Another way to define colors using:

- **Hue** (color angle on the color wheel)
- **Saturation** (intensity of the color)
- **Lightness** (brightness)

```
h2 {
  color: hsl(240, 100%, 50%);
}
```

---

6. HSLA (HSL + Alpha)

Same as HSL, but with transparency.

```
section {
  background-color: hsla(120, 60%, 70%, 0.3);
}
```

Applying Colors in CSS

You can use color properties in many different places:

```
h1 {
  color: navy;                 /* Text color */
  background-color: #e0e0e0;   /* Background color */
  border: 2px solid #333;      /* Border color */
}
```

Transparent and CurrentColor

- `transparent` → Makes an element's color fully transparent.
- `currentColor` → Inherits the current value of the `color` property.

```
button {
  color: blue;
  border: 2px solid currentColor;
}
```

Summary

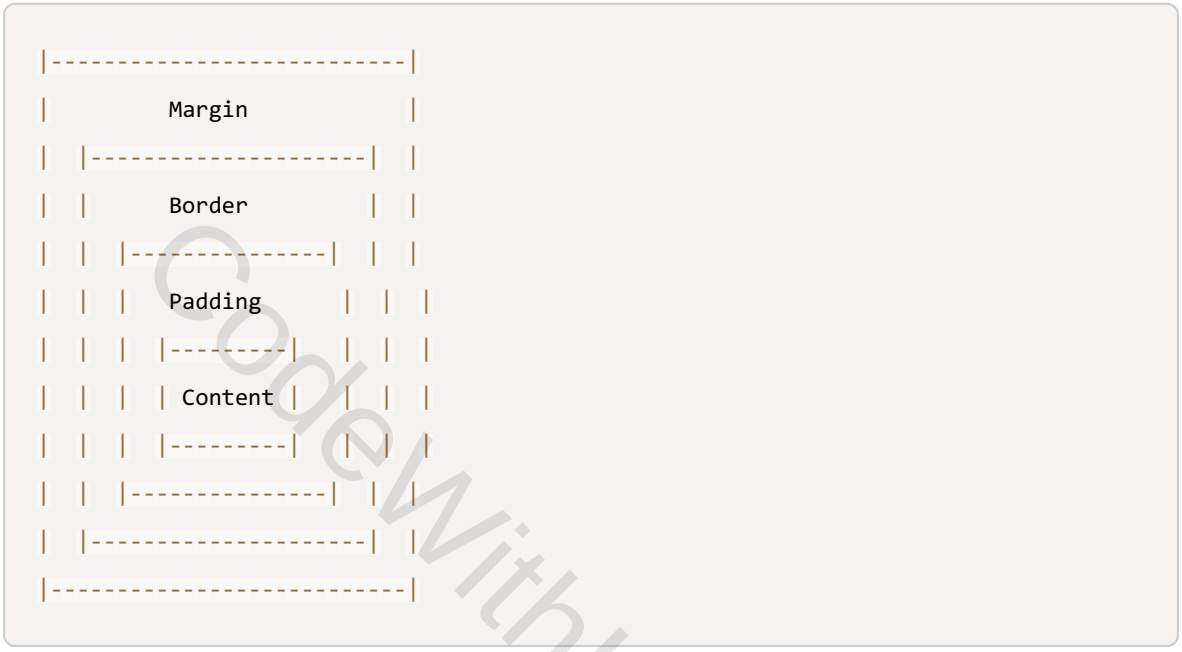
- Use color to enhance readability, structure, and aesthetics.
- Choose the format (HEX, RGB, HSL) that suits your workflow.
- Learn to use `rgba` or `hsla` for transparency effects.

The CSS Box Model

Every HTML element on a page is a **rectangular box** in the browser, and the **Box Model** defines how that box behaves. It's the foundation of spacing, layout, and sizing in CSS.

What is the Box Model?

The box model consists of **four layers**, from innermost to outermost:



The Four Parts

1. Content

The actual text, image, or element inside the box.

```
width: 200px;
height: 100px;
```

---

2. Padding

Space **inside** the box, between content and border.

```
padding: 20px;
```

It pushes the content **inward**, increasing the total box size.

---

3. Border

The border around the padding and content.

```
border: 2px solid black;
```

You can control its width, style, and color.

---

4. Margin

Space **outside** the border. Used to create distance between elements.

```
margin: 30px;
```

Margins do not have a background color and are completely transparent.

Example

```
.box {
  width: 300px;
  height: 150px;
  padding: 20px;
  border: 5px solid gray;
  margin: 40px;
}
```

The **actual space** this element occupies:

- Width: `300 + 2*20 (padding) + 2*5 (border)` = **350px**
- Height: `150 + 2*20 (padding) + 2*5 (border)` = **200px**

Margin is **outside** of this box, adding extra space between elements.

Box Sizing: `content-box` vs `border-box`

By default, CSS uses `content-box`, where `width` and `height` apply **only to the content**, not padding or border.

To include padding and border **inside** the specified dimensions, use:

```
* {
  box-sizing: border-box;
}
```

With `border-box`, the total width stays fixed, and padding/border are adjusted **inside** the box.

Visual Example

```
.card {
  width: 400px;
  padding: 20px;
  border: 10px solid black;
  box-sizing: border-box;
}
```

In this case, the **total width remains 400px**, including padding and border.

Summary

- The box model controls **how elements take up space**.
- Understand how content, padding, border, and margin interact.
- Use `box-sizing: border-box` to make layout calculations easier.

Units in CSS

CSS units define the size, spacing, and positioning of elements on a web page. Understanding units is essential for building layouts that are consistent, responsive, and easy to manage.

Two Categories of Units

1. Absolute Units

These do **not change** based on screen size or parent element. Use them for fixed-size elements (use cautiously in responsive designs).

Unit	Description
`px`	Pixels (most common absolute unit)
`pt`	Points (1/72 of an inch)
`cm`	Centimeters
`mm`	Millimeters
`in`	Inches

Example:

```
h1 {
  font-size: 24px;
}
```


---

2. Relative Units

These are **responsive** and scale based on parent elements, root font size, or viewport size.

Unit	Description
`%`	Relative to parent element
`em`	Relative to parent's font size
`rem`	Relative to root font size (usually `<html>`)
`vw`	1% of viewport width
`vh`	1% of viewport height
`vmin`	1% of smaller viewport dimension
`vmax`	1% of larger viewport dimension

Commonly Used Units

`px` (Pixels)

```
p {
  margin: 10px;
}
```

Fixed spacing that does not scale with screen size.

`%` (Percentage)

```
div {
  width: 80%;
}
```

Useful for making widths or heights relative to parent elements.

`em` vs `rem`

`em`: Relative to the font size of the parent.

```
div {
  font-size: 2em; /* 2 times the parent's font size */
}
```

`rem`: Relative to the font size of the root (`html`) element.

```
html {
  font-size: 16px;
}

h1 {
  font-size: 2rem; /* 32px */
}
```

Use `rem` for consistency in modern responsive design.

`vw` and `vh`

```
.container {
  width: 100vw;   /* Full width of the viewport */
  height: 100vh;  /* Full height of the viewport */
}
```

These units are powerful for creating fullscreen layouts.

`calc()` Function

Combine different units using `calc()`:

```
section {
  width: calc(100% - 200px);
}
```

Best Practices

- Use **`rem`** for typography for consistency and scalability.
- Use **`%`**, **`vw`**, **`vh`** for responsive layouts.
- Avoid overusing **`px`** in responsive designs.

Summary

CSS units help control the size and spacing of elements. Choosing the right unit is key to making layouts flexible, scalable, and consistent across devices.

Typography in CSS

Typography is how text appears on a web page — its **font, size, spacing, alignment, weight**, and overall readability. Good typography improves user experience and design quality.

Basic Text Properties

1. `font-family`

Sets the typeface for your text.

```
body {
  font-family: Arial, sans-serif;
}
```

- You can specify a list of fallback fonts.
- Always end with a **generic family** like `sans-serif`, `serif`, or `monospace`.

Common font stacks:

```
font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
font-family: Georgia, 'Times New Roman', serif;
font-family: 'Courier New', Courier, monospace;
```

---

2. `font-size`

Controls the size of the text.

```css
h1 {
  font-size: 36px;
}
```

You can use units like `px`, `em`, `rem`, `%`.

```css
p {
  font-size: 1.2rem;
}
```

---

3. `font-weight`

Defines the boldness of text.

```css
strong {
  font-weight: bold;
}
```

You can use keywords like `normal`, `bold`, or numeric values like `100`, `400`, `700`, `900`.

---

4. `font-style`

Sets text to normal, italic, or oblique.

```css
em {
  font-style: italic;
}
```

---

5. `text-align`

Aligns text horizontally.

```
h2 {
  text-align: center;
}
```

Values: `left`, `right`, `center`, `justify`

---

6. `line-height`

Controls the space between lines of text.

```
p {
  line-height: 1.6;
}
```

This improves readability, especially for paragraphs.

---

7. `letter-spacing`

Controls space between characters.

```
h1 {
  letter-spacing: 2px;
}
```

---

8. `word-spacing`

Controls space between words.

```css
p {
  word-spacing: 5px;
}
```

---

9. `text-transform`

Changes the case of text.

```css
.upper {
  text-transform: uppercase;
}

.lower {
  text-transform: lowercase;
}

.capitalize {
  text-transform: capitalize;
}
```

---

10. `text-decoration`

Controls underlining, overlining, and line-through.

```css
a {
  text-decoration: none;
}
```

Using Google Fonts

To use custom fonts, you can load them from Google Fonts.

HTML

```
<link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="styl
esheet">
```

CSS

```
body {
  font-family: 'Roboto', sans-serif;
}
```

Summary

Typography affects the readability and tone of your website. Key things to remember:

- Use `rem` for font sizing to keep things scalable.
- Set appropriate `line-height` and `font-family` for comfortable reading.
- Align and style text to match your design's personality.

Backgrounds and Borders in CSS

CSS allows you to customize how elements **look and feel** by adding backgrounds and borders. You can apply colors, images, gradients, and control borders precisely around elements.

Background Properties

1. `background-color`

Sets a solid color behind an element.

```
div {
  background-color: lightblue;
}
```

---

2. `background-image`

Adds an image as the background.

```
body {
  background-image: url('background.jpg');
}
```

You can use local images or remote URLs.

---

3. `background-repeat`

Controls if the background image repeats.

```
background-repeat: repeat;      /* Default */
background-repeat: no-repeat;
background-repeat: repeat-x;    /* Only horizontally */
background-repeat: repeat-y;    /* Only vertically */
```

---

4. `background-size`

Sets the size of the background image.

```
background-size: cover;     /* Fill container and crop */
background-size: contain;   /* Fit image without cropping */
background-size: 100px 200px; /* Custom dimensions */
```

---

5. `background-position`

Positions the background image within the element.

```
background-position: center;
background-position: top right;
background-position: 50% 50%;
```

---

6. `background-attachment`

Controls scroll behavior.

```
background-attachment: scroll;   /* Default */
background-attachment: fixed;    /* Stays in place during scroll */
```

7. Shorthand Property: `background`

You can combine all background properties in one line.

```
div {
  background: url('bg.jpg') no-repeat center center / cover;
}
```

Border Properties

1. `border-width`

Sets the thickness of the border.

```
div {
  border-width: 3px;
}
```

---

2. `border-style`

Defines the style of the border.

```
border-style: solid;     /* Common */
border-style: dashed;
border-style: dotted;
border-style: double;
border-style: none;
```

---

3. `border-color`

Sets the color of the border.

```
border-color: darkgray;
```

---

4. Shorthand: `border`

You can combine width, style, and color.

```
div {
  border: 2px solid #333;
}
```

---

5. Individual Sides

```
border-top: 1px solid black;
border-right: 2px dashed red;
border-bottom: none;
border-left: 3px dotted green;
```

---

6. `border-radius`

Rounds the corners of an element.

```
button {
  border-radius: 10px;
}
```

You can also use percentages to make circular shapes:

```
img {
  border-radius: 50%; /* Perfect circle for square images */
}
```

Summary

- Use background properties to add color, images, and gradients.
- Borders help separate and define content.
- Use `border-radius` to soften corners and create modern designs.

Margin and Padding in CSS

Margin and **Padding** are two of the most commonly used properties in CSS to control **spacing** around elements. They are part of the CSS **Box Model** and play a crucial role in layout and visual structure.

Padding vs Margin

Property	Affects	Where the space appears
`padding`	Inside the element	Between the content and the border
`margin`	Outside the element	Between the element and others

Padding

Apply space inside the border, around the content.

```
.box {
  padding: 20px;
}
```

This adds 20px space inside all four sides of the `.box`.

Individual sides

```
padding-top: 10px;
padding-right: 15px;
```

```
padding-bottom: 10px;
padding-left: 15px;
```

Shorthand

```
padding: 10px 15px 10px 15px;  /* top right bottom left */
padding: 10px 15px;            /* top-bottom | right-left */
padding: 10px;                 /* all sides */
```

Margin

Adds space outside the border of an element.

```
.card {
  margin: 30px;
}
```

This separates the `.card` from nearby elements.

Individual sides

```
margin-top: 20px;
margin-right: 0;
margin-bottom: 20px;
margin-left: auto;
```

Shorthand

```
margin: 20px 40px 20px 40px;  /* top right bottom left */
margin: 20px 40px;            /* top-bottom | right-left */
margin: 0 auto;               /* top-bottom: 0, left-right: auto (used for
centering) */
```

Auto Margin (Horizontal Centering)

```
.container {
  width: 500px;
  margin: 0 auto;
}
```

This centers the container **horizontally** if a fixed width is set.

Margin Collapse

When two vertical margins meet (e.g., margin-bottom of one element and margin-top of the next), the **larger one wins**, not their sum.

```
h1 {
  margin-bottom: 30px;
}

p {
  margin-top: 20px;
}
```

The space between them will be **30px**, not 50px.

Summary

- **Padding** pushes content **inward**.
- **Margin** pushes the element **outward**.
- Use shorthand to simplify your CSS.
- Be aware of **margin collapsing** in vertical spacing.

The Display Property in CSS

The `display` property controls **how an element is rendered** on the page — whether it takes up a full line, shares space with others, behaves like a container, or is completely hidden.

Understanding how display works is critical to mastering layout in CSS.

Common Display Values

1. `block`

- The element takes up the **full width** of its container.
- Starts on a **new line**.

Examples of block elements: `<div>`, `<p>`, `<h1>` – `<h6>`, `<section>`, `<article>`

```
div {
  display: block;
}
```

---

2. `inline`

- The element takes up **only as much width** as its content.
- Can appear **next to other inline elements**.
- **Cannot** set width, height, margin-top/bottom, or padding-top/bottom effectively.

Examples: `<span>`, `<a>`, `<strong>`, `<em>`

```
span {
  display: inline;
}
```

---

3. `inline-block`

- Behaves like `inline` but **allows width, height, margin, and padding** to be set.
- Does **not** force a line break.

```
button {
  display: inline-block;
  width: 150px;
  height: 40px;
}
```

---

4. `none`

- Completely **hides** the element from the page.
- The element is **not rendered**, and does **not take up any space**.

```
.alert {
  display: none;
}
```

Useful for toggling visibility dynamically (e.g., with JavaScript).

---

5. `flex` and `grid` (Coming Soon)

These are modern layout tools you'll learn in upcoming lessons:

- `flex` enables 1D flexible layouts.
- `grid` enables 2D grid layouts.

Visual Example

```
<div style="display: inline-block; width: 100px; background: lightgray;">
  Box 1
</div>
<div style="display: inline-block; width: 100px; background: lightblue;">
  Box 2
</div>
```

These boxes appear **side by side** with a fixed width.

Changing Display in Practice

```
nav {
  display: block;
}

nav a {
  display: inline-block;
  padding: 10px;
}
```

Summary

- `block` : Full width, starts new line.
- `inline` : Sits within a line, no block features.
- `inline-block` : Inline behavior with block features.
- `none` : Removes the element completely.
- `flex` and `grid` : Modern layouts covered soon.

Positioning in CSS

CSS positioning allows you to **move elements** from their default flow and place them **precisely** where you want on the page. It's an essential part of creating modern, interactive layouts.

The `position` Property

There are **five main values**:

Value for Position	Description
`static`	Default. Element stays in the normal document flow
`relative`	Moves the element **relative to its normal position**
`absolute`	Removes from flow; positions **relative to nearest positioned ancestor**
`fixed`	Positions the element **relative to the browser window**, even on scroll
`sticky`	Behaves like `relative`, but **sticks to a position while scrolling**

---

1. `static` (Default)

Every element is positioned statically by default.

```css
div {
  position: static;
}
```

You **can't move** statically positioned elements with `top`, `left`, etc.

---

2. `relative`

Moves the element **relative to where it would normally be**.

```css
.box {
  position: relative;
  top: 20px;
  left: 10px;
}
```

It stays in the document flow, but shifts slightly.

---

3. `absolute`

- Removes the element from normal flow.
- Positions it **relative to the closest ancestor with `position` set** (not `static`).
- If no positioned ancestor, it uses the `<html>` element.

```css
.parent {
  position: relative;
}

.child {
  position: absolute;
  top: 0;
```

```
  right: 0;
}
```

The `.child` will stick to the top-right of `.parent`.

---

4. `fixed`

- Stays in a fixed position **relative to the viewport**.
- Does **not move** when scrolling.

```
.banner {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
}
```

Great for sticky headers, floating buttons, or back-to-top links.

---

5. `sticky`

- Acts like `relative` until a scroll threshold is reached, then behaves like `fixed`.

```
.heading {
  position: sticky;
  top: 0;
  background: white;
}
```

Sticky headers or sidebars often use this behavior.

`top`, `right`, `bottom`, `left`

These properties only work with `relative`, `absolute`, `fixed`, or `sticky`.

```
.box {
  position: absolute;
  top: 50px;
  left: 100px;
}
```

`z-index`

Controls the **stacking order** of overlapping elements.

```
.modal {
  position: absolute;
  z-index: 100;
}
```

Higher `z-index` values appear **above** lower ones.

