😬

WPT 1

Created	@June 22, 2025 6:59 PM

🌐 How does the Internet work?

👉 The internet is like a huge network of networks that connects computers around the world.

👉 Each computer or device has a unique **IP address** (like your house address) so it can send and receive data.

What happens when you open a website:

1️⃣ You type a website name (like `www.google.com`) in your browser.

2️⃣ Your browser asks the **DNS (Domain Name System)** to find the IP address for that name.

3️⃣ The browser sends a request to that IP address (the server).

4️⃣ The server sends back the website data (like HTML, images, etc.).

5️⃣ Your browser shows the website on your screen.

🌍 Internet Protocol (IP)

- **IP (Internet Protocol)** is the set of rules that defines how data travels over the internet.
- Data is broken into **packets** (small pieces).
- Each packet has the destination's IP address, so it knows where to go.

Example:

If you send a message, it's split into packets → packets travel → packets are reassembled at the destination.

🔗 HTTP (Hypertext Transfer Protocol)

- **HTTP** is the protocol (set of rules) that the browser and server use to talk to each other.
- When you visit a website, your browser sends an **HTTP request**, and the server responds with the web page data.

🌐 Domain Names

- A **domain name** is the human-friendly name of a website (like `facebook.com`).
- It is easier to remember than IP addresses (like `142.250.195.36`).
- **DNS** converts the domain name to the correct IP address.

📝 HTTP Protocol Versions

Difference between HTTP 1.0 and HTTP 1.1

Feature	HTTP 1.0	HTTP 1.1
Connection handling	One request per connection	Multiple requests per connection (keep-alive)
Host header requirement	Optional	Required (supports hosting multiple websites on one server)
Caching	Basic	Advanced caching supported
Data transfer	No chunking	Supports chunked transfer

⚡ HTTP Methods

- **GET** → Request data from the server (e.g., load a web page).
- **POST** → Send new data to the server (e.g., submit a form).
- **HEAD** → Like GET, but only ask for headers (no content).
- **PUT** → Upload or update a resource on the server.

- **DELETE** → Delete a resource on the server.

🚦 HTTP Status Codes

Code	Meaning
200	OK (request was successful)
404	Not Found (resource not found)
500	Internal Server Error (server problem)
301	Moved Permanently (redirect)
403	Forbidden (access denied)

⚠ Stateless Nature of HTTP

👉 **HTTP is stateless** — this means each request is treated separately, and the server doesn't remember anything about previous requests.

➡️ Example:

If you log in to a site, the server doesn't automatically remember you on the next page. That's why **cookies** or **sessions** are used to keep track of users.

🔐 HTTPS (HTTP Secure)

- **HTTPS = HTTP + security (encryption with SSL/TLS)**
- Data sent between browser and server is encrypted so that hackers can't steal it.
- URLs start with `https://` and show a lock icon in the browser.

🏛️ Architecture of the Web

A typical website has this flow:

```
scss

Browser (client)
```

```
   ↓
Internet (network)
   ↓
Web server (like Apache or IIS)
   ↓
App code (PHP, Java, Node.js, etc.)
   ↓
Database (MySQL, MongoDB)
```

➡️ The browser requests the page → server processes it → server asks the database if needed → sends back the result to the browser.

⚙️ Web Servers

1️⃣ IIS (Internet Information Services)

- A web server software made by Microsoft.
- Runs on Windows servers.
- Best for websites built using .NET or ASP.NET technologies.
- Easy to manage using graphical tools.

2️⃣ Apache HTTP Server

- A free, open-source web server.
- Works on Linux, Windows, etc.
- Commonly used for PHP, Python, and Java apps.
- Configured using text files (`httpd.conf`).

Feature	IIS	Apache
Platform	Windows	Linux, Windows, Mac
Best for	.NET apps	PHP, Python, Java apps
Cost	Comes with Windows license	Free (open source)

Configuration style	GUI + config files	Mainly text config files

💡 Summary

✅ The internet connects computers using IP addresses and protocols like HTTP.

✅ HTTP defines how browsers and servers communicate.

✅ HTTPS ensures communication is secure.

✅ Web servers (Apache, IIS) serve website files to browsers.

🌟 Introduction to HTML5

👉 **HTML5** is the latest version of HTML (HyperText Markup Language) —

It is used to **structure web pages** and tell the browser **how to display content** (text, images, links, videos, etc.).

✅ New features in HTML5:

- Supports **audio**, **video**, and **canvas** (for graphics/drawing)
- Cleaner syntax (no need to always close empty tags with `/`)
- Better support for mobile-friendly pages
- Semantic tags like `<header>`, `<footer>`, `<section>`, etc.

🏷️ Basic HTML Tags

🌟 What is a tag?

👉 A **tag** is a keyword in HTML that tells the browser **how to display content** on the web page.

👉 Tags usually come in **pairs**:

- **Opening tag** — starts an element
- **Closing tag** — ends that element

👉 Tags are written inside **angle brackets** `< >`.

Let’s go through some important tags that you’ll use all the time 👇

1️⃣ Headings

HTML provides 6 levels of headings:

```
html

<h1>This is Heading 1 (biggest)</h1>
<h2>This is Heading 2</h2>
<h3>This is Heading 3</h3>
<h4>This is Heading 4</h4>
<h5>This is Heading 5</h5>
<h6>This is Heading 6 (smallest)</h6>
```

✅ Use headings to give structure to your page content (like titles, subheadings).

2️⃣ Paragraph

```
html

<p>This is a paragraph of text. It can contain multiple lines of content and will automatically have space above and below.</p>
```

✅ Used for normal body text.

3️⃣ Anchor (Links)

```
html

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

✅ `<a>` creates a hyperlink.

👉 `href` attribute sets the URL where the link points.

4️⃣ Image

```
html

<img src="image.jpg" alt="Description of image" width="300" height="200">
```

✅ `<img>` is used to display images.

👉 `src` = path to image file

👉 `alt` = text shown if image doesn't load (also good for accessibility)

5️⃣ Alignment (using CSS or old align attribute)

➡️ In modern HTML5, use **CSS for alignment**:

```
html

<p style="text-align: center;">This text is centered.</p>
<p style="text-align: right;">This text is right-aligned.</p>
<p style="text-align: left;">This text is left-aligned.</p>
```

✅ `text-align` aligns text inside elements like `<p>`, `<div>`, etc.

6️⃣ Lists

✅ **Ordered List (numbered):**

```
html

<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
```

```
</ol>
```

✅ **Unordered List (bulleted):**

```
html

<ul>
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

7️⃣ Tables

```
html

<table border="1">
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Alice</td>
    <td>25</td>
  </tr>
  <tr>
    <td>Bob</td>
    <td>30</td>
  </tr>
</table>
```

✅ `<table>` defines a table

✅ `<tr>` = table row

✅ `<th>` = table header (bold + centered)

✅ `<td>` = table data (cell)

8️⃣ Iframes

👉 **Iframes embed another web page or document inside your page:**

```
html

<iframe src="https://www.wikipedia.org" width="600" height="400"></iframe>
```

✅ Example use: embed YouTube videos, Google Maps, external pages, etc.

💡 Example Putting it all Together:

```
html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My First HTML5 Page</title>
</head>
<body>

  <h1>Welcome to My Webpage</h1>
  <p style="text-align: center;">This is a centered paragraph.</p>

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

  <p>Here is an image:</p>
  <img src="logo.png" alt="Company Logo" width="200">
```

```
  <h2>My Favorite Fruits</h2>
  <ul>
    <li>Mango</li>
    <li>Apple</li>
    <li>Banana</li>
  </ul>

  <h2>Team Members</h2>
  <table border="1">
    <tr><th>Name</th><th>Role</th></tr>
    <tr><td>Alice</td><td>Developer</td></tr>
    <tr><td>Bob</td><td>Designer</td></tr>
  </table>

  <h2>Embedded Site</h2>
  <iframe src="https://www.wikipedia.org" width="500" height="300"></ifra
me>

</body>
</html>
```

⚡ Final tips

✅ HTML5 focuses on **clean code** — avoid outdated things like `<font>` tag, use CSS instead.

✅ Always close your tags properly!

✅ Use semantic tags for better SEO and structure (`<header>`, `<footer>`, `<nav>`, etc.)

🌟 HTML5 — New Features

HTML5 is an upgraded version of HTML that focuses on:

✅ Rich media (audio, video, graphics)

✅ Better page structure (semantic tags)

✅ Mobile-friendliness

✅ Accessibility

✅ Cleaner and more powerful code

🆕 1️⃣ New Elements (Semantic & Media elements)

👉 **Semantic elements** (these give meaning to the structure of the page — helps browsers, search engines, and developers)

```
html

<header> ... </header>   <!-- Page header -->
<footer> ... </footer>   <!-- Page footer -->
<nav> ... </nav>         <!-- Navigation links -->
<article> ... </article> <!-- Independent content -->
<section> ... </section> <!-- Section of page -->
<aside> ... </aside>     <!-- Sidebar or extra info -->
<main> ... </main>       <!-- Main content of page -->
```

👉 **Media elements**

```
<video> ... </video>     <!-- Embed video without Flash -->
<audio> ... </audio>     <!-- Embed audio -->
<canvas> ... </canvas>   <!-- Draw graphics using JavaScript -->
```

👉 **Other elements**

```
<figure> ... </figure>   <!-- Self-contained content, like images -->
<figcaption> ... </figcaption>  <!-- Caption for <figure> -->
```

```
<mark> ... </mark>      <!-- Highlighted text -->
<time> ... </time>      <!-- Date/time info -->
```

🆕 2️⃣ New Attributes

👉 HTML5 introduced new attributes for existing tags:

- `placeholder` → Shows hint in input box

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

- `required` → Makes a form field mandatory

```
<input type="email" required>
```

- `autofocus`, `autocomplete`, `pattern`, `form` (for inputs)
- `download` → For `<a>` tag, allows file download

```
<a href="file.pdf" download>Download File</a>
```

🆕 3️⃣ Link Relations

👉 Link relations describe the **relationship** between the current page and linked resources:

```
html

<link rel="stylesheet" href="styles.css">  <!-- CSS file -->
<link rel="icon" href="favicon.ico">      <!-- Favicon icon -->
```

```
<link rel="author" href="author-info.html"> <!-- Author info -->
<link rel="prev" href="page1.html">       <!-- Previous page -->
<link rel="next" href="page3.html">       <!-- Next page -->
```

🆕 4️⃣ Microdata

👉 Microdata lets you **embed extra data (metadata)** about your content so search engines (like Google) can understand it better (for rich results).

Example (for product info):

```
html

<div itemscope itemtype="http://schema.org/Product">
  <span itemprop="name">Laptop XYZ</span>
  <span itemprop="price">$999</span>
</div>
```

🌟 What is `rel` in a `<link>` tag?

👉 `rel` stands for **relationship**.

👉 It tells the browser **what kind of relationship** the linked file or resource has with the current HTML document.

👉 Think of it as **explaining the purpose of the link** to the browser.

📝 Common values for `rel` and what they mean

rel value	What it means	Example
`stylesheet`	The linked file is a CSS stylesheet for this page	`<link rel="stylesheet" href="styles.css">`
`icon` or `shortcut icon`	The linked file is the favicon (browser tab icon)	`<link rel="icon" href="favicon.ico">`

`author`	The linked file contains author information	`<link rel="author" href="author.html">`
`prev`	This link points to the previous document in a series (like previous page in pagination)	`<link rel="prev" href="page1.html">`
`next`	This link points to the next document in a series	`<link rel="next" href="page3.html">`
`alternate`	Link to an alternate version (e.g. different language, feed, or mobile version)	`<link rel="alternate" href="feed.xml">`

🆕 5️⃣ ARIA (Accessible Rich Internet Applications)

👉 ARIA attributes help make websites accessible to users with disabilities (screen readers, etc.)

Example:

```
html

<button aria-label="Close" onclick="closeWindow()">X</button>
```

✅ `aria-label` tells assistive technology what this button does.

👉 ARIA adds **extra information** so the assistive technology knows:

- What the element is
- What its role is
- What its state is (open/closed, checked/unchecked, etc.)

Other ARIA examples:

- `role="navigation"`
- `aria-hidden="true"`
- `aria-checked="false"`

🆕 6️⃣ Objects

👉 `<object>` tag can be used to embed external content (like PDFs, Flash, or other documents).

Example:

```
html

<object data="file.pdf" type="application/pdf" width="500" height="600"></object>
```

🆕 7️⃣ Events (New in HTML5 + JS)

👉 HTML5 made it easier to handle **events** (things that happen on the page):

- `oninput` → when user types in an input box
- `onblur` → when element loses focus
- `onfocus` → when element gets focus
- `oncanplay` → when audio/video is ready to play
- `ondrag` → when element is dragged
- `ondrop` → when dragged element is dropped

Example:

```
html

<input type="text" oninput="console.log('Typing...')">
```

🆕 8️⃣ Canvas

👉 `<canvas>` is used for **drawing graphics, charts, games, or animations** using JavaScript.

Example:

```
html

<canvas id="myCanvas" width="300" height="150"></canvas>
<script>
  var c = document.getElementById("myCanvas");
  var ctx = c.getContext("2d");
  ctx.fillStyle = "red";
  ctx.fillRect(10, 10, 100, 50);
</script>
```

✅ This draws a red rectangle on the page!

🚀 Summary of New Features

Feature	What it provides
New elements	Better page structure (header, nav, section, etc.), multimedia support (video, audio, canvas)
New attributes	Enhanced forms, better usability
Link relations	Clear relation between resources (icon, author, prev/next)
Microdata	Extra data for search engines (SEO)
ARIA	Accessibility for screen readers
Objects	Embed external files (PDF, multimedia)
Events	More user interaction possibilities
Canvas	Draw 2D graphics on the fly

🌟 What is HTML5 Validation?

👉 **HTML5 validation** is a built-in way to **check user input in forms** before the form is submitted to the server.

👉 The browser itself checks if the entered data follows certain rules, without needing JavaScript.

👉 Example:

- Required fields left empty? ❌ Browser stops submission.
- Wrong email format? ❌ Browser shows error.
- Number out of range? ❌ Browser warns the user.

✅ The goal:

👉 **Reduce errors**

👉 **Improve user experience**

👉 **Save server processing by catching errors early**

📝 How does HTML5 Validation work?

👉 You add certain **attributes** to your form elements.

👉 When the user submits the form, the browser checks if the input **matches the rules** set by these attributes.

👉 If not, it shows an error message and stops submission.

⚡ Common HTML5 validation attributes

Let's see the most important ones 👇

Attribute	What it does	Example
`required`	Field cannot be left empty	`<input type="text" required>`
`min` / `max`	Set minimum/maximum value for numbers	`<input type="number" min="1" max="10">`
`minlength` / `maxlength`	Set minimum/maximum length of text	`<input type="text" minlength="3" maxlength="10">`
`pattern`	Set a regular expression pattern	`<input type="text" pattern="[A-Za-z]{3,}">`
`type`	Certain types auto-validate format (like `email`, `url`)	`<input type="email">`
`step`	Set stepping for numbers	`<input type="number" step="5">`
`placeholder`	Gives hint text (not validation, but improves UX)	`<input type="text" placeholder="Enter name">`

`readonly` / `disabled`	`readonly` lets you view but not edit; `disabled` makes field inactive	`<input type="text" readonly>`

📝 Example form with validation

```
<form>
  <label>Email:</label>
  <input type="email" required><br><br>

  <label>Age (18-99):</label>
  <input type="number" min="18" max="99" required><br><br>

  <label>Password (min 6 chars):</label>
  <input type="password" minlength="6" required><br><br>

  <label>Website:</label>
  <input type="url"><br><br>

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

👉 What happens?

✅ Browser will check:

- Email looks like `abc@xyz.com`
- Age is between 18 and 99
- Password is at least 6 characters

If any check fails, the form won't submit and browser will show an error.

💡 What kind of error messages?

Browser shows **default messages** like:

- "Please fill out this field."

- “Please enter a valid email address.”
- “Value must be greater than or equal to 18.”

✅ You can customize messages with JavaScript if you want (optional).

🚀 Advantages of HTML5 validation

✅ No need for extra JavaScript code for basic checks

✅ Faster feedback for users

✅ Works in modern browsers

✅ Saves server load by catching bad data early

⚠️ Limitations

❌ It’s **not a replacement for server-side validation** — people can disable client-side validation or tamper with data

❌ Browser default messages may not look the same everywhere

✨ Summary

👉 **HTML5 validation** = Built-in form checks using attributes.

👉 Helps users fill forms correctly before submitting.

👉 Common attributes: `required`, `type="email"`, `min`, `max`, `pattern`, `minlength`, `maxlength`.

👉 Always combine with server-side validation for security!

🎵 HTML5 Audio Support

👉 HTML5 introduced the `<audio>` tag so we can **play audio files directly in the browser** — no need for plugins like Flash!

📌 Syntax:

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

✅ Explanation:

- `<audio>` — the tag that tells browser you want to embed audio
- `controls` — adds play, pause, volume controls
-
- `<source>` — specifies the file(s) and format(s)
- Fallback text — shown if browser can't play audio

💡 Example output:

👉 Browser shows a small player where user can play/pause/adjust volume.

🎬 HTML5 Video Support

👉 HTML5 also introduced the `<video>` tag so you can **play videos** directly in the browser without needing external plugins.

📌 Syntax:

```html
<video width="400" controls>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.webm" type="video/webm">
  Your browser does not support the video tag.
</video>
```

```
</video>
```

✅ Explanation:

- `<video>` — tag for embedding video
- `width` — sets video display width
- `controls` — shows play, pause, seek, volume controls
- `<source>` — lists video files in different formats

👉 The browser picks the first format it supports.

🔑 Additional attributes:

- `autoplay` → Video starts playing automatically
- `loop` → Video keeps replaying
- `muted` → Video starts muted
- `poster="image.jpg"` → Shows an image before video plays

🌍 HTML5 Geolocation Support

👉 Geolocation API lets you **get the location (latitude/longitude)** of the user's device (with their permission).

👉 Example uses:

✅ Maps

✅ Show nearby stores

✅ Track deliveries

✅ Check-in features

📌 Example:

```
html
```

```
<button onclick="getLocation()">Get My Location</button>
<p id="output"></p>

<script>
function getLocation() {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(showPosition);
  } else {
    document.getElementById("output").innerHTML = "Geolocation is not supp
orted by this browser.";
  }
}

function showPosition(position) {
  document.getElementById("output").innerHTML =
  "Latitude: " + position.coords.latitude +
  "<br>Longitude: " + position.coords.longitude;
}
</script>
```

✅ What happens?

👉 When the user clicks **Get My Location**:

- Browser asks for permission
- If allowed, it shows their latitude & longitude

⚡ Summary of these features

Feature	Tag/API	What it does
Audio	`<audio>`	Play sound/music in browser
Video	`<video>`	Play videos in browser
Geolocation	`navigator.geolocation`	Get user's location (coordinates)

🚀 Advantages

✅ No need for external plugins like Flash

✅ Works across modern browsers

✅ Gives better user experience

🌟 What is an HTML Form?

👉 An HTML **form** is used to collect **user input** — like name, email, password, choices, etc.

📌 Basic Form Structure:

```
<form action="submit_page.php" method="post">
  <!-- form controls go here -->
</form>
```

- `action` = where to send form data after submit (a file or URL)
- `method` = `post` (secure) or `get` (shows data in URL)

🧾 1️⃣ Input Field (Single line)

👉 Used for text, email, password, etc.

```
<label>Username:</label>
<input type="text" name="username">
```

Common `input` types:

Type	What it does
`text`	Basic text input
`email`	Validates email format
`password`	Hides text as you type

`number`	Only allows numbers
`date`	Date picker
`file`	File upload field

✍️2️⃣ Textarea (Multiline input)

👉 For longer input like messages or comments.

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

🔘3️⃣ Radio Buttons (Single Choice from a group)

```
<label>Gender:</label><br>
<input type="radio" name="gender" value="male"> Male<br>
<input type="radio" name="gender" value="female"> Female<br>
```

👉 Only **one** radio button from a group (same `name`) can be selected.

3. `name="gender"`

👉 The `name` attribute **groups radio buttons together**.

👉 Radio buttons that have the **same** `name` belong to the same group, and only **one of them can be selected** at a time.

3️⃣ `value="male"`

👉 This defines **what value will be sent to the server** when the form is submitted **if this radio button is selected**.

☑️4️⃣ Checkboxes (Multiple Selections allowed)

```
<label>Choose your hobbies:</label><br>
<input type="checkbox" name="hobby" value="reading"> Reading<br>
<input type="checkbox" name="hobby" value="music"> Music<br>
<input type="checkbox" name="hobby" value="travel"> Travel<br>
```

👉 You can select **more than one** option.

🔽 5️⃣ Dropdown / Select Box

```
<label>Choose a city:</label><br>
<select name="city">
  <option value="mumbai">Mumbai</option>
  <option value="delhi">Delhi</option>
  <option value="pune">Pune</option>
</select>
```

👉 Only **one option** can be selected.

🟢 6️⃣ Submit Button

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

👉 Submits the form to the URL in the `action` attribute.

🔄 7️⃣ Reset Button

```
<input type="reset" value="Clear Form">
```

👉 Clears **all input fields** in the form.

✅ Full Example Form:

```html
<form action="submit.php" method="post">

  <label>Full Name:</label><br>
  <input type="text" name="fullname" required><br><br>

  <label>Email:</label><br>
  <input type="email" name="email" required><br><br>

  <label>Gender:</label><br>
  <input type="radio" name="gender" value="male"> Male
  <input type="radio" name="gender" value="female"> Female<br><br>

  <label>Hobbies:</label><br>
  <input type="checkbox" name="hobby" value="reading"> Reading
  <input type="checkbox" name="hobby" value="sports"> Sports<br><br>

  <label>City:</label><br>
  <select name="city">
    <option value="mumbai">Mumbai</option>
    <option value="pune">Pune</option>
    <option value="delhi">Delhi</option>
  </select><br><br>

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

  <input type="submit" value="Submit">
  <input type="reset" value="Clear">
</form>
```

🔐 Tips

- Always use `name=""` — this is how data is identified when submitted.
- Use `required` to make a field mandatory.
- Group radio buttons by giving them the **same name**.

🧠 Summary Table

Control	Tag Used	Description
Input field	`<input type="text">`	Single-line text
Textarea	`<textarea>`	Multiline text
Radio button	`<input type="radio">`	One choice
Checkbox	`<input type="checkbox">`	Multiple choices
Dropdown	`<select><option>`	One from list
Submit	`<input type="submit">`	Send form
Reset	`<input type="reset">`	Clear all fields

🌟 What is DOM (Document Object Model)?

👉 The **DOM** is a programming interface for **web pages**.

👉 When a web page loads, the browser converts the **HTML document** into a **DOM tree** — a structured, tree-like representation of the page's content.

👉 **DOM = HTML as objects + hierarchy**

- The **document** is the whole HTML page.
- Each **HTML element** (like `<p>`, `<div>`, `<img>`, etc.) becomes an **object** in the tree.
- These objects can be **accessed, modified, deleted, or created** using programming languages like **JavaScript**.

📌 What does the DOM represent?

👉 Think of the DOM as:

- A **live structure** of the page.

- A **tree of nodes** or **objects** that represents everything in your HTML.

🖼️ DOM Tree Example

Given this HTML:

```
html

<html>
  <body>
    <h1>Hello World</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

👉 The DOM looks like this:

```
css

Document
 └── html
     └── body
         ├── h1
         │    └── "Hello World"
         └── p
              └── "This is a paragraph."
```

✅ Each tag is a **node**

✅ Each text inside tags is also a **node**

🔑 Why is the DOM important?

👉 The DOM lets you:

✅ **Access elements** → Get a tag or content (`document.getElementById()` , etc.)

✅ **Change elements** → Change text, styles, attributes

✅ **Add elements** → Insert new HTML dynamically

✅ **Remove elements** → Delete part of the page

👉 The DOM lets **JavaScript control your web page**!

💻 Basic JavaScript DOM Examples

```
html

<p id="demo">Hello</p>
<button onclick="changeText()">Click me</button>

<script>
function changeText() {
  document.getElementById("demo").innerHTML = "You clicked the button!";
}
</script>
```

👉 Here, we:

- **Accessed the DOM** (`getElementById("demo")`)
- **Modified the content** of `<p>`

⚡ Main parts of DOM

Part	What it is
Document node	The whole HTML document
Element node	Each HTML tag (e.g. `<p>` , `<div>`)
Text node	The text inside elements
Attribute node	Attributes like `class` , `id` , etc.

🚀 Summary

👉 The **Document Object Model (DOM)** is:

- The browser's internal model of your HTML page
- A **tree structure** where every tag, text, and attribute is a node
- A way for JavaScript to **interact with and manipulate** the page dynamically

---

🌟 1️⃣ What is CSS (Cascading Style Sheets)?

👉 CSS is a **language used to style HTML elements**.

👉 It controls **how your page looks** — things like:

- Colors
- Fonts
- Sizes
- Layouts
- Spacing
- Positioning

👉 Without CSS → Your HTML is plain, unstyled.

👉 With CSS → You can create beautiful, professional-looking websites!

📝 Example: Basic HTML vs CSS

```
html

<!-- Plain HTML -->
<p>This is a paragraph</p>

<!-- Styled with CSS -->
```

```
<p style="color: blue; font-size: 20px;">This is a styled paragraph</p>
```

➡️ The second paragraph appears **blue** and **larger** because of CSS.

🌈2️⃣ Styling HTML with CSS

💡 3 ways to apply CSS:

Method	Example	When to use?
Inline CSS	`<p style="color:red;">Hello</p>`	Quick test, 1 element
Internal CSS	`<style>p { color: red; }</style>`	Small site, 1 page
External CSS	`<link rel="stylesheet" href="style.css">`	Best for large/multi-page sites

✅ Example of External CSS:

👉 `style.css`

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

h1 {
  color: darkblue;
  text-align: center;
}

p {
  font-size: 18px;
  color: #333;
}
```

👉 `index.html`

```
<link rel="stylesheet" href="style.css">
<h1>Welcome</h1>
<p>This is my page styled with CSS!</p>
```

🖼️ 3️⃣ Structuring Pages with CSS

👉 CSS helps **arrange and design the layout of your web page** — it structures how things appear on the screen.

✨ Examples of structuring:

✅ Position elements

```
header {
  position: fixed;
  top: 0;
  width: 100%;
}
```

➡️ Keeps the header at the top of the page always.

✅ Create columns / grids

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr;
}
```

➡️ Divides the page into columns — for sidebar + main content.

✅ Control spacing

```css
p {
  margin: 10px;
  padding: 5px;
}
```

➡️ Controls space **outside (margin)** and **inside (padding)** elements.

✅ **Make responsive pages**

```css
@media (max-width: 600px) {
  body {
    background-color: lightgray;
  }
}
```

➡️ Changes styles for small screens (mobile-friendly design).

⚡ Summary

What?	Purpose
CSS	Controls the appearance of HTML elements
Styling HTML	Change colors, fonts, sizes, borders, etc.
Structuring pages	Arrange layout (grids, flexbox, spacing, positioning)

💡 **Key point:**

👉 **HTML = content/structure**

👉 **CSS = style/layout**

💻 Example to try:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    body { background-color: #e0f7fa; }
    h1 { color: #00796b; text-align: center; }
    p { font-size: 18px; color: #004d40; }
  </style>
</head>
<body>
  <h1>My Styled Page</h1>
  <p>This page is styled using CSS!</p>
</body>
</html>
```

🌟 1️⃣ Types of CSS: Inline, Internal, External

🟣 Inline CSS

👉 CSS written **directly inside an HTML element** using the `style` attribute.

✅ Example:

```
<p style="color: blue; font-size: 20px;">This is inline styled text</p>
```

✅ **When to use:**

- Quick changes
- One-off styling

- Not recommended for big projects (hard to maintain)

🟣 Internal CSS

👉 CSS written **inside a** `<style>` **tag** in the `<head>` section of the HTML file.

✅ Example:

```
<head>
  <style>
    p {
      color: green;
      font-size: 18px;
    }
  </style>
</head>
<body>
  <p>This is internal CSS styled text</p>
</body>
```

✅ **When to use:**

- Small websites
- Single page with unique styles

🟣 External CSS

👉 CSS stored in a **separate .css file** and linked using `<link>` in HTML.

✅ Example:

style.css

```
p {
  color: red;
  font-size: 16px;
```

```
}
```

index.html

```
<head>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p>This is external CSS styled text</p>
</body>
```

✅ **When to use:**

- Best for large websites
- Easy to manage and reuse styles across pages

🌟 2️⃣ Multiple Styles (Which one wins?)

👉 If you apply **inline + internal + external** CSS to the same element:

- **Inline CSS wins** (highest priority)
- **Then internal CSS**
- **Then external CSS**

✅ Example:

```
<head>
  <style>
    p { color: green; }
  </style>
  <link rel="stylesheet" href="style.css"> <!-- style.css says: p { color: blue; }
-->
```

```
</head>
<body>
  <p style="color: red;">This will be red because inline wins!</p>
</body>
```

✅ **Priority order (highest to lowest):**

```
1️⃣ Inline style
2️⃣ Internal style
3️⃣ External style
```

👉 And finally, **browser default styles** if no CSS is applied.

🌟 3️⃣ CSS Fonts

👉 CSS lets you control **how text looks** using font-related properties.

✅ **Common font properties:**

Property	Example	What it does
`font-family`	`font-family: Arial, sans-serif;`	Sets the font type
`font-size`	`font-size: 16px;`	Sets size of the text
`font-style`	`font-style: italic;`	Makes text italic
`font-weight`	`font-weight: bold;`	Makes text bold
`line-height`	`line-height: 1.5;`	Sets space between lines

✅ Example:

```
p {
  font-family: 'Times New Roman', serif;
  font-size: 18px;
  font-weight: bold;
  font-style: italic;
```

```
}
```

✅ Example in HTML:

```
<p style="font-family: Arial; font-size: 20px; font-weight: bold;">Styled font e
xample</p>
```

🌟 Example Full HTML with Fonts + Multiple Styles

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="style.css"> <!-- style.css has p { color: blue; } -
->
  <style>
    p { color: green; font-family: Verdana; }
  </style>
</head>
<body>
  <p style="color: red; font-size: 18px;">This is inline styled text (will be red)</
p>
</body>
</html>
```

👉 Even though external CSS says blue, and internal says green → **inline wins → red!**

⚡ Summary

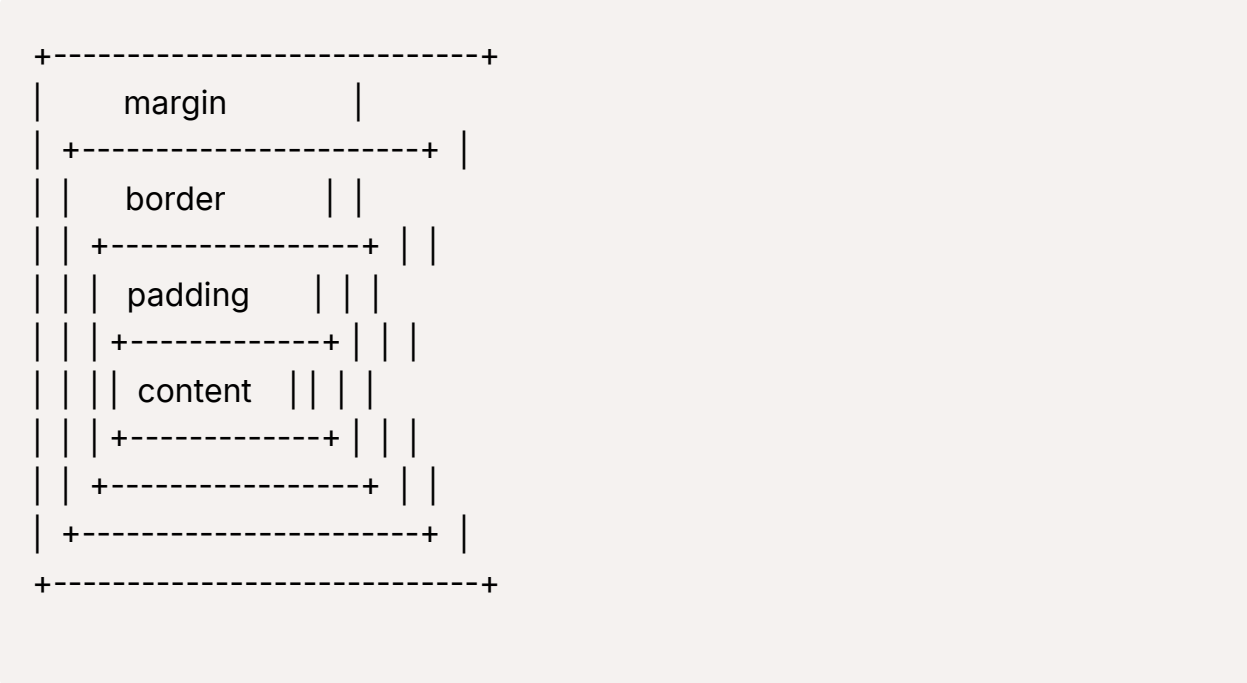
CSS Type	Where applied	Use case
Inline CSS	In the tag using `style=""`	Quick, one-time styles

Internal CSS	In `<style>` tag in `<head>`	Unique styles for 1 page
External CSS	Linked `.css` file	Best for large/multi-page sites
Multiple styles	Inline > Internal > External	Follows this priority order
CSS fonts	Style text (font-family, size, weight, style)	Makes typography attractive

🌟 What is the CSS Box Model?

👉 The **CSS Box Model** is **how browsers see and calculate space around HTML elements**.

👉 Every HTML element is treated as a **rectangular box**, and this box is made of:



✅ The box has **4 parts**:

1️⃣ **Content** → The actual text/image

2️⃣ **Padding** → Space inside the border, around content

3️⃣ **Border** → Line around padding

4️⃣ **Margin** → Space outside the border (space between this element and others)

🔑 Parts of the Box Model

Part	What it does	Example CSS
Content	The actual stuff (text, image, etc.)	`width: 200px; height: 100px;`
Padding	Inside space between content & border	`padding: 10px;`
Border	The border line around the box	`border: 2px solid black;`
Margin	Space outside the border	`margin: 20px;`

💡 Example CSS

```
.box {
  width: 200px;
  height: 100px;
  padding: 10px;
  border: 2px solid black;
  margin: 20px;
}
```

✅ The **total space this box takes up on the page** will be:

```
width = content width + left padding + right padding + left border + right bord
er + left margin + right margin
= 200 + 10 + 10 + 2 + 2 + 20 + 20 = 264px

height = same idea
= 100 + 10 + 10 + 2 + 2 + 20 + 20 = 164px
```

⚠ Box-sizing

By default:

```
box-sizing: content-box;
```

👉 The `width` / `height` includes **only content size** (padding + border + margin added outside).

If you want total width to include padding and border:

```
box-sizing: border-box;
```

👉 Now the `width` / `height` includes padding + border — easier for layout.

✨ Visual example

✅ Without a diagram image, picture:

```
[Margin: empty space outside border]

[Border: visible line around element]

[Padding: empty space inside border, before content]

[Content: the text/image you see]
```

🚀 Sample HTML + CSS

```
<div class="box">Hello Box Model!</div>
```

```
.box {
```

```
  width: 200px;
  padding: 10px;
  border: 3px solid blue;
  margin: 15px;
  background-color: lightyellow;
}
```

👉 You will see:

- Yellow box of content
- Blue border
- Space around the box (margin)

✅ Summary

Part	Position	Function
Content	Inside the box	Actual text, image, etc.
Padding	Between content & border	Adds inside space
Border	Around padding	Draws outline around element
Margin	Outside border	Adds outside spacing between elements

💡 **Tip:** Use `box-sizing: border-box;` → easier to manage sizes because padding/border are inside the width/height.

🌟 What is `id` attribute?

👉 The `id` attribute:

- Gives a **unique name** to a single HTML element.
- You can use it to style, access, or manipulate that element using CSS or JavaScript.
- **No two elements should have the same `id`.**

📌 Example:

```
<p id="special">This is a special paragraph.</p>
```

✅ CSS using `id` :

```
css
Copy code
#special {
  color: blue;
  font-weight: bold;
}
```

👉 The `#` selector in CSS is used for `id`.

🌟 What is `class` attribute?

👉 The `class` attribute:

- Gives a **name to a group of elements**.
- You can apply the same style or behavior to **many elements**.
- **Multiple elements can share the same class.**
- One element can have **multiple classes** too.

📌 Example:

```
<p class="highlight">This is a highlighted paragraph.</p>
<p class="highlight">This is another highlighted paragraph.</p>
```

✅ CSS using `class` :

```css
.highlight {
  background-color: yellow;
  font-size: 18px;
}
```

👉 The `.` selector in CSS is used for `class`.

🔑 Difference between `id` and `class`

Feature	`id`	`class`
Uniqueness	Unique to 1 element	Can be shared by many elements
CSS selector	`#idname`	`.classname`
Use case	Target single unique element	Target multiple elements
Example	`<div id="header">`	`<div class="menu">`

💡 Can you use both together?

✅ Yes! Example:

```html
<p id="mainPara" class="highlight">This is both id and class</p>
```

👉 In CSS:

```css
#mainPara {
  color: red;
}
.highlight {
  background-color: yellow;
}
```

➡️ The text will be **red** with **yellow background**.

🚀 Quick Example

```
<!DOCTYPE html>
<html>
<head>
<style>
#uniqueTitle {
  color: green;
}

.commonPara {
  font-size: 20px;
  color: blue;
}
</style>
</head>
<body>

<h1 id="uniqueTitle">Welcome</h1>
<p class="commonPara">Paragraph 1</p>
<p class="commonPara">Paragraph 2</p>

</body>
</html>
```

👉 The `<h1>` will be **green**

👉 Both `<p>` will be **blue**, size **20px**

✅ Summary

Attribute	Purpose	CSS Selector
`id`	Unique identifier for one element	`#idname`
`class`	Group name for styling multiple elements	`.classname`

💡 **Tip:**

👉 Use `id` for special cases (like one header, one footer).

👉 Use `class` for styles shared by many elements (like buttons, paragraphs).

1. `<style>` Tag

This is used to define **internal CSS** (Cascading Style Sheets) in an HTML document. It's placed within the `<head>` section.

Syntax:

```
<head>
  <style>
    body {
      background-color: lightblue;
    }

    h1 {
      color: navy;
      text-align: center;
    }

    p {
      font-size: 18px;
    }
  </style>
</head>
```

- Styles written here apply to the whole document.
- This is best for small or simple websites.

---

2. Inline Style Attribute

The `style` **attribute** is used inside any HTML tag to apply CSS **directly to that element**.

Example:

```
<h1 style="color: green; font-size: 30px;">Hello, Inline Style!</h1>
```

- Useful for quick changes or overrides.
- Not recommended for large-scale styling due to maintainability issues.

---

3. External Stylesheet Link

Not a style tag itself, but a common way to apply CSS by linking to a `.css` file using:

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

This keeps HTML and CSS separate and organized.

Summary:

Type	Location	Best For
`<style>` tag	Inside `<head>`	Small sites or page-specific CSS
`style` attribute	Inline in tags	Quick one-off styles
`<link>` tag	Inside `<head>`	Reusable external CSS files

✅ 1. External CSS (Recommended)

Link a separate `.css` file using the `<link>` tag inside the `<head>` of your HTML.

🔶 HTML:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Web Page</title>
  <link rel="stylesheet" href="styles.css">
</head>
```

```
<body>
  <h1>Hello World</h1>
</body>
</html>
```

🔶 styles.css:

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

- ✅ Best for large websites
- ✅ Keeps design separate from content
- ✅ Easy to reuse styles

✅ 2. Internal CSS

Use the `<style>` tag inside the `<head>` section.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    h1 {
      color: green;
    }
    p {
      font-family: Arial;
    }
  </style>
```

```
</head>
<body>
  <h1>Welcome</h1>
  <p>This is an internal style.</p>
</body>
</html>
```

- Best for styling a single page only

✅ 3. Inline CSS

Use the `style` attribute **directly on HTML elements**.

Example:

```
<h1 style="color: red; text-align: right;">Inline Styled Heading</h1>
```

- Good for quick styling
- ❌ Not ideal for maintainability

📌 Summary Table:

Method	Where it Goes	Use Case
External CSS	`<link>` in `<head>`	Large sites, reusable styles
Internal CSS	`<style>` in `<head>`	Single page styling
Inline CSS	`style=""` in tag	Quick fixes or one-time changes

🧩 What Is UI Scripting?

UI Scripting is:

- The process of **controlling the UI** of an application through **scripts or code**

- Common in **automated testing** (e.g., Selenium, Cypress)
- Also used in **macros**, **game bots**, **accessibility tools**, and **desktop automation**

🛠️ Why Use UI Scripting?

Purpose	Example Tool	Use Case
🧪 Automated Testing	Selenium, Cypress	Simulate user actions for regression testing
⚙️ Automation	AutoHotKey, AppleScript	Automate UI workflows on desktop apps
🔍 Accessibility	UI Automation APIs	Help users with disabilities navigate UIs
🎮 Game Macros	AutoIt, Python PyAutoGUI	Perform repeated in-game actions

---

🧪 UI Scripting in Web Development

For **web applications**, scripting often happens through:

- **JavaScript** (DOM manipulation, event handling)
- **Testing libraries** (e.g., Puppeteer, Playwright)

Example (JavaScript):

```
// Click a button with JavaScript
document.getElementById("submitBtn").click();
```

🖥️ UI Scripting in Desktop Apps

✅ JavaScript Code (Equivalent to PyAutoGUI)

```
const robot = require("robotjs");
```

```
// Move mouse to x:100, y:100
robot.moveMouse(100, 100);

// Perform a mouse click
robot.mouseClick();

// Type out "Hello World"
robot.typeString("Hello World");
```

🔍 What this does:

Function	Description
`robot.moveMouse(x, y)`	Moves the mouse to the given coordinates
`robot.mouseClick()`	Performs a left-click
`robot.typeString(str)`	Types out a string like keyboard input

⚠️ Note:

- `robotjs` works on **Windows, macOS, and Linux**.
- Some antivirus or OS settings may block scripting tools.
- You must run the script with permission to control the mouse/keyboard.

💡 Use Case Examples:

- Automate opening and filling desktop apps
- Auto-login scripts
- Automated data entry or GUI testing

🔐 Key Concepts in UI Scripting

Concept	Meaning
Element Locators	How scripts find UI elements (e.g., ID, class, XPath)

Events	Actions like `click`, `hover`, `input`, etc.
Wait/Timeout	Wait for UI to load before acting
Assertions	Check if the expected outcome is met

📦 Popular Tools for UI Scripting

Tool	Platform	Use
Selenium	Web	UI Testing (Browser)
Puppeteer	Web (Chrome)	Headless browser testing
AutoHotKey	Windows	Desktop UI Automation
PyAutoGUI	Cross-platform	Mouse & keyboard control
Playwright	Web	Modern UI testing

✅ Conclusion:

UI Scripting bridges the gap between **manual UI interaction** and **automation**. Whether you're testing a web app or automating a desktop workflow, UI scripting is a powerful way to **interact with the user interface programmatically**.

🌐 Bootstrap Overview

Bootstrap is a popular **open-source front-end framework** used to design **responsive** and **mobile-first websites** quickly and easily.

🔧 It includes:

- HTML templates
- CSS stylesheets
- JavaScript components

📦 Created By:

- Twitter (initially in 2011)
- Currently maintained as Bootstrap v5+

💡 Why Use Bootstrap? (The Need)

Feature	Benefit
✅ **Responsive Design**	Adapts to all screen sizes (mobile, tablet, desktop)
✅ **Pre-designed Components**	Ready-to-use UI elements like navbars, buttons, modals
✅ **Grid System**	Layout system using rows and columns (12-column layout)
✅ **Cross-browser Compatibility**	Works consistently across all major browsers
✅ **Faster Development**	Saves time with reusable code and components
✅ **Customizable**	You can override or extend Bootstrap styles easily

🚀 How Bootstrap Works

Bootstrap can be used in 2 main ways:

🔶 1. Via CDN (No Download Needed)

```
<!-- Add this in the <head> section of your HTML -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
```

🔶 2. Download and Use Locally

You can download Bootstrap and include the files in your project.

🎨 Example: Button in Bootstrap

```
<button class="btn btn-primary">Click Me</button>
```

- `btn` = base button style
- `btn-primary` = blue theme

📱 Example: Responsive Grid Layout

```
<div class="container">
  <div class="row">
    <div class="col-md-6">Left Column</div>
    <div class="col-md-6">Right Column</div>
  </div>
</div>
```

This splits the layout into 2 columns on medium+ screens, and stacks them on smaller screens.

🧰 Key Components in Bootstrap

Component	Use Case
Navbar	Site navigation bar
Card	Display content in boxes
Modal	Pop-up windows
Carousel	Image sliders
Form Controls	Styled form inputs
Alerts	Messages and warnings

📌 Summary

- **Bootstrap** makes web design faster, consistent, and responsive.
- It’s beginner-friendly and powerful for developers.
- You can use it via **CDN** or by downloading files.
- Ideal for **prototypes**, **dashboards**, and **full websites**.

📐 Bootstrap Grid System

The **Bootstrap Grid System** is a flexible, responsive layout system based on **12 columns**. It allows you to build complex layouts by dividing a page into rows and columns.

✅ Key Features:

- Based on a **12-column** layout
- Fully **responsive** (adjusts across devices)
- Built using **Flexbox**
- Easy to **nest** columns

🧱 Basic Structure of a Bootstrap Grid

```
<div class="container">        <!-- Fixed-width container -->
  <div class="row">            <!-- Row -->
    <div class="col">Column 1</div>  <!-- Column -->
    <div class="col">Column 2</div>
  </div>
</div>
```

OR with specific column sizes:

```
<div class="container">
  <div class="row">
    <div class="col-6">Left (6 columns)</div>
    <div class="col-6">Right (6 columns)</div>
  </div>
</div>
```

🏷️ Bootstrap Grid Classes

🔷 Column Width Classes (Fixed)

Class	Description
`.col-1` to `.col-12`	Fixed number of columns (out of 12)
`.col`	Auto-sizes based on available space

🔷 Responsive Grid Classes

Class	Screen Size	Usage Example
`col-sm-`	≥576px (small)	`<div class="col-sm-6">`
`col-md-`	≥768px (medium)	`<div class="col-md-4">`
`col-lg-`	≥992px (large)	`<div class="col-lg-3">`
`col-xl-`	≥1200px (extra large)	`<div class="col-xl-2">`
`col-xxl-`	≥1400px	`<div class="col-xxl-1">`

🧩 Example: Responsive Grid

```
<div class="container">
  <div class="row">
    <div class="col-sm-12 col-md-6">Left Column</div>
    <div class="col-sm-12 col-md-6">Right Column</div>
  </div>
</div>
```

- On small screens: Columns stack (`col-sm-12`)
- On medium+ screens: 2 equal columns (`col-md-6` + `col-md-6`)

🧠 Important Grid Classes Summary

Class	Purpose
`.container`	Fixed-width layout wrapper
`.container-fluid`	Full-width layout across screen
`.row`	Creates a horizontal group of columns
`.col`	Automatically sized column
`.col-4` , `.col-8`	Defines column width out of 12
`.g-3` , `.gx-4` , `.gy-2`	Add grid spacing (gutters)

⬅️ END Summary:

- Bootstrap Grid = 12 columns + responsive design
- Use `.container` , `.row` , and `.col-*` classes
- Supports different layouts for different screen sizes
- Makes page layout fast and mobile-friendly

✍️ Bootstrap Typography

Typography in Bootstrap refers to the **predefined styles** for text, including **headings**, **paragraphs**, **alignment**, **fonts**, **text utilities**, and more. It helps you format content cleanly without writing custom CSS.

🔠 1. Headings

Bootstrap supports all HTML headings (`<h1>` to `<h6>`) with proper font sizes and spacing.

Example:

```
<h1>h1 Heading</h1>
<h2>h2 Heading</h2>
<h3>h3 Heading</h3>
```

Custom Heading Class:

```
<p class="h1">Bootstrap styled h1</p>
<p class="h2">Bootstrap styled h2</p>
```

📄 2. Paragraphs and Lead Text

Regular Paragraph:

```
<p>This is a normal paragraph.</p>
```

Lead Paragraph (larger and lighter):

```
<p class="lead">This is a lead paragraph. It stands out more.</p>
```

🖌️ 3. Font Weight and Style

Bootstrap provides utility classes to style font weight and italics.

Class	Description
`.fw-bold`	Bold text
`.fw-semibold`	Semi-bold
`.fw-normal`	Normal weight
`.fw-light`	Light font
`.fst-italic`	Italic text
`.fst-normal`	Remove italic

```
<p class="fw-bold">Bold text</p>
<p class="fst-italic">Italic text</p>
```

📐 4. Text Alignment

Class	Alignment
`.text-start`	Left
`.text-center`	Center
`.text-end`	Right

```
<p class="text-center">Centered Text</p>
```

🌈 5. Text Colors and Background

Text Color Class	Background Color Class
`.text-primary`	`.bg-primary`
`.text-success`	`.bg-success`
`.text-danger`	`.bg-danger`
`.text-muted`	Muted (gray)

```
<p class="text-success">Success message</p>
<p class="bg-warning text-dark">Warning with dark text</p>
```

🔤 6. Text Transformations

Class	Effect
`.text-uppercase`	UPPERCASE
`.text-lowercase`	lowercase
`.text-capitalize`	Capitalize Each Word

```
<p class="text-uppercase">uppercase text</p>
```

💡 7. Display Headings

For larger display titles, Bootstrap offers:

```
<h1 class="display-1">Display 1</h1>
<h1 class="display-4">Display 4</h1>
```

- `display-1` to `display-6`: Larger and bolder than normal headings

⬅️ Summary

Feature	Class or Tag Example
Headings	`<h1>` or `.h1`
Paragraphs	`<p>`, `.lead`
Font weight/style	`.fw-bold`, `.fst-italic`
Alignment	`.text-center`, `.text-end`
Color/Background	`.text-danger`, `.bg-info`
Transformation	`.text-uppercase`
Display headings	`.display-1` to `.display-6`

Components of bootstap

🔷 1. Tables

Bootstrap enhances tables with `.table` class for clean and styled layouts.

Basic Example:

```
<table class="table">
  <thead>
    <tr><th>#</th><th>Name</th><th>Email</th></tr>
  </thead>
  <tbody>
    <tr><td>1</td><td>Alice</td><td>alice@example.com</td></tr>
  </tbody>
</table>
```

Other Classes:

- `.table-striped`: Alternating row colors
- `.table-bordered`: Adds borders to all cells
- `.table-hover`: Adds hover effect
- `.table-condensed`: Reduces padding

🔷 2. Images

Bootstrap provides responsive image styles.

Classes:

- `.img-responsive` (v3) / `.img-fluid` (v4+): Makes image scale with parent
- `.img-thumbnail`: Adds border and padding like a photo frame
- `.rounded`, `.rounded-circle`: Rounded corners / circular

```
<img src="img.jpg" class="img-fluid rounded" alt="Responsive Image">
```

🔷 3. Badges / Labels

Used to display counters or small highlights.

Bootstrap 3:

```
<span class="label label-success">New</span>
```

Bootstrap 4+:

```
<span class="badge bg-success">New</span>
```

Can be used inside buttons too:

```
<button type="button" class="btn btn-primary">
  Messages <span class="badge bg-light text-dark">4</span>
</button>
```

🔷 4. Progress Bars

Shows progress (e.g., file upload).

Additional Classes:

- `.progress-bar-striped`
- `.progress-bar-animated` (v4+)
- `.bg-success`, `.bg-danger`, etc.

🔷 5. Pagination

Pagination is a technique used in websites and applications to **divide large sets of data or content into smaller, manageable pages**

Adds paginated navigation links.

```
<ul class="pagination">
  <li class="page-item"><a class="page-link" href="#">Previous</a></li>
  <li class="page-item"><a class="page-link" href="#">1</a></li>
</ul>
```

Variants:

- `.pagination-lg`, `.pagination-sm`
- `.active`, `.disabled` classes on `<li>`

🔷 6. List Groups

Used for menus, sidebars, task lists, etc.

```
<ul class="list-group">
  <li class="list-group-item active">Dashboard</li>
  <li class="list-group-item">Profile</li>
</ul>
```

With badges:

```
<li class="list-group-item d-flex justify-content-between align-items-center">
  Inbox
  <span class="badge bg-primary">14</span>
</li>
```

🔷 7. Panels (Bootstrap 3 Only – replaced by Cards in v4+)

Used for boxes/containers.

```
<div class="panel panel-default">
  <div class="panel-heading">Panel Heading</div>
  <div class="panel-body">Panel content</div>
</div>
```

🔷 8. Dropdowns

Used for menus and options.

```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle" data-bs-toggle="drop
down">
    Dropdown
  </button>
  <ul class="dropdown-menu">
    <li><a class="dropdown-item" href="#">Action</a></li>
  </ul>
</div>
```

🔷 9. Collapse

Expands or collapses content using JavaScript.

```
<a data-bs-toggle="collapse" href="#collapseExample">Toggle</a>
<div class="collapse" id="collapseExample">
  <div class="card card-body">
    Collapsible content.
  </div>
```

```
</div>
```

🔷 10. Tabs / Pills

Navigation between tabbed content.

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link active" data-bs-toggle="tab" href="#home">Home</a>
  </li>
</ul>

<div class="tab-content">
  <div class="tab-pane fade show active" id="home">Tab 1 content</div>
</div>
```

`.nav-pills` is an alternative to `.nav-tabs` with pill-style look.

🔷 11. Navbar

Used for headers, menus, search bars.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Brand</a>
</nav>
```

Expandable + responsive features: Use `.navbar-toggler`, `.collapse`, and `.navbar-collapse`.

🔷 12. Jumbotron (Bootstrap 3/4)

Large box for highlighting content (replaced by utility classes in v5).

```
<div class="jumbotron">
  <h1>Welcome!</h1>
  <p>This is a simple hero unit.</p>
</div>
```

🔷 13. Wells (Bootstrap 3 Only)

Used for inset content areas.

```
<div class="well">Look! I'm in a well!</div>
```

(In Bootstrap 4/5, use `card` or spacing utilities instead.)

🔷 14. Alerts

Used for feedback messages.

```
<div class="alert alert-success" role="alert">
  Successfully saved!
</div>
```

Types:

- `alert-primary`, `alert-danger`, `alert-warning`, etc.

🔷 15. Buttons

Bootstrap has several button styles.

```
<button class="btn btn-primary">Primary</button>
```

Variants:

- `.btn-secondary`, `.btn-success`, `.btn-danger`, etc.
- Sizes: `.btn-lg`, `.btn-sm`
- States: `.disabled`, `.active`

🔷 16. Button Groups

Group related buttons together.

```
<div class="btn-group" role="group">
  <button class="btn btn-primary">Left</button>
  <button class="btn btn-primary">Middle</button>
  <button class="btn btn-primary">Right</button>
</div>
```

✅ Summary Table

Component	Bootstrap Class(es) Used
Tables	`.table`, `.table-striped`, `.table-bordered`
Images	`.img-fluid`, `.img-thumbnail`, `.rounded`
Badges/Labels	`.badge`, `.label`
Progress Bars	`.progress`, `.progress-bar`
Pagination	`.pagination`, `.page-item`, `.page-link`
List Groups	`.list-group`, `.list-group-item`
Panels	`.panel`, `.panel-heading` (v3 only)
Dropdowns	`.dropdown`, `.dropdown-toggle`, `.dropdown-menu`
Collapse	`.collapse`, data attributes
Tabs/Pills	`.nav`, `.nav-tabs`, `.tab-content`
Navbar	`.navbar`, `.navbar-brand`, `.navbar-nav`
Jumbotron	`.jumbotron` (deprecated in v5)

Wells	`.well` (v3 only)
Alerts	`.alert` , `.alert-dismissible`
Buttons	`.btn` , `.btn-primary` , `.btn-sm`
Button Groups	`.btn-group`

📄 Bootstrap Forms & Inputs – Theory

🔷 What is a Form?

A **form** is a section of a webpage that allows users to **input data** that can be sent to a server for processing. Bootstrap enhances the appearance and behavior of HTML forms using **predefined classes** for structure, spacing, validation, and responsiveness.

✅ Purpose of Using Bootstrap for Forms

- Provides a **clean and consistent layout** across browsers.
- Supports **responsive behavior** (adapts to screen sizes).
- Reduces the need to write CSS manually.
- Built-in **form validation**, **input styling**, and **layout options**.

🧱 Basic Structure of a Bootstrap Form

```
<form>
  <div class="mb-3">
    <label for="exampleInputEmail" class="form-label">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail">
  </div>
</form>
```

🧩 Key Bootstrap Classes for Forms

Class	Purpose
`form-control`	Styles inputs, textareas, selects (full width, padding, border).
`form-label`	Styles labels associated with form controls.
`form-check`	Used for checkboxes and radio buttons.
`form-check-input`	Styles input checkbox/radio.
`form-check-label`	Styles label next to checkbox/radio.
`form-select`	Styles `<select>` dropdowns.
`form-text`	Used for helper text below input fields.
`form-group` (v4)	Wraps label + input together (v5 uses spacing utilities like `mb-3`).

 Types of Input Controls

Control Type	Description	Bootstrap Class
Text Input	Single-line input	`form-control`
Password	Hidden characters	`form-control`
Email	Validates email format	`form-control`
Textarea	Multi-line input	`form-control`
Checkbox	Binary option	`form-check-input`
Radio	Single choice in a group	`form-check-input`
Select	Drop-down list	`form-select`
File Upload	File picker	`form-control`

 Form Layouts in Bootstrap

1. Vertical (Default)

Stacked inputs.

```
<form>
  <div class="mb-3">
    <label class="form-label">Name</label>
    <input type="text" class="form-control">
```

```
  </div>
</form>
```

2. Horizontal Forms

Label and input in the same row (often in columns).

```
<form class="row g-3">
  <div class="col-md-4">
    <label class="form-label">First Name</label>
    <input type="text" class="form-control">
  </div>
</form>
```

3. Inline Forms

Compact form elements in a single line.

```
<form class="row row-cols-lg-auto g-3 align-items-center">
  <input type="text" class="form-control" placeholder="Search">
  <button type="submit" class="btn btn-primary">Go</button>
</form>
```

🛑 Form Validation (Visual Feedback)

Bootstrap provides built-in styles for **client-side validation** using classes like:

Class	Use
`is-valid`	Marks input as valid (green border)
`is-invalid`	Marks input as invalid (red border)
`invalid-feedback`	Shows error message
`valid-feedback`	Shows success message

Example:

```
<input type="text" class="form-control is-invalid">
<div class="invalid-feedback">This field is required.</div>
```

🎨 Custom Form Controls

- Custom checkboxes/radios:

```
<div class="form-check">
  <input class="form-check-input" type="checkbox" id="check1">
  <label class="form-check-label" for="check1">I agree</label>
</div>
```

- Custom range sliders:

```
<input type="range" class="form-range">
```

- Switches:

```
<div class="form-check form-switch">
  <input class="form-check-input" type="checkbox" role="switch">
  <label class="form-check-label">Toggle Me</label>
</div>
```

📋 Summary Table

Feature	Bootstrap Class	Purpose
Text Input	`form-control`	Default styling for inputs
Label	`form-label`	Input label
Group Spacing	`mb-3` , `form-group` (v4)	Margin between fields
Checkbox/Radio	`form-check` , `form-check-input` , `form-check-label`	Styled options

Select Dropdown	`form-select`	Style `<select>` elements
Validation	`is-valid`, `is-invalid`	Show feedback styles
Inline Form	`row row-cols-lg-auto` etc.	Compact layout

🔷 What is a Bootstrap Theme?

A **Bootstrap Theme** is a pre-designed visual style built on top of the Bootstrap framework. It customizes the **look and feel** of the Bootstrap components (buttons, navbars, forms, etc.) by applying unique styles like:

- Colors
- Fonts
- Spacing
- Layouts
- Animations

Think of it like changing the **skin** of your Bootstrap UI while keeping the **functionality and structure** the same.

✅ Why Use a Theme?

- Saves **design time**
- Ensures **visual consistency**
- Makes your site look **professional**
- Offers **brand customization** (fonts, palettes, etc.)

📁 What is a Bootstrap Template?

A **Bootstrap Template** is a **ready-made webpage or web app layout** created using Bootstrap.

It includes:

- Full HTML structure
- Styled Bootstrap components
- Predefined navigation, header, footer
- Often includes JS plugins and demo content

✅ Why Use a Template?

- Jump-starts development
- Used for specific purposes like:
  - Admin dashboards
  - Portfolios
  - E-commerce pages
  - Blogs or personal websites
- Can be **customized** easily

🎯 Difference Between Themes and Templates

Feature	Theme	Template
Purpose	Changes appearance (style)	Pre-built page layout
Includes HTML?	Not always (mostly CSS/SCSS files)	Yes, full HTML with Bootstrap components
Includes JS?	Usually no	Often includes extra JS scripts
Usage	Applied over any Bootstrap project	Used as a starting point for a full site
Examples	Bootswatch, Custom UI themes	AdminLTE, Start Bootstrap, Creative

---

Here's a **clear and detailed theoretical explanation** of the fundamental concepts of **JavaScript**, structured for study, teaching, or documentation.

🟨 Introduction to JavaScript

📌 What is JavaScript?

JavaScript is a **high-level, interpreted, lightweight programming language** primarily used to **add interactivity, logic, and dynamic behavior** to websites. It is a **core technology** of the web, alongside HTML and CSS.

✅ Key Features:

- Runs in the browser (client-side)
- Can also run on servers (Node.js)
- Used for DOM manipulation, event handling, form validation, animations, etc.
- Dynamically typed language
- Supports object-oriented, functional, and imperative programming styles

---

🟩 JavaScript Variables

📌 What is a Variable?

A **variable** is a container that holds data that can be changed later.

Hoisting-

> JavaScript remembers the declaration of variables and functions before running any code, even if you wrote them at the bottom of the file.

🎯 Variable Keywords:

Keyword	Scope Type	Reassignable?	Hoisting	Introduced In
`var`	Function	Yes	Yes	ES5
`let`	Block	Yes	No	ES6
`const`	Block	No	No	ES6

📝 Syntax:

```
var name = "Alice";
let age = 25;
const country = "India";
```

🟦 JavaScript Statements

📌 What is a Statement?

A **JavaScript statement** is a command that tells the browser to perform an action.

🔷 Examples:

```
let x = 5;              // Assignment Statement
console.log(x);         // Function call statement
if (x > 0) { ... }      // Conditional statement
```

👉 Statements end with a **semicolon** `;` (optional but recommended).

🟥 JavaScript Operators

📌 What is an Operator?

An **operator** performs a specific operation on operands (values/variables).

The `typeof` operator is a **unary operator** in JavaScript that is used to **determine the data type** of a given value or variable.

It returns a **string** that represents the **type of the operand**.

```
typeof 123          // "number"
typeof "hello"      // "string"
typeof true         // "boolean"
typeof undefined    // "undefined"
typeof null         // "object"  ← (This is a known JS quirk)
```

```
typeof [1, 2, 3]      // "object"
typeof {a: 1}         // "object"
typeof function() {}  // "function"
```

The `===` operator in JavaScript is called the **strict equality operator**.

> It checks whether both the value and the data type are the same without doing any type conversion.

Expression	Result	Explanation
`5 === 5`	`true`	Same value and type (number)
`"5" === 5`	`false`	String vs. number (different type)
`true === 1`	`false`	Boolean vs. number

🔷 Types of Operators:

Type	Examples	Description
Arithmetic	`+`, `-`, `*`, `/`, `%`	Math operations
Assignment	`=`, `+=`, `-=`	Assign values
Comparison	`==`, `===`, `!=`, `>`, `<`	Compare values
Logical	`&&`, `	
String	`+` (concatenation)	Join strings
Unary	`typeof`, `++`, `--`	One operand only
Ternary	`condition ? val1 : val2`	Conditional expression

🟨 JavaScript Comments

📌 What is a Comment?

Comments are non-executable lines used to explain code, improve readability, or disable code during debugging.

📝 Syntax:

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

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

🟪 JavaScript Expressions

📌 What is an Expression?

An **expression** is a code fragment that evaluates to a value.

🔷 Examples:

```
3 + 4       // Arithmetic expression ⇒ 7
"Hi" + "JS"  // String expression ⇒ "HiJS"
x > 5       // Boolean expression ⇒ true or false
```

💡 Expression vs. Statement:

- **Expression** produces a value.
- **Statement** performs an action.

🟫 JavaScript Control Structures

📌 What are Control Structures?

Control structures are constructs that control the **flow of execution** of code based on conditions or repetition.

🔷 1. Conditional Statements

Used for decision-making.

Type	Syntax
if	`if (condition) { }`
if-else	`if (...) { } else { }`
else-if	`if (...) { } else if (...) { }`
switch-case	`switch(value) { case 'A': ... }`

```
if (score > 50) {
  console.log("Passed");
} else {
  console.log("Failed");
}
```

🔷 2. Loops

Used for repetition.

Loop Type	Description
`for`	Loop with a counter
`while`	Loop while condition is true
`do-while`	Executes at least once
`for...of`	Iterates over array values
`for...in`	Iterates over object keys

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

🧠 Summary Table

Concept	Description
JavaScript	Scripting language for web interactivity
Variables	Used to store data (`var` , `let` , `const`)

Statements	Instructions that perform actions
Operators	Symbols to perform operations on data
Comments	Code annotations ignored by the browser
Expressions	Code that evaluates to a value
Control Structures	Structures to control code execution flow

🔷 JavaScript Scopes

📌 What is Scope?

Scope refers to the **visibility or accessibility** of variables in different parts of your code.

🔷 Types of Scope in JavaScript:

Scope Type	Description
Global Scope	Variables declared **outside** any function or block. Available everywhere.
Function Scope	Variables declared **inside a function** using `var`. Accessible only within that function.
Block Scope	Variables declared with `let` or `const` **inside blocks** (`{}`) are limited to that block.

✅ Example:

```
var globalVar = "I'm global"; // Global scope

function testScope() {
  var functionVar = "I'm inside function"; // Function scope
  if (true) {
    let blockVar = "I'm block-scoped";     // Block scope
    console.log(blockVar);                 // ✅ Accessible here
  }
  // console.log(blockVar); ❌ Error
}
```

🔶 Strings & String Methods

📌 What is a String?

A **string** is a **sequence of characters** used to represent text.

```
let name = "JavaScript";
```

🔷 Common String Methods:

Method	Description	Example
`length`	Returns length of the string	`"abc".length → 3`
`toUpperCase()`	Converts to uppercase	`"abc".toUpperCase()`
`toLowerCase()`	Converts to lowercase	`"ABC".toLowerCase()`
`charAt(index)`	Returns character at given index	`"hello".charAt(1) → 'e'`
`includes(str)`	Checks if string contains substring	`"code".includes("od")`
`indexOf(str)`	Returns index of first occurrence	`"hello".indexOf("l") → 2`
`slice(start, end)`	Extracts a part of string	`"hello".slice(0, 2) → 'he'`
`replace(old, new)`	Replaces substring with another	`"abc".replace("a", "z") → 'zbc'`
`trim()`	Removes whitespace from both ends	`" hi ".trim() → 'hi'`
`split(separator)`	Converts string to array	`"a,b,c".split(",") → ['a', 'b', 'c']`

🔶 Numbers & Number Methods

📌 What is a Number?

JavaScript has **only one numeric type**: both integers and floats are treated as **Number**.

```
let a = 10;
let b = 5.5;
```

🔷 Common Number Methods/Properties:

Method / Property	Description	Example
`toFixed(n)`	Rounds number to `n` decimal places	`3.14159.toFixed(2) → '3.14'`
`toString()`	Converts number to string	`(123).toString()`
`parseInt(str)`	Converts string to integer	`parseInt("123px") → 123`
`parseFloat(str)`	Converts string to float	`parseFloat("3.14") → 3.14`
`Number.isNaN(val)`	Checks if value is NaN	`Number.isNaN("hi") → false`
`Number.isInteger(val)`	Checks if value is an integer	`Number.isInteger(5) → true`
`Math.round()`	Rounds to nearest integer	`Math.round(2.6) → 3`
`Math.floor()`	Rounds down	`Math.floor(2.9) → 2`
`Math.ceil()`	Rounds up	`Math.ceil(2.1) → 3`
`Math.random()`	Returns random number between 0–1	`Math.random()`

🔶 Boolean Values

📌 What is a Boolean?

A **Boolean** represents a logical entity with **only two values**:

- `true`
- `false`

```
let isAvailable = true;
```

🔷 Falsy Values in JavaScript:

These values evaluate to `false` when used in a boolean context:

- `false`
- `0`
- `""` (empty string)
- `null`

- `undefined`
- `NaN`

🔷 Truthy Values:

All other values (non-empty strings, numbers except 0, objects, arrays) evaluate to `true`.

✅ Boolean in Conditions:

```
let age = 18;
if (age >= 18) {
  console.log("Adult"); // true condition
}
```

🔷 Boolean Functions:

Method	Description
`Boolean(value)`	Converts a value to true/false
`!!value`	Double NOT for conversion (e.g., `!!"hello"` → `true`)

🧠 Summary Table

Concept	Description	Example
Scope	Where variables are accessible	Global, Function, Block
String	Text data, many methods for use	`slice()`, `replace()`
Number	Integer/decimal, rounded or parsed	`parseFloat()`, `toFixed()`
Boolean	Logic: `true` or `false`	Used in `if`, comparisons

🗓️ JavaScript Dates

📌 What is a Date?

JavaScript provides the `Date` **object** to work with **dates and times**. It can represent:

- Current date/time
- Custom date/time
- Time difference (duration)

🛠️ Creating a Date Object

```
let now = new Date();                  // current date & time
let date1 = new Date("2024-12-25");    // specific date
let date2 = new Date(2025, 5, 24);     // year, month (0-indexed), day
let date3 = new Date(0);               // Jan 1, 1970 (Unix Epoch)
```

📆 Common Date Formats

Format	Example
ISO Date	`"2025-06-24"`
Short Date (MM/DD/YYYY)	`"06/24/2025"`
Long Date	`"June 24, 2025"`
Full DateTime (ISO)	`"2025-06-24T10:00:00Z"`
Milliseconds since Jan 1, 1970	`Date.now()`

🔧 Useful Date Methods

Method	Description	Example
`getFullYear()`	Gets the 4-digit year	`d.getFullYear()`
`getMonth()`	Gets the month (0–11)	`d.getMonth()`
`getDate()`	Gets the day of the month (1–31)	`d.getDate()`
`getDay()`	Gets day of week (0=Sunday to 6=Sat)	`d.getDay()`

`getHours(), getMinutes()`	Time parts	`d.getHours()`
`setFullYear(), setMonth()`	Modify date	`d.setMonth(11)`
`toDateString()`	Returns date as readable string	`"Tue Jun 24 2025"`
`toISOString()`	Returns ISO format	`"2025-06-24T09:00:00.000Z"`
`Date.now()`	Returns current time in ms	`1719210000000`

📚 JavaScript Arrays

📌 What is an Array?

An **array** is a special object that holds **a list of values** (called elements), which can be of **any data type**.

```
let fruits = ["apple", "banana", "cherry"];
let mixed = [1, "two", true, [4, 5]];
```

- Arrays are **zero-indexed**
- Dynamic in size
- Can contain any type of element

🔧 Common Array Methods

Method	Description	Example
`push()`	Add to end	`arr.push("item")`
`pop()`	Remove last item	`arr.pop()`
`shift()`	Remove first item	`arr.shift()`
`unshift()`	Add to beginning	`arr.unshift("start")`
`length`	Returns number of elements	`arr.length`
`indexOf()`	Finds index of element	`arr.indexOf("apple")`
`includes()`	Checks if element exists	`arr.includes("banana")`
`slice(start, end)`	Extracts part of array	`arr.slice(1, 3)`

`splice(start, count)`	Remove/replace items	`arr.splice(1, 2)`
`concat()`	Combines arrays	`arr1.concat(arr2)`
`join()`	Converts array to string	`arr.join(", ")`
`reverse()`	Reverses the array	`arr.reverse()`
`sort()`	Sorts elements (lexicographically)	`arr.sort()`
`map()`	Transforms each element	`arr.map(x ⇒ x * 2)`
`filter()`	Filters based on condition	`arr.filter(x ⇒ x > 10)`
`forEach()`	Runs a function for each element	`arr.forEach(alert)`
`reduce()`	Reduces to a single value (e.g., sum)	`arr.reduce((a, b) ⇒ a + b)`

🧠 Summary Table

Concept	Description
Date	Used for date/time manipulation
Date Methods	Get/set parts of date, format output
Array	Ordered collection of elements
Array Methods	Modify, loop, filter, transform arrays

🧰 JavaScript Objects – Full Theory

🟦 What is an Object in JavaScript?

In JavaScript, an **object** is a **collection of key-value pairs**, where:

- Keys are called **properties**
- Values can be **data (primitive or complex)** or **functions** (called **methods**)

Objects allow you to group related data and behaviors into a single structure.

🔷 Object Definitions

✅ 1. Object Literal (most common)

```
let person = {
  name: "Alice",
  age: 25,
  isStudent: true
};
```

✅ 2. Using `new Object()`

```
let person = new Object();
person.name = "Alice";
person.age = 25;
```

✅ 3. Using Constructor Functions

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
let p1 = new Person("Bob", 30);
```

✅ 4. Using ES6 Classes

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
let p2 = new Person("Charlie", 28);
```

🔶 Object Properties

Properties are **key-value pairs** inside an object.

```
let car = {
  brand: "Toyota",
  model: "Camry",
  year: 2020
};
```

🔧 Accessing Properties:

```
car.brand       // Dot notation
car["model"]    // Bracket notation (useful for dynamic keys)
```

✏️ Modifying Properties:

```
car.year = 2022;
car["color"] = "blue";
```

❌ Deleting Properties:

```
delete car.year;
```

🔶 Object Methods

A **method** is a **function stored as a property** of an object.

```
let user = {
  name: "John",
  greet: function () {
    return "Hello, " + this.name;
  }
};
```

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

🔷 Shorthand Method Syntax (ES6):

```
let user = {
  name: "John",
  greet() {
    return `Hello, ${this.name}`;
  }
};
```

🔷 `this` Keyword

In object methods, `this` refers to the **object calling the method**.

```
let dog = {
  name: "Bruno",
  speak() {
    return this.name + " says Woof!";
  }
};
```

🧬 Object Prototypes

📌 What is a Prototype?

Every JavaScript object has an internal property called `[[Prototype]]`, which points to **another object**.

This forms the **Prototype Chain**, which is used for **inheritance**.

✅ Example:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

console.log(rabbit.eats); // true — inherited from animal
```

🧠 Prototype Inheritance Summary:

Concept	Explanation
`__proto__`	Links to prototype object
`Object.getPrototypeOf(obj)`	Gets the prototype
`Object.setPrototypeOf(obj, proto)`	Sets prototype
Inherited Properties	If a property is not found on object, JS looks up the prototype chain

🏗️ Defining Methods on Prototypes

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  return "Hi, I'm " + this.name;
};

let p = new Person("Tom");
```

```
p.greet(); // "Hi, I'm Tom"
```

This approach saves memory by creating the method once and sharing it among all instances.

🔚 Summary Table

Concept	Description
Object	A key-value container
Property	Variable inside an object
Method	Function inside an object
this	Refers to the calling object
Prototype	Inheritance mechanism
Prototype Chain	Series of linked objects

🔧 JavaScript Functions – Theory Guide

🟢 1. What is a Function?

A **function** is a **block of reusable code** designed to perform a specific task.

You define it **once**, and **reuse it** as many times as needed — with different values (parameters).

🔷 2. Function Definitions

✅ Function Declaration (Named Function)

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

- **Hoisted**: Can be used before it's declared.

✅ Function Expression

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

- Stored in a variable.
- **Not hoisted** like declarations.

✅ Arrow Function (ES6+)

```
const greet = () => {
  console.log("Hey!");
};
```

- Shorter syntax.
- No own `this` keyword (important for closures and objects).

🔶 3. Function Parameters & Arguments

Parameters = placeholders in definition

Arguments = real values passed when calling

```
function add(a, b) {    // a and b are parameters
  return a + b;
}
add(5, 3);            // 5 and 3 are arguments
```

Default Parameters

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

greet(); // Hello, Guest
```

Rest Parameters

The **Rest Parameters** syntax allows a **function to accept an indefinite number of arguments** as an **array**.

It is used when you don't know how many arguments will be passed to the function.

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b, 0);
}

sum(1, 2, 3); // 6
```

🔷 4. Function Invocation (Calling a Function)

📌 How to call a function:

```
greet();          // Simple call
person.sayHello();  // Method call on object
add(4, 5);        // With arguments
```

📌 Function Invocation Types

Type	Example	Notes
Function call	`greet()`	Simple call

Method call	`obj.method()`	`this` refers to object
Constructor call	`new Person()`	Creates instance
`call` / `apply` / `bind`	`func.call(obj)`	Set custom `this` context

🧠 5. Closures in JavaScript

📌 What is a Closure?

A **closure** is a function that **remembers and has access to variables from its outer scope**, even after the outer function has finished executing.

✅ Example of Closure:

```
function outer() {
  let count = 0;

  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer();  // outer runs once
counter(); // 1
counter(); // 2
counter(); // 3
```

Here, `inner` has access to `count`, **even though** `outer` **is finished**. This is the essence of a **closure**.

🔍 Why Closures Matter

Use Case	Explanation

Data Privacy	Keep variables hidden from outside
Function Factories	Create custom functions with preset data
Event Handlers	Maintain access to original state
Stateful Functions	Remember values across function calls

🧠 Summary Table

Concept	Description
Function	Reusable block of code
Declaration vs Expression	Named (hoisted) vs Assigned (not hoisted)
Parameters/Arguments	Inputs and values
Invocation	Calling a function
Closure	Function that "remembers" its outer scope

🔷 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm centered around **objects**, which are entities that contain:

- **Properties** (also called **attributes** or **data**)
- **Methods** (functions that define behavior)

The goal of OOP is to organize code so it's more **modular**, **reusable**, and **scalable**.

In JavaScript, everything (except primitive types) is an object, and OOP is fully supported via **prototypes** and **classes**.

🔷 1. Method

📌 Definition:

A **method** is a function that is defined inside an object and is used to represent the **behavior** of that object.

🧠 Example in life:

In a **Car object**, the action `drive()` is a method that describes the car's ability to move.

```
let car = {
brand: "Honda",
start() {
console.log("Car is starting...");
}
};
```

✅ Key Concept:

- Methods operate **on the data** inside the object (using `this`).
- They help achieve **encapsulation** by bundling data + behavior.

🔷 2. Constructor

📌 Definition:

A **constructor** is a blueprint or template used to create **multiple instances** (objects) with the same structure but different values.

🧠 Example in life:

Think of a **cookie cutter**. The cutter is the constructor, and each cookie is an object created from it.

```
class Person {
constructor(name, age) {
this.name = name;
this.age = age;
}
greet() {
console.log(Hi, I'm ${this.name});
}
}
```

✅ Key Concept:

- Every object has a **constructor** that defines its **initial structure** (properties + optional default behaviors).
- Used to create objects dynamically rather than manually every time.

🔷 3. Inheritance

📌 Definition:

Inheritance is the process by which one object (child) can **acquire properties and methods** from another object (parent).

🧠 Example in life:

A **Dog** inherits from **Animal**. All dogs are animals, and they inherit properties like `breathe`, `eat`, etc.

```
class Animal {
constructor(name) {
this.name = name;
}speak() {
console.log(${this.name} makes a sound.);
}
}class Dog extends Animal {
speak() {
console.log(${this.name} barks.);
}
}let d = new Dog("Bruno");
d.speak(); // Bruno barks.
```

✅ Key Concept:

- Promotes **code reuse** (write once, reuse many times).
- JavaScript uses **prototypes** or `class extends` to achieve inheritance.
- The child can **override** or **extend** the behavior of the parent.

🔷 4. Encapsulation

📌 Definition:

Encapsulation is the OOP principle of **restricting direct access to some of the object's components**, exposing only what's necessary.

🧠 Example in life:

You drive a **car** using the steering wheel, pedals, and gear — but you don't need to know how the engine or gearbox works internally.

✅ Key Concept:

- Keeps the internal implementation **hidden** from the outside world.
- Only certain **methods are exposed** to interact with the object.
- In JavaScript, encapsulation is done using **private fields/methods** (`#`) or closures.

🔷 5. Abstraction

📌 Definition:

Abstraction means showing only the **essential features** of an object and hiding the **complex background details**.

🧠 Example in life:

Using a **smartphone** — you touch icons to perform actions without knowing how the circuitry and software behind it works.

✅ Key Concept:

- Simplifies usage by hiding complexity.
- Helps in **focusing on what the object does**, not how it does it.
- Achieved by using **public methods** to expose only necessary functionality.

🔷 6. Polymorphism

📌 Definition:

Polymorphism means "**many forms**" — it allows objects of different types to be **treated using a common interface**, with behavior that may vary depending on the object.

🧠 Example in life:

A **remote control** works for a TV, AC, or music system — you press the "power" button, and each device reacts differently.

✅ Key Concept:

- Allows **different objects** to use the **same method name**, but perform different tasks.
- Enhances **flexibility** and **extensibility**.
- Achieved through **method overriding** in subclasses.

📘 Summary Table – OOP Concepts in Plain English

Concept	Meaning
Method	A function that describes object behavior
Constructor	A template for creating objects
Inheritance	Child objects reuse and extend parent object behavior
Encapsulation	Hiding internal details, exposing only the needed parts
Abstraction	Hiding complexity, showing only relevant actions
Polymorphism	Same method name behaves differently depending on the object

🧩 Why Use OOP in JavaScript?

- Organizes code around **real-world entities**
- Enhances **code readability** and **reusability**
- Helps create **modular** and **maintainable** code
- Widely used in **modern JavaScript frameworks** (React, Angular, Vue)

😬

WPT 2

Created	@June 25, 2025 8:03 PM

🌐 Document Object Model (DOM) in JavaScript – Full Theory

🔷 What is the DOM?

DOM (Document Object Model) is a **programming interface** for HTML and XML documents.

It represents the **structure of a web page** as a **tree of objects** that JavaScript can **access and manipulate**.

> ✅ The DOM connects HTML and JavaScript, allowing dynamic changes to the page content, structure, and style.

🔷 Object Hierarchy in JavaScript (DOM Tree)

When a web page is loaded, the browser creates a **DOM Tree** that mirrors the structure of the HTML document.

📁 DOM Object Hierarchy (simplified):

```
window
 └── document
     └── html
         ├── head
         │    └── title
```

```
└── body
    ├── div
    │   └── p
    └── script
```

🔑 Key Objects:

Object	Description
`window`	The browser window (global object)
`document`	Represents the entire HTML document
`element`	Any individual tag in the document
`attribute`	HTML attributes (like `id`, `class`)
`text node`	Text inside HTML tags

📚 HTML DOM

The **HTML DOM** is a standard that defines how to:

- Access HTML elements
- Change element content, structure, or style
- Respond to user actions (events)

> Every HTML element becomes a JavaScript object, and all properties/methods can be accessed using the document object.

🔶 DOM Elements

A **DOM Element** is any HTML element that exists in the document.

✅ Examples:

- `<p>` → `document.querySelector("p")`

- `<div id="box">` → `document.getElementById("box")`

Common DOM element types:

- `HTMLElement` – Any standard HTML element
- `HTMLInputElement` – Specific to inputs
- `HTMLAnchorElement` – Specific to links

🔶 DOM Events

DOM Events are actions that occur in the browser and can be detected using JavaScript.

Event Type	Example	Description
Mouse	`click`, `mouseover`, `mouseout`	Mouse interaction
Keyboard	`keydown`, `keyup`, `keypress`	Keyboard input
Form	`submit`, `change`, `focus`	Form activity
Window	`load`, `resize`, `scroll`	Browser-level

> Events allow developers to make web pages interactive.

🔧 Event Handling (Conceptually):

- **Add event listener** to an element.
- Browser **calls the function** when the event occurs.

🔧 DOM Methods

These are the functions used to **access**, **create**, or **modify** elements in the DOM.

📌 Access Methods:

Method	Description
`getElementById(id)`	Selects by ID

`getElementsByClassName()`	Selects by class name (collection)
`getElementsByTagName()`	Selects all tags (collection)
`querySelector(cssSelector)`	First match (powerful)
`querySelectorAll(selector)`	All matches (NodeList)

🔨 DOM Manipulation

These are techniques to **modify** content, structure, or styles of the web page using JavaScript.

✏️ Content Manipulation:

Task	Method
Change text	`element.textContent` or `innerText`
Change HTML	`element.innerHTML`
Add HTML	`insertAdjacentHTML()`

🧱 Structure Manipulation:

Task	Method
Create element	`document.createElement()`
Append child	`parent.appendChild()`
Remove element	`parent.removeChild()`
Replace element	`parent.replaceChild()`

🎨 Style Manipulation:

Task	Method
Set inline style	`element.style.color = "red"`
Change class	`element.classList.add()`
Toggle class	`element.classList.toggle()`

🔁 Summary Table

Concept	Description
DOM	Tree structure of the page
Object Hierarchy	From `window` → `document` → `html`
DOM Elements	Individual HTML tags as objects
DOM Events	User actions (clicks, keys, etc.)
DOM Methods	Tools to find/manipulate elements
DOM Manipulation	Dynamically changing structure/content/style

📝 Forms in JavaScript – Theory Guide

🟦 1. What is a Form?

A **form** in HTML is used to **collect user input** and send it to a **server** or process it using **JavaScript**.

📌 Examples of form elements:

- `<input>` – for text, number, email, etc.
- `<textarea>` – for multiline text
- `<select>` – for dropdowns
- `<button>` – to submit or reset

Forms are wrapped in the `<form>` tag:

```
<form action="/submit" method="POST">
  <input type="text" name="username">
  <input type="submit" value="Send">
</form>
```

🟦 2. Forms API (JavaScript Perspective)

The **Forms API** provides JavaScript access to form data, elements, and events.

✅ How JavaScript sees a form:

- Forms are accessed using `document.forms`
- Each form and its elements can be referenced like an object

🧠 Example:

```
document.forms[0]; // first form on the page
document.forms["loginForm"]; // form with name="loginForm"
```

🔶 Common properties and methods:

API Part	Description
`form.elements`	All elements inside the form
`form.submit()`	Programmatically submits the form
`form.reset()`	Resets all fields to default
`input.value`	Gets/sets the value of an input
`input.checked`	Checks if a checkbox/radio is selected

🟦 3. Form Validation

Form validation ensures that the user provides correct and complete input before the form is submitted.

Validation can be:

- **HTML-based (built-in)**
- **JavaScript-based (custom logic)**

✅ A. HTML Form Validation (Client-Side)

You can add validation rules directly in HTML:

Attribute	Purpose

`required`	Field must be filled out
`type="email"`	Must be a valid email format
`min`, `max`	Numeric limits
`maxlength`	Max number of characters
`pattern`	Regex-based custom validation

Example:

```
<input type="email" required>
<input type="text" pattern="[A-Za-z]{3,}" required>
```

✅ B. JavaScript Form Validation (Custom Logic)

JS allows more complex checks and dynamic feedback:

Example (Conceptual):

```
let form = document.getElementById("myForm");
form.addEventListener("submit", function (e) {
  let email = form.elements["email"].value;

  if (!email.includes("@")) {
    alert("Please enter a valid email.");
    e.preventDefault(); // prevent form submission
  }
});
```

✅ Types of Form Validation

Type	Description
Required Fields	User must fill the input
Input Length	Set max/min characters
Data Type Check	Validate emails, numbers, etc.

Pattern Matching	Use RegEx to allow only specific patterns
Password Match	Compare values of two fields

🧠 Why Form Validation Matters

- Prevents invalid data from being processed or sent to the server
- Improves **user experience** by providing instant feedback
- Enhances **security** by avoiding bad inputs (e.g., SQL injection, XSS)

🧾 Summary Table

Concept	Description
Form	A collection of input elements for user data
Forms API	JavaScript access to form data and controls
Validation	Ensures input correctness before submission
HTML Validation	Built-in using attributes like `required`, `type`
JS Validation	Custom logic for advanced or dynamic checks

🔍 Regular Expressions (RegEx) in JavaScript – Theoretical Guide

📘 What is a Regular Expression?

A **Regular Expression (RegEx)** is a **pattern-matching tool** used to search, match, and manipulate **text or strings**.

> Think of it as a rule or formula that defines what kind of text you're trying to find or validate.

✅ Why Use Regular Expressions?

Use Case	Description
✅ **Validation**	Emails, passwords, phone numbers, etc.
🔍 **Searching**	Find specific patterns in large text
✂️ **Extracting**	Pull specific parts of a string
🔁 **Replacing**	Swap matched patterns with something else

📚 Components of a Regular Expression

A regular expression is made of **literals** and **special characters** (also called **metacharacters**) that define a pattern.

🔷 1. Literal Characters

These match themselves exactly.

- `a` matches the letter a
- `123` matches the string "123"

🔷 2. Metacharacters (Symbols with special meaning)

Symbol	Meaning
`.`	Matches any single character
`^`	Start of a string
`$`	End of a string
`*`	Matches 0 or more times
`+`	Matches 1 or more times
`?`	Matches 0 or 1 time (optional)
`[]`	Character set (any one of...)
`[^]`	Negated set (not any of...)
`{}`	Quantifier (number of repetitions)
`()`	Grouping

🔷 3. Character Classes (Shorthand sets)

Notation	Matches
`\d`	Any digit (0–9)
`\D`	Not a digit
`\w`	Word character (a–z, A–Z, 0–9, _)
`\W`	Not a word character
`\s`	Whitespace
`\S`	Not whitespace

🔷 4. Quantifiers

Used to specify **how many times** a character/pattern should occur.

Symbol	Meaning
`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{n}`	Exactly n times
`{n,}`	n or more times
`{n,m}`	Between n and m times

🧠 Conceptual Examples (Theory)

Use Case	Pattern Example	Description
Email Validation	`/^\w+@\w+\.\w+$/`	Checks basic email structure
Digits Only	`/^\d+$/`	Only numbers allowed
Phone Number	`/^\d{10}$/`	Exactly 10 digits
Password	`/^(?=.*[A-Z])(?=.*\d).{8,}$/`	At least 1 capital, 1 number, 8+ length

🔶 How RegEx Works in JavaScript

In theory:

- You **define** a pattern using `/pattern/`

- You **apply** it to a string to **test** or **search**

Common methods (no code here, just concept):

Method	Purpose
`test()`	Returns true/false if matched
`match()`	Extracts matched value(s)
`replace()`	Replaces matched value
`search()`	Returns index of match
`split()`	Breaks string based on pattern

 Summary Table

Concept	Description
RegEx	Pattern-based text matching system
Metacharacters	Special symbols (`^` , `.` , `$` , etc.)
Quantifiers	How many times a pattern should match
Character Classes	Built-in shortcuts (`\d` , `\w` , etc.)
Usage	Validations, searches, formatting

🎯 Real-Life Use Cases of Regular Expressions

Scenario	Purpose
Form Validation	Ensure correct email, phone, password
Text Processing	Find hashtags, mentions, emails in text
Search/Replace	Auto-correct typos, reformat numbers
Log Parsing	Extract data from logs and reports

📘 What is an Error?

In programming, an **error** is an issue that **disrupts the normal flow of execution**.

In JavaScript, errors occur when:

- Code is written incorrectly (syntax error)
- Logic is wrong (runtime error)
- External resources fail (network, APIs)
- User input is invalid

🧰 Types of Errors in JavaScript

1. Syntax Errors

- Occur when code violates language grammar rules.
- Detected before code runs (compile-time errors).

🧠 Example: Missing a closing bracket or semicolon

💬 "Unexpected token" or "missing }"

---

2. Runtime Errors

- Occur while the code is running.
- Usually caused by undefined variables, logic mistakes, or bad user input.

🧠 Example: Accessing a property of `undefined`

---

3. Logical Errors

- Code runs without crashing but gives **incorrect results**.
- Hardest to detect because there's no error message.

🧠 Example: A loop that calculates wrong totals

---

4. Type Errors

- Happen when a value is used in the wrong way (e.g., calling a number like a function).

🧠 Example: `"abc".push()` → Error (string has no `push` method)

🔷 JavaScript Error Objects

JavaScript represents errors using the `Error` **object**, which contains information like:

- `name`: Type of error (e.g., SyntaxError, ReferenceError)
- `message`: Description of what went wrong
- `stack`: Trace of function calls that led to the error

🔶 Common Error Types:

Error Type	Meaning
`ReferenceError`	Variable not defined
`TypeError`	Wrong type usage (e.g., null as object)
`SyntaxError`	Code syntax is invalid
`RangeError`	Number out of range
`EvalError`	Error in use of `eval()`
`URIError`	Malformed URI/URL

🛠️ Debugging in JavaScript

Debugging is the process of **identifying, analyzing, and fixing errors** or unexpected behaviors in your code.

✅ Debugging Tools & Techniques

1. Using `console` methods

Method	Use
`console.log()`	Print values to inspect logic
`console.error()`	Highlight error messages
`console.warn()`	Display non-breaking warnings
`console.table()`	Display arrays/objects in table format

🧠 *Good for checking variables and flow of logic.*

---

2. Browser Developer Tools (DevTools)

- Built into browsers like Chrome, Firefox
- Access with `F12` or `Right click → Inspect → Console`
- Lets you:
  - View errors
  - Set breakpoints
  - Step through code line by line
  - Watch variables in real time

---

3. Breakpoints

- Manual “pauses” in code using **DevTools**
- Useful to stop execution and examine state step-by-step

---

4. `debugger;` Statement

- A keyword in JavaScript that triggers the browser’s debugger

```
function test() {
  let x = 10;
  debugger; // pauses here
  let y = x + 5;
}
```

---

5. Try-Catch Block (Error Handling)

- Use `try...catch` to catch and handle errors without crashing the app.

🧠 Syntax:

```
try {
  // risky code
} catch (error) {
  // handle error gracefully
}
```

You can even add `finally {}` to run cleanup code whether an error occurs or not.

🔑 Best Practices for Debugging

Practice	Why it's important
Use `console.log()` wisely	To avoid cluttering your output
Understand the error message	It usually tells you the exact issue
Reproduce the error	Make it happen consistently
Isolate the code	Narrow down which part is failing
Keep code clean and readable	Easier to debug later

🧾 Summary Table

Concept	Description
Error	An issue that breaks or disrupts code
Types	Syntax, Runtime, Logical, Type
Error Object	Holds details like `message`, `stack`
Debugging	Process to find and fix errors
Tools	Console, DevTools, Breakpoints, Debugger
Try-Catch	Gracefully handle errors in code

🧠 What are Developer Tools?

Browser Developer Tools (DevTools) are **built-in features** available in all modern web browsers (like Chrome, Firefox, Edge) that allow **developers to inspect, debug, and optimize** their websites or web applications.

> They help developers understand what the browser sees and how it renders and runs a webpage.

✅ Why Use DevTools?

Purpose	Example
🧪 Debug JavaScript	Find and fix script errors
🎨 Inspect & Edit CSS/HTML	Modify layout or style live
📊 Monitor Performance	Analyze page loading speed
🔐 Security Checks	Check HTTPS, CORS, headers
📪 Track Network Requests	See API calls and responses
📱 Test Responsiveness	See how site looks on mobile

🧱 Key Panels/Sections of DevTools

1. Elements Panel

- Displays the **HTML structure** (DOM) of the page.
- Lets you **inspect and edit HTML** and **CSS styles**.
- Useful for testing layout changes live.

> Used heavily for CSS debugging, layout fixing.

2. Console Panel

- Displays **errors, warnings**, and `console.log()` outputs.
- Allows you to **run JavaScript interactively**.
- Great for **debugging and testing logic**.

> You can directly type code and inspect results.

3. Sources Panel

- Shows all **JS, CSS, and other source files**.
- Allows you to set **breakpoints**, pause execution, and step through code.
- You can use the `debugger` tool visually here.

> Most powerful for JavaScript error tracking.

---

4. Network Panel

- Tracks all **HTTP requests** (API calls, assets, files).
- Shows status codes (200, 404, etc.), timing, payloads, etc.
- Helps diagnose **slow loading**, **API errors**, or **CORS issues**.

> Essential for backend/frontend API integration debugging.

---

5. Performance Panel

- Records and analyzes **page load time** and interactions.
- Helps improve **rendering performance** and reduce lag.

---

6. Application Panel

- Manages **cookies**, **local storage**, **session storage**, **indexedDB**, and **service workers**.
- Allows you to test how the app stores data on the client side.

> Important for offline storage or login/session systems.

---

7. Security Panel

- Shows HTTPS status, certificate information, and insecure resources.

---

8. Lighthouse Panel (in Chrome)

- Audits your site for **performance, SEO, accessibility, and best practices**.
- Gives a score and suggestions for improvement.

🎯 Benefits of DevTools

Feature	Advantage
Real-time Editing	See changes instantly
Error Visibility	Easily find script or resource issues
Performance Insights	Optimize for faster load times
Mobile Simulation	Preview how site looks on phones/tablets
Data Inspection	Debug network, cookies, storage, etc.

🧾 Summary

Panel	Use
Elements	View/edit HTML & CSS
Console	Log errors and run JS
Sources	Debug JavaScript line-by-line
Network	Inspect HTTP/API traffic
Performance	Analyze speed and rendering
Application	Manage storage, cookies, cache
Security	Check HTTPS and certificates

📘 What is JSLint?

JSLint is a **static code analysis tool** for **JavaScript**, created by **Douglas Crockford**, one of JavaScript's early influencers.

> It scans your JavaScript code and detects potential problems, bad practices, and violations of coding conventions.

🎯 Objective of JSLint

The **goal of JSLint** is not just error detection but to **enforce a disciplined, high-quality style** of writing JavaScript.

It answers questions like:

- Is your code clean?
- Is it consistent?
- Is it safe and maintainable?

🔍 What JSLint Checks For

Category	What It Checks
✅ **Syntax Errors**	Missing semicolons, brackets, quotes
🚫 **Bad Practices**	Use of `eval`, `with`, global vars
⚠️ **Unused Variables**	Declared but never used
🧹 **Code Style Issues**	Inconsistent indentation, spacing
📛 **Naming Problems**	Non-standard naming of variables
❌ **Equality Issues**	Use of `==` instead of `===`
🔐 **Security Warnings**	Unsafe patterns (e.g., global scope)

🧠 How Does JSLint Work?

1. You paste or upload your JavaScript code into JSLint (web tool or command-line).
2. It analyzes the code **without running it** (static analysis).
3. It shows a list of **warnings, errors, and suggestions**.

✅ Benefits of Using JSLint

Benefit	Description
✅ Enforces clean code	Reduces messy or ambiguous syntax

🔍 Early bug detection	Finds problems before runtime
🔐 Improves security	Flags dangerous patterns
📐 Encourages consistency	Helps teams write similar code
📚 Educational	Teaches good JavaScript practices

📌 Example Warnings JSLint Might Show

Issue	JSLint Warning Example
Missing semicolon	"Expected ';' and instead saw '}'."
Using undeclared variable	"'x' is not defined."
Using global variable	"Global variable 'window'."
Using `==` instead of `===`	"Expected '===' and instead saw '=='."
Too many nested blocks	"This function is too complex."

🛠️ Where Can You Use JSLint?

Environment	How to Use
Web Interface	Visit jslint.com and paste code
Node.js	Use CLI with `npm install jslint`
IDE Plugins	Some editors like VS Code support JSLint extensions

🧾 Summary Table

Concept	Description
JSLint	A strict JavaScript code quality tool
Checks	Syntax, style, security, logic
Purpose	Push toward better, maintainable code
Method	Static code analysis (no execution)
Benefit	Improves code consistency, safety

🔎 JSLint vs JSHint vs ESLint

Tool	Strictness	Customizable	Use Case
JSLint	Very strict	❌ Not customizable	Best for learning best practices
JSHint	Medium	✅ Yes	More relaxed than JSLint
ESLint	Flexible	✅ Highly	Widely used in modern projects

🟦 What is JSON?

JSON (JavaScript Object Notation) is a lightweight, text-based **data format** used to **store and exchange** data.

It is:

- Easy to read/write for humans
- Easy to parse/generate by machines

> Originally derived from JavaScript, but now supported in all programming languages (Java, Python, PHP, etc.)

🎯 Why JSON is Needed (Uses & Importance)

Use Case	Purpose
🌐 **Data Exchange**	Between web browser and server (AJAX, REST API)
💾 **Configuration**	Used in config files (e.g., `package.json`)
☁️ **APIs**	JSON is the most common format in APIs
🗃️ **Data Storage**	Lightweight database entries or local storage

XML was used earlier, but **JSON is shorter, cleaner, and faster**.

🧾 JSON Syntax Rules

Rule	Description
🔤 **Data is in key/value pairs**	Like `name: "John"`

🔁 **Data is separated by commas**	Like in arrays or objects
🟦 **Keys are strings (in double quotes)**	`"name": "value"`
📦 **Objects are wrapped in** `{}`	Key-value pairs
📚 **Arrays are wrapped in** `[]`	List of values

> ✅ Strings must be in double quotes
>
> ❌ No functions, comments, or variables allowed

🔧 Example JSON Structure

```
{
  "name": "Alice",
  "age": 25,
  "isMember": true,
  "skills": ["JavaScript", "Python", "HTML"],
  "address": {
    "city": "New York",
    "zip": "10001"
  }
}
```

📘 JSON Objects

A **JSON Object** is a collection of **key-value pairs** surrounded by `{}`.

```
{
  "id": 1,
  "title": "Book",
  "price": 100
}
```

> Similar to JavaScript object but with stricter formatting (double quotes required for keys).

📘 JSON Arrays

A **JSON Array** is an **ordered list** of values (can be strings, numbers, objects, or even other arrays) surrounded by `[]`.

```
[
  "apple",
  "banana",
  "cherry"
]
```

You can also have an array of objects:

```
[
  { "name": "John", "age": 22 },
  { "name": "Sara", "age": 24 }
]
```

📁 JSON Files

A JSON file:

- Has a `.json` extension
- Contains a single valid JSON object or array
- Is often used to store data, configs, or to fetch from APIs

Example:

```
{
  "language": "JavaScript",
```

```
  "framework": "React"
}
```

🔄 JSON Parsing

Parsing means converting a JSON string into a usable **JavaScript object**.

In JavaScript:

Task	Method
Parse JSON	`JSON.parse(jsonString)`
Convert to JSON	`JSON.stringify(jsObject)`

> JSON.parse() turns text → object
>
> `JSON.stringify()` turns object → text

✅ Summary Table

Topic	Description
JSON	Lightweight data format
Syntax	Key-value pairs, strict rules
Object	`{ "key": "value" }`
Array	`[ value1, value2 ]`
File	`.json` file with JSON data
Parsing	`parse()` to read, `stringify()` to write

✨ Key Benefits of JSON

Feature	Advantage
💡 Human-readable	Easy to edit and debug
💬 Language-independent	Used across all platforms

🚀 Lightweight	Faster than XML
🔧 Structured	Ideal for nested or grouped data

💡 jQuery – Theoretical Overview

📘 Introduction to jQuery

jQuery is a fast, small, and feature-rich **JavaScript library**.

It simplifies **HTML document traversal, manipulation, event handling**, and **animation**.

> Motto: “Write less, do more.”

✅ Why jQuery?

- Simplifies JavaScript code
- Cross-browser compatibility
- Easier DOM manipulation
- Built-in animation and AJAX support

🎯 Core jQuery Syntax

```
$(selector).action()
```

- `$` = jQuery function
- `selector` = selects HTML elements
- `action()` = applies action (e.g., hide/show)

🔷 1. jQuery Selectors

Selectors are used to **select HTML elements**, similar to CSS.

Selector Type	Example	Meaning
Element Selector	`$("p")`	All `<p>` tags
ID Selector	`$("#id")`	Element with that ID
Class Selector	`$(".class")`	Elements with that class
Attribute Selector	$("[type='text'])"	Input fields of type text
Universal Selector	$("*")	All elements

> Uses CSS-style selectors internally.

🔷 2. jQuery Events

Events respond to user interactions like clicks, hovers, typing, etc.

Event Type	Example
Click	`$("#btn").click(function(){...})`
Hover	`$("p").hover(inFn, outFn)`
Keypress	`$("#input").keypress()`
Submit	`$("form").submit()`
Change	`$("#select").change()`

> jQuery uses the .on() method for advanced event handling.

🔷 3. jQuery Animation Effects

Built-in **effects/animations** make it easy to add visual feedback.

Effect Function	Purpose
`.hide()` / `.show()`	Hide/show elements
`.fadeIn()` / `.fadeOut()`	Smooth fading
`.slideDown()` / `.slideUp()`	Slide vertically
`.animate()`	Custom animations (e.g., movement, size)

`.toggle()`	Switch between show/hide

> You can also control duration, callback, and easing.

🔷 4. DOM Traversal and Manipulation

jQuery makes navigating the **DOM tree** and **editing HTML elements** easier.

Traversal Methods:

Method	Description
`.parent()`	Get immediate parent
`.children()`	Get all child elements
`.next()` / `.prev()`	Sibling elements
`.find()`	Find elements within an element

Manipulation Methods:

Method	Use
`.html()`	Get/set HTML content
`.text()`	Get/set plain text
`.val()`	Get/set form values
`.attr()`	Get/set attributes
`.css()`	Get/set styles
`.append()` / `.prepend()`	Insert content inside
`.remove()`	Delete elements

🔷 5. Data Attributes and Templates

Data Attributes:

- HTML5 allows **custom data attributes**, like:

```
<div data-user-id="101"></div>
```

In jQuery:

```
let id = $("div").data("user-id");
```

> Used to store extra data without cluttering the DOM.

Templates:

- jQuery can work with client-side templating engines (like Mustache.js or Handlebars) to **render data-driven UI**.
- While jQuery itself doesn't have a template engine, you can combine it with `.html()` or `.append()` to inject dynamic content.

🔷 6. jQuery DOM Utility Functions

Function	Use
`.each()`	Loop through elements
`.is()`	Checks if element matches condition
`.hasClass()` / `.addClass()` / `.removeClass()`	Manage classes
`.clone()`	Clone elements
`.empty()`	Remove content inside element
`.width()` / `.height()`	Get/set dimensions

> These utilities simplify common DOM tasks without verbose code.

🔷 7. jQuery Plugins

jQuery Plugins are external packages that extend jQuery's functionality.

Examples	Purpose
jQuery UI	Animations, widgets, drag/drop
DataTables	Interactive tables
OwlCarousel	Carousels/sliders
Lightbox	Image modals
Form Validator	Client-side form validation

How to use:

1. Include the plugin's `.js` and `.css` files
2. Call the plugin method on a selected element

 Summary Table

Feature	Description
Selectors	Find elements using CSS-style syntax
Events	React to user actions
Effects	Hide, show, fade, slide, animate
DOM Traversal	Navigate the HTML document
Manipulation	Modify HTML, text, attributes, values
Data Attributes	Store custom data
Utility Functions	General tools for DOM/logic
Plugins	Add prebuilt functionality

✅ Benefits of jQuery

- ✨ Clean, readable syntax
- 💻 Works across all browsers
- 🔧 Quick development of UI features
- 📦 Rich plugin ecosystem
- 🎯 Great for beginners and rapid prototyping

🔄 AJAX – Asynchronous JavaScript and XML

📘 Introduction to AJAX

AJAX is not a programming language, but a **technique** that allows **web pages to communicate with a server in the background** without reloading the entire page.

> It enables dynamic, fast, and interactive user experiences on websites.

🔑 Key Idea:

AJAX allows web applications to:

- Send and receive **data from a server asynchronously**
- **Update parts of a page** without refreshing the whole page

---

🧱 Technologies Involved in AJAX

Technology	Role
HTML/XHTML	Structure of web page
CSS	Styling
JavaScript	Logic and event handling
DOM	To update page content dynamically
XMLHttpRequest or **Fetch API**	To communicate with the server
XML/JSON	Format of data being exchanged

> Today, JSON is preferred over XML due to its simplicity and JavaScript compatibility.

🏗️ AJAX Architecture

✅ Step-by-Step AJAX Workflow:

1. **Event Triggered**

   (e.g., user clicks a button)

2. **JavaScript Creates an XMLHttpRequest**

   or uses `fetch()` or `$.ajax()` in jQuery

3. **Request Sent to Server (Asynchronously)**

   Server processes request (e.g., database query)

4. **Server Responds with Data**

   Often in **JSON**

5. **JavaScript Receives Response**

   Parses and updates the DOM (part of the page)

🔄 Flow Diagram

```
[User Action]
     ↓
[JavaScript (AJAX Call)]
     ↓
[Web Server]
     ↓
[Data (JSON/XML)]
     ↓
[JavaScript Updates the DOM]
```

🌐 Web Services and AJAX

Web Services are APIs on the server side that provide data in a machine-readable format (usually JSON or XML).

AJAX calls these services to **retrieve or send data** dynamically.

> Examples: RESTful APIs, GraphQL, SOAP services (less common today)

Common Operations:

- GET (fetch data)
- POST (submit data)
- PUT/PATCH (update data)
- DELETE (remove data)

AJAX enables these API calls without page reload.

💡 AJAX Using jQuery

jQuery simplifies AJAX with easy-to-use methods that abstract away browser complexities.

1. `$.ajax()` — Most powerful method

```
$.ajax({
  url: "https://api.example.com/data",
  type: "GET",
  success: function(response) {
    // handle successful response
  },
  error: function(error) {
    // handle error
  }
})
```

2. `$.get()` – Simplified GET request

```
$.get("data.json", function(data) {
  console.log(data);
```

```
});
```

3. `$.post()` – Simplified POST request

```
$.post("submit.php", { name: "Alice" }, function(response) {
  alert("Data saved");
});
```

4. Load Method – Load part of an HTML file

```
$("#content").load("page.html");
```

> jQuery automatically parses JSON responses.

✅ Advantages of AJAX

Benefit	Description
🚀 Faster interactions	No full page reload
💬 Real-time updates	Live chat, search suggestions
📉 Reduced server load	Only necessary data sent
🖥️ Better user experience	Smoother UI/UX
🔧 Works with any back-end	PHP, Node.js, .NET, etc.

⚠️ Challenges / Considerations

- Cannot access **cross-origin** data unless CORS is enabled
- **SEO limitations** for AJAX-rendered content
- Requires **JavaScript support** in browser

🧾 Summary Table

Concept	Description
AJAX	Asynchronous communication between browser & server
Architecture	Uses JavaScript, XMLHttpRequest/fetch, and DOM
Web Services	APIs that serve data used by AJAX
jQuery AJAX	Simplified syntax for making requests
Common Formats	JSON (preferred), XML

🔁 Axios – A Promise-Based HTTP Client for JavaScript

📘 What is Axios?

Axios is a **promise-based HTTP client** for JavaScript used to make **asynchronous requests** to servers from the browser or Node.js.

> It's commonly used for calling REST APIs and works seamlessly with modern JavaScript features like async/await.

✅ Why Use Axios?

Feature	Benefit
🌐 HTTP Requests Made Easy	Simplifies `GET`, `POST`, `PUT`, `DELETE`
🔒 Handles JSON by default	Automatically parses JSON responses
📦 Browser + Node Support	Works on both client and server
🔁 Interceptors	Modify request/response globally
❌ Auto error handling	Detects HTTP status-based errors

🔷 Axios Syntax Basics

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

🧰 Axios Instance and Its Config

Instead of calling Axios directly, you can create an **Axios instance** to apply **default configuration** across all requests.

🛠️ Creating an Axios Instance

```
const api = axios.create({
  baseURL: 'https://api.example.com',
  timeout: 5000,
  headers: { 'Authorization': 'Bearer token123' }
});
```

> This is useful when you want centralized control over API behavior (e.g., base URL, tokens).

⚙️ Axios Configuration Options

Config Option	Purpose
`baseURL`	Sets base for relative URLs
`headers`	Custom headers like auth
`timeout`	Max time to wait for response
`params`	URL query parameters

`data`	Payload for POST/PUT
`withCredentials`	For sending cookies (CORS)

Example:

```
axios.get('/user', {
  params: { id: 123 },
  timeout: 3000
});
```

🔄 Handling Requests and Responses

1. Request

You can use different methods based on the type of HTTP request:

```
axios.get('/users');
axios.post('/users', { name: "Alice" });
axios.put('/users/1', { name: "Updated" });
axios.delete('/users/1');
```

2. Response

The response object contains useful properties:

Property	Meaning
`data`	The actual response body (usually JSON)
`status`	HTTP status code (e.g. 200, 404)
`headers`	HTTP response headers
`config`	The config used for the request

Example:

```
axios.get('/api').then(response => {
  console.log(response.status); // 200
```

```
  console.log(response.data);   // API data
});
```

❌ Handling Errors

Axios errors can occur due to:

- HTTP error codes (like 404, 500)
- Network issues (like timeout)
- Server or client validation failures

🔍 Error Handling Pattern

```
axios.get('/invalid-url')
  .then(response => {
    // success
  })
  .catch(error => {
    if (error.response) {
      // Server responded with a non-2xx status code
      console.log(error.response.status); // 404, 500, etc.
      console.log(error.response.data);   // error details
    } else if (error.request) {
      // No response received
      console.log("No response from server");
    } else {
      // Something went wrong while setting up the request
      console.log("Error", error.message);
    }
  });
```

🔄 Interceptors (Advanced)

Interceptors are functions that run **before a request is sent** or **after a response is received**.

Example: Adding Authorization Token

```
axios.interceptors.request.use(config => {
  config.headers.Authorization = "Bearer mytoken";
  return config;
});
```

Summary Table

Feature	Description
Axios	Promise-based HTTP client
Axios Instance	Custom version with defaults (baseURL, headers)
Config Options	Set headers, params, timeouts
Request Handling	Use `.get()`, `.post()` etc.
Response Structure	Use `response.data`, `response.status`
Error Handling	Use `.catch()` and inspect `error.response`
Interceptors	Modify request/response globally

😬

WPT 3

Created	@June 25, 2025 11:42 PM

📦 Node.js Modules – Theory Guide

vedant bhi padh looooooooooooooooooooooooo

📘 What is a Node.js Module?

A **module** in Node.js is simply a **reusable block of code** that can be split into separate files and used throughout your application.

> Node uses a modular architecture — each file is treated as a separate module.

✅ Why Use Modules?

Benefit	Description
🧱 Code Reusability	Write once, use anywhere
🔄 Separation of Concerns	Keeps code clean and maintainable
📉 Avoid Global Scope	Modules keep variables local
🔐 Encapsulation	Only share what's needed using `exports`

🧰 Types of Modules in Node.js

Module Type	Description
Core Modules	Built into Node.js (e.g., `fs`, `http`, `path`)

Local Modules	Custom modules you create (your own `.js` files)
Third-party Modules	Installed using npm (e.g., `express`, `axios`)

🔷 `require()` – Importing Modules

The `require()` function is used to **import** a module into another file.

```
const fs = require('fs'); // Core module
const myModule = require('./myModule'); // Local module
```

How It Works:

- Loads and runs the module file once
- Returns the value from `module.exports`
- Caches the result for future `require()` calls

🔷 `exports` and `module.exports` – Exporting from Modules

To **share variables, functions, or objects** from a module, you use `exports` or `module.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)); // 5
```

> You can also directly assign a function or value:

```
module.exports = function greet(name) {
  return `Hello, ${name}`;
};
```

🧠 Difference Between `exports` and `module.exports`

`exports`	`module.exports`
Shortcut to export multiple items	The actual object returned by `require()`
Must not be reassigned directly	Can be reassigned (e.g., function/class)

> Best Practice: Use module.exports for exporting a single item (function/class) and exports. for multiple properties.

🔄 Real-world Analogy

Think of modules like **files in a toolbox**:

- `require()` = taking a tool from the box
- `exports` = putting a tool inside for others to use

🧾 Summary Table

Concept	Description
Module	A JavaScript file with its own scope
require()	Imports the contents of a module

exports	Adds properties/functions to the module
module.exports	Sets the final object returned by `require()`
Core Modules	Built into Node.js (e.g., `http`, `fs`)
Local Modules	Created by the developer (`./file.js`)

✅ Quick Example (Putting It All Together)

📁 greet.js

```
function greet(name) {
  return `Hello, ${name}`;
}
module.exports = greet;
```

📁 app.js

```
const greet = require('./greet');
console.log(greet('Alice')); // Hello, Alice
```

📦 Introduction to npm (Node Package Manager)

📘 What is npm?

npm is the **default package manager** for **Node.js**. It is used to:

- **Install, update, and manage** third-party packages/libraries
- **Share reusable code** with others
- Track dependencies of your project in a `package.json` file

> npm is also an online repository of over 1 million open-source packages.

📂 What is `package.json`?

The `package.json` file is the **metadata file** for a Node.js project.

It describes the project, dependencies, and scripts.

📄 Sample `package.json`

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "A sample Node.js app",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo 'Running tests...'"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

🧠 Key Fields:

Field	Purpose
`name`	Name of the project
`version`	Current version
`scripts`	Custom commands (`npm run`)
`dependencies`	Required packages to run the app
`devDependencies`	Used only during development

🛠️ npm Commands to Manage Packages

📥 Installing Packages

✅ Locally (default):

```
npm install <package-name>
```

E.g.:

```
npm install express
```

This:

- Installs in `node_modules/`
- Adds to `dependencies` in `package.json`

✅ Globally:

```
npm install -g <package-name>
```

E.g.:

```
npm install -g nodemon
```

> Global packages are available system-wide from the command line.

🔄 Updating Packages

```
npm update <package-name>
```

Updates a specific package to its latest **compatible** version based on `package.json`.

🗑️ Removing Packages

```
npm uninstall <package-name>
```

📂 Local vs Global Packages

Feature	Local Packages	Global Packages
Installed In	`node_modules` inside your project folder	System-wide (e.g., `/usr/lib/node_modules`)
Used For	Project-specific libraries (e.g., Express, Mongoose)	CLI tools (e.g., nodemon, eslint)
Accessible In Code?	Yes	No (unless linked locally)
Shared Between Projects?	No	Yes

🧪 Other Useful npm Commands

Command	Description
`npm init`	Starts a wizard to create `package.json`
`npm list`	Shows installed packages and versions
`npm outdated`	Shows which packages need updates
`npm install`	Installs all packages from `package.json`
`npm run <script>`	Runs a custom script (from `"scripts"`)

📁 `node_modules/` and `.package-lock.json`

- `node_modules/`: Folder where npm installs all packages.
- `package-lock.json`: Records exact versions installed (helps with consistency across environments).

🧾 Summary Table

Concept	Description
npm	Node Package Manager
`package.json`	Project metadata and dependencies
Local Packages	Installed per project
Global Packages	Installed system-wide
`npm install`	Installs a package
`npm uninstall`	Removes a package
`npm update`	Updates a package
`npm init`	Creates `package.json`
`npm run`	Executes scripts like build/start/test

📁 Node.js File I/O – Synchronous & Asynchronous Methods

Node.js provides the `fs` **(File System)** module to perform file operations such as reading, writing, deleting, and renaming files.

🔄 Synchronous File I/O

- Executes blocking code (waits for operation to complete).
- Slows down performance if used in web apps.

Example:

```
const fs = require('fs');

const data = fs.readFileSync('file.txt', 'utf8');
console.log(data); // Outputs file content
```

> ✅ Easy to write
>
> ❌ Blocks execution until complete

🔁 Asynchronous File I/O

- Non-blocking — allows other code to run in parallel.
- Recommended for **server-side applications**.

Example:

```
const fs = require('fs');

fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data); // Outputs file content
});
```

> ✅ Improves performance
>
> ✅ Handles many requests at once

Common File Operations

Operation	Sync Method	Async Method
Read File	`fs.readFileSync()`	`fs.readFile()`
Write File	`fs.writeFileSync()`	`fs.writeFile()`
Append File	`fs.appendFileSync()`	`fs.appendFile()`
Delete File	`fs.unlinkSync()`	`fs.unlink()`
Create Folder	`fs.mkdirSync()`	`fs.mkdir()`

🌐 Node.js HTTP Module – Building an HTTP Server

Node.js provides a built-in `http` **module** to create web servers **without using any external library**.

✅ Why Use It?

- Lightweight and fast
- Good for learning how web servers work
- Foundation for frameworks like Express.js

🔨 Creating a Basic HTTP Server

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js server!');
});

server.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Breakdown:

Part	Purpose
`http.createServer()`	Creates server with request/response handler
`res.writeHead()`	Sets HTTP status and headers
`res.end()`	Sends response and ends connection
`server.listen()`	Binds server to a port

🌐 Handling Routes in HTTP Server

```
if (req.url === '/about') {
  res.end('About Page');
} else if (req.url === '/') {
  res.end('Home Page');
```

```
} else {
  res.writeHead(404);
  res.end('Page Not Found');
}
```

🛠️ Developing a Simple Node.js Web Application (Without Express)

Project Structure:

```
bash
Copy code
project/
│
├── app.js          # Main application
├── data.txt        # Sample file
└── package.json    # Metadata
```

Sample Application Logic (in `app.js`):

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    fs.readFile('data.txt', 'utf8', (err, data) => {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(data);
    });
  } else {
    res.writeHead(404);
    res.end('404 Not Found');
```

```
  }
});

server.listen(3000, () => {
  console.log('Server started at http://localhost:3000');
});
```

🧾 Summary Table

Topic	Summary
File I/O (Sync)	Blocking operations (`fs.readFileSync`)
File I/O (Async)	Non-blocking, uses callbacks
HTTP Module	Built-in module to create web servers
Web App	Created using core modules (`http` , `fs`)

🚀 Introduction to Express.js

📘 What is Express?

Express.js is a **minimal and flexible Node.js web application framework** that provides a robust set of features to build web and mobile applications.

> It simplifies routing, request/response handling, and middleware use in Node.js applications.

🔧 Why Use Express?

Feature	Benefit
📦 Lightweight	Fast and minimal by design
🔁 Easy Routing	Clean URL handling with HTTP methods
⚙️ Middleware Support	Add custom logic between request and response

🔌 Integration Friendly	Works with databases, template engines, and APIs
🧱 Scalable	Good for both small and large apps

✅ Getting Started with Express

🔨 1. Install Express using npm

```
npm install express
```

📝 2. Create a Basic Express App

```
const express = require('express');
const app = express();

// Route for GET request
app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

// Start server
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

⚙️ Key Concepts in Express

1. Application Object (`app`)

The `app` object is the **main Express application**.

It is created by calling `express()` and is used to:

- Define routes: `app.get()` , `app.post()` , etc.
- Use middleware: `app.use()`

- Listen for requests: `app.listen()`

> It acts as the central control unit of your Express web app.

---

2. Request Object (`req`)

`req` is an object representing the **HTTP request** and contains information such as:

Property	Description
`req.url`	The request path
`req.method`	HTTP method (`GET` , `POST` , etc.)
`req.body`	Data sent in the body (used in POST/PUT)
`req.params`	Route parameters
`req.query`	URL query string (`?key=value`)

---

3. Response Object (`res`)

`res` is an object used to **send the response back** to the client.

Method	Description
`res.send()`	Sends text or object as response
`res.json()`	Sends JSON-formatted data
`res.status(code)`	Sets the HTTP status code
`res.redirect(url)`	Redirects the request
`res.render()`	Renders a view (used with template engines)

✅ Example with Request & Response:

```
app.get('/hello', (req, res) => {
  const user = req.query.name || "Guest";
  res.send(`Hello, ${user}`);
});
```

Request: `http://localhost:3000/hello?name=Alice`

Response: `Hello, Alice`

🧾 Summary Table

Concept	Description
Express	Web framework for Node.js
`app`	Main application object
`req`	Represents the client's request
`res`	Represents the server's response
Middleware	Functions run before sending response

🛣️ Routes in Express.js

📘 What Are Routes?

Routing refers to defining how your application responds to **client requests (HTTP methods like GET, POST, etc.)** for specific **URLs (endpoints)**.

> In simple terms, routes map incoming URLs to logic that handles them.

🛠️ Route Syntax

```
app.METHOD(PATH, HANDLER)
```

- `METHOD` = HTTP method (GET, POST, etc.)
- `PATH` = route endpoint (e.g., `/home`)
- `HANDLER` = function with `req` and `res` parameters

✅ Example

```
app.get('/', (req, res) => {
  res.send('Home Page');
```

```
});

app.post('/submit', (req, res) => {
  res.send('Form submitted');
});
```

🔄 Types of Route Parameters

🔷 Route Parameters (`:param`)

Used for dynamic values:

```
app.get('/user/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

Request: `/user/101` → Response: `User ID: 101`

🔷 Query Strings

Accessed using `req.query`:

```
app.get('/search', (req, res) => {
  res.send(`Search: ${req.query.term}`);
});
```

Request: `/search?term=node` → Response: `Search: node`

⚙️ Middleware in Express.js

📘 What Is Middleware?

A **middleware** is a **function** that runs **between receiving a request and sending a response**.

> It can modify the req and res objects or terminate the request-response cycle.

🔁 Syntax

```
app.use((req, res, next) => {
  console.log('Request received');
  next(); // Move to next middleware or route
});
```

✅ Common Uses

Use Case	Example Middleware
Logging	`morgan`
Parsing request body	`express.json()`
Authentication	Custom or passport.js
Serving static files	`express.static()`

🧰 Built-in Middlewares

- `express.json()` – Parse JSON body
- `express.urlencoded()` – Parse URL-encoded body
- `express.static()` – Serve static files like images, CSS, JS

🎨 Templates and Template Engines

📘 What Is a Template?

A **template** is an HTML file with placeholders (like `{{name}}`) that are **dynamically filled** with data before being sent to the browser.

🔥 What Is a Template Engine?

A **template engine** takes template files and **renders** them into complete HTML pages using dynamic data.

Popular Engines:

Engine	Extension	Notes
EJS	`.ejs`	Easy, uses plain JS code
Pug	`.pug`	Indentation-based syntax
Handlebars	`.hbs`	Uses `{{}}` style markup

🛠️ Setting a Template Engine in Express

```
app.set('view engine', 'ejs');
app.set('views', './views'); // directory for templates
```

🖼️ Rendering Views

Use `res.render()` to **combine a template and data** into HTML.

Example:

📁 views/welcome.ejs

```
<h1>Welcome, <%= name %>!</h1>
```

📄 In your route:

```
app.get('/greet', (req, res) => {
  res.render('welcome', { name: 'Alice' });
});
```

Response HTML:

```
html
Copy code
<h1>Welcome, Alice!</h1>
```

 Summary Table

Concept	Description
Routes	Define app behavior for each URL + method
Middleware	Functions executed before final response
Template Engine	Dynamically creates HTML from templates
res.render()	Renders a view with dynamic data
EJS/Pug	Common engines for rendering