MERN Full-Stack

# Full Stack Development

Fullstack development refers to the practice of building both the front-end (client-side) and back-end (server-side) components of a web application. Fullstack developers are skilled in a wide range of technologies that enable them to handle the complete lifecycle of an application, from designing the user interface to managing databases and server logic.

### **Key Components of Fullstack Development**

1. **Front-End Development**
   * **Purpose**: Focuses on the user interface (UI) and user experience (UX).
   * **Technologies**:
     + **Languages**: HTML, CSS, JavaScript.
     + **Frameworks/Libraries**: React.js, Angular, Vue.js, Bootstrap.
   * **Responsibilities**:
     + Designing responsive and interactive web pages.
     + Ensuring cross-browser compatibility.
     + Implementing designs from UI/UX wireframes.
2. **Back-End Development**
   * **Purpose**: Focuses on server-side logic, database management, and API integration.
   * **Technologies**:
     + **Languages**: Node.js, Python, Java, PHP, Ruby.
     + **Frameworks**: Express.js, Django, Spring Boot, Flask.
     + **Databases**: MySQL, MongoDB, PostgreSQL, Firebase.
     + **APIs**: RESTful APIs, GraphQL.
   * **Responsibilities**:
     + Managing server-side logic and database queries.
     + Developing and maintaining APIs.
     + Ensuring application security and performance.
3. **DevOps and Deployment**
   * **Purpose**: Focuses on deploying and maintaining applications in a production environment.
   * **Technologies**: Docker, Kubernetes, AWS, Azure, CI/CD pipelines.
   * **Responsibilities**:
     + Automating deployment processes.
     + Monitoring application performance.
     + Ensuring scalability and reliability.

### **Popular Fullstack Tech Stacks**

1. **MERN Stack**: MongoDB, Express.js, React.js, Node.js.
2. **MEAN Stack**: MongoDB, Express.js, Angular, Node.js.
3. **LAMP Stack**: Linux, Apache, MySQL, PHP.
4. **Django Stack**: Django, Python, PostgreSQL.
5. **Spring Boot Stack**: Spring Boot, Java, MySQL.

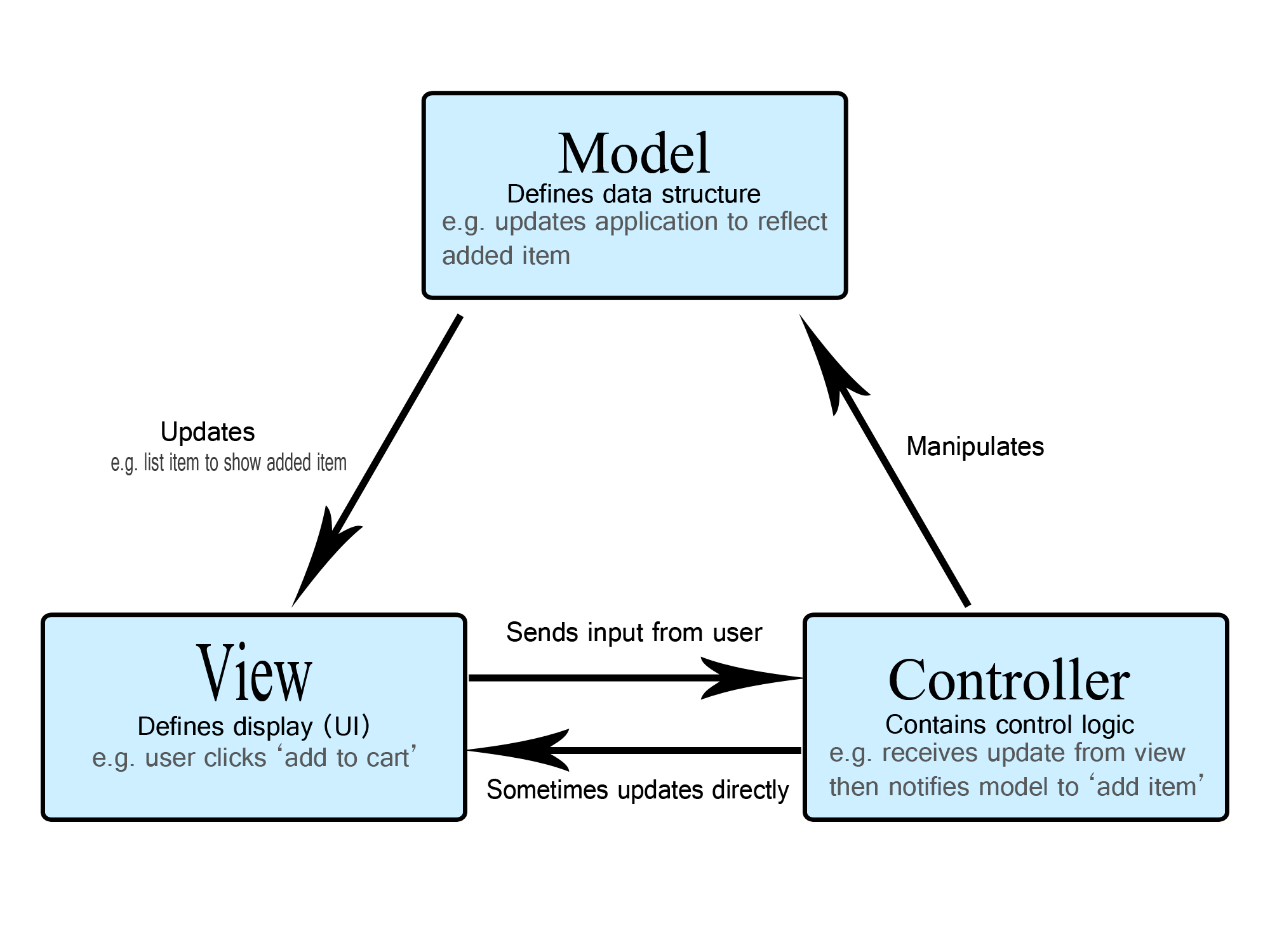
### **Challenges in Fullstack Development**

1. **Continuous Learning**: Keeping up with the rapid evolution of technologies.
2. **Workload Management**: Balancing front-end and back-end responsibilities.
3. **Depth vs. Breadth**: Achieving expertise in both areas can be challenging.