Summary

- Use `relative` for minor adjustments.
- Use `absolute` to fully control placement inside containers.
- Use `fixed` for elements that stay on screen while scrolling.
- Use `sticky` for scroll-based sticky behaviors.
- Always understand the **positioning context** — especially when using `absolute`.

Flexbox in CSS

Flexbox (Flexible Box Layout) is a powerful layout system in CSS that allows you to **align, space, and distribute elements** easily — especially when building responsive layouts.

It's ideal for **one-dimensional layouts** (either a row or a column).

Getting Started

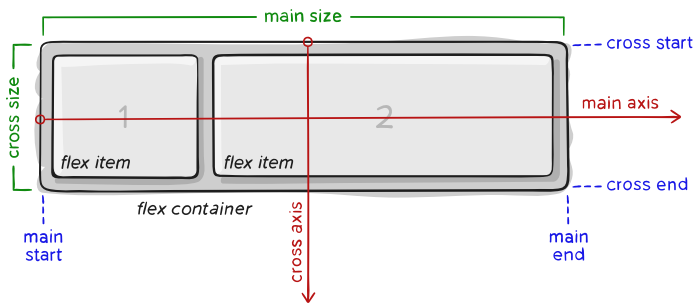
To use Flexbox, set the parent container's `display` to `flex`:

```
.container {
  display: flex;
}
```

Now, all **direct children** of `.container` become **flex items**.

Main Concepts

Term	Description
Main Axis	The primary direction (`row` by default)
Cross Axis	Perpendicular to main axis
Flex Container	The parent element with `display: flex`
Flex Items	The children inside the container



Flex Direction

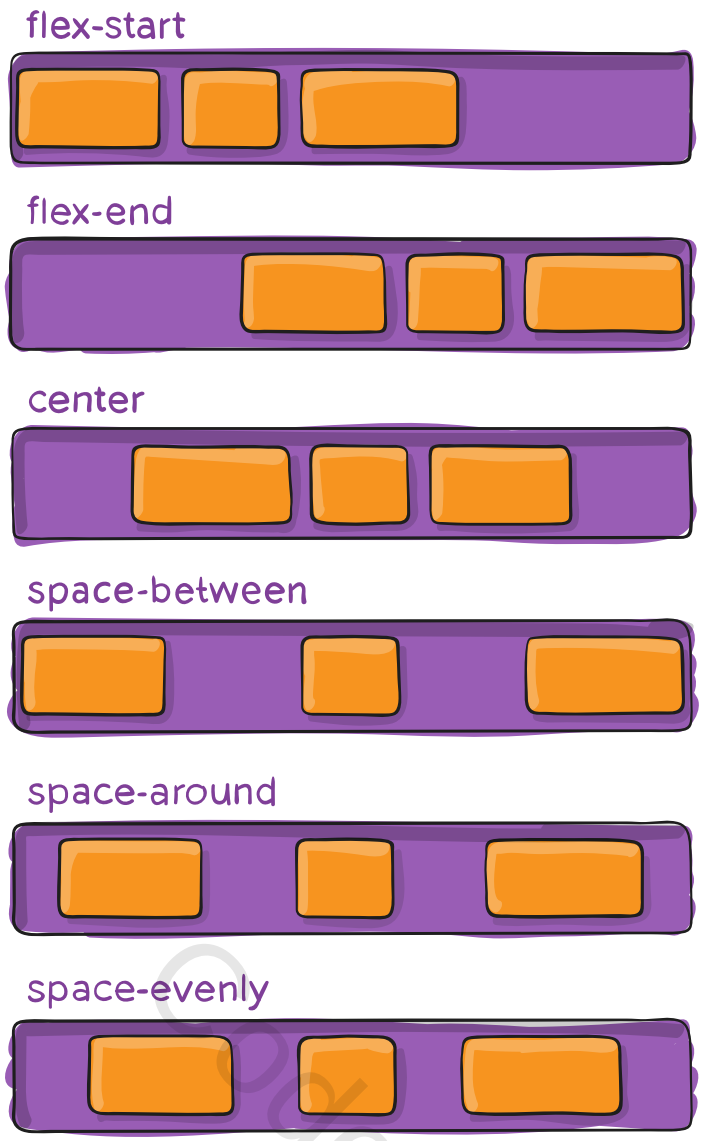
Controls the direction of flex items.

```
.container {
  display: flex;
  flex-direction: row;      /* default */
  flex-direction: row-reverse;
  flex-direction: column;
  flex-direction: column-reverse;
}
```

Justify Content (Main Axis Alignment)

Controls how items are **aligned along the main axis** (horizontal by default).

```
.container {
  justify-content: flex-start;   /* default */
  justify-content: flex-end;
  justify-content: center;
  justify-content: space-between;
  justify-content: space-around;
  justify-content: space-evenly;
}
```

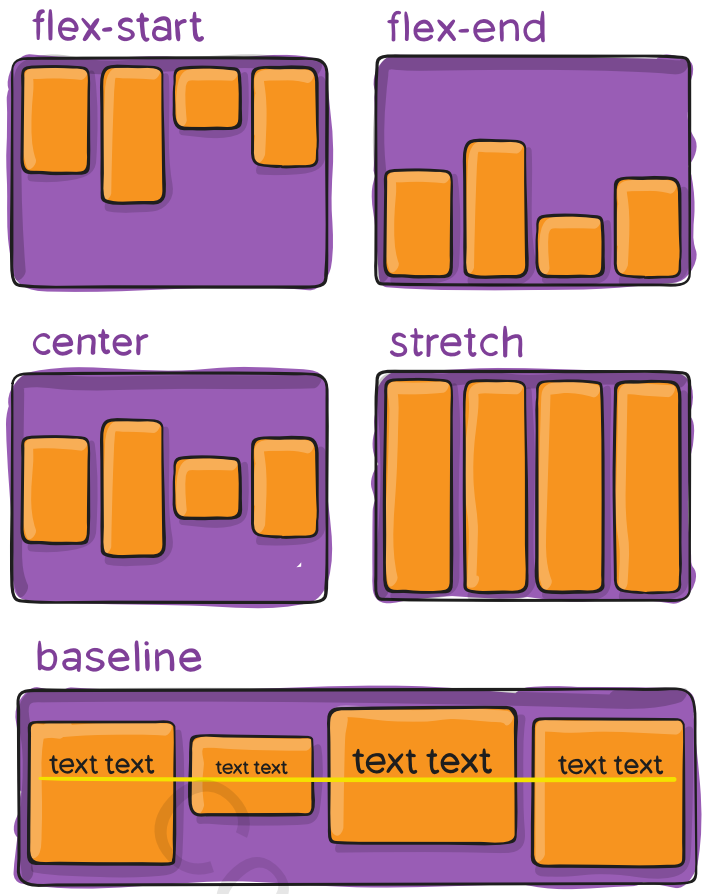


Align Items (Cross Axis Alignment)

Controls how items are **aligned on the cross axis** (vertical by default).

```
.container {
  align-items: stretch;     /* default */
  align-items: flex-start;
  align-items: flex-end;
  align-items: center;
```

```
  align-items: baseline;
}
```



Align Self

Allows individual items to override `align-items`.

```
.item {
  align-self: flex-end;
}
```

Flex Wrap

By default, items try to fit into a single line. Use `flex-wrap` to wrap them:

```
.container {
  flex-wrap: wrap;
  flex-wrap: nowrap;       /* default */
  flex-wrap: wrap-reverse;
}
```

Gap (Spacing Between Items)

```
.container {
  gap: 20px;
}
```

This replaces the need for margins between flex items.

Flex Grow, Shrink, Basis

Control how items grow, shrink, or have an initial size:

```
.item {
  flex-grow: 1;    /* takes remaining space */
  flex-shrink: 1;  /* shrink if needed */
  flex-basis: 200px; /* default size */
}
```

Shorthand:

```css
.item {
  flex: 1 1 200px;
}
```

Example Layout

```html
<div class="container">
  <div class="item">One</div>
  <div class="item">Two</div>
  <div class="item">Three</div>
</div>
```

```css
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 10px;
}
.item {
  background: lightgray;
  padding: 20px;
  flex: 1;
}
```

Summary

- `display: flex` turns a container into a Flexbox layout.
- Use `justify-content`, `align-items`, and `flex-direction` to control layout flow.
- `flex` shorthand (`grow shrink basis`) gives you fine-grained sizing control.
- Use `gap` instead of margins for consistent spacing.

CSS Grid

CSS Grid Layout is a **two-dimensional** layout system that allows you to design web pages in **rows and columns**. It gives you complete control over both axes, unlike Flexbox which is mostly one-dimensional.

Enabling Grid

Set the container's `display` property to `grid`:

```
.container {
  display: grid;
}
```

All direct children of this container become **grid items**.

Defining Rows and Columns

You use `grid-template-columns` and `grid-template-rows` to define the grid structure:

```
.container {
  display: grid;
  grid-template-columns: 200px 1fr 1fr;
  grid-template-rows: 100px auto;
}
```

- `1fr` means "1 fraction of remaining space"
- You can mix fixed units (e.g. `px`) with flexible ones (`fr`)

Repeat Syntax

To avoid repeating values:

```
.container {
  grid-template-columns: repeat(3, 1fr);
}
```

Creates 3 equal-width columns.

Grid Gap

Adds spacing between rows and columns:

```
.container {
  gap: 20px; /* shorthand for row-gap and column-gap */
}
```

Placing Items

You can control where an item appears in the grid using `grid-column` and `grid-row`.

```
.item {
  grid-column: 1 / 3; /* spans column 1 to 2 (exclusive of 3) */
  grid-row: 2 / 3;
}
```

You can also use `span`:

```css
.item {
  grid-column: span 2;
}
```

Named Areas (Optional but Powerful)

Define areas using `grid-template-areas`:

```css
.container {
  display: grid;
  grid-template-areas:
    "header header"
    "sidebar content"
    "footer footer";
  grid-template-columns: 1fr 3fr;
  grid-template-rows: auto 1fr auto;
}
```

Then assign each item:

```css
.header { grid-area: header; }
.sidebar { grid-area: sidebar; }
.content { grid-area: content; }
.footer { grid-area: footer; }
```

Auto-Placement

Grid can automatically place items:

```css
.container {
  display: grid;
```

```
  grid-template-columns: repeat(auto-fill, minmax(150px, 1fr));
}
```

This makes the layout responsive, automatically filling space with flexible-width items.

Complete Example

```
<div class="container">
  <div class="item header">Header</div>
  <div class="item sidebar">Sidebar</div>
  <div class="item content">Content</div>
  <div class="item footer">Footer</div>
</div>
```

```
.container {
  display: grid;
  grid-template-areas:
    "header header"
    "sidebar content"
    "footer footer";
  grid-template-columns: 1fr 3fr;
  grid-template-rows: auto 1fr auto;
  gap: 10px;
}

.header  { grid-area: header; background: #ddd; }
.sidebar { grid-area: sidebar; background: #bbb; }
.content { grid-area: content; background: #eee; }
.footer  { grid-area: footer; background: #ccc; }

.item {
  padding: 20px;
}
```

Summary

- CSS Grid is **perfect for page layouts** with rows and columns.
- `grid-template-columns` and `grid-template-rows` define structure.
- Use `grid-column` and `grid-row` to place or span items.
- Named grid areas make your layout more readable and semantic.
- Auto-fill and auto-fit allow responsive grids.

CSS Media Queries

Media Queries allow you to create responsive designs by applying CSS rules based on the device's characteristics — such as screen width, height, orientation, and resolution.

They are essential for building **mobile-first**, **responsive** websites that adapt to various screen sizes (phones, tablets, desktops).

Basic Syntax

```
@media (condition) {
  /* CSS rules */
}
```

Example: Target screens smaller than 768px

```
@media (max-width: 768px) {
  body {
    background-color: lightgray;
  }
}
```

This CSS will apply **only** when the screen width is **768px or less**.

Common Conditions

Media Feature	Description	Example
`max-width`	Target screens **up to** a width	`@media (max-width: 600px)`
`min-width`	Target screens **starting from** a width	`@media (min-width: 1024px)`
`orientation`	Target `portrait` or `landscape` mode	`@media (orientation: landscape)`
`max-height`	Target screen height	`@media (max-height: 500px)`
`resolution`	Target pixel density	`@media (min-resolution: 2dppx)`

Responsive Layout Example

```
.container {
  padding: 20px;
  font-size: 18px;
}

@media (max-width: 768px) {
  .container {
    padding: 10px;
    font-size: 16px;
  }
}

@media (max-width: 480px) {
  .container {
    font-size: 14px;
```

```
  }
}
```

This approach ensures your layout adjusts smoothly as screen sizes change.

Combining Multiple Conditions

```
@media (min-width: 600px) and (max-width: 1024px) {
  .sidebar {
    display: none;
  }
}
```

You can combine conditions using `and`, `or`, or `not`.

Mobile-First Approach

Start with styles for small screens, then use `min-width` to add enhancements for larger screens.

```
/* Mobile-first (default) */
.card {
  font-size: 14px;
}

/* Tablet and up */
@media (min-width: 768px) {
  .card {
    font-size: 16px;
  }
}

/* Desktop and up */
@media (min-width: 1024px) {
```

```
  .card {
    font-size: 18px;
  }
}
```

Media Queries for Print

```
@media print {
  body {
    background: white;
    color: black;
  }

  .no-print {
    display: none;
  }
}
```

Used to style web pages when printed.

Summary

- Media queries help build **responsive** and **accessible** websites.
- Use `min-width` for mobile-first, scalable layouts.
- Combine media queries for precise control across devices.

CSS Variables and Custom Properties

CSS Variables, also called **Custom Properties**, allow you to store values in a reusable way — making your CSS more maintainable and dynamic.

They follow the pattern of:

```
--custom-name: value;
```

Declaring a CSS Variable

Variables are declared inside a selector using the `--` prefix:

```
:root {
  --primary-color: #3498db;
  --font-size: 16px;
}
```

- `:root` is the highest-level selector (like `html`) — variables here are global.
- Variables declared inside `:root` can be used throughout your stylesheet.

Using a CSS Variable

Use the `var()` function to apply the variable:

```
body {
  color: var(--primary-color);
```

```
  font-size: var(--font-size);
}
```

Why Use CSS Variables?

☑ **Consistency** ☑ **Easy to update** (change in one place) ☑ **Theme support** (light/dark mode) ☑ **Cleaner, scalable CSS**

Example: Theming with CSS Variables

```
:root {
  --bg-color: white;
  --text-color: black;
}

body {
  background-color: var(--bg-color);
  color: var(--text-color);
}
```


You can override these in a different class for themes:

```
.dark-theme {
  --bg-color: #121212;
  --text-color: #ffffff;
}
```

Now just add `class="dark-theme"` to `<body>` or a wrapper div to switch themes.

Fallback Values

If a variable isn't defined, you can specify a fallback:

```
h1 {
  color: var(--heading-color, blue);
}
```

If `--heading-color` is not set, `blue` will be used instead.

Scoped Variables

Variables can also be scoped to a class or element:

```
.card {
  --border-radius: 10px;
  border-radius: var(--border-radius);
}
```

Only elements within `.card` can access this variable.

Real-World Example

```
:root {
  --btn-padding: 12px 24px;
  --btn-color: #fff;
  --btn-bg: #2ecc71;
}

.button {
  padding: var(--btn-padding);
  color: var(--btn-color);
  background-color: var(--btn-bg);
```

```
  border: none;
  border-radius: 6px;
  cursor: pointer;
}
```

Update theme by just changing variables in `:root`.

Summary

- Use `--variable-name` to declare and `var(--variable-name)` to use.
- Declare in `:root` for global usage.
- Support theming, dynamic styles, and cleaner code.
- Can be scoped or overridden for flexibility.

CSS Transitions and Animations

CSS provides powerful tools for creating smooth, engaging user experiences through **transitions** and **animations**. These effects can enhance the visual appeal and usability of your website without needing JavaScript.

---

1. CSS Transitions

A **transition** is used to change CSS properties **smoothly** over a given duration.

Basic Syntax

```
selector {
  transition: property duration timing-function delay;
}
```

- `property` : The CSS property to animate (e.g., `background-color` , `transform` , etc.)
- `duration` : How long the transition lasts (e.g., `0.3s` , `1s`)
- `timing-function` : The pace of the transition (`ease` , `linear` , `ease-in` , `ease-out` , etc.)
- `delay` : Optional delay before starting

Example

```
.button {
  background-color: blue;
  color: white;
  transition: background-color 0.3s ease;
}
```

```
.button:hover {
  background-color: green;
}
```

This smoothly changes the button's background color on hover.

Shorthand vs Longhand

Shorthand:

```
transition: all 0.5s ease;
```

Longhand:

```
transition-property: background-color;
transition-duration: 0.5s;
transition-timing-function: ease;
transition-delay: 0s;
```

---

2. CSS Animations

CSS animations allow more **complex, keyframe-based** changes over time.

Basic Syntax

```
selector {
  animation: animation-name duration timing-function delay iteration-count
direction;
}
```

Keyframes

Define how the animation should behave at different points:

```css
@keyframes slideIn {
  from {
    transform: translateX(-100%);
    opacity: 0;
  }
  to {
    transform: translateX(0);
    opacity: 1;
  }
}
```

Example

```css
.box {
  width: 100px;
  height: 100px;
  background-color: red;
  animation: slideIn 1s ease-in-out;
}
```

Animation Properties

Property	Description
`animation-name`	Name of the `@keyframes` to use
`animation-duration`	How long the animation takes
`animation-delay`	Delay before starting
`animation-iteration-count`	Number of times to run (or `infinite`)
`animation-direction`	`normal`, `reverse`, `alternate`
`animation-fill-mode`	Defines final state: `forwards`, `backwards`, `both`

Property	Description
`animation-play-state`	`running` or `paused`

Looping Animations

```
.pulse {
  animation: pulse 2s infinite;
}

@keyframes pulse {
  0%, 100% {
    transform: scale(1);
  }
  50% {
    transform: scale(1.1);
  }
}
```

Combining Transitions and Animations

Transitions are great for hover and interactive effects. Animations are better for more **dynamic**, self-running effects.

Example using both:

```
.card {
  transition: transform 0.3s;
}

.card:hover {
  transform: scale(1.05);
}

@keyframes fadeIn {
```

```
  from { opacity: 0; }
  to { opacity: 1; }
}

.card {
  animation: fadeIn 1s ease;
}
```

Summary

- Use **transitions** for smooth changes on hover, focus, etc.
- Use **animations** for keyframe-driven effects like entrance, bounce, etc.
- Keep animations subtle and purposeful — avoid overwhelming the user.

CSS Transformations

CSS **transforms** allow you to visually manipulate elements by rotating, scaling, skewing, or translating them. Transforms are applied using the `transform` property.

---

1. `transform` Property

Syntax:

```
selector {
  transform: function(value);
}
```

Multiple functions can be combined:

```
transform: translateX(50px) rotate(45deg) scale(1.2);
```

---

2. Types of Transformations

a. `translate()`

Moves an element from its current position.

```
.box {
  transform: translateX(50px); /* Move 50px to the right */
}
```

Other variations:

- `translateY(30px)` – moves vertically
- `translate(50px, 30px)` – moves on both axes

---

b. `rotate()`

Rotates the element clockwise by default.

```
.box {
  transform: rotate(45deg); /* Rotate 45 degrees */
}
```

Use negative values to rotate counter-clockwise:

```
transform: rotate(-45deg);
```

---

c. `scale()`

Scales the size of an element.

```
.box {
  transform: scale(1.5); /* Increase size by 1.5x */
}
```

- `scaleX(2)` – scales horizontally
- `scaleY(0.5)` – scales vertically

---

d. `skew()`

Slants an element along the X and/or Y axis.

```css
.box {
  transform: skew(20deg, 10deg); /* Skew in X and Y */
}
```

Individual axis:

- `skewX(20deg)`
- `skewY(10deg)`

---

e. `matrix()`

A shorthand to apply multiple transformations using a 2D matrix. Rarely used directly because it's less readable.

---

3. Transform Origin

By default, transforms are applied relative to the **center** of the element. You can change this with `transform-origin`.

```css
.box {
  transform: rotate(45deg);
  transform-origin: top left;
}
```

---

4. Combining Multiple Transforms

```css
.box {
  transform: translateX(100px) rotate(30deg) scale(1.2);
}
```

The **order matters**: transforms are applied from left to right.

---

5. 3D Transforms (Intro Only)

- `rotateX()`, `rotateY()`, and `rotateZ()` add 3D rotation.
- `perspective` property is needed to see 3D depth.

Example:

```
.box {
  transform: rotateY(45deg);
  transform-style: preserve-3d;
}
```

Summary

Transform Function	Description
`translate()`	Moves element
`rotate()`	Rotates element
`scale()`	Resizes element
`skew()`	Slants element
`matrix()`	Combines multiple transforms

CSS Transforms are foundational for building modern UI effects — often combined with **transitions** and **animations**.

Introduction to JavaScript

What is JavaScript?

JavaScript is a **programming language** used to make web pages **interactive**. While **HTML** structures the page and **CSS** styles it, **JavaScript adds behavior**.

For example:

- Want to show a popup when a user clicks a button? Use JavaScript.
- Want to build a game, form validation, or fetch data from a server? Use JavaScript.

JavaScript runs **in the browser**, meaning it executes on the user's device.

Why JavaScript?

- Works on **all modern browsers**
- Essential for **front-end development**
- Used by frameworks like **React**, **Vue**, **Angular**
- Can also be used on the **backend** (Node.js)

Where Can You Write JavaScript?

There are 3 common ways to write JavaScript in a webpage:

Inline (Not recommended)

```
<button onclick="alert('Hello!')">Click Me</button>
```

In a `<script>` tag inside HTML

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello World</h1>
    <script>
      console.log("JavaScript is working!");
    </script>
  </body>
</html>
```

External JavaScript File (Recommended)

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello</h1>
    <script src="script.js"></script>
  </body>
</html>
```

```
// script.js
console.log("Hello from external JS file!");
```

Your First JavaScript Code

Let's start with the simplest code that prints something:

```
console.log("Hello JavaScript");
```

> `console.log()` is used to print messages to the **browser console**.

How to Open the Console

- Right-click on the page > Inspect
- Go to the **Console** tab
- You'll see outputs from `console.log()` here

JavaScript is Case-Sensitive

```
let x = 5;
let X = 10;
console.log(x); // 5
console.log(X); // 10
```

Comments in JavaScript

Use comments to explain your code:

```
// This is a single-line comment

/*
  This is a
  multi-line comment
*/
```

Summary

- JavaScript makes your website interactive.
- You can write it inside HTML or in separate `.js` files.
- Use `console.log()` to test your code.
- Practice by writing simple scripts and watching them work in the browser.

Variables and Data Types in JavaScript

What is a Variable?

A **variable** is a named container for storing data. In JavaScript, we can declare variables using:

```
let name = "Harry";
const age = 25;
var city = "Delhi";
```

`let` vs `const` vs `var`

Keyword	Reassignable?	Block Scoped?	Hoisted?
`let`	Yes	Yes	Yes
`const`	No	Yes	Yes
`var`	Yes	No	Yes (but undefined)

Use `let` when:

- You plan to **change the value** later.

```
let score = 0;
score = 10;
```

Use `const` when:

- The value **should not change**.

```
const pi = 3.14159;
// pi = 3.14; Error
```

Avoid `var`

- It behaves inconsistently due to hoisting and lack of block scoping.

JavaScript Data Types

JavaScript has two categories of data types:

---

1. Primitive (Value) Data Types

These are **immutable** and **stored directly in memory**.

Data Type	Example
`string`	`"Hello World"`
`number`	`42` , `3.14` , `-100`
`boolean`	`true` , `false`
`null`	`null`
`undefined`	`undefined`
`bigint`	`12345678901234567890n`
`symbol`	`Symbol("id")`


```
let name = "Harry";        // string
let age = 25;              // number
let isCool = true;         // boolean
let noValue = null;        // null
let notDefined;            // undefined
```

Note: `typeof null` is `"object"` due to a long-standing bug in JS.

---

2. Non-Primitive (Reference) Data Types

These hold **references** to memory, not actual values.

Type	Example
Object	`{ name: "Harry" }`
Array	`[1, 2, 3]`
Function	`function() {}`
Date, RegExp, etc.	Built-in Objects

```
let person = { name: "Harry", age: 25 }; // Object
let colors = ["red", "blue", "green"];   // Array
let greet = function() { console.log("Hi") }; // Function
```

Differences Between Primitive and Reference Types

Feature	Primitive	Reference
Stored as	Value	Memory address (reference)
Mutable?	Immutable	Mutable
Copied as	Value	Reference

```
let a = 10;
let b = a; // Copy by value
b = 20;
console.log(a); // 10 (unchanged)

let obj1 = { x: 1 };
```

```
let obj2 = obj1; // Copy by reference
obj2.x = 2;
console.log(obj1.x); // 2 (both point to same object)
```

`typeof` Operator

Use `typeof` to check the type of a variable:

```
console.log(typeof "Hello");     // string
console.log(typeof 100);         // number
console.log(typeof true);        // boolean
console.log(typeof undefined);   // undefined
console.log(typeof null);        // object (quirk!)
console.log(typeof {});          // object
console.log(typeof []);          // object (array is also object)
console.log(typeof function(){});// function
```

Summary

- Use `let` for changeable values, `const` for constants.
- Understand the difference between **primitive** (copied by value) and **reference** types (copied by reference).
- `typeof` is useful but not perfect (e.g., `typeof null === "object"`).

Naming Variables in JavaScript

Rules for Naming Variables

1. **Variable names must begin with a letter, underscore `_`, or dollar sign `$`.**
   Valid examples:

   ```
   let name;
   let _count;
   let $price;
   ```

   Invalid example:

   ```
   let 1user; // Invalid: cannot start with a number
   ```

2. **Use camelCase for variable names.** In JavaScript, the convention is to use **camelCase**, where the first word is lowercase and each new word starts with an uppercase letter:

   ```
   let userName;
   let totalAmount;
   let isLoggedIn;
   ```

3. **Be descriptive and meaningful.** Use names that describe the purpose of the variable:

   ```
   let age = 25;            // Good
   let userAge = 25;        // Better
   let a = 25;              // Poor
   ```

4. **Avoid using reserved keywords.** JavaScript has reserved words that cannot be used as variable names:

```
let let;       // Invalid
let function;  // Invalid
```

5. **No spaces or special characters.** Variable names must not contain spaces or symbols like `@`, `-`, or `#`:

```
let user name;     // Invalid
let user-name;     // Invalid
let user_name;     // Valid, but camelCase is preferred
```

Good Examples

```
let firstName = "John";
let totalPrice = 100;
let isAvailable = true;
```

Bad Examples

```
let x = "John";           // Not descriptive
let user_123_abc = 25;    // Unclear naming
let 1user = "Alice";      // Invalid syntax
```

Boolean Variable Naming

When naming variables that store `true` or `false`, use prefixes like `is`, `has`, or `can` to make their intent clear:

```
let isLoggedIn = true;
let hasAccess = false;
let canEdit = true;
```

Summary

- Use camelCase.
- Choose meaningful names.
- Start names with a letter, `_`, or `$`.
- Avoid JavaScript keywords.
- Avoid spaces and special characters.

Operators in JavaScript

Operators are symbols used to **perform operations** on values and variables. For example, you use `+` to add two numbers, `=` to assign values, and `==` to compare values.

---

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example	Result
`+`	Addition	`5 + 2`	`7`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`5 * 2`	`10`
`/`	Division	`10 / 2`	`5`
`%`	Modulus (Remainder)	`5 % 2`	`1`
`**`	Exponentiation	`2 ** 3`	`8`
`++`	Increment	`let x = 1; x++`	`2`
`--`	Decrement	`let x = 2; x--`	`1`

---

2. Assignment Operators

Used to assign values to variables.

Operator	Example	Same As
`=`	`x = 5`	Assign 5 to x
`+=`	`x += 2`	`x = x + 2`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 4`	`x = x * 4`
`/=`	`x /= 2`	`x = x / 2`
`%=`	`x %= 2`	`x = x % 2`

---

3. Comparison Operators

Used to compare two values. They return a Boolean (`true` or `false`).

Operator	Description	Example	Result
`==`	Equal (loose)	`5 == '5'`	`true`
`===`	Strict equal (type + value)	`5 === '5'`	`false`
`!=`	Not equal	`5 != '5'`	`false`
`!==`	Strict not equal	`5 !== '5'`	`true`
`>`	Greater than	`5 > 3`	`true`
`<`	Less than	`5 < 3`	`false`
`>=`	Greater than or equal	`5 >= 5`	`true`
`<=`	Less than or equal	`3 <= 5`	`true`

---

4. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example	Result
`&&`	Logical AND	`true && false`	`false`
`||`	Logical OR	`true || false`	`true`
`!`	Logical NOT	`!true`	`false`

Example:

```
let age = 20;
if (age > 18 && age < 60) {
  console.log("You are eligible");
}
```

---

5. Ternary Operator

A shorthand way to write simple `if...else`.

```
let age = 17;
let result = age >= 18 ? "Adult" : "Minor";
console.log(result); // "Minor"
```

Summary

- Use arithmetic operators to perform math.
- Use assignment operators to update variables.
- Use comparison and logical operators for decision making.
- Ternary operator is a cleaner way to write simple conditions.

If-Else Statements in JavaScript

What is an If-Else Statement?

An `if` statement is used to run a block of code only if a specified condition is true. You can use `else` or `else if` to run different blocks of code based on different conditions.

Basic Syntax

```
if (condition) {
  // code to run if condition is true
} else {
  // code to run if condition is false
}
```

Example

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

---

if-else-if Ladder

You can check multiple conditions using `else if`:

```
let score = 85;

if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else {
  console.log("Grade: F");
}
```

Nested if Statements

You can place one `if` statement inside another:

```
let age = 25;
let hasID = true;

if (age >= 18) {
  if (hasID) {
    console.log("Access granted.");
  } else {
    console.log("ID required.");
  }
} else {
  console.log("Access denied. You must be at least 18.");
}
```

Using the Ternary Operator (Short Form)

The ternary operator is a shorter way to write simple `if-else` statements:

```
let isLoggedIn = true;

let message = isLoggedIn ? "Welcome back!" : "Please log in.";
console.log(message);
```

Summary

- Use `if` to test a condition.
- Use `else` for an alternative block if the condition is false.
- Use `else if` to test multiple conditions.
- The ternary operator is a compact way to write simple `if-else` logic.

Objects in JavaScript – Deep Dive

Objects in JavaScript are used to store collections of **key-value pairs**. They are one of the most important and widely used data types in the language.

Creating an Object

You can create an object using **object literal syntax**:

```
let person = {
  name: "Alice",
  age: 30,
  isEmployed: true
};
```

Accessing Object Properties

You can access properties using **dot notation** or **bracket notation**:

```
console.log(person.name);       // "Alice"
console.log(person["age"]);     // 30
```

Use bracket notation when the property name is stored in a variable or contains special characters:

```
let key = "isEmployed";
console.log(person[key]);       // true
```

Modifying Object Properties

```
person.age = 31;
person["name"] = "Bob";
```

Adding New Properties

```
person.city = "Delhi";
person["hobby"] = "Reading";
```

Deleting Properties

```
delete person.hobby;
```

Checking if a Property Exists

```
console.log("age" in person);          // true
console.log(person.hasOwnProperty("city")); // true
```

Looping Through Object Properties

Use `for...in` to iterate over an object's keys:

```
for (let key in person) {
  console.log(key + ": " + person[key]);
}
```

Nested Objects

Objects can contain other objects:

```
let student = {
  name: "John",
  address: {
    city: "Mumbai",
    pin: 400001
  }
};

console.log(student.address.city);  // "Mumbai"
```

Objects and Functions

Objects can have **methods** (functions defined inside them):

```
let user = {
  name: "Sara",
  greet: function () {
    console.log("Hello, " + this.name);
  }
};

user.greet(); // "Hello, Sara"
```

You can also use shorthand syntax for methods:

```
let user = {
  name: "Sara",
  greet() {
    console.log("Hello, " + this.name);
  }
};
```

Summary

- Objects store key-value pairs.
- Access and modify properties with dot or bracket notation.
- Use `for...in` to loop through properties.

Loops in JavaScript

Loops allow you to execute a block of code multiple times, which is useful for tasks like iterating over arrays or repeating operations until a condition changes.

---

1. `for` Loop

Syntax

```
for (initialization; condition; finalExpression) {
  // code to execute on each iteration
}
```

Example

```
for (let i = 0; i < 5; i++) {
  console.log(i); // prints 0, 1, 2, 3, 4
}
```

- **initialization**: executed once before the loop starts (e.g., `let i = 0`)
- **condition**: checked before each iteration; if `true`, the loop continues
- **finalExpression**: executed at the end of each iteration (e.g., `i++`)

---

2. `while` Loop

Syntax

```
while (condition) {
  // code to execute while condition is true
}
```

Example

```
let count = 0;
while (count < 3) {
  console.log(count); // prints 0, 1, 2
  count++;
}
```

- The condition is evaluated before each iteration.
- If the condition is initially `false`, the loop body may never run.

---

3. `do...while` Loop

Syntax

```
do {
  // code to execute
} while (condition);
```

Example

```
let num = 0;
do {
  console.log(num); // prints 0
  num++;
} while (num < 1);
```

- The loop body executes **at least once**, then the condition is checked.

---

4. `for...of` Loop

Purpose: Iterate over iterable objects (arrays, strings, etc.).

Syntax

```
for (variable of iterable) {
  // code using variable
}
```

Example

```
let colors = ["red", "green", "blue"];
for (let color of colors) {
  console.log(color);
}
```

---

5. `for...in` Loop

Purpose: Iterate over the keys of an object.

Syntax

```
for (key in object) {
  // code using key and object[key]
}
```

Example

```
let person = { name: "Alice", age: 30 };
for (let prop in person) {
  console.log(prop + ": " + person[prop]);
// prints "name: Alice" and then "age: 30"
}
```

Summary

- `for` : general-purpose loop with counter.
- `while` : loop based on a condition.
- `do...while` : condition checked after first execution.
- `for...of` : iterate over iterable values.
- `for...in` : iterate over object keys.

Control Flow in JavaScript

Control flow means **how your code runs step-by-step**, and how you can make decisions or repeat actions.

JavaScript runs code **from top to bottom**, but you can control the flow using:

- Conditional statements (`if` , `else` , `switch`)
- Loops (`for` , `while` , `do...while`)

---

1. `if` , `else if` , and `else`

Use these to **run code only if a condition is true**.

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult");
} else if (age >= 13) {
  console.log("You are a teenager");
} else {
  console.log("You are a child");
}
```

The first condition that is true will run, others will be skipped.

---

2. `switch` Statement

Use this when you have **multiple fixed cases** to check.

```
let day = "Monday";

switch (day) {
  case "Monday":
    console.log("Start of the week");
    break;
  case "Friday":
    console.log("End of the week");
    break;
  default:
    console.log("Midweek day");
}
```

- Use `break` to stop after each case.
- `default` runs if no cases match.

---

3. Loops

Loops let you **run code multiple times**.

a. `for` Loop

Used when you know how many times to repeat.

```
for (let i = 1; i <= 5; i++) {
  console.log("Count:", i);
}
```

b. `while` Loop

Used when the number of repetitions is unknown.

```
let i = 1;
while (i <= 3) {
  console.log("While loop:", i);
```

```
  i++;
}
```

c. `do...while` Loop

Same as `while`, but runs **at least once**, even if the condition is false.

```
let i = 1;
do {
  console.log("Do while:", i);
  i++;
} while (i <= 3);
```

---

4. `break` and `continue`

- `break` = exit the loop immediately
- `continue` = skip the current iteration

Example:

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue; // skip 3
  console.log(i);        // 1, 2, 4, 5
}
```

Summary

- Use `if`, `else if`, and `else` to control logic.
- Use `switch` for multiple case checking.
- Use loops to repeat actions.
- Use `break` to stop loops, and `continue` to skip one turn.

`break` and `continue` in JavaScript

`break` Statement

The `break` statement is used to exit a loop prematurely, before the loop condition evaluates to false.

Syntax

```
break;
```

Example: Exiting a loop when a condition is met

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break; // exits the loop when i equals 5
    }
    console.log(i);
}
```

Output:

```
0
1
2
3
4
```

Use Cases

- Exiting a `for`, `while`, or `do...while` loop early

`continue` Statement

The `continue` statement skips the current iteration of a loop and proceeds to the next one.

Syntax

```
continue;
```

Example: Skipping a loop iteration

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue; // skips even numbers
    }
    console.log(i);
}
```

Output:

```
1
3
5
7
9
```

Use Cases

- Skipping specific iterations based on a condition
- Useful in filtering or avoiding certain values during iteration

Summary

Statement	Purpose	Effect
`break`	Exit the loop entirely	Stops loop execution
`continue`	Skip current iteration	Moves to the next iteration immediately

Both statements help in controlling loop execution flow more precisely based on conditions.

Functions in JavaScript

A **function** is a block of code that performs a specific task. Instead of repeating the same code again and again, you can write it once in a function and call it whenever needed.

---

1. Function Declaration

This is the most common way to define a function.

```
function greet() {
  console.log("Hello, JavaScript!");
}

greet(); // Call the function
```

---

2. Function with Parameters

You can pass **data into functions** using parameters.

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

greetUser("Ali"); // Output: Hello, Ali
```

---

3. Function with Return Value

Functions can **return** values using the `return` keyword.

```
function add(a, b) {
  return a + b;
}

let result = add(5, 3); // 8
console.log(result);
```

---

4. Function Expressions

Functions can also be stored in variables.

```
const sayHi = function() {
  console.log("Hi there!");
};

sayHi();
```

---

5. Arrow Functions (ES6+)

A shorter way to write functions.

```
const square = (num) => {
  return num * num;
};

console.log(square(4)); // 16
```

One-liner version (if only returning a value):

```
const multiply = (a, b) => a * b;
console.log(multiply(2, 3)); // 6
```

---

6. Scope (Simple Explanation)

Variables declared inside a function are **local** and can't be accessed outside.

```
function showAge() {
  let age = 25;
  console.log(age);
}

showAge();
// console.log(age); // Error: age is not defined
```

---

7. Why Use Functions?

- Organize code into reusable pieces
- Avoid repetition
- Make your code cleaner and easier to understand
- Useful in handling events and user interactions

Summary

- Use `function` to define reusable code blocks.
- Pass data using parameters and return results with `return`.
- Store functions in variables or use arrow functions for brevity.
- Functions help you write cleaner and modular code.

Arrays in JavaScript

An **array** is a collection of items stored in a single variable. It lets you store multiple values — like a list of names, numbers, or even other arrays.

---

1. Creating Arrays

```
let fruits = ["apple", "banana", "mango"];
let numbers = [10, 20, 30, 40];
```

You can mix data types, but it's better to keep arrays consistent.

```
let mixed = [1, "hello", true];
```


---

2. Accessing and Modifying Elements

Arrays use **zero-based indexing**.

```
let fruits = ["apple", "banana", "mango"];
console.log(fruits[0]); // "apple"
fruits[1] = "orange";   // change "banana" to "orange"
console.log(fruits);    // ["apple", "orange", "mango"]
```

---

3. Array Length

```
let colors = ["red", "green", "blue"];
console.log(colors.length); // 3
```

---

4. Common Array Methods

a. `push()` – Add item at the end

```
colors.push("yellow");
```

b. `pop()` – Remove last item

```
colors.pop();
```

c. `shift()` – Remove first item

```
colors.shift();
```

d. `unshift()` – Add item at the beginning

```
colors.unshift("pink");
```

---

5. Looping Through Arrays

Using `for` loop:

```
let fruits = ["apple", "banana", "mango"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

Using `for...of` loop:

```
for (let fruit of fruits) {
  console.log(fruit);
}
```

---

6. `map()` – Transform Array

Creates a new array by applying a function to each item.

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

---

7. `filter()` – Filter Items

Returns a new array of items that match a condition.

```
let scores = [30, 50, 90, 20];
let passed = scores.filter(score => score >= 50);
console.log(passed); // [50, 90]
```

8. Other Useful Methods

Method	Description
`includes()`	Checks if an item exists in the array
`indexOf()`	Finds index of an item
`slice()`	Extracts part of the array
`join()`	Joins array into a string

Example:

```
let names = ["Ali", "Sara", "John"];
console.log(names.includes("Sara")); // true
console.log(names.indexOf("John"));  // 2
console.log(names.join(" - "));      // Ali - Sara - John
```


Summary

- Arrays store lists of data in one variable.
- Use indexing to access or update items.
- Common methods like `push`, `pop`, `map`, `filter` make array handling easier.
- Looping is essential for displaying or processing lists.

Common Array Methods

a. `push()` – Add item at the end

```
colors.push("yellow");
```

b. `pop()` – Remove last item

```
colors.pop();
```

c. `shift()` – Remove first item

```
colors.shift();
```

d. `unshift()` – Add item at the beginning

```
colors.unshift("pink");
```

Looping Through Arrays

Using `for` loop:

```
let fruits = ["apple", "banana", "mango"];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

Using `for...of` loop:

```
for (let fruit of fruits) {
  console.log(fruit);
}
```


`map()` – Transform Array

Creates a new array by applying a function to each item.

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

`filter()` – Filter Items

Returns a new array of items that match a condition.

```
let scores = [30, 50, 90, 20];
let passed = scores.filter(score => score >= 50);
console.log(passed); // [50, 90]
```

Other Useful Methods

Method	Description
`includes()`	Checks if an item exists in the array
`indexOf()`	Finds index of an item
`slice()`	Extracts part of the array
`join()`	Joins array into a string

Example:

```
let names = ["Ali", "Sara", "John"];
console.log(names.includes("Sara")); // true
console.log(names.indexOf("John"));  // 2
console.log(names.join(" - "));      // Ali - Sara - John
```

Summary

- Arrays store lists of data in one variable.
- Use indexing to access or update items.
- Common methods like `push`, `pop`, `map`, `filter` make array handling easier.
- Looping is essential for displaying or processing lists.

Strings in JavaScript

What is a String?

A **string** is a sequence of characters used to represent text.
It can contain letters, numbers, symbols, or even be empty.

In JavaScript, strings are written inside **single quotes**, **double quotes**, or **backticks**.

```
let single = 'Hello';
let double = "World";
let template = `Hello World`;
```

All three are valid, but backticks (`` ` ``) are useful for string interpolation (covered later).

Declaring Strings

```
let message = "Welcome to JavaScript!";
let name = 'Salman';
```

String Length

You can check how many characters are in a string using the `.length` property.

```
let msg = "Hello";
console.log(msg.length); // 5
```

Multiline Strings

Using `\n` (newline character):

```
let text = "Line 1\nLine 2\nLine 3";
console.log(text);
```

Using template literals (backticks):

```
let multiline = `This is line 1
This is line 2
This is line 3`;
console.log(multiline);
```

String Indexing

Strings are indexed like arrays — the first character is at position `0`.

```
let word = "JavaScript";
console.log(word[0]); // "J"
console.log(word[4]); // "S"
```

Strings are Immutable

You **cannot change** a specific character in a string directly.

```
let greeting = "Hello";
greeting[0] = "Y"; // This will not work
console.log(greeting); // Still "Hello"
```

To change a string, you have to **create a new one**.

Concatenation (Combining Strings)

Using `+` operator:

```
let firstName = "Salman";
let lastName = "Khan";
let fullName = firstName + " " + lastName;
console.log(fullName); // "Salman Khan"
```

Summary

- Strings are sequences of characters.
- Use single, double, or backticks to define them.
- Strings are immutable.
- You can access characters using indices.
- Use `+` for concatenation or backticks for string interpolation.

Template Literals in JavaScript

Template literals are a way to work with strings in JavaScript that allow for easier multi-line strings and string interpolation.

```
let name = "Salman";
let message = `Hello, ${name}!`;
console.log(message); // "Hello, Salman!"
```

Template literals are enclosed in backticks (`` ` ``) and can contain placeholders for variables or expressions, which are wrapped in `${}`.

Multiline Strings with Template Literals

You can create multi-line strings without using escape characters:

```
let multiline = `This is line 1
This is line 2
This is line 3`;
console.log(multiline);
```

Summary

- Template literals use backticks (`` ` ``).
- They allow for string interpolation with `${}`.
- Multi-line strings can be created easily without escape characters.

String Methods in JavaScript

JavaScript provides many built-in methods to **work with strings** — to inspect, modify, or extract information from them.

All string methods return a **new string** or value. The original string remains unchanged.

---

1. `length`

Returns the number of characters in the string.

```
let text = "JavaScript";
console.log(text.length); // 10
```

---

2. `toUpperCase()` and `toLowerCase()`

Convert a string to upper or lower case.

```
let name = "Ali";
console.log(name.toUpperCase()); // "ALI"
console.log(name.toLowerCase()); // "ali"
```

---

3. `trim()`

Removes extra whitespace from both ends of a string.

```
let input = "   Hello World   ";
console.log(input.trim()); // "Hello World"
```

---

4. `includes()`

Checks if a string contains another string.

```
let msg = "Learn JavaScript";
console.log(msg.includes("Java")); // true
console.log(msg.includes("Python")); // false
```

---

5. `indexOf()` and `lastIndexOf()`

Returns the index of the first/last occurrence of a substring. Returns `-1` if not found.

```
let str = "banana";
console.log(str.indexOf("a")); // 1
console.log(str.lastIndexOf("a")); // 5
```

---

6. `startsWith()` and `endsWith()`

Check if a string starts or ends with a specific value.

```
let title = "Frontend Developer";
console.log(title.startsWith("Front")); // true
console.log(title.endsWith("Dev")); // false
```

---

7. `slice(start, end)`

Extracts part of a string. `end` is not included.

```
let word = "JavaScript";
console.log(word.slice(0, 4)); // "Java"
console.log(word.slice(4));    // "Script"
```

---

8. `substring(start, end)`

Similar to `slice()`, but cannot accept negative indexes.

```
let text = "Coding";
console.log(text.substring(0, 3)); // "Cod"
```

---

9. `replace(old, new)`

Replaces the first occurrence of a substring.

```
let msg = "I love Python";
console.log(msg.replace("Python", "JavaScript")); // "I love JavaScript"
```

Note: Only the first match is replaced. To replace all, use a regular expression with `/g`.

```
let msg = "apple apple apple";
console.log(msg.replace(/apple/g, "banana")); // "banana banana banana"
```

---

10. `split(separator)`

Splits a string into an array based on the given separator.

```
let data = "HTML,CSS,JavaScript";
let parts = data.split(",");
console.log(parts); // ["HTML", "CSS", "JavaScript"]
```

---

11. `charAt(index)`

Returns the character at a specific position.

```
let lang = "JavaScript";
console.log(lang.charAt(0)); // "J"
```

---

12. `repeat(count)`

Repeats a string multiple times.

```
let wow = "ha";
console.log(wow.repeat(3)); // "hahaha"
```

Summary

Method	Description
`.length`	String length
`.toUpperCase()`	Convert to uppercase

Method	Description
`.toLowerCase()`	Convert to lowercase
`.trim()`	Remove spaces from both ends
`.includes()`	Check if string contains a value
`.indexOf()`	Index of first occurrence
`.lastIndexOf()`	Index of last occurrence
`.startsWith()`	Check if string starts with a substring
`.endsWith()`	Check if string ends with a substring
`.slice()`	Extract part of a string
`.replace()`	Replace a part of the string
`.split()`	Convert string to array
`.charAt()`	Get character at specific position
`.repeat()`	Repeat a string multiple times

What is the DOM?

Introduction

The **DOM (Document Object Model)** is a programming interface provided by the browser that represents an HTML or XML document as a structured tree of objects. Each element, attribute, and piece of text in the HTML document becomes a **node** in the DOM tree.

This allows JavaScript to interact with the HTML and CSS of a web page — you can use JavaScript to read and modify the page's structure, content, and style dynamically.

Key Points:

- The DOM is **not** part of JavaScript, but it is provided by the browser's **Web APIs**.
- The browser turns HTML into a tree structure called the DOM.
- JavaScript can use this tree to:
    - Select elements
    - Modify content or style
    - Add or remove elements
    - Respond to user interactions

Example HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Example</title>
  </head>
  <body>
```

```
    <h1>Hello, DOM!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

How the DOM Sees It

The browser parses the above HTML and creates a tree-like structure:

- `document`
  - `html`
    - `head`
      - `title`
    - `body`
      - `h1`
      - `p`

Accessing the DOM in JavaScript

The `document` object is the entry point to the DOM in JavaScript.

```
console.log(document); // Logs the entire DOM tree
console.log(document.body); // Logs the <body> element
console.log(document.title); // Logs the content of <title>
```

Summary

- The DOM represents the page so that JavaScript can interact with it.
- You can access and modify HTML elements using JavaScript through the DOM.
- Understanding the DOM is essential for web development and dynamic page interactions.

Accessing the DOM

Introduction

To manipulate or interact with elements in an HTML document using JavaScript, you must first **access the DOM**. The browser provides two main global objects for this:

- `window`
- `document`

The `window` Object

The `window` object represents the browser window. It is the **global scope** in a browser environment. All global variables and functions become properties of the `window` object.

```
console.log(window); // Logs the global window object
console.log(window.innerWidth); // Gets the width of the browser window
```

You don't usually need to reference `window` explicitly, because it's the default context:

```
alert("Hello"); // Same as window.alert("Hello")
```

The `document` Object

The `document` object is a property of the `window` and serves as the main entry point to the web page's DOM.

```
console.log(document); // Logs the entire HTML document as a DOM tree
console.log(document.URL); // Gets the current page URL
```

Common Properties of `document`

Property	Description
`document.body`	Refers to the `<body>` element
`document.head`	Refers to the `<head>` element
`document.title`	Gets or sets the document's title
`document.URL`	Returns the full URL of the page

Example

```
<!DOCTYPE html>
<html>
<head>
  <title>Accessing the DOM</title>
</head>
<body>
  <h1>Hello World</h1>
  <script>
    console.log(document.title); // "Accessing the DOM"
    document.title = "New Title";
  </script>
</body>
</html>
```

Summary

- The `window` object is the global context and represents the browser window.
- The `document` object gives you access to the DOM structure of the HTML page.
- You use `document` to navigate and manipulate HTML elements via JavaScript.

Selecting Elements in JavaScript

When working with the DOM (Document Object Model), selecting elements is often the first step to manipulate them. JavaScript provides multiple methods to select HTML elements based on their ID, class, tag name, or CSS selector.

1. `getElementById`

Selects a single element by its unique `id`.

2. `getElementsByClassName`

Returns a live HTMLCollection of all elements with the specified class name.

3. `getElementsByTagName`

Returns a live HTMLCollection of all elements with the specified tag name (e.g., `div`, `p`, `h1`, etc.).

4. `querySelector`

Returns the first element that matches a specified CSS selector.

5. `querySelectorAll`

Returns a static NodeList of all elements that match a specified CSS selector.

These methods allow you to access and manipulate elements dynamically. We'll explore each of them in detail with examples next.

Changing `textContent`, `innerHTML`, `value`, and `style` in JavaScript

In this section, you'll learn how to dynamically change content and appearance on a webpage using JavaScript.

---

1. `textContent`

The `textContent` property sets or returns the text content of a node and its descendants. It ignores any HTML tags.

```
<p id="demo">Hello <strong>World</strong></p>
```

```
const para = document.getElementById("demo");
console.log(para.textContent); // Hello World
para.textContent = "New text content";
```

---

2. `innerHTML`

The `innerHTML` property sets or returns the HTML content of an element.

```
<p id="demo">Hello</p>
```

```
const para = document.getElementById("demo");
para.innerHTML = "<strong>Bold Text</strong>";
```

> Use `innerHTML` carefully to avoid XSS vulnerabilities when inserting user-provided data.

---

3. `value`

The `value` property is used to get or set the value of form elements such as `<input>`, `<textarea>`, and `<select>`.

```
<input type="text" id="username" value="John">
```

```
const input = document.getElementById("username");
console.log(input.value); // John
input.value = "Harry";
```

---

4. `style`

You can change the inline style of an element using the `style` property.

```
<p id="demo">This is a paragraph.</p>
```

```
const para = document.getElementById("demo");
para.style.color = "red";
para.style.fontSize = "20px";
para.style.backgroundColor = "#f0f0f0";
```

> Note: CSS property names with dashes (e.g., `background-color`) become camelCase (`backgroundColor`) in JavaScript.

Summary

Property	Purpose
`textContent`	Changes plain text (ignores HTML)
`innerHTML`	Changes or inserts HTML content
`value`	Reads or sets form input values
`style`	Dynamically changes inline CSS

Working with Attributes and Classes in JavaScript

JavaScript allows you to dynamically control HTML element attributes and CSS classes, enabling powerful, interactive web experiences.

---

1. Setting and Getting Attributes

Use `setAttribute()` to add or update an attribute and `getAttribute()` to retrieve its value.

Example:

```
<img id="myImage" src="logo.png">
```

```
const img = document.getElementById("myImage");

// Set a new attribute
img.setAttribute("alt", "Company Logo");

// Get an attribute value
console.log(img.getAttribute("src")); // "logo.png"

// Update an existing attribute
img.setAttribute("src", "new-logo.png");
```


---

2. Removing Attributes

Use `removeAttribute()` to completely remove an attribute from an element.

Example:

```
const img = document.getElementById("myImage");

// Remove the alt attribute
img.removeAttribute("alt");
```

---

3. Adding Classes

Use `classList.add()` to add one or more CSS classes to an element.

Example:

```
<div id="box"></div>
```

```
const box = document.getElementById("box");

box.classList.add("active");
box.classList.add("highlight", "shadow"); // Multiple classes
```

---

4. Removing Classes

Use `classList.remove()` to remove one or more classes.

```
box.classList.remove("highlight");
```

---

5. Toggling Classes

Use `classList.toggle()` to add a class if it doesn't exist, or remove it if it does.

```
box.classList.toggle("hidden");
```

---

6. Checking for a Class

Use `classList.contains()` to check whether an element has a specific class.

```
if (box.classList.contains("active")) {
  console.log("The box is active");
}
```

Summary

Task	Method
Set attribute	`element.setAttribute()`
Get attribute	`element.getAttribute()`
Remove attribute	`element.removeAttribute()`
Add class	`element.classList.add()`
Remove class	`element.classList.remove()`
Toggle class	`element.classList.toggle()`
Check class existence	`element.classList.contains()`

Creating, Appending, and Removing Elements in JavaScript

JavaScript allows you to dynamically create, insert, and remove elements from the DOM, enabling highly interactive web applications.

---

1. Creating Elements

Use `document.createElement()` to create a new HTML element in memory (not yet in the DOM).

Example:

```
const newDiv = document.createElement("div");
newDiv.textContent = "Hello, I was created dynamically!";
```

You can also set attributes or classes:

```
newDiv.setAttribute("id", "dynamicDiv");
newDiv.classList.add("box", "highlight");
```

---

2. Appending Elements

Use methods like `appendChild()` or `append()` to insert an element into the DOM.

Example:

```
<div id="container"></div>
```

```
const container = document.getElementById("container");
container.appendChild(newDiv); // Adds the newDiv as the last child
```

Or using `append()` which allows multiple nodes or strings:

```
container.append("Another text node", document.createElement("hr"));
```

---

3. Prepending Elements

Use `prepend()` to insert an element at the beginning of the parent.

```
const heading = document.createElement("h2");
heading.textContent = "Welcome!";
container.prepend(heading);
```

---

4. Removing Elements

To remove an element, you can use:

a) `removeChild()` (from parent)

```
container.removeChild(newDiv);
```

b) `element.remove()` (directly on the element)

```
heading.remove(); // Removes the heading element directly
```

> Note: `element.remove()` is widely supported but not in very old browsers.

Summary

Task	Method
Create element	`document.createElement()`
Set text content	`element.textContent`
Add attribute or class	`setAttribute()`, `classList.add()`
Append to parent	`appendChild()`, `append()`
Prepend to parent	`prepend()`
Remove from parent	`parent.removeChild(child)`
Remove element directly	`element.remove()`

Introduction to Events in JavaScript

In web development, **events** are actions or occurrences that happen in the browser, often triggered by the user. JavaScript allows us to **listen for these events** and respond with custom behavior.

Common examples of events include:

- Clicking a button
- Pressing a key
- Submitting a form
- Hovering over an element
- Scrolling the page

Why Events Matter

Events make your web pages interactive. They allow users to engage with your content dynamically, rather than just reading static information.

Types of Common Events

Here are some commonly used event types in JavaScript:

1. `click`

Triggered when an element is clicked.

```
const button = document.querySelector('button');
button.addEventListener('click', () => {
  alert('Button clicked!');
});
```

2. `keyup`

Fires when the user releases a key on the keyboard.

```
const input = document.querySelector('input');
input.addEventListener('keyup', (event) => {
  console.log(`You typed: ${event.key}`);
});
```

3. `submit`

Occurs when a form is submitted.

```
const form = document.querySelector('form');
form.addEventListener('submit', (event) => {
  event.preventDefault(); // Prevents the page from reloading
  console.log('Form submitted');
});
```

4. `mouseover` and `mouseout`

Fired when the mouse enters or leaves an element.

```
const box = document.querySelector('.box');
box.addEventListener('mouseover', () => {
  box.style.backgroundColor = 'lightblue';
});
box.addEventListener('mouseout', () => {
  box.style.backgroundColor = '';
});
```


Attaching Event Listeners

To handle events, use the `addEventListener` method:

```
element.addEventListener('eventName', callbackFunction);
```

Example:

```
document.querySelector('h1').addEventListener('click', () => {
  console.log('Heading clicked!');
});
```

Summary

- Events allow JavaScript to react to user interactions.
- Use `addEventListener` to attach event handlers.
- Prevent default behavior using `event.preventDefault()` if needed (e.g., in forms).
- Always test your event listeners to ensure the intended behavior works across different browsers.

Attaching Event Listeners in JavaScript

In JavaScript, you can use the `addEventListener()` method to handle user interactions like clicks, typing, mouse movements, etc.

Syntax:

```
element.addEventListener(event, callback);
```

Common Events:

- `click` – when an element is clicked
- `mouseover` – when the mouse hovers over an element
- `keydown` – when a key is pressed
- `submit` – when a form is submitted

Examples:

```
<button id="clickBtn">Click Me</button>
<input id="inputBox" placeholder="Type something" />
<form id="myForm">
  <input type="text" required />
  <button type="submit">Submit</button>
</form>

<script>
  // Click event
  document.getElementById("clickBtn").addEventListener("click", () => {
    alert("Button was clicked!");
  });

  // Keydown event
  document.getElementById("inputBox").addEventListener("keydown", (e) => {
```

```
    console.log("Key pressed:", e.key);
  });

  // Submit event
  document.getElementById("myForm").addEventListener("submit", (e) => {
    e.preventDefault();
    alert("Form submitted!");
  });
</script>
```

Event Bubbling and Delegation in JavaScript

Understanding **event bubbling** and **event delegation** is essential for writing clean, efficient event-driven code in JavaScript.

What is Event Bubbling?

When an event occurs on a DOM element, it **bubbles up** through its ancestors. This means the event is first captured and handled by the **target element**, and then propagated upward to its parent, grandparent, and so on.

Example:

```
<div id="parent">
  <button id="child">Click Me</button>
</div>
```

```
document.getElementById('child').addEventListener('click', () => {
  console.log('Child clicked');
});

document.getElementById('parent').addEventListener('click', () => {
  console.log('Parent clicked');
});
```


Output when button is clicked:

```
Child clicked
Parent clicked
```

This shows that the event bubbles from the child to the parent.

Stopping Event Bubbling

To stop the event from bubbling up the DOM tree, use:

```
event.stopPropagation();
```

```
button.addEventListener('click', (event) => {
  event.stopPropagation();
  console.log('This won't bubble up');
});
```

What is Event Delegation?

Event delegation is a technique where a single event listener is added to a **common parent** instead of individual child elements. This works because of event bubbling.

Why Use Event Delegation?

- Improved performance with many child elements
- Simplified code
- Useful for dynamic elements added after page load

Example:

```
<ul id="menu">
  <li>Home</li>
  <li>About</li>
  <li>Contact</li>
</ul>
```

Instead of attaching a click listener to every `li`, delegate it to the `ul`:

```
document.getElementById('menu').addEventListener('click', (event) => {
  if (event.target.tagName === 'LI') {
```

```
    console.log(`You clicked on ${event.target.textContent}`);
  }
});
```

This even works if new `<li>` elements are added later using JavaScript.

Summary

- **Event bubbling**: Events move up the DOM tree from child to parent.
- Use `event.stopPropagation()` to prevent bubbling.
- **Event delegation**: Handle events at a parent level for better performance and maintainability.

Preventing Default Behavior in JavaScript

Many HTML elements have **default behaviors**. For example:

- Clicking a link navigates to a new URL
- Submitting a form reloads the page
- Pressing certain keys triggers built-in browser actions

In JavaScript, you can **override** these default behaviors using `event.preventDefault()`.

Syntax

```
element.addEventListener('eventType', (event) => {
  event.preventDefault();
});
```

This method tells the browser **not to perform** the default action associated with the event.

Common Use Cases

1. Preventing Form Submission

By default, submitting a form reloads the page. You can prevent this to handle form data using JavaScript:

```
<form id="myForm">
  <input type="text" />
```

```
  <button type="submit">Submit</button>
</form>
```

```
const form = document.getElementById('myForm');

form.addEventListener('submit', (event) => {
  event.preventDefault();
  console.log('Form submission prevented');
});
```

2. Preventing Link Navigation

You may want to handle navigation manually using JavaScript:

```
<a href="https://codewithharry.com" id="myLink">Go to CodeWithHarry</a>
```

```
const link = document.getElementById('myLink');

link.addEventListener('click', (event) => {
  event.preventDefault();
  console.log('Navigation prevented');
});
```

3. Preventing Checkbox Default Toggle

In special cases, you might want to keep a checkbox from changing state:

```
const checkbox = document.querySelector('input[type="checkbox"]');

checkbox.addEventListener('click', (event) => {
  event.preventDefault();
  console.log('Checkbox toggle prevented');
});
```

When Not to Use `preventDefault()`

Avoid using `preventDefault()` unless necessary. Overusing it can confuse users and break expected behavior (like tabbing between fields or pressing Enter to submit).

Summary

- Use `event.preventDefault()` to cancel default browser actions.
- Common in form handling, link overrides, or input control.
- Always test to ensure usability and accessibility are not harmed.

Working with `localStorage` in JavaScript

The `localStorage` API allows you to store key-value data **in the browser**, with no expiration date. This data **persists across page reloads and browser sessions**, unless manually cleared.

Key Features

- Stores data as **strings**
- Synchronous API (blocking)
- Maximum storage capacity: around 5MB (varies by browser)
- Data is domain-specific

Basic Syntax

```
// Set item
localStorage.setItem('key', 'value');

// Get item
const value = localStorage.getItem('key');

// Remove item
localStorage.removeItem('key');

// Clear all items
localStorage.clear();
```

Example: Storing and Retrieving Data

```
localStorage.setItem('username', 'haris');

const user = localStorage.getItem('username');
console.log(user); // Output: haris
```

Storing Objects

Since `localStorage` only stores strings, you must convert objects using `JSON.stringify()` and retrieve them using `JSON.parse()`:

```
const user = {
  name: 'Harry',
  age: 25
};

// Store the object
localStorage.setItem('user', JSON.stringify(user));

// Retrieve the object
const storedUser = JSON.parse(localStorage.getItem('user'));
console.log(storedUser.name); // Output: Harry
```


Checking if Key Exists

```
if (localStorage.getItem('theme')) {
  console.log('Theme is set');
}
```

Use Cases

- Saving user preferences (e.g., theme, language)

- Caching small amounts of data
- Keeping users logged in (store tokens)
- Temporarily saving form data

Limitations

- Synchronous (can block main thread)
- Storage limit (~5MB)
- No automatic expiration (unlike `sessionStorage` or cookies)
- Not secure for sensitive data (easily inspectable via DevTools)

Summary

- `localStorage` provides simple, persistent browser storage.
- Use `setItem`, `getItem`, `removeItem`, and `clear` methods.
- Always stringify objects before storing.
- Avoid storing sensitive data.

Parsing JSON in JavaScript

JSON (JavaScript Object Notation) is a lightweight data format used for exchanging data between a server and a client. JavaScript provides built-in methods to **parse** and **stringify** JSON.

What is JSON?

JSON is a **string representation** of data structured in key-value pairs.

Example JSON string:

```
{
  "name": "Harry",
  "age": 25,
  "skills": ["JavaScript", "Python"]
}
```

This looks like a JavaScript object, but it's actually a string.

Parsing JSON with `JSON.parse()`

To convert a JSON string into a JavaScript object, use `JSON.parse()`:

```
const jsonString = '{"name":"Harry","age":25}';

const user = JSON.parse(jsonString);
console.log(user.name); // Output: Harry
```

Note:

- The string **must be valid JSON**, otherwise `JSON.parse()` will throw an error.
- All keys must be enclosed in double quotes (`"`).

Stringifying JavaScript Objects with `JSON.stringify()`

To convert a JavaScript object into a JSON string, use `JSON.stringify()`:

```
const user = {
  name: 'Harry',
  age: 25
};

const json = JSON.stringify(user);
console.log(json); // Output: {"name":"Harry","age":25}
```

Common Use Case: `localStorage`

Since `localStorage` can only store strings, you often use JSON methods to store and retrieve objects:

```
// Store object
localStorage.setItem('user', JSON.stringify(user));

// Retrieve and parse object
const storedUser = JSON.parse(localStorage.getItem('user'));
```

Handling Errors

Wrap parsing in a `try...catch` block to handle invalid JSON:

```
try {
  const data = JSON.parse(badJSONString);
} catch (error) {
  console.error('Invalid JSON:', error.message);
}
```

Summary

- Use `JSON.parse()` to convert a JSON string into a JavaScript object.
- Use `JSON.stringify()` to convert a JavaScript object into a JSON string.
- These methods are commonly used with `localStorage`, APIs, and data exchange.

Error Handling in JavaScript

Errors are inevitable in any application. JavaScript provides robust tools to **detect**, **handle**, and **respond** to runtime errors, allowing your code to fail gracefully.

Types of Errors

- **Syntax Errors**: Mistakes in the code structure (e.g., missing brackets)
- **Runtime Errors**: Occur during execution (e.g., accessing undefined variables)
- **Logical Errors**: Code runs but doesn't behave as intended

---

The `try...catch` Statement

Use `try...catch` to handle errors without crashing the entire script.

Syntax:

```
try {
  // Code that may throw an error
} catch (error) {
  // Handle the error
}
```

Example:

```
try {
  let result = 10 / x; // x is not defined
} catch (error) {
  console.log('An error occurred:', error.message);
}
```

The `finally` Block

The `finally` block is optional and always runs, **whether an error occurred or not**.

```
try {
  // Risky code
} catch (error) {
  // Handle error
} finally {
  // Always runs
  console.log('Cleanup complete');
}
```

Throwing Custom Errors

You can manually throw errors using the `throw` statement.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Cannot divide by zero');
  }
  return a / b;
}

try {
  divide(5, 0);
} catch (error) {
  console.error(error.message); // Output: Cannot divide by zero
}
```

Catching Specific Error Types

The `catch` block receives an `Error` object with useful properties:

```
try {
  JSON.parse('invalid JSON');
} catch (error) {
  console.log(error.name);    // SyntaxError
  console.log(error.message); // Unexpected token i in JSON
}
```

Best Practices

- Always handle expected errors (e.g., API failures, user input)
- Avoid swallowing errors silently — log them
- Use specific error messages for debugging
- Don't use try-catch for control flow in performance-critical code

---

Summary

- Use `try...catch` to prevent your app from crashing.
- Add `finally` for cleanup logic.
- Use `throw` to raise custom errors.
- Proper error handling makes applications more stable and user-friendly.

Timers and Intervals in JavaScript

JavaScript provides built-in functions to schedule code execution after a delay or at repeated intervals. These are useful for animations, delayed actions, auto-saving, and more.

`setTimeout()`: Run Code After a Delay

The `setTimeout()` function executes code **once** after a specified delay (in milliseconds).

Syntax

```
setTimeout(callback, delay);
```

Example

```
setTimeout(() => {
  console.log('This runs after 2 seconds');
}, 2000);
```

Canceling a Timeout

Use `clearTimeout()` to stop a scheduled timeout.

```
const timeoutId = setTimeout(() => {
  console.log('Will not run');
}, 3000);

clearTimeout(timeoutId);
```

`setInterval()` : Run Code Repeatedly

The `setInterval()` function executes code **repeatedly** at a specified interval (in milliseconds).

Syntax

```
setInterval(callback, interval);
```

Example

```
setInterval(() => {
  console.log('Runs every second');
}, 1000);
```

Canceling an Interval

Use `clearInterval()` to stop the repetition.

```
const intervalId = setInterval(() => {
  console.log('Repeating...');
}, 1000);

setTimeout(() => {
  clearInterval(intervalId);
  console.log('Interval stopped');
}, 5000);
```


Use Cases

- Delayed popups or notifications
- Auto-refreshing data (like a clock or stock ticker)

- Game loops and animations
- Debouncing or throttling user input

Important Notes

- Delays are **not guaranteed** to be exact — they depend on the event loop and execution stack.
- Avoid overly frequent intervals (`<10ms`) as it may block the main thread.

Summary

Function	Purpose	Cancel with
`setTimeout()`	Run code once after delay	`clearTimeout()`
`setInterval()`	Run code repeatedly with delay	`clearInterval()`

Introduction to Node.js

Why Do We Need a Backend?

In web development, **frontend** and **backend** are two key parts of an application.

- The **frontend** is what users see and interact with—like buttons, forms, and text on a webpage. It runs in the browser.
- The **backend** is like the behind-the-scenes part. It handles things like:
    - Storing and retrieving data from databases
    - Authenticating users (login/signup)
    - Processing business logic
    - Securing sensitive operations
    - Handling API requests from the frontend

Without a backend, a website can't store user data, communicate with a database, or perform secure tasks. The frontend would just be a static page with limited interactivity.

Example

Imagine a to-do list app:

- The **frontend** displays the tasks and lets the user add or remove them.
- The **backend** saves these tasks to a database, so they're still there when the user comes back later.

Introduction to Node.js

Node.js is a runtime environment that lets you run JavaScript on the **server**, not just in the browser.

- Normally, JavaScript runs only in the browser (client-side).
- Node.js allows you to run JavaScript on your computer or server (server-side).

With Node.js, you can build the backend of your application using JavaScript—the same language you use for the frontend. This makes development faster and easier, especially for beginners.

Key Features of Node.js

- Built on Chrome's V8 JavaScript engine
- Handles many requests at once using non-blocking (asynchronous) code
- Has a large ecosystem of libraries (called npm)
- Great for building APIs, real-time apps (like chat apps), and full-stack JavaScript projects

---

Server-side vs. Client-side JavaScript

Feature	Client-side (Browser)	Server-side (Node.js)
Runs on	User's browser	Web server
Use case	Displaying content, UI interaction	Storing data, handling logic
Access to system resources	Limited (sandboxed)	Full access (file system, network)
Performance	Depends on user's device	Depends on server
Examples	Form validation, animations	Database queries, user authentication

Why the Difference Matters

- Running JavaScript in the **client** is good for creating interactive webpages, but it's limited and not secure for sensitive operations.
- Running JavaScript in the **server** (with Node.js) allows you to perform secure tasks and manage application logic centrally.

Summary

- A **backend** is essential for dynamic websites that need to store data, handle users, or connect to databases.
- **Node.js** lets you write backend code using JavaScript, making full-stack development more accessible.
- **Client-side** JavaScript is for user interaction; **server-side** JavaScript (via Node.js) handles the logic and data behind the scenes.

Installing Node.js and npm

To run JavaScript on the server and build backend applications with Node.js, you first need to install **Node.js**. When you install Node.js, **npm** (Node Package Manager) is installed automatically.

Step 1: Download Node.js

1. Go to the official Node.js website: https://nodejs.org
2. You'll see two versions:
    1. **LTS (Long-Term Support):** Recommended for most users. It's stable and reliable.
    2. **Current:** Has the latest features but may not be as stable.
3. Download the **LTS** version for your operating system (Windows, macOS, or Linux).

Step 2: Install Node.js On Windows and macOS

1. Run the installer you downloaded.
2. Follow the setup instructions:
    1. Accept the license agreement.
    2. Choose the default options.
3. After installation, Node.js and npm will be available globally on your system.

Step 3: Verify Installation

After installation, open a terminal (Command Prompt, Terminal, or shell) and check the versions:

```
node -v
```

This prints the installed Node.js version.

```
npm -v
```

This prints the installed npm version.

If you see version numbers for both, the installation was successful.

What is npm?

npm (Node Package Manager) is a tool that comes with Node.js. It allows you to:

- Install open-source packages and libraries (called “modules”)
- Manage project dependencies
- Run scripts for building, testing, and starting your app

You’ll use npm often when building projects with Node.js.

Summary

- Install Node.js from https://nodejs.org
- Node.js includes npm, which is used to manage packages
- Use your terminal to verify the installation with `node -v` and `npm -v`

You’re now ready to start building backend applications with JavaScript and Node.js. Let me know if you want a guide on starting your first Node.js project.

Using npm Packages in Node.js (with Express)

In this guide, we'll install and use an npm package in a Node.js project. We'll use **Express**, a popular web framework for Node.js. Don't worry about the details of Express for now—we'll cover that later. The goal here is simply to show how to install and use packages with npm.

Step 1: Initialize a New Project

Create a new project folder and initialize it:

```
mkdir my-npm-app
cd my-npm-app
npm init -y
```

This creates a `package.json` file that keeps track of your project's dependencies.

Step 2: Install Express

Use `npm install` to install the Express package:

```
npm install express
```

After installing, you'll see a `node_modules` folder and a `package-lock.json` file created. The `package.json` file will also list Express under `dependencies`.

Step 3: Create a Simple Server with Express

Create a new file called `app.js`:

```
touch app.js
```

Add the following code to `app.js`:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

const port = 3000;
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

We'll explore what this code does later. For now, it just starts a basic server.

Step 4: Run Your App

To start the app, run:

```
node app.js
```

Visit http://localhost:3000 in your browser. You should see:

```
Hello from Express!
```

Step 5: Use Node.js Watch Mode

Node.js now supports **watch mode** natively (from version 18 and above)

To run your app in watch mode:

```
node --watch app.js
```

This means Node.js will automatically restart whenever you save changes to `app.js` or other imported files.

> Note: Watch mode works best in modern versions of Node.js. You can check your Node.js version using:

```
node -v
```

Step 6: Uninstall a Package

If you ever want to remove a package (like Express), use:

```
npm uninstall express
```

This removes it from `node_modules` and updates your `package.json`.

Summary

- Initialized a Node.js project using `npm init -y`
- Installed an npm package (`express`)
- Used it in a basic server file (`app.js`)
- Ran the app using both normal and watch mode (`node` and `node --watch`)
- Learned how to uninstall packages with `npm uninstall`

Creating a Simple Node.js Application

Now that Node.js and npm are installed, you can create your first Node.js application. This guide walks you through building a basic “Hello, World” server.

Step 1: Create a New Project Folder

Open your terminal and create a new folder for your project:

```
mkdir my-node-app
cd my-node-app
```

Step 2: Initialize the Project

Run the following command to create a `package.json` file, which keeps track of your project settings and dependencies:

```
npm init -y
```

This generates a basic `package.json` file with default values.

Step 3: Create the Application File

Create a new file named `app.js` (or any name you prefer):

```
touch app.js
```

Open `app.js` in your code editor and add the following code:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

This creates a basic HTTP server that responds with “Hello, World!” to every request.

Step 4: Run the Application

In your terminal, start the server:

```
node app.js
```

If everything is set up correctly, you should see this message in the terminal:

```
Server running at http://localhost:3000/
```

Open your web browser and visit http://localhost:3000. You should see “Hello, World!” displayed.

Summary

- You created a new folder and initialized a Node.js project.
- You wrote a simple server using Node's built-in `http` module.
- You started the server and accessed it in your browser.

Node.js Modules

Modules in Node.js are reusable pieces of code that help organize programs into separate files and components.

Types of Modules

1. **Core Modules**
   Built into Node.js, no need to install.
   Example: `fs`, `http`, `path`

```
const fs = require('fs');
const data = fs.readFileSync('file.txt', 'utf8');
console.log(data);
```

2. **Local Modules** Custom modules created in your project.

   Example: `math.js`

```
// math.js
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

```
// app.js
const math = require('./math');
console.log(math.add(5, 3));
```

3. **Third-party Modules** Installed via npm (Node Package Manager). Example: `express`, `lodash`

```
npm install express
```

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(3000);
```

Exporting and Importing

- Use `module.exports` to export
- Use `require()` to import

Summary

Node.js modules keep code organized, reusable, and maintainable. Whether you're using built-in modules, custom files, or third-party packages, modules are fundamental to every Node.js project.

ES6 Modules vs CommonJS in Node.js

Node.js supports two main module systems:

1. **CommonJS (CJS)** – Default in older Node.js versions
2. **ES6 Modules (ESM)** – Modern standard, used in frontend and supported natively in recent Node.js versions

---

1. CommonJS

- **File Extension:** `.js`
- **Import Syntax:** `require()`
- **Export Syntax:** `module.exports` or `exports`

Example:

```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = { add };
```

```
// app.js
const math = require('./math');
console.log(math.add(2, 3));
```


---

2. ES6 Modules

- **File Extension:** `.mjs` or `.js` with `"type": "module"` in `package.json`

- **Import Syntax:** `import`
- **Export Syntax:** `export` / `export default`

Example:

```
// math.mjs
export function add(a, b) {
  return a + b;
}
```

```
// app.mjs
import { add } from './math.mjs';
console.log(add(2, 3));
```

Key Differences

Feature	CommonJS (CJS)	ES6 Modules (ESM)
Syntax	`require`, `module.exports`	`import`, `export`
File extension	`.js`	`.mjs` or `.js` with `"type": "module"`
Loading style	Synchronous	Asynchronous
Support	Default in Node.js	Modern Node.js (14+)
Top-level await	Not supported	Supported
Used in	Node.js traditionally	Both frontend and backend

When to Use What

- Use **CommonJS** if you're working on older Node.js projects or need compatibility with legacy packages.
- Use **ES6 Modules** for new projects to align with modern JavaScript standards and browser support.

Mixing CJS and ESM

Mixing module types can be tricky:

- You can't use `require()` to import an ES6 module.
- You can't use `import` to load a CommonJS module unless it's default-exported.

Conclusion

Both module systems help organize and reuse code, but **ES6 modules are the future**, offering cleaner syntax and better interoperability across frontend and backend.

Understanding the Node.js Wrapper Function and Special Variables

In Node.js, every JavaScript file is wrapped inside a special function before it is executed. This allows each file to have its own private scope, preventing variables from leaking into the global scope.

The Wrapper Function

Node.js wraps your code like this internally:

```
(function(exports, require, module, __filename, __dirname) {
  // Your entire file code lives here
});
```

This is known as the **Module Wrapper Function**. Because of this, your Node.js file gets access to the following special variables:

Special Variables in Node.js

`__filename`

- Returns the absolute path of the current file.

```
console.log(__filename);
// Example: /Users/haris/project/app.js
```

`__dirname`

- Returns the absolute path of the directory that contains the current file.

```
console.log(__dirname);
// Example: /Users/haris/project
```

`exports` and `module.exports`

- Used to export variables or functions from a module.

`require`

- Used to import modules.

`module`

- Represents the current module and contains metadata about it.

Why This Matters

This wrapper allows each file to be treated as a separate module. It ensures:

- Code isolation (no global scope pollution)
- Access to helpful variables like `__dirname` and `__filename`
- A consistent module system using CommonJS

Example

```
// test.js
console.log('Filename:', __filename);
console.log('Directory:', __dirname);
```

When you run this with `node test.js`, it will print the full path of the file and its directory.

Understanding the wrapper function and special variables is key to mastering how Node.js modules work internally.

Asynchronous JavaScript

JavaScript runs code **one line at a time** — it's **single-threaded**. This means only one task can happen at any moment. Still, JavaScript can do things like wait for a timer or handle user clicks without stopping everything else.

This is because JavaScript uses **asynchronous behavior** for certain tasks.

Synchronous Code

This is how JavaScript normally works — **top to bottom**.

```
console.log("A");
console.log("B");
console.log("C");

// Output:
// A
// B
// C
```

Each line waits for the previous one to finish. That's **synchronous** execution.

Asynchronous Code

Here's where it gets interesting:

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

setTimeout(() => {
```

```
  console.log("B");
}, 1000);

console.log("C");

// Output:
// A
// C
// B (after about 1 second)
```

Even though `setTimeout` is written before `"C"`, it runs **after**. Why?

What Happens Behind the Scenes

When JavaScript sees `setTimeout`, it:

1. Sends the task (along with its delay) to the **browser** (or **Node.js**, if you're running it there).
2. Continues running the rest of the code — it doesn't wait.
3. After the timer finishes, the function you passed to `setTimeout` is **sent back** to JavaScript to be run.
4. But it will only run **after** the current code is done.

This system of handling things is managed by the **event loop**.

The Event Loop (In Simple Words)

The **event loop** is a mechanism that:

- Keeps checking if JavaScript is done running your current code.
- If yes, it picks up any pending tasks (like the `setTimeout` callback) and runs them.

Real-Life Analogy

Imagine you're cooking:

- You put rice on the stove and set a timer.
- You don't just stand there — you cut vegetables.
- When the timer rings, you go back to check the rice.

Similarly, JavaScript **schedules** tasks like timers to run **later**, and continues with the rest of the code.

Another Example

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

setTimeout(() => {
  console.log("Waiting over");
}, 2000);

console.log("End");

// Output:
// Start
// End
// Waiting over (after ~2 seconds)
```

Even with 2 seconds delay, `"End"` appears **immediately** after `"Start"` — that's the power of async.

Summary

- JavaScript is single-threaded: one thing at a time.
- Functions like `setTimeout` are **asynchronous**.
- These async tasks are handled by the browser/Node and returned later.

- The **event loop** checks when JavaScript is free to run those returned tasks.

Understanding this helps you write programs that don’t get “stuck” waiting and can handle things like user input, network requests, and timers smoothly.

Introduction to JavaScript Promises

A **Promise** in JavaScript is a way to handle **asynchronous operations**. It lets you write code that runs **after something finishes**, without getting stuck in messy nested callbacks.

Think of a Promise like a **placeholder for a value** that will be available in the future.

Why Do We Need Promises?

With callbacks, things can quickly become hard to read and maintain, especially when we have to wait for multiple things.

Example of **callback hell**:

```
doTask1(function (result1) {
  doTask2(result1, function (result2) {
    doTask3(result2, function (result3) {
      console.log("All tasks done");
    });
  });
});
```

This kind of nested code becomes difficult to manage. **Promises solve this** by allowing a cleaner, more readable structure.

Basic Promise Syntax

```
const promise = new Promise(function (resolve, reject) {
  // Do some work...
```

```
  // Call resolve(result) if successful
  // Call reject(error) if there's an error
});
```

Once a Promise is created, we can handle its result using `.then()` and `.catch()`:

```
promise
  .then(function (result) {
    // This runs if the promise was resolved
  })
  .catch(function (error) {
    // This runs if the promise was rejected
  });
```

A Simple Example: Fake Async Task

Let's create a Promise that waits for 2 seconds and then resolves.

```
function waitTwoSeconds() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("Done waiting");
    }, 2000);
  });
}

console.log("Start");

waitTwoSeconds()
  .then(function (message) {
    console.log(message); // "Done waiting"
  })
  .catch(function (error) {
    console.log("Something went wrong");
  });
```

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

Output:

```
Start
End
Done waiting
```

Even though the Promise is written earlier, it runs **after** the rest of the synchronous code — just like with callbacks.

Solving Callback Hell with Promises

You can chain multiple `.then()` calls instead of nesting:

```
doTask1()
  .then(function (result1) {
    return doTask2(result1);
  })
  .then(function (result2) {
    return doTask3(result2);
  })
  .then(function (result3) {
    console.log("All tasks done");
  })
  .catch(function (error) {
    console.log("Something failed", error);
  });
```


This is **much cleaner** than deeply nested callbacks.

Summary

- A **Promise** is an object representing a value that may be available now, later, or never.
- It has three states:
  - **Pending**: not yet finished
  - **Resolved**: finished successfully
  - **Rejected**: finished with an error
- Use `.then()` to handle success and `.catch()` to handle errors.
- Promises help write **cleaner async code**, especially when chaining tasks.

JavaScript `async` and `await`

Writing asynchronous code using `.then()` and `.catch()` works well, but as your code grows, it can still feel a bit hard to follow.

JavaScript gives us a cleaner way to work with Promises:
`async` and `await`

What is `async` ?

If you put the keyword `async` before a function, it **automatically returns a Promise**.

```
async function greet() {
  return "Hello";
}

greet().then(function (message) {
  console.log(message); // "Hello"
});
```

Even though we just returned a string, `greet()` becomes a Promise.

What is `await` ?

The `await` keyword is used **inside an `async` function**. It tells JavaScript to **wait for the Promise to resolve**, then continue.

Basic Example

Let's simulate a delay using `setTimeout` wrapped in a Promise:

```
function waitTwoSeconds() {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve("Waited for 2 seconds");
    }, 2000);
  });
}

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

  const result = await waitTwoSeconds();
  console.log(result);

  console.log("End");
}

runTask();
```

Output:

```
Start
Waited for 2 seconds
End
```

Why Use `async` and `await`?

Let's compare the same logic using `.then()`:

```
waitTwoSeconds().then(function (result) {
  console.log(result);
});
```

It works, but once you have **multiple async operations**, the `.then()` style gets harder to follow.

With `await`, your code looks more like regular, synchronous code — even though it's asynchronous.

Handling Errors with `try...catch`

If a Promise rejects, you can catch the error using `try...catch`.

```
function fakeTask(fail) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      if (fail) {
        reject("Something went wrong");
      } else {
        resolve("Task completed");
      }
    }, 1000);
  });
}

async function run() {
  try {
    const result = await fakeTask(false);
    console.log(result);
  } catch (error) {
    console.log("Caught error:", error);
  }
}
```

```
run();
```

Summary

- `async` makes a function return a Promise.
- `await` pauses inside the function until the Promise is done.
- You can use `try...catch` to handle errors just like synchronous code.
- `async` / `await` makes asynchronous code easier to **read** and **write**.

JavaScript `async` and `await`

Writing asynchronous code using `.then()` and `.catch()` works well, but as your code grows, it can still feel a bit hard to follow.

JavaScript gives us a cleaner way to work with Promises:
`async` and `await`

What is `async` ?

If you put the keyword `async` before a function, it **automatically returns a Promise**.

```
async function greet() {
  return "Hello";
}

greet().then(function (message) {
  console.log(message); // "Hello"
});
```

Even though we just returned a string, `greet()` becomes a Promise.

What is `await` ?

The `await` keyword is used **inside an `async` function**. It tells JavaScript to **wait for the Promise to resolve**, then continue.

Basic Example

Let's simulate a delay using `setTimeout` wrapped in a Promise:

```
function waitTwoSeconds() {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve("Waited for 2 seconds");
    }, 2000);
  });
}

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

  const result = await waitTwoSeconds();
  console.log(result);

  console.log("End");
}

runTask();
```

Output:

```
Start
Waited for 2 seconds
End
```

Why Use `async` and `await`?

Let's compare the same logic using `.then()`:

```
waitTwoSeconds().then(function (result) {
  console.log(result);
});
```

It works, but once you have **multiple async operations**, the `.then()` style gets harder to follow.

With `await`, your code looks more like regular, synchronous code — even though it's asynchronous.

Handling Errors with `try...catch`

If a Promise rejects, you can catch the error using `try...catch`.

```
function fakeTask(fail) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      if (fail) {
        reject("Something went wrong");
      } else {
        resolve("Task completed");
      }
    }, 1000);
  });
}

async function run() {
  try {
    const result = await fakeTask(false);
    console.log(result);
  } catch (error) {
    console.log("Caught error:", error);
  }
}
```

```
run();
```

Summary

- `async` makes a function return a Promise.
- `await` pauses inside the function until the Promise is done.
- You can use `try...catch` to handle errors just like synchronous code.
- `async` / `await` makes asynchronous code easier to **read** and **write**.

JavaScript Callbacks

A **callback** is simply a **function passed as an argument to another function**, to be **called later**.

This might sound confusing at first, but once you see it in action, it becomes very easy to understand.

Why Do We Need Callbacks?

Sometimes, you don't want a function to run immediately.
Instead, you want it to **run later**, **when something else happens** — for example:

- When a timer finishes
- When a user clicks a button
- When some data is ready

Callbacks help us do that.

Basic Example of a Callback

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

function processUser(callback) {
  const userName = "Harry";
  callback(userName);
}
```

```
processUser(greet);
```

What's Happening Here?

- `greet` is a function that prints a greeting.
- `processUser` is another function that receives a function (callback) and **calls it later**.
- We pass `greet` as an argument to `processUser`.

So `processUser(greet)` ends up calling: `greet("Harry")`

Callbacks in Asynchronous Code

Callbacks are often used with async functions like `setTimeout`.

```
function showMessage() {
  console.log("This runs after 2 seconds");
}

setTimeout(showMessage, 2000);

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

Output:

```
This runs first
This runs after 2 seconds
```

Even though `showMessage` is written before the timer, it **runs later** — after 2 seconds. That's because we passed it as a callback to `setTimeout`.

Writing Inline Callback Functions

Instead of defining the function first, we can also write it directly:

```
setTimeout(function () {
  console.log("Hello after 1 second");
}, 1000);
```

This is still a callback — just written directly where it's used.

Another Example with User Input (Browser Only)

If you're in a browser and use something like this:

```
document.getElementById("btn").addEventListener("click", function () {
  console.log("Button clicked");
});
```

The second argument to `addEventListener` is a callback. It runs **only when the user clicks** the button.

Summary

- A **callback** is just a function passed to another function.
- It can be used to run code **later**.
- Callbacks are commonly used in:
  - `setTimeout`
  - Event listeners
  - Many asynchronous operations

Callbacks are the foundation for working with asynchronous JavaScript. Once you're comfortable with them, you're ready to learn more advanced things like **Promises** and `async/await`.

Introduction to Express.js

What is Express.js?

Express.js is a fast, unopinionated, and minimalist web framework for Node.js. It simplifies the process of building web servers and APIs using JavaScript.

Instead of writing raw HTTP code in Node.js, Express gives us a higher-level set of tools to build robust backend applications quickly and efficiently.

Why Use Express.js Over Node.js Core Modules?

Raw Node.js:

- You need to manually parse requests and handle routes.
- No built-in support for things like middleware, form data, sessions, or routing.

Express.js:

- Built-in routing support.
- Middleware support for handling requests and responses.
- Easily serve static files.
- Simplifies API and web app development.

Real-World Use Cases

Express is used in many production-grade applications, such as:

- REST APIs

- Web applications (with HTML templating)
- Backend for mobile and single-page apps
- Server-side rendering setups

Installing Express.js

Before using Express, make sure Node.js and npm are installed.

To install Express:

```
npm install express
```

Your First Express App

Here's a basic example of an Express server:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```


- `express()` creates an Express app.
- `app.get()` defines a route handler.
- `app.listen()` starts the server.

Visit `http://localhost:3000` in your browser to test it.

Installing and Setting Up Express.js

Project Setup

Before we start writing code, let's set up a new project folder.

1. Create a new folder:

```
mkdir express-intro
cd express-intro
```

1. Initialize a new Node.js project:

```
npm init -y
```

This creates a `package.json` file with default settings.

1. Install Express:

```
npm install express
```

You'll now see `express` listed in your `dependencies`.

Creating Your First Express Server

Create a file called `index.js`:

```
const express = require('express');
const app = express();
```

```
// Define a basic GET route
app.get('/', (req, res) => {
  res.send('Welcome to Express.js!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Testing the Server

1. Run the server using:

```
node index.js
```

1. Open your browser and visit:

```
http://localhost:3000
```

You should see the message: `Welcome to Express.js!`

Understanding the Code

- `express()` initializes the app.
- `app.get()` sets up a route for GET requests to the `/` path.
- `res.send()` sends a simple text response.
- `app.listen()` starts the server on the specified port.

Routing in Express.js

What is Routing?

Routing refers to how an application responds to client requests to specific paths (URLs) using HTTP methods such as GET, POST, PUT, DELETE, etc.

In Express.js, routes define the logic for what should happen when a user visits a particular URL.

Basic GET Route

```
app.get('/', (req, res) => {
  res.send('Home Page');
});
```

- This responds to a GET request to the root (`/`) URL.
- `res.send()` sends a response back to the client.

Route Methods in Express

Express provides methods for all standard HTTP methods:

```
app.get('/about', (req, res) => {
  res.send('About Page');
});

app.post('/contact', (req, res) => {
  res.send('Contact form submitted');
```

```
});

app.put('/user/:id', (req, res) => {
  res.send(`Update user with ID ${req.params.id}`);
});

app.delete('/user/:id', (req, res) => {
  res.send(`Delete user with ID ${req.params.id}`);
});
```

Route Parameters

Route parameters are named segments of the URL prefixed with a colon (`:`). They allow you to capture values from the URL.

```
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID is ${userId}`);
});
```

Query Parameters

Query parameters are added to the URL after a `?` and are accessible using `req.query` .

```
// URL: /search?term=node
app.get('/search', (req, res) => {
  const searchTerm = req.query.term;
  res.send(`You searched for ${searchTerm}`);
});
```

Summary

- Use `app.METHOD()` to define routes.
- Use `req.params` to get route parameters.
- Use `req.query` for query parameters.

What is MongoDB?

Introduction

MongoDB is a **NoSQL document-oriented database** designed for modern application development. It stores data in flexible, JSON-like documents, which makes it easy to work with dynamic or semi-structured data.

Unlike traditional relational databases (like MySQL or PostgreSQL), MongoDB does **not use tables or rows**. Instead, it uses:

- **Databases** → which contain
- **Collections** → which contain
- **Documents** (individual records in JSON/BSON format)

Key Features of MongoDB

- **Document-Based Storage**: Data is stored in BSON (binary JSON) documents.
- **Schema-less**: Each document can have a different structure.
- **High Performance**: Built for high read/write throughput.
- **Horizontal Scalability**: Supports sharding to handle large datasets.
- **Rich Query Language**: Supports nested queries, filters, and aggregation.

MongoDB vs Relational Databases

Feature	MongoDB	Relational DB (e.g., MySQL)
Data Model	Document	Table/Row
Schema	Flexible (Schema-less)	Fixed (Predefined Schema)
Relationships	Embedded or Referenced	Foreign Keys

Feature	MongoDB	Relational DB (e.g., MySQL)
Query Language	BSON-based	SQL
Best Use Cases	Real-time apps, analytics	Financial systems, complex joins

When to Use MongoDB

MongoDB is ideal for use cases like:

- Content management systems (CMS)
- Social media applications
- Product catalogs or inventory systems
- Real-time analytics dashboards
- IoT and sensor data storage

Conclusion

MongoDB is a powerful, flexible, and scalable database solution suited for applications where data structure may evolve over time or performance at scale is critical.

Setting Up MongoDB

Local Installation (Optional for Beginners)

To install MongoDB on your system:

On Windows

1. Go to MongoDB Community Download Center.
2. Download the MSI installer for your version.
3. Follow the installation steps and **enable MongoDB as a service**.
4. Use the terminal to run:

   ```
   mongod
   ```

   This starts the MongoDB server.

On macOS (Using Homebrew)

```
brew tap mongodb/brew
brew install mongodb-community
brew services start mongodb-community
```

On Linux (Ubuntu Example)

```
sudo apt update
sudo apt install -y mongodb
sudo systemctl start mongodb
```

To check if MongoDB is running:

```
mongo
```

This will open the MongoDB shell if it's installed correctly.

Using MongoDB Atlas (Cloud Setup)

MongoDB Atlas is the easiest way to get started without installing anything.

Steps to Create a Cluster:

1. Visit https://www.mongodb.com/cloud/atlas
2. Sign up and create a **free cluster** (Shared Tier).
3. Choose cloud provider and region.
4. Create a **username and password** for database access.
5. Whitelist your IP address or allow all IPs (`0.0.0.0/0`) for testing.
6. Get your connection string, which looks like:

```
mongodb+srv://<username>:<password>@cluster0.mongodb.net/myDatabase?
retryWrites=true&w=majority
```

MongoDB Compass (Optional GUI)

MongoDB Compass is a GUI client to visually interact with your databases.

- Download from: https://www.mongodb.com/products/compass
- Connect using the same Atlas URI or local URI like:

```
mongodb://localhost:27017
```

You can:

- View collections and documents
- Run queries
- Inspect schema visually

Connecting to MongoDB with Node.js

Install the official MongoDB driver:

```
npm install mongodb
```

Sample connection code:

```
const { MongoClient } = require('mongodb');

const uri = 'your_connection_string';
const client = new MongoClient(uri);

async function run() {
  await client.connect();
  const db = client.db('test');
  const collection = db.collection('students');
  const result = await collection.find().toArray();
  console.log(result);
  await client.close();
}

run();
```

Create and Read Documents

In MongoDB, data is stored in **documents** (which are JSON-like objects) inside **collections**. You can perform **Create** and **Read** operations using simple methods.

This section will cover:

- `insertOne()`, `insertMany()`
- `find()`, `findOne()`
- Basic filters and projections

Note: All code examples in this section are written for **MongoDB Compass** (MongoDB Shell syntax). You can run these directly in the MongoDB Compass shell or mongosh.

Setting Up Sample Data

Before we start, let's create a school database with students and teachers. Run this in MongoDB Compass:

```
// Switch to school database
use school

// Insert sample teachers
db.teachers.insertMany([
  {
    _id: ObjectId("507f1f77bcf86cd799439011"),
    name: 'Dr. Kumar',
    subject: 'MongoDB',
    experience: 5
  },
  {
    _id: ObjectId("507f1f77bcf86cd799439012"),
```

```
    name: 'Prof. Sharma',
    subject: 'Node.js',
    experience: 8
  },
  {
    _id: ObjectId("507f1f77bcf86cd799439013"),
    name: 'Ms. Patel',
    subject: 'Express',
    experience: 3
  }
])

// Insert sample students with teacher references
db.students.insertMany([
  {
    name: 'Ali',
    age: 22,
    course: 'MongoDB',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439011"),
    grades: [85, 90, 88]
  },
  {
    name: 'Sara',
    age: 20,
    course: 'Node.js',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439012"),
    grades: [92, 88, 95]
  },
  {
    name: 'Ahmed',
    age: 24,
    course: 'Express',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439013"),
    grades: [78, 82, 85]
  },
  {
    name: 'Fatima',
```

```
    age: 21,
    course: 'MongoDB',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439011"),
    grades: [95, 93, 97]
  },
  {
    name: 'Ravi',
    age: 23,
    course: 'Node.js',
    enrolled: false,
    teacherId: ObjectId("507f1f77bcf86cd799439012"),
    grades: [70, 75, 72]
  }
])
```

Inserting Documents

insertOne

Use this to insert a single document into a collection.

```
db.students.insertOne({
  name: 'Priya',
  age: 19,
  course: 'MongoDB',
  enrolled: true,
  teacherId: ObjectId("507f1f77bcf86cd799439011"),
  grades: [88, 91, 89]
})
```

insertMany

Use this to insert multiple documents at once.

```
db.students.insertMany([
  {
    name: 'Kabir',
    age: 20,
    course: 'Node.js',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439012"),
    grades: [84, 87, 86]
  },
  {
    name: 'Zara',
    age: 22,
    course: 'Express',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439013"),
    grades: [90, 92, 94]
  }
])
```

Reading Documents

findOne

Returns the first document that matches the filter.

```
db.students.findOne({ name: 'Ali' })
```

find

Returns all matching documents. In MongoDB Compass, results are automatically displayed.

```
db.students.find({ course: 'MongoDB' })
```

To get all students:

```
db.students.find({})
```

Projections

Use projections to include or exclude specific fields.

```
// Include only name and age (exclude _id)
db.students.find({}, { _id: 0, name: 1, age: 1 })
```

```
// Include name, course, and grades
db.students.find({}, { name: 1, course: 1, grades: 1 })
```

Other Options

Limiting Results

```
db.students.find().limit(3)
```

Sorting Results

```
// Sort by age (descending)
db.students.find().sort({ age: -1 })
```

Combining Operations

```
// Find MongoDB students, show only name and grades, sorted by age
db.students.find(
  { course: 'MongoDB' },
```

```
  { name: 1, grades: 1, _id: 0 }
).sort({ age: 1 })
```

Working with Teachers Collection

```
// Find all teachers
db.teachers.find()

// Find teacher by subject
db.teachers.findOne({ subject: 'Node.js' })

// Find experienced teachers (more than 5 years)
db.teachers.find({ experience: { $gt: 5 } })
```

Summary

- Use `insertOne()` and `insertMany()` to add data
- Use `findOne()` and `find()` to read data
- Use projections to control which fields are returned
- Use `.sort()` and `.limit()` to customize results
- All code runs directly in MongoDB Compass shell

Update and Delete Documents

MongoDB provides powerful methods to **update** or **remove** documents from a collection.

This section covers:

- `updateOne()`, `updateMany()`, `$set`, `$inc`
- `deleteOne()`, `deleteMany()`
- `replaceOne()`
- Understanding `ObjectId`

Note: All code examples are for MongoDB Compass shell (mongosh).

Updating Documents

updateOne

Updates the **first** document that matches the filter.

```
// Change Ali's course to Advanced MongoDB
db.students.updateOne(
  { name: 'Ali' },
  { $set: { course: 'Advanced MongoDB' } }
)
```

updateMany

Updates **all** documents that match the filter.

```
// Add 1 year to age of all enrolled students
db.students.updateMany(
```

```
  { enrolled: true },
  { $inc: { age: 1 } }
)

// Update all MongoDB students to have a new teacher
db.students.updateMany(
  { course: 'MongoDB' },
  { $set: { teacherId: ObjectId("507f1f77bcf86cd799439011") } }
)
```

Common Update Operators

- `$set` : Sets the value of a field
- `$unset` : Removes a field
- `$inc` : Increments a numeric field
- `$push` : Adds an item to an array
- `$pull` : Removes an item from an array
- `$addToSet` : Adds unique items to an array

Examples:

```
// Add a new grade to Sara's grades array
db.students.updateOne(
  { name: 'Sara' },
  { $push: { grades: 96 } }
)

// Remove enrolled field from Ravi
db.students.updateOne(
  { name: 'Ravi' },
  { $unset: { enrolled: "" } }
)

// Increment teacher's experience by 1
db.teachers.updateOne(
  { name: 'Dr. Kumar' },
```

```
  { $inc: { experience: 1 } }
)
```

Replacing a Document

replaceOne

Replaces the entire document with a new one (except the _id).

```
db.students.replaceOne(
  { name: 'Ravi' },
  {
    name: 'Ravi Kumar',
    age: 24,
    course: 'Python',
    enrolled: true,
    teacherId: ObjectId("507f1f77bcf86cd799439012"),
    grades: [80, 85, 82],
    email: 'ravi.kumar@school.edu'
  }
)
```

Deleting Documents

deleteOne

Deletes the **first** matching document.

```
// Delete a specific student
db.students.deleteOne({ name: 'Kabir' })
```

deleteMany

Deletes **all** matching documents.

```
// Delete all students who are not enrolled
db.students.deleteMany({ enrolled: false })

// Delete all students with low average grades
db.students.deleteMany({
  $expr: {
    $lt: [{ $avg: "$grades" }, 75]
  }
})
```

Working with ObjectId

Each document in MongoDB has a unique `_id` field of type `ObjectId`.

In MongoDB Compass, ObjectId is available globally:

```
// Find a student by _id
db.students.findOne({
  _id: ObjectId("64bd2e183dd4e6402f10388f")
})

// Update a teacher by _id
db.teachers.updateOne(
  { _id: ObjectId("507f1f77bcf86cd799439011") },
  { $set: { office: "Room 301" } }
)
```

Practical Examples

Update Student's Teacher

```
// Move all Express students to a different teacher
db.students.updateMany(
  { course: 'Express' },
  { $set: {
    teacherId: ObjectId("507f1f77bcf86cd799439012"),
    course: 'Advanced Express'
  }}
)
```

Bulk Grade Update

```
// Add bonus points to all students of a specific teacher
db.students.updateMany(
  { teacherId: ObjectId("507f1f77bcf86cd799439011") },
  { $push: { grades: 5 } }  // Add 5 bonus points
)
```

Summary

- Use `updateOne()` or `updateMany()` with operators like `$set`, `$inc`, `$push`
- Use `replaceOne()` to completely replace documents
- Use `deleteOne()` or `deleteMany()` to remove documents
- ObjectId is globally available in MongoDB Compass
- Always verify your filters before running update/delete operations

Query Operators and Filtering

MongoDB provides powerful operators to filter and search documents in flexible ways.

In this section, you will learn how to use:

- Comparison operators (`$gt` , `$lt` , `$eq` , `$ne` , `$in` , `$nin`)
- Logical operators (`$or` , `$and` , `$not` , `$nor`)
- Array and embedded field queries
- Sorting and pagination

Note: All code examples are for MongoDB Compass shell (mongosh).

Comparison Operators

$gt, $gte, $lt, $lte

Find students older than 21:

```
db.students.find({ age: { $gt: 21 } })
```

Find students aged between 18 and 25:

```
db.students.find({ age: { $gte: 18, $lte: 25 } })
```

Find teachers with more than 5 years experience:

```
db.teachers.find({ experience: { $gt: 5 } })
```

$eq, $ne

Find students who are **not enrolled**:

```
db.students.find({ enrolled: { $ne: true } })
```

$in, $nin

Find students enrolled in either “MongoDB” or “Node.js”:

```
db.students.find({ course: { $in: ['MongoDB', 'Node.js'] } })
```

Find teachers NOT teaching Express or Python:

```
db.teachers.find({ subject: { $nin: ['Express', 'Python'] } })
```

Logical Operators

$or

Find students enrolled in “Python” **or** age less than 20:

```
db.students.find({
  $or: [
    { course: 'Python' },
    { age: { $lt: 20 } }
  ]
})
```

$and (default behavior)

Find students aged above 20 **and** enrolled:

```
db.students.find({
  age: { $gt: 20 },
  enrolled: true
})
```

You can also write it explicitly:

```
db.students.find({
  $and: [
    { age: { $gt: 20 } },
    { enrolled: true }
  ]
})
```

$not

Find students **not** enrolled in “Node.js”:

```
db.students.find({
  course: { $not: { $eq: 'Node.js' } }
})
```

Complex Query Example

Find enrolled students who are either: - Taking MongoDB with age > 21, OR - Taking Node.js with high grades (average > 90)

```
db.students.find({
  enrolled: true,
  $or: [
    { course: 'MongoDB', age: { $gt: 21 } },
    {
      course: 'Node.js',
      $expr: { $gt: [{ $avg: "$grades" }, 90] }
    }
```

```
  ]
})
```

Array Queries

Working with the `grades` array in our student documents:

Matching Array Elements

Find students who scored exactly 95 in any test:

```
db.students.find({ grades: 95 })
```

$all - Multiple Values

Find students who scored both 90 and 95 at some point:

```
db.students.find({
  grades: { $all: [90, 95] }
})
```

$size - Array Length

Find students with exactly 3 grades recorded:

```
db.students.find({
  grades: { $size: 3 }
})
```

$elemMatch - Complex Array Conditions

Find students with at least one grade above 90:

```
db.students.find({
  grades: { $elemMatch: { $gt: 90 } }
})
```

Working with Average Grades

Find students with average grade above 85:

```
db.students.find({
  $expr: {
    $gt: [{ $avg: "$grades" }, 85]
  }
})
```

Sorting and Pagination

sort()

Sort students by age (descending):

```
db.students.find().sort({ age: -1 })
```

Sort by multiple fields:

```
// Sort by course (ascending), then by age (descending)
db.students.find().sort({ course: 1, age: -1 })
```

skip() and limit()

Get the first 3 students:

```
db.students.find().limit(3)
```

Implement pagination (skip first 2, then get next 3):

```
db.students.find().skip(2).limit(3)
```

Combined Example

Find top 3 performing MongoDB students:

```
db.students.find({
  course: 'MongoDB'
}).sort({
  grades: -1
}).limit(3)
```

Advanced Queries with Teachers

Join-like Queries

Find all students taught by Dr. Kumar:

```
// First, find Dr. Kumar's ID
db.teachers.findOne({ name: 'Dr. Kumar' })

// Then find all students with that teacherId
db.students.find({
  teacherId: ObjectId("507f1f77bcf86cd799439011")
})
```

Count Operations

Count students per course:

```
db.students.countDocuments({ course: 'MongoDB' })
```

Count all enrolled students:

```
db.students.countDocuments({ enrolled: true })
```

Summary

- Use comparison operators (`$gt` , `$lt` , `$in` , etc.) for filtering
- Combine with logical operators (`$or` , `$and`) for complex queries
- Array operators (`$all` , `$size` , `$elemMatch`) for array fields
- Use `.sort()` , `.skip()` , and `.limit()` for result control
- All queries run directly in MongoDB Compass shell

Introduction to React

In this lecture, we will set the foundation for working with React using Vite. The goal is to understand what React is, why we use it, and how to quickly set up a React project with Vite.

What is React?

React is a popular JavaScript library for building user interfaces. Instead of manipulating the DOM directly, React allows you to build applications using reusable components. These components make your code more organized, modular, and easier to maintain.

Key benefits of React:

- Component-based structure
- Declarative code style
- Efficient updates with the virtual DOM
- Large ecosystem and community support

Why use Vite?

Traditionally, React apps were created using Create React App (CRA). While CRA works, it can feel slow and bloated. Vite is a modern alternative that offers:

- Extremely fast startup times
- Hot Module Replacement (HMR) for instant updates
- Simpler configuration
- Lightweight builds

Vite makes it much easier to get started with React development.

Setting Up Your Environment

Before we can use Vite, make sure you have the following installed:

- Node.js (LTS version recommended)
- npm (comes bundled with Node.js)

You can check if Node.js and npm are installed by running:

```
node -v
npm -v
```

Creating a React Project with Vite

To create a new React project with Vite, run the following command in your terminal:

```
npm create vite@latest my-react-app
```

This will prompt you to choose a framework. Select **React** (or **React + JavaScript** if you see multiple options).

Once the project is created, navigate into the folder and install dependencies:

```
cd my-react-app
npm install
```

Now, run the development server:

```
npm run dev
```

You should see a development URL, usually `http://localhost:5173`, where your React app will be running.

Recap

- React is a JavaScript library for building user interfaces using components.
- Vite is a modern build tool that makes React development faster and simpler.
- We created a new React app using Vite and ran it locally.

Project Setup and Folder Structure

In this lecture, we will explore the project created with Vite and understand the key files and folders. Knowing the structure will make it easier to navigate and organize your React application.

Vite Project Overview

After creating a project with Vite, you will see a structure similar to this:

```
my-react-app/
├── node\_modules/
├── public/
├── src/
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .gitignore
├── index.html
├── package.json
├── vite.config.js
└── README.md
```

Let's break this down.

Important Files and Folders

- **index.html**
  The entry point of the application. Unlike traditional setups, Vite uses this as the main HTML file. Notice the `<div id="root"></div>` where your React components will be rendered.

- **src/**
  This folder contains your React code.
  - `main.jsx` is the entry point for React. It mounts the root React component (`App.jsx`) into the `#root` div in `index.html` .
  - `App.jsx` is your main React component. You can think of it as the starting point of your UI.
  - `App.css` and `index.css` are for styling your components and global styles.
- **public/**
  Any static files like images or icons can be placed here. Files in this folder are served directly without processing.
- **package.json**
  Contains the project metadata, dependencies, and scripts. The important scripts are:

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

- **vite.config.js** Vite's configuration file. For beginners, you won't need to change this much.

Running the Project

Use the following command to start the development server:

```
npm run dev
```

Open the provided local URL (usually `http://localhost:5173`) in your browser. You will see the default Vite + React starter page.

Recap

- Vite creates a simple, lightweight project structure.
- `index.html` contains the root element.
- `main.jsx` connects React with the DOM.
- `App.jsx` is the root component where your UI begins.

Understanding React Components

In this lecture, we will learn what React components are and how to create them. Components are the building blocks of any React application.

What is a Component?

A component is a reusable piece of code that represents part of the user interface. Components can be as small as a button or as large as an entire page.

React components make your code modular and easier to maintain.

Types of Components

React supports two types of components:

- **Function Components**: These are the modern and recommended way to write components.
- **Class Components**: An older way of writing components (not commonly used today). We will focus only on function components.

Creating a Simple Component

Open `App.jsx` and you will see something like this:

```
function App() {
  return (
    <div>
      <h1>Hello Vite + React</h1>
    </div>
  )
}
```

```
export default App
```

This is a function component named `App`. It returns JSX, which looks like HTML but is actually JavaScript.

JSX in Action

JSX allows us to write HTML-like code inside JavaScript. For example:

```
function Greeting() {
  return <h2>Welcome to React</h2>
}
```

Here, `Greeting` is a component that can be reused anywhere in your application.

Using Components

To use a component, simply include it inside another component like this:

```
function App() {
  return (
    <div>
      <Greeting />
    </div>
  )
}
```

Notice that we use `<Greeting />` as if it were an HTML tag. This is how components are composed to build larger applications.

Recap

- Components are reusable building blocks in React.
- Function components are the modern way of creating components.

- JSX looks like HTML but allows JavaScript expressions inside.
- You can include components inside other components.

Passing Data with Props

In this lecture, we will learn how to pass data to components using props. Props allow components to be dynamic and reusable.

What are Props?

Props (short for "properties") are a way of passing data from a parent component to a child component. They make components more flexible because the same component can display different content depending on the props it receives.

Example: Card Component

Let's create a simple `Card` component inside the `src` folder:

```
function Card(props) {
  return (
    <div style={{ border: "1px solid gray", padding: "10px", margin: "10px" }}>
      <h3>{props.title}</h3>
      <p>{props.description}</p>
    </div>
  )
}

export default Card
```

Here, `Card` accepts `title` and `description` as props and displays them.

Using the Card Component

Now open `App.jsx` and use the `Card` component:

```
import Card from "./Card"

function App() {
  return (
    <div>
      <Card title="React Basics" description="Learn the fundamentals of React." />
      <Card title="Props in React" description="Understand how to pass data into components." />
    </div>
  )
}

export default App
```

Each `Card` is the same component, but it displays different content based on the props passed.

Props Are Read-Only

It's important to note that props are **read-only**. A component should never change its props. Instead, they should be used to display or work with the data received from the parent component.

Recap

- Props allow you to pass data into components.
- Props make components reusable and dynamic.
- Props are read-only and should not be modified inside the component.

Managing Data with State

In this lecture, we will learn about state in React. While props let us pass data into components, state allows components to manage their own data and update the user interface when that data changes.

What is State?

State is like a variable that belongs to a component. Unlike normal variables, when state changes, React automatically re-renders the component to reflect the new data.

The useState Hook

React provides a special function called `useState` to create and manage state. It is called a “hook” because it hooks into React features.

Example:

```
import { useState } from "react"

function App() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  )
}

export default App
```

How it works:

- `useState(0)` creates a state variable named `count` with an initial value of `0`.
- `setCount` is the function used to update the state.
- Whenever `setCount` is called, React re-renders the component with the new state value.

Multiple State Variables

You can have more than one state variable inside a component:

```
function Profile() {
  const [name, setName] = useState("Harry")
  const [age, setAge] = useState(19)

  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
    </div>
  )
}
```

Each call to `useState` creates a separate piece of state.

Recap

- State is used to store data inside a component.
- The `useState` hook provides a way to declare and update state.
- Updating state automatically re-renders the component.

Handling Events in React

In this lecture, we will learn how to handle user interactions in React using events. Events make applications interactive, such as responding to clicks, typing, or form submissions.

Event Handling in React

React events are very similar to DOM events in plain JavaScript, but there are a few key differences:

- Event names are written in **camelCase** (e.g., `onClick` instead of `onclick`).
- You pass a function as the event handler, not a string.

Example: Button Click

```
function App() {
  function handleClick() {
    alert("Button was clicked!")
  }

  return (
    <div>
      <button onClick={handleClick}>Click Me</button>
    </div>
  )
}

export default App
```

Here, when the button is clicked, the `handleClick` function runs.

Inline Event Handlers

You can also define the function inline:

```
<button onClick={() => alert("Inline click!")}>Click Inline</button>
```

While this works, using separate functions is better for readability when the logic gets more complex.

Example: Updating State with Events

Let's combine events with state:

```
import { useState } from "react"

function App() {
  const [text, setText] = useState("Hello")

  function updateText() {
    setText("You clicked the button!")
  }

  return (
    <div>
      <p>{text}</p>
      <button onClick={updateText}>Change Text</button>
    </div>
  )
}

export default App
```

Clicking the button updates the state, and React re-renders the UI.

Recap

- React event handlers use camelCase names like `onClick` and `onChange`.
- Event handlers are functions that run in response to user actions.
- Combining events with state allows us to create dynamic, interactive components.

Rendering Lists and Using Keys

In this lecture, we will learn how to display lists of data in React. Often, you will need to render multiple items, such as a list of products, users, or tasks. React makes this easy using the JavaScript `map` function.

Rendering a List

Suppose we have an array of fruits. We can render them like this:

```
function App() {
  const fruits = ["Apple", "Banana", "Orange"]

  return (
    <ul>
      {fruits.map(fruit => (
        <li>{fruit}</li>
      ))}
    </ul>
  )
}

export default App
```

Here, we loop through the `fruits` array using `map` and return a `<li>` for each fruit.

The Key Prop

When rendering lists, React requires a special `key` prop. Keys help React identify which items have changed, been added, or removed. Without keys, React may re-render inefficiently.

```
function App() {
  const fruits = ["Apple", "Banana", "Orange"]

  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  )
}
```

In this example, we use the array index as the key. For real applications, it's better to use a unique ID if available.

Example: Rendering Objects

```
function App() {
  const users = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
    { id: 3, name: "Charlie" }
  ]

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  )
}
```

Each user has a unique `id`, which is perfect for the `key` prop.

Recap

- Use the JavaScript `map` function to render lists in React.
- Always add a unique `key` prop to list items.
- Keys help React efficiently update the UI when lists change.

Conditional Rendering in React

In this lecture, we will learn how to show or hide parts of the user interface based on conditions. This is called conditional rendering.

Using if Statements

You can use a normal JavaScript `if` statement inside a component:

```
function App() {
  const isLoggedIn = true

  if (isLoggedIn) {
    return <h2>Welcome back!</h2>
  } else {
    return <h2>Please log in.</h2>
  }
}

export default App
```

Here, the UI changes depending on the value of `isLoggedIn`.

Using the Ternary Operator

A shorter way to write conditional logic is the ternary operator:

```
function App() {
  const isLoggedIn = false

  return (
    <div>
      {isLoggedIn ? <h2>Welcome back!</h2> : <h2>Please log in.</h2>}
```

```
    </div>
  )
}
```

Using Logical AND (&&)

When you only want to render something if a condition is true, you can use the logical AND operator:

```
function App() {
  const showMessage = true

  return (
    <div>
      <h1>Hello User</h1>
      {showMessage && <p>This is a special message!</p>}
    </div>
  )
}
```

If `showMessage` is `false`, the paragraph will not be rendered.

Example: Toggle Visibility

Let's combine state and conditional rendering:

```
import { useState } from "react"

function App() {
  const [visible, setVisible] = useState(true)

  return (
    <div>
      <button onClick={() => setVisible(!visible)}>
        {visible ? "Hide" : "Show"}
      </button>
```

```
      {visible && <p>This text can be toggled on and off.</p>}
    </div>
  )
}

export default App
```

Clicking the button toggles the paragraph on and off.

Recap

- Use `if` statements, the ternary operator, or logical AND (`&&`) for conditional rendering.
- Conditional rendering allows the UI to change based on state or props.
- You can combine state and conditions to build interactive features.

Forms and Two-Way Binding

In this lecture, we will learn how to handle forms in React. Forms allow users to input data, and with two-way binding, we can keep the input value in sync with component state.

Controlled Components

In React, form inputs are usually controlled by state. This means the value of the input is stored in state and updated on every change.

Example with a text input:

```
import { useState } from "react"

function App() {
  const [name, setName] = useState("")

  function handleChange(event) {
    setName(event.target.value)
  }

  return (
    <div>
      <input type="text" value={name} onChange={handleChange} />
      <p>Your name is: {name}</p>
    </div>
  )
}

export default App
```

Here:

- `value={name}` means the input's value comes from state.

- `onChange` updates the state whenever the user types.

This is called two-way binding: the input updates the state, and the state updates the input.

Handling Multiple Inputs

For multiple inputs, you can have separate state variables:

```
import { useState } from "react"

function LoginForm() {
  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")

  function handleSubmit(event) {
    event.preventDefault()
    alert(`Email: ${email}, Password: ${password}`)
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email: </label>
        <input type="email" value={email} onChange={e =>
setEmail(e.target.value)} />
      </div>
      <div>
        <label>Password: </label>
        <input type="password" value={password} onChange={e =>
setPassword(e.target.value)} />
      </div>
      <button type="submit">Login</button>
    </form>
  )
}

export default LoginForm
```

The `handleSubmit` function prevents the page from refreshing and shows the input values in an alert.

Recap

- Controlled components keep form inputs in sync with state.
- Two-way binding means user input updates state, and state updates the input field.
- Forms in React usually require event handlers for `onChange` and `onSubmit`.

Styling Components in React

In this lecture, we will learn different ways to style components in React. Styling is important to make your application look polished and user-friendly.

Using Regular CSS Files

You can write your styles in a separate CSS file and import it into your component.

`App.css`:

```
.container {
  text-align: center;
  padding: 20px;
  border: 1px solid gray;
}
```

`App.jsx`:

```
import "./App.css"

function App() {
  return <div className="container">Styled with CSS file</div>
}

export default App
```

Here, we use `className` instead of `class` because `class` is a reserved word in JavaScript.

Inline Styles

You can also apply styles directly to elements using inline style objects.

```
function App() {
  const style = {
    color: "blue",
    fontSize: "20px",
    padding: "10px"
  }

  return <h2 style={style}>Styled with Inline Styles</h2>
}

export default App
```

Notice that CSS properties are written in **camelCase** (e.g., `fontSize` instead of `font-size`).

CSS Modules

For component-specific styles, you can use CSS Modules. This prevents class name conflicts.

`Button.module.css`:

```
.button {
  background-color: green;
  color: white;
  padding: 10px;
  border: none;
  border-radius: 5px;
}
```

`Button.jsx`:

```
import styles from "./Button.module.css"

function Button() {
  return <button className={styles.button}>Click Me</button>
}
```

```
export default Button
```

Here, `styles.button` applies the scoped CSS class to the button.

Recap

- You can style React components with regular CSS files, inline styles, or CSS Modules.
- `className` is used instead of `class`.
- CSS Modules are useful to avoid class name conflicts.

Navigation with React Router

In this lecture, we will learn how to add navigation to a React app using **React Router**. This allows us to create multiple pages within a single-page application.

Installing React Router

First, install React Router with npm:

```
npm install react-router-dom
```

Setting Up Routes

Open `main.jsx` and wrap your app in a `BrowserRouter`:

```
import React from "react"
import ReactDOM from "react-dom/client"
import { BrowserRouter } from "react-router-dom"
import App from "./App"
import "./index.css"

ReactDOM.createRoot(document.getElementById("root")).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)
```

Defining Routes

Inside `App.jsx`, use the `Routes` and `Route` components to define paths.

```jsx
import { Routes, Route, Link } from "react-router-dom"

function Home() {
  return <h2>Home Page</h2>
}

function About() {
  return <h2>About Page</h2>
}

function Contact() {
  return <h2>Contact Page</h2>
}

function App() {
  return (
    <div>
      <nav>
        <Link to="/">Home</Link> |{" "}
        <Link to="/about">About</Link> |{" "}
        <Link to="/contact">Contact</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </div>
  )
}

export default App
```

Here's what happens:

- The navigation bar uses `Link` to change pages without reloading.
- `Routes` and `Route` define which component shows for each path.

- For example, `/about` displays the `About` component.

Recap

- React Router enables multi-page navigation in React apps.
- Use `BrowserRouter` to enable routing.
- Define routes with `Routes` and `Route`.
- Use `Link` for navigation instead of `<a>` tags.

React Hooks: useEffect

The **useEffect** hook allows you to perform side effects in React components. Side effects are tasks that run outside the normal component rendering process, such as fetching data, updating the document title, or setting up subscriptions.

What is a Side Effect?

Examples of side effects include:

- Fetching data from an API
- Directly updating the DOM
- Setting up timers or intervals
- Subscribing to events

Syntax

```
import { useState, useEffect } from "react";

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `Clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
```

```
    </div>
  );
}
```

How it works

- `useEffect` runs after every render by default.
- In the above example, whenever `count` changes, the document title is updated.

Using Dependencies

You can control when the effect runs by passing a dependency array as the second argument.

```
useEffect(() => {
  console.log("Effect ran!");
}, [count]);
```

- The effect will only run when `count` changes.
- If you pass an empty array `[]`, the effect runs only once after the component mounts.

Cleanup with useEffect

Some effects require cleanup, like removing event listeners or clearing intervals. You can return a function from `useEffect` for cleanup.

```
useEffect(() => {
  const timer = setInterval(() => {
```

```
    console.log("Timer running");
  }, 1000);

  return () => clearInterval(timer);
}, []);
```

- The cleanup function runs before the component unmounts or before the effect re-runs.

Key Points

- `useEffect` replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- Always specify dependencies clearly to avoid unnecessary re-renders.
- Use cleanup to prevent memory leaks.