### **Applications of Fullstack Development**

1. **E-Commerce Platforms** (e.g., Amazon, Flipkart).
2. **Social Media Platforms** (e.g., Instagram, Facebook).
3. **Content Management Systems** (e.g., WordPress, Wix).
4. **Business Management Tools** (e.g., CRM software).

# Model-View-Controller (MVC) Architecture

****

1. **Model**
   * **Purpose**: Represents the application's data and business logic.
   * **Responsibilities**:
     + Manages the data, logic, and rules of the application.
     + Communicates with the database or external APIs to fetch and store data.
     + Notifies the view of any changes in the data.
   * **Example**:
     + In a blog application, the Model might handle blog posts and user data, including fetching posts from the database.
2. **View**
   * **Purpose**: Represents the user interface (UI) of the application.
   * **Responsibilities**:
     + Displays data to the user as provided by the Model.
     + Sends user inputs to the Controller.
   * **Example**:
     + The web page displaying a list of blog posts with titles, authors, and timestamps.
3. **Controller**
   * **Purpose**: Acts as an intermediary between the Model and the View.
   * **Responsibilities**:
     + Handles user inputs and routes them to the appropriate Model methods.
     + Updates the View with data from the Model.
   * **Example**:
     + In a blog application, the Controller processes a user request to add a new post by passing the data to the Model and refreshing the View.

### **How MVC Works**

1. **User Interaction**: The user interacts with the View (e.g., clicks a button or submits a form).
2. **Controller Handling**: The Controller processes the input, interprets it, and calls the appropriate Model method.
3. **Model Update**: The Model performs the required operations, such as updating the database or retrieving data.
4. **View Update**: The View is updated with new data from the Model and re-rendered for the user.

### 

### **Applications of MVC**

1. **Web Development**
   * Frameworks like **Django**, **Ruby on Rails**, and **Spring MVC** follow this architecture.
2. **Desktop Applications**
   * Used in frameworks like **JavaFX** or **Microsoft's .NET Framework**.
3. **Mobile Applications**
   * Adopted in **iOS (Cocoa MVC)** and **Android development**.

## Client-Server Model

The **client-server model** is a network architecture where two types of entities—clients and servers—communicate with each other to perform tasks. The server provides resources or services, and the client requests them. This model underpins the design of most modern networked systems, including web applications, email systems, and file-sharing platforms.

### **Key Components of the Client-Server Model**

1. **Client**
   * **Purpose**: A client is a device or application that initiates a request for resources or services from a server.
   * **Characteristics**:
     + Typically runs on end-user devices like PCs, smartphones, or tablets.
     + Sends requests to the server and processes the responses.
     + Examples: Web browsers, email clients, mobile apps.
2. **Server**
   * **Purpose**: A server is a system or application that provides resources, services, or data to clients.
   * **Characteristics**:
     + Typically hosted on powerful machines or cloud platforms.
     + Waits for client requests and responds to them.
     + Examples: Web servers, database servers, file servers.
3. **Network**
   * **Purpose**: Connects clients and servers to facilitate communication.
   * **Characteristics**:
     + Can be a local network (LAN) or the internet (WAN).
     + Uses protocols like HTTP, FTP, or SMTP for communication.

### **How the Client-Server Model Works**

1. **Request**: The client sends a request to the server using a specific protocol.
2. **Processing**: The server processes the request using its resources or services.
3. **Response**: The server sends the requested data or an appropriate response back to the client.

### **Examples of the Client-Server Model**

1. **Web Browsing**
   * **Client**: Web browsers like Chrome or Firefox.
   * **Server**: Web servers like Apache or Nginx serve websites.
2. **Email Systems**
   * **Client**: Email applications like Outlook or Gmail.
   * **Server**: Email servers using protocols like SMTP, IMAP, or POP3.
3. **Online Gaming**
   * **Client**: Gaming consoles or apps.
   * **Server**: Game servers handle multiplayer interactions.
4. **Database Access**
   * **Client**: Applications or software requesting data.
   * **Server**: Database servers like MySQL or PostgreSQL

### **Client-Server Protocols**

1. **HTTP/HTTPS**
   * Used for web applications to exchange data between web clients and servers.
2. **FTP**
   * Used for file transfer between a client and a server.
3. **SMTP, IMAP, POP3**
   * Used in email systems for sending and retrieving messages.
4. **SQL**
   * Used for querying and managing databases on database servers.

Frontend Development

**Frontend development** involves building the part of a web application that users interact with directly—the user interface (UI). It focuses on creating visually appealing, responsive, and user-friendly designs while ensuring seamless functionality and compatibility across devices and browsers.

### **Key Aspects of Frontend Development**

1. **Structure and Layout (HTML)**
   * **HTML (HyperText Markup Language)**: The backbone of web pages, used to structure content such as text, images, and links.
2. **Styling and Design (CSS)**
   * **CSS (Cascading Style Sheets)**: Controls the appearance of web pages, including colors, fonts, layouts, and responsiveness.
3. **Interactivity and Logic (JavaScript)**
   * Adds dynamic behaviors to web pages, such as animations, form validation, and interactive menus.

### **Frontend Development Frameworks and Libraries**

1. **Frameworks**
   * **React.js**: A library for building dynamic user interfaces with reusable components.
   * **Angular**: A complete framework for building single-page applications (SPAs).
   * **Vue.js**: A lightweight and versatile framework for building UIs.
2. **Styling Frameworks**
   * **Bootstrap**: A framework for responsive and mobile-first design.
   * **Tailwind CSS**: A utility-first CSS framework for custom designs.

### **Key Frontend Skills**

* HTML5 and CSS3
* JavaScript (ES6 and beyond)
* Knowledge of responsive design principles
* Understanding of API integration

Hypertext Markup Language

**HTML (HyperText Markup Language)**:

* The standard language used to create and design the structure of web pages.
* It defines elements like headings, paragraphs, images, links, and more.

## Basic Structure of an HTML Document

<!DOCTYPE html>

<html>

<head>

<title>Document Title</title>

</head>

<body>

<h1>Welcome to HTML</h1>

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

</body>

</html>

**Explanation:**

* <!DOCTYPE html>: Declares the document as HTML5.
* <html>: Root element containing all HTML content.
* <head>: Contains metadata (e.g., title, styles, links).
* <body>: Contains visible content of the web page.

**Commonly Used HTML Tags**

| **Tag** | **Description** |
| --- | --- |
| <h1> to <h6> | Headings, <h1> is the largest. |
| <p> | Paragraph text. |
| <a> | Hyperlink. |
| <img> | Displays images. |
| <ul> | Unordered list. |
| <ol> | Ordered list. |
| <li> | List item for <ul> or <ol>. |
| <div> | Generic container for block content. |
| <span> | Generic container for inline content. |
| <table> | Creates a table. |
| <form> | Creates an input form. |

### **Attributes**

* Attributes provide additional information about an element.
* **Syntax**: <tag attribute="value">

Example:

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

<img src="image.jpg" alt="Sample Image" width="300" height="200">

### **Types of Lists**

* **Unordered List**:

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

* **Ordered List**:

<ol>

<li>First Item</li>

<li>Second Item</li>

</ol>

``>

---

### \*\*6. Images\*\*

```html

<img src="path/to/image.jpg" alt="Description" width="300" height="200">

* src: Path to the image file.
* alt: Alternative text for accessibility.

### **Links**

<a href="https://example.com" target="\_blank">Click Here</a>

href: URL of the link.

target="\_blank": Opens link in a new tab.

### **Tables**

<table border="1">

<tr>

<th>Heading 1</th>

<th>Heading 2</th>

</tr>

<tr>

<td>Data 1</td>

<td>Data 2</td>

</tr>

</table>

* <table>: Creates a table.
* <tr>: Table row.
* <th>: Table header.
* <td>: Table data.

### **Forms**

<form action="/submit" method="POST">

<label for="name">Name:</label>

<input type="text" id="name" name="name">

<button type="submit">Submit</button>

</form>

action: URL to send the form data.

method: HTTP method (GET or POST).

### **Semantic HTML**

Semantic tags clearly describe their purpose:

* <header>: Page or section header.
* <footer>: Page footer.
* <article>: Independent content.
* <section>: Thematic grouping of content.
* <nav>: Navigation links.

### **Inline vs Block Elements**

* **Inline**: Does not start a new line (e.g., <span>, <a>).
* **Block**: Starts a new line and takes full width (e.g., <div>, <p>).

### **Comments in HTML**

Comments are not displayed on the webpage.

Example: <!-- This is a comment →

Cascading Style Sheets

**CSS (Cascading Style Sheets)**: A language used to style HTML elements, controlling the appearance and layout of web pages.

CSS allows you to change colors, fonts, spacing, layouts, and more.

**Basic CSS syntax:**

The CSS Syntax consists of a set of rules

selector{

property:value;

}

Example:

h1{

color:blue; font-size: 12px;

}

In the example, h1 is a selector, colour and font size are properties, blue and 12px are values, property and value together known as declaration and each declaration will be separated by semicolon

The selector points to the HTML element(tag) you want to style

A selector is an HTML tag at which style will be applied. This could be any tag like h1, li, p  etc.

* The declaration block contains one or more declarations separated by semicolon

The property is a type of attribute of HTML tag, values are assigned to property.

* For example, color property can have values either blue or red

Multiple CSS declarations are separated with semicolon and declaration blocks are surrounded by curly braces

### **Levels of stylesheet**

There are three levels of stylesheet used to associate CSS styles with your HTML document

* Inline
* Internal(document)
* External

#### Inline CSS - The style attribute

* Inline style sheet rules will be applied to the content of the one element or tag.
* Inline style sheet rules are specified as the value of the style attribute.
* Syntax: <element style = "style rule" >
* Example: <h1 style="color: red;">This is inline</h1>

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

#### Document Level or Internal CSS- The Style element

* Document level style sheet rule will be applied to all the elements available in the document. The Internal style is defined in the <style> element, inside the head section.

<style>

p {

color: green;

font-size: 16px;

}

</style>

#### External stylesheet

* With an external style sheet, you can change the look of an entire website by changing just one file!
* Each HTML page must include a reference to the external style sheet file inside the <link> element, inside the head section.

<link rel="stylesheet" href="styles.css">

## CSS Selectors

* **Universal Selector**: \* (Applies to all elements)

\* { margin: 0; padding: 0; }

* **Type Selector(Element selector)**: Targets a specific tag (e.g., p, h1)

p {

font-size: 14px;

}

* **Class Selector**: Targets elements with a specific class (.)

.highlight { background-color: yellow; }

* **ID Selector**: Targets an element with a specific ID (#)

#header { color: blue; }

* **Group Selector**: Targets multiple elements

h1, h2, h3 { font-family: Arial, sans-serif; }

* **Descendant Selector**: Targets elements within a parent

div p {

color: gray;

}

**Common Properties**

| **Property** | **Description** | **Example** |
| --- | --- | --- |
| color | Text color | color: red; |
| background-color | Background color | background-color: yellow; |
| font-size | Size of text | font-size: 20px; |
| font-family | Font style | font-family: Arial; |
| margin | Space outside an element | margin: 10px; |
| padding | Space inside an element | padding: 5px; |
| border | Border around an element | border: 1px solid black; |
| text-align | Text alignment | text-align: center; |
| width | Width of an element | width: 100%; |
| height | Height of an element | height: 300px; |

### **Box Model**

* Describes the layout and spacing of elements, including:
  1. **Content**: The actual content inside the element.
  2. **Padding**: Space between content and border.
  3. **Border**: The boundary around padding.
  4. **Margin**: Space outside the border.

Example:

div {

width: 200px;

padding: 10px;

border: 2px solid black;

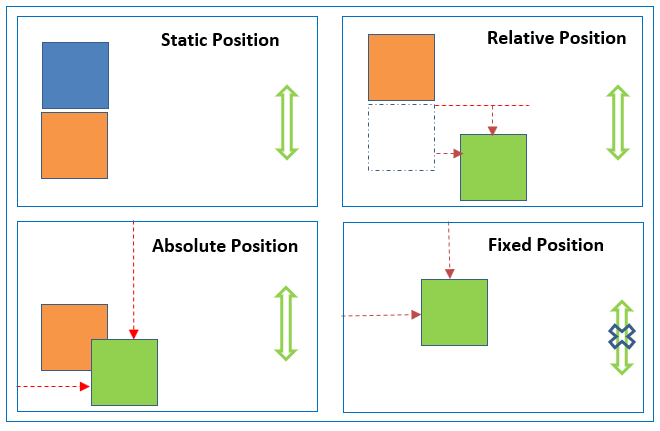
margin: 20px;

}

### **Positioning**

Controls the placement of elements:

* **Static**: Default positioning.
* **Relative**: Positioned relative to its normal position.
* **Absolute**: Positioned relative to its nearest positioned ancestor.
* **Fixed**: Positioned relative to the viewport.
* **Sticky**: Toggles between relative and fixed.



### CSS Flexbox

**Flexbox (Flexible Box Layout)** is a CSS layout model designed to align and distribute items efficiently in a container, even when the size of the items is unknown or dynamic.

Useful for creating responsive designs and handling spacing/alignment issues.

### **Key Concepts**

1. **Flex Container**: The parent element with display: flex; or display: inline-flex;.
2. **Flex Items**: The child elements of the flex container.

**Properties for Flex Containers**

| **Property** | **Description** | **Values** |
| --- | --- | --- |
| flex-direction | Defines the direction of flex items. | row (default), row-reverse, column, column-reverse |
| justify-content | Aligns items along the main axis. | flex-start, flex-end, center, space-between, space-around, space-evenly |
| align-items | Aligns items along the cross axis. | stretch (default), flex-start, flex-end, center, baseline |
| align-content | Aligns multiple lines of content (if wrapping is enabled). | stretch, flex-start, flex-end, center, space-between, space-around |
| flex-wrap | Specifies whether items wrap or stay on one line. | nowrap (default), wrap, wrap-reverse |

**Properties for Flex Items**

| **Property** | **Description** | **Values** |
| --- | --- | --- |
| order | Controls the order of the items. | Integer (default is 0) |
| flex-grow | Specifies how much a flex item should grow relative to the others. | Number (default is 0) |
| flex-shrink | Specifies how much a flex item should shrink relative to the others. | Number (default is 1) |
| flex-basis | Sets the initial size of the flex item before growing or shrinking. | auto, <length> |
| align-self | Overrides align-items for a specific item. | auto, flex-start, flex-end, center, baseline, stretch |

**Example**

<!--- html–>

<div class="container">

<div class="item">1</div>

<div class="item">2</div>

<div class="item">3</div>

</div>

.container {

display: flex;

flex-direction: row; /\* Items arranged horizontally \*/

justify-content: space-around; /\* Space around items \*/

align-items: center; /\* Items centered vertically \*/

height: 200px;

}

.item {

background-color: lightblue;

padding: 20px;

border: 1px solid blue;

}

### CSS z-index

The **z-index** property in CSS determines the **stacking order** of elements along the **z-axis** (visual depth).

It defines which elements appear in front of or behind others.

Works only on elements that have a **position** other than static (e.g., relative, absolute, fixed, or sticky).

**Values of z-index**

| **Value** | **Description** |
| --- | --- |
| auto | Default value. The stack order of the element is determined by its order in the DOM. |
| Positive Integers (1, 10, 100) | Places the element above elements with lower values. |
| Negative Integers (-1, -10) | Places the element below elements with higher values or 0. |
| 0 | Places the element at the default stacking context. |

**How z-index Works**

* Higher Values Appear on Top:
  + Elements with a higher z-index value are rendered above elements with a lower value.
* Stacking Context:
  + A stacking context is created in the following cases:
    - When an element has a z-index and a positioned property (relative, absolute, etc.).
    - CSS properties like opacity (< 1), transform, or filter also create a stacking context.
  + Elements within the same stacking context only compare their z-index values within that context.

Example

<div class="box1">Box 1</div>

<div class="box2">Box 2</div>

<div class="box3">Box 3</div>

.box1 {

position: relative;

z-index: 1;

background: lightblue;

width: 100px;

height: 100px;

}

.box2 {

position: relative;

z-index: 2;

background: lightgreen;

width: 100px;

height: 100px;

margin-top: -50px;

}

.box3 {

position: relative;

z-index: 0;

background: lightcoral;

width: 100px;

height: 100px;

margin-top: -50px;

}

## CSS Grid

### **What is CSS Grid?**

* **CSS Grid** is a powerful layout system that allows you to create complex two-dimensional layouts (both rows and columns) in a flexible and efficient way.
* It provides more control over the arrangement of items in a grid than traditional layout methods (like flexbox or floats).

### **Basic Terminology**

* **Grid Container**: The element that defines the grid layout. It is the parent element with display: grid;.
* **Grid Items**: The direct child elements of the grid container, which are placed within the grid cells.
* **Grid Lines**: The horizontal and vertical lines that divide the grid into rows and columns.
* **Grid Tracks**: The space between two grid lines (i.e., rows or columns).
* **Grid Cells**: The individual spaces within the grid formed by intersecting grid lines.

### **Defining Rows and Columns**

* **grid-template-columns**: Specifies the width of each column.
* **grid-template-rows**: Specifies the height of each row

### **Creating Gaps Between Rows and Columns**

* **grid-gap**: Defines the space between grid items.
* **grid-column-gap**: Defines space between columns.
* **grid-row-gap**: Defines space between rows.

### **Justify and Align Content in the Grid**

* **justify-content**: Aligns the entire grid container's content horizontally.
* **align-content**: Aligns the entire grid container's content vertically.

Example:

<div class="container">

<div class="item1">Item 1</div>

<div class="item2">Item 2</div>

<div class="item3">Item 3</div>

<div class="item4">Item 4</div>

</div>

.container {

display: grid;

grid-template-columns: 1fr 1fr 1fr;

grid-template-rows: 100px 100px;

grid-gap: 10px;

}

.item1 {

background: lightblue;

grid-column: 1 / 3; /\* Spans columns 1 to 3 \*/

}

.item2 {

background: lightgreen;

}

.item3 {

background: lightcoral;

}

.item4 {

background: lightyellow;

}

**Responsive Grid Layouts**

.container {

display: grid;

grid-template-columns: 1fr 1fr 1fr;

grid-gap: 10px;

}

@media (max-width: 600px) {

.container {

grid-template-columns: 1fr; /\* Single column on small screens \*/

}

}

### **Benefits of Using CSS Grid**

1. **2D Layouts**: Grid allows both row and column alignment, unlike flexbox, which is one-dimensional.
2. **Precision**: Grid allows you to create precise, complex layouts with less effort.
3. **Responsiveness**: Easily create responsive designs using fr units, auto, and media queries.
4. **Align Items and Content**: Aligning both grid items and the entire grid content becomes much easier.

## CSS Responsive Design

* **Responsive Web Design (RWD)** ensures that web pages adapt to different screen sizes and orientations. It aims to provide an optimal viewing experience across a wide range of devices, from desktop monitors to smartphones.
* It uses flexible layouts, images, and media queries to create a fluid, adaptable design.

### **Key Principles of Responsive Design**

1. **Fluid Layouts**:
   * Use flexible units like percentages (%), vw (viewport width), and vh (viewport height) to set widths, margins, and padding. This allows the layout to scale according to the size of the viewport.
2. **Flexible Images**:
   * Images should scale with the size of the container. Use CSS to set max-width: 100% to make images responsive.
3. **Media Queries**:
   * Media queries are key to responsive design. They apply CSS styles based on specific conditions like screen width, height, or device orientation.

### **Fluid Layouts**

A fluid layout in CSS adapts to the screen size, allowing the design to adjust as the window is resized. You can achieve this by using relative units like percentages or vw (viewport width), rather than fixed pixel values.

* Use **percentages**, **vw**, **vh**, and **em**/rem units instead of fixed pixel values.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Fluid Layout Example</title>

<style>

body {

margin: 0;

font-family: Arial, sans-serif;

}

.container {

width: 100%;

margin: 0 auto;

padding: 20px;

}

.header, .footer {

background-color: #333;

color: white;

text-align: center;

padding: 20px;

}

.main {

display: flex;

flex-wrap: wrap;

justify-content: space-between;

}

.main > .box {

width: 48%; /\* Fluid width, taking up half of the container \*/

background-color: #f4f4f4;

padding: 20px;

margin-bottom: 20px;

box-sizing: border-box;

}

@media (max-width: 768px) {

.main > .box {

width: 100%; /\* Stack the boxes on smaller screens \*/

}

}

</style>

</head>

<body>

<div class="container">

<div class="header">

<h1>Fluid Layout Example</h1>

</div>

<div class="main">

<div class="box">

<h2>Box 1</h2>

<p>This box will take up 48% of the width on larger screens.</p>

</div>

<div class="box">

<h2>Box 2</h2>

<p>This box will also take up 48% of the width on larger screens.</p>

</div>

</div>

<div class="footer">

<p>&copy; 2025 Fluid Layout Example</p>

</div>

</div>

</body>

</html>

### **Flexible Images**

To make images flexible in CSS, you can use the max-width property along with height: auto;. This ensures that the images scale appropriately while maintaining their aspect ratio

* **max-width: 100%** ensures images scale with their parent container, preventing overflow.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Flexible Images Example</title>

<style>

body {

margin: 0;

font-family: Arial, sans-serif;

}

.container {

width: 80%;

margin: 0 auto;

padding: 20px;

text-align: center;

}

img {

max-width: 100%; /\* Makes the image scale to the container's width \*/

height: auto; /\* Maintains aspect ratio \*/

display: block; /\* Removes space below the image \*/

margin: 0 auto; /\* Centers the image \*/

}

</style>

</head>

<body>

<div class="container">

<h1>Flexible Image Example</h1>

<img src="https://via.placeholder.com/1200x800" alt="Placeholder Image">

<p>The image will resize automatically according to the container size while maintaining its aspect ratio.</p>

</div>

</body>

</html>

### **Explanation:**

* The img tag has a max-width: 100%; property, which ensures the image will not exceed the width of its parent container (in this case, .container).
* The height: auto; property maintains the image's aspect ratio as it scales up or down.
* display: block; is used to remove any unwanted space beneath the image (which is typically caused by inline-block behavior of images).
* margin: 0 auto; centers the image within its container.

### **Media Queries**

* Media queries allow you to apply CSS styles based on conditions such as viewport width, height, orientation, and more.

**Common Conditions**:

* max-width: Applies styles when the viewport is smaller than the specified width.
* min-width: Applies styles when the viewport is larger than the specified width.
* orientation: Detects the screen's orientation (portrait or landscape).

### **Breakpoints**

* **Breakpoints** are specific widths at which the layout should adjust.
* Common breakpoints:
  + 320px: Small mobile devices.
  + 600px: Medium-sized devices (tablets).
  + 768px: Tablet-sized screens.
  + 1024px: Desktop-sized screens.
  + 1200px: Large desktop screens.

### **Mobile-First Design**

* **Mobile-first** approach involves designing for smaller screens (mobile devices) first, then using media queries to adjust styles for larger screens.
* This ensures that mobile users get a faster-loading, optimized experience.

### **Viewport Meta Tag (For Mobile)**

* To control the viewport's size and scaling on mobile devices, use the <meta> tag inside the <head> of the HTML document.

**width=device-width**: Makes the width of the page match the width of the device.

**initial-scale=1.0**: Sets the initial zoom level when the page is loaded.

Example:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Responsive Design Example</title>

<style>

/\* Mobile-first base styles \*/

body {

margin: 0;

font-family: Arial, sans-serif;

padding: 0;

}

.container {

width: 100%;

padding: 10px;

box-sizing: border-box;

}

.header, .footer {

background-color: #333;

color: white;

text-align: center;

padding: 10px;

}

.main {

text-align: center;

margin-top: 20px;

}

.main > .box {

width: 100%; /\* Full width on small screens \*/

background-color: #f4f4f4;

padding: 20px;

margin-bottom: 20px;

box-sizing: border-box;

}

/\* Tablet \*/

@media (min-width: 600px) {

.main {

display: flex;

justify-content: space-around;

}

.main > .box {

width: 45%; /\* 2 boxes side by side on tablets \*/

}

}

/\* Laptop / Desktop \*/

@media (min-width: 992px) {

.main > .box {

width: 30%; /\* 3 boxes side by side on larger screens \*/

}

}

/\* Large Desktop \*/

@media (min-width: 1200px) {

.container {

width: 80%; /\* Center content with more space on larger screens \*/

}

}

</style>

</head>

<body>

<div class="container">

<div class="header">

<h1>Responsive Design Example</h1>

</div>

<div class="main">

<div class="box">

<h2>Box 1</h2>

<p>This box will take up 100% of the width on mobile devices, 45% on tablets, and 30% on desktops.</p>

</div>

<div class="box">

<h2>Box 2</h2>

<p>This box will adjust similarly depending on screen size.</p>

</div>

<div class="box">

<h2>Box 3</h2>

<p>On larger screens, you will see 3 boxes in a row. On mobile, the boxes stack.</p>

</div>

</div>

<div class="footer">

<p>&copy; 2025 Responsive Design Example</p>

</div>

</div>

</body>

</html>

### **Explanation:**

1. **Mobile-First Approach:**
   * The base styles are designed for mobile devices (small screens).
   * The .box elements have width: 100%, meaning they take up the entire width of the container on mobile devices.
2. **Media Queries for Larger Screens:**
   * **@media (min-width: 600px)**: Targets devices with a screen width of at least 600px (tablets and larger). The .box elements will be displayed side by side with width: 45%.
   * **@media (min-width: 992px)**: Targets devices with a screen width of at least 992px (laptops and larger). The .box elements will be displayed in 3 columns, each with width: 30%.
   * **@media (min-width: 1200px)**: For screens larger than 1200px (large desktops), the .container is given a width: 80% to center it with some space around it.

### **Breakpoints:**

* **Mobile (default)**: No media query required, as the base styles are mobile-friendly.
* **Tablet (≥600px)**: @media (min-width: 600px)
* **Laptop/Desktop (≥992px)**: @media (min-width: 992px)
* **Large Desktop (≥1200px)**: @media (min-width: 1200px)

### **Key Concepts:**

* **Mobile-first**: Start with styles for mobile devices and add breakpoints to adjust for larger screens.
* **Flexible Layouts**: Use percentages (%) or flexbox to create responsive designs that adjust to different screen sizes.
* **Media Queries**: Use media queries to apply styles based on specific screen widths, making the design responsive.

## CSS Animations

### **What are CSS Animations?**

* **CSS Animations** allow you to animate HTML elements without the need for JavaScript.
* They provide a way to change CSS properties smoothly over time, creating dynamic effects like movement, color changes, scaling, rotations, and more.

Syntax:

@keyframes animationName {

from {

/\* Initial state of the animation \*/

property: value;

}

to {

/\* Final state of the animation \*/

property: value;

}

}

.element {

animation: animationName duration timing-function delay iteration-count direction;

}

**@keyframes**: Defines the animation and its behavior over time.

**animation**: A shorthand property that includes animation name, duration, timing function, delay, iteration count, and direction.

### **Key Properties for CSS Animations**

* **@keyframes**: Defines the sequence of frames in the animation.
  + **from**: Starting point of the animation (0%).
  + **to**: Ending point of the animation (100%)
* **animation-name**: The name of the animation (the name defined in @keyframes).
* **animation-duration**: Defines how long the animation lasts.
* **animation-timing-function**: Specifies the speed curve of the animation (how the transition happens over time).
* **Common values:**
  + linear: Constant speed.
  + ease: Starts slow, speeds up, then slows down.
  + ease-in: Starts slow and speeds up.
  + ease-out: Starts fast and slows down.
  + ease-in-out: Starts and ends slow, speeds up in the middle.
* **animation-delay**: Defines a delay before the animation starts.
* **animation-iteration-count**: Specifies how many times the animation should run.
* **infinite**: The animation runs indefinitely.
* **animation-direction**: Defines whether the animation should play forward, backward, or alternate between forward and backward.
  + normal: Default, plays from start to finish.
  + reverse: Plays the animation in reverse.
  + alternate: Alternates between forward and backward.
* **animation-fill-mode**: Defines the style of the element after the animation ends.
  + forwards: The animation will retain the styles of the last frame.
  + backwards: The animation will retain the styles of the first frame.
  + both: Both forwards and backwards.

Example:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

<title>Simple Animation Example</title>

<style>

/\* Base styling for the rectangle \*/

.rectangle {

width: 100px;

height: 100px;

background-color: #4CAF50;

position: absolute;

top: 50%;

left: 50%;

transform-origin: center; /\* To rotate around the center \*/

animation: rotateAndMove 4s linear infinite;

}

/\* Keyframes for the animation \*/

@keyframes rotateAndMove {

0% {

transform: translate(-50%, -50%) rotate(0deg); /\* Initial position and rotation \*/

}

50% {

transform: translate(150px, -50%) rotate(180deg); /\* Move right and rotate \*/

}

100% {

transform: translate(300px, -50%) rotate(360deg); /\* Move further right and complete rotation \*/

}

}

</style>

</head>

<body>

<div class="rectangle"></div>

</body>

</html>

# 

# JavaScript

**JavaScript** is a **high-level, interpreted programming language** that is used to create interactive and dynamic websites. It enables the manipulation of web page elements, control of browser events, communication with servers, and more.

It is an essential part of the **web development stack**, alongside HTML (structure) and CSS (style).

**Variables in Javascript**

Variables are used to store data values that can be referenced and manipulated in a program. They act as containers for data, allowing developers to store and retrieve values during the execution of the program.

### **Declaring Variables**

JavaScript provides three keywords to declare variables:

* var
* let
* const

**Types of Variables**

#### **var (Older Method)**

* Declares a variable that can be re-assigned and re-declared.
* Has **function-scoped** or **global-scoped** behavior.
* Not recommended for modern JavaScript, as it can lead to issues like hoisting and scoping problems.

var name = "John"; // Declare a variable using var

name = "Alice"; // Reassigning the variable

#### **let (Modern Method)**

* Declares a variable with block scope (limited to the block, statement, or expression where it's used).
* Can be reassigned but cannot be redeclared within the same scope.
* Preferred for most cases where re-assignment is needed.

let age = 25; // Declare a variable using let

age = 30; // Reassigning the variable

#### **const (Constant)**

* Declares a constant, meaning the variable cannot be reassigned after initialization.
* Also block-scoped.
* Typically used for values that are not expected to change, like configuration values.

const birthYear = 1995; // Declare a constant using const

// birthYear = 2000; // Error! Cannot reassign a constant

#### Variable Hoisting

* JavaScript **hoists** declarations, but **not initializations**. This means that var and let declarations are moved to the top of their respective scopes during compilation.

console.log(x); //

Undefined var x = 5;

console.log(y); // Error: Cannot access 'y' before initialization

let y = 10;

#### Variable Scope

* **Global Scope**: A variable declared outside of any function or block is in the global scope. It is accessible anywhere in the code.
* **Function Scope**: Variables declared inside a function using var are available only within that function.
* **Block Scope**: Variables declared using let and const are limited to the block (enclosed in {}) in which they are declared.

#### Best Practices

* Use let and const instead of var for better scoping and avoid potential issues with hoisting and accidental re-declarations.
* Use const for variables that should not be reassigned.
* Keep variable names descriptive and follow naming conventions (camelCase for variables).

#### **Data Types**

JavaScript supports various data types:

1. **Primitive Types**:
   * **Number**: let a = 5;
   * **String**: let str = "Hello";
   * **Boolean**: let isTrue = true;
   * **Null**: let obj = null;
   * **Undefined**: let data;
   * **Symbol**: let sym = Symbol('id');
2. **Object Types**:
   * **Object**: let person = { name: "John", age: 30 };
   * **Array**: let arr = [1, 2, 3];
   * **Function**: Functions are also objects.

### **Operators**

1. **Arithmetic Operators**:
   * +, -, \*, /, %, ++, --
2. **Assignment Operators**:
   * =, +=, -=, \*=, /=
3. **Comparison Operators**:
   * ==, ===, !=, !==, >, <, >=, <=
4. **Logical Operators**:
   * && (AND), || (OR), ! (NOT)

### **Control Flow**

1. **Conditional Statements**:
   * if, else, else if

if (x > y) {

console.log("x is greater than y");

} else if (x < y) {

console.log("x is smaller than y");

} else {

console.log("x and y are equal");

}

* Switch Statement:

switch (day) {

case 1:

console.log("Sunday");

break;

case 2:

console.log("Monday");

break;

default:

console.log("Invalid day");

}

**Loops**:

* **for loop**: Runs a block of code for a specified number of times.
* **while loop**: Continues until a condition is false.
* **do...while loop**: Executes code at least once, then repeats while the condition is true.

for (let i = 0; i < 5; i++) {

console.log(i);

}

### 

### **Functions**

A function in JavaScript is a block of reusable code designed to perform a specific task. Functions help make code modular, reusable, and easier to maintain.

#### Declaring a Function

A function is declared using the function keyword, followed by a function name, parentheses (), and a block {} that contains the code.

**Syntax:**

function functionName(parameters) {

// Function body

return value; // (optional)

}

#### Function Parameters & Arguments

* **Parameters** are placeholders defined in the function declaration.
* **Arguments** are actual values passed to the function when calling it.

function greetUser(name) {

console.log("Hello, " + name + "!");

}

greetUser("Alice"); // Output: Hello, Alice!

#### Function Return Statement

The return statement is used to send a value back to the caller.

function multiply(x, y) {

return x \* y;

}

let result = multiply(3, 4);

console.log(result); // Output: 12

* Functions **without return** simply execute code but do not return a value.

#### Function Expressions (Anonymous Functions)

A function can also be stored inside a variable. These are called **function expressions**.

let square = function(num) {

return num \* num;

};

console.log(square(5)); // Output: 25

* Function expressions **do not have names** (anonymous functions).
* They must be defined **before** calling them.

#### Arrow Functions (ES6)

Arrow functions provide a shorter syntax for writing functions.

**Syntax:**

const functionName = (parameters) => expression;

Example:

const add = (a, b) => a + b;

console.log(add(4, 6)); // Output: 10

**Single Parameter (No Parentheses Needed)**

const greet = name => console.log("Hello, " + name);

greet("John"); // Output: Hello, John

##### Multi-line Arrow Function

const multiply = (a, b) => {

let result = a \* b;

return result;

};

console.log(multiply(3, 7)); // Output: 21

#### Function Scope

* **Global Scope**: Variables declared outside a function can be accessed anywhere.
* **Local Scope**: Variables declared inside a function are only accessible within that function.

let globalVar = "I am global";

function testScope() {

let localVar = "I am local";

console.log(globalVar); // Accessible

console.log(localVar); // Accessible

}

testScope();

console.log(globalVar); // Accessible

// console.log(localVar); // Error! Not defined outside function

#### Default Parameters (ES6)

If an argument is not passed, the function will use a default value.

function greet(name = "Guest") {

console.log("Hello, " + name);

}

greet(); // Output: Hello, Guest

greet("Alice"); // Output: Hello, Alice

#### Immediately Invoked Function Expression (IIFE)

A function that runs immediately after being defined.

(function() {

console.log("I am an IIFE!");

})(); // Output: I am an IIFE!

**Why use IIFE?**

* To avoid polluting the global scope.
* Useful for initialization code.

#### Higher-Order Functions

Functions that take another function as an argument or return a function.

function operate(a, b, operation) {

return operation(a, b);

}

let sum = operate(5, 3, (x, y) => x + y);

console.log(sum); // Output: 8

#### Function Hoisting

* **Function declarations** are hoisted to the top of their scope.
* **Function expressions** are **not hoisted**.

hoistedFunction(); // Works fine

function hoistedFunction() {

console.log("This function is hoisted!");

}

—-------------------------------------------------

nonHoistedFunction(); // Error! Cannot access before initialization

const nonHoistedFunction = function() {

console.log("This function is NOT hoisted!");

};

#### Closures

A closure is a function that remembers its surrounding variables even after execution

function outer() {

let count = 0;

return function inner() {

count++;

console.log(count);

};

}

let counter = outer();

counter(); // Output: 1

counter(); // Output: 2

## JavaScript Objects

An object in JavaScript is a collection of key-value pairs where values (properties) can be of any data type, including functions. Objects allow us to store and manage related data efficiently.

Example:

let person = {

name: "John",

age: 30,

isStudent: false

};

### Creating an Object

#### Method 1: Object Literal (Most Common)

let car = {

brand: "Toyota",

model: "Camry",

year: 2022

};

#### Method 2: Using new Object() (Less Common)

let person = new Object();

person.name = "Alice";

person.age = 25;

#### Method 3: Using a Constructor Function

function Person(name, age) {

this.name = name;

this.age = age;

}

let john = new Person("John", 30);

#### Method 4: Using Object.create()

let personPrototype = {

greet: function() {

console.log("Hello!");

}

};

let newPerson = Object.create(personPrototype);

newPerson.greet(); // Output: Hello!

### Accessing Object Properties

#### Dot Notation (Recommended)

console.log(car.brand); // Output: Toyota

#### Bracket Notation (For Dynamic Keys)

console.log(car["model"]); // Output: Camry

#### Adding and Modifying Properties

**Adding New Properties**

car.color = "Red";

console.log(car.color); // Output: Red

**Modifying Existing Properties**

car.year = 2023;

console.log(car.year); // Output: 2023

**Deleting Properties**

delete car.color;

console.log(car.color); // Output: undefined

#### Object Methods (Functions Inside Objects)

An object method is a function stored as a property.

let person = {

name: "Alice",

age: 25,

greet: function() {

console.log("Hello, " + this.name);

}

};

person.greet(); // Output: Hello, Alice

#### **Checking if a Property Exists**

* **Using in Operator**

console.log("model" in car); // Output: true

console.log("color" in car); // Output: false

* **Using hasOwnProperty()**

console.log(car.hasOwnProperty("brand")); // Output: true

#### **Looping Through an Object**

* **Using for...in Loop**

for (let key in car) {

console.log(key + ": " + car[key]);

}

#### **Object Methods**

**Object.keys() – Get All Keys**

console.log(Object.keys(car)); // Output: ["brand", "model", "year"]

**Object.values() – Get All Values**

console.log(Object.values(car)); // Output: ["Toyota", "Camry", 2022]

### **Object.entries() – Get Key-Value Pairs**

console.log(Object.entries(car));

// Output: [["brand", "Toyota"], ["model", "Camry"], ["year", 2022]]

### **Object.assign() – Clone or Merge Objects**

let newCar = Object.assign({}, car); // Cloning an object

#### Nested Objects

Objects can contain other objects.

let student = {

name: "Mike",

marks: {

math: 90,

science: 85

}

};

console.log(student.marks.math); // Output: 90

### **JSON (JavaScript Object Notation)**

JSON is a lightweight data format that looks like JavaScript objects but is used for data exchange.

#### Converting an Object to JSON

let jsonData = JSON.stringify(student);

console.log(jsonData);

// Output: '{"name":"Mike","marks":{"math":90,"science":85}}'

#### Parsing JSON Back to an Object

let parsedData = JSON.parse(jsonData);

console.log(parsedData);

### this Keyword in Objects

* this refers to the object it belongs to.
* Used in methods to access object properties.

let user = {

name: "John",

greet: function() {

console.log("Hello, " + this.name);

}

};

user.greet(); // Output: Hello, John

### Object Destructuring

Extract values from objects into variables.

let user = { name: "Alice", age: 25 };

let { name, age } = user;

console.log(name); // Output: Alice

console.log(age); // Output: 25

### Spread Operator (...) with Objects

### **Merging Objects**

let obj1 = { a: 1, b: 2 };

let obj2 = { c: 3, d: 4 };

let merged = { ...obj1, ...obj2 };

console.log(merged); // Output: { a: 1, b: 2, c: 3, d: 4 }

### **Classes and Objects (ES6)**

Objects can also be created using classes.

class Car {

constructor(brand, model) {

this.brand = brand;

this.model = model;

}

display() {

console.log(this.brand + " " + this.model);

}

}

let myCar = new Car("Honda", "Civic");

myCar.display(); // Output: Honda Civic

### Object Freezing & Sealing

Object.freeze() – Prevent Modifications

let obj = { name: "John" };

Object.freeze(obj);

obj.name = "Alice"; // No effect

console.log(obj.name); // Output: John

Object.seal() – Allow Updates but No New Properties

let obj2 = { age: 30 };

Object.seal(obj2);

obj2.age = 35; // Allowed

obj2.city = "New York"; // Not Allowed

console.log(obj2); // Output: { age: 35 }

## Asynchronous JavaScript

JavaScript is single-threaded, meaning it executes code line by line. However, it uses an asynchronous model to handle tasks like:

* Fetching data from an API
* Reading files
* Timers (e.g., setTimeout)
* User interactions

This prevents blocking the main thread and keeps the app responsive.

#### **Synchronous vs. Asynchronous Execution**

##### **Synchronous Code (Blocking)**

Executes line by line, waiting for each operation to complete before moving to the next.

console.log("Start");

for (let i = 0; i < 1000000000; i++) {} // Blocking operation

console.log("End");

// Output:

// Start

// End (after delay)

##### **Asynchronous Code (Non-Blocking)**

Allows other operations to continue while waiting.

console.log("Start");

setTimeout(() => console.log("Async Task"), 2000);

console.log("End");

// Output:

// Start

// End

// (After 2 seconds) Async Task

## **Callbacks**

A callback is a function passed as an argument to another function, executed later.

#### **Why Use Callbacks?**

* JavaScript is **asynchronous** (especially in I/O operations, API calls, and event handling).
* Callbacks ensure that certain code executes only **after** another function completes execution.
* Helps in managing **asynchronous operations** like file reading, database queries, and API calls.

**Synchronous vs. Asynchronous Callbacks**

##### Synchronous Callbacks

* These are executed immediately within the function that calls them.
* Example:

function greet(name, callback) {

console.log("Hello, " + name);

callback();

}

function sayGoodbye() {

console.log("Goodbye!");

}

greet("John", sayGoodbye);

Output:

Hello, John

Goodbye!

##### **Asynchronous Callbacks**

* These are executed after an asynchronous operation completes (e.g., setTimeout, API calls, file reading).
* Example:

function fetchData(callback) {

setTimeout(() => {

console.log("Data fetched!");

callback();

}, 2000);

}

function processData() {

console.log("Processing data...");

}

fetchData(processData);

**Output (after 2 seconds):**

Data fetched!

Processing data…

**Callback in Asynchronous JavaScript (API Call Example)**

function getUserData(id, callback) {

setTimeout(() => {

console.log(`User data for ID: ${id}`);

callback();

}, 2000);

}

getUserData(101, function() {

console.log("Processing user data...");

});

#### **Callback Hell (Pyramid of Doom)**

* When multiple callbacks are nested, it leads to unreadable and difficult-to-maintain code.
* Example of **callback hell**:

setTimeout(() => {

console.log("Step 1");

setTimeout(() => {

console.log("Step 2");

setTimeout(() => {

console.log("Step 3");

}, 1000);

}, 1000);

}, 1000);

**Solution:** Use **Promises** or **async/await** instead of deep-nested callbacks.

#### **Replacing Callbacks with Promises**

Instead of nesting callbacks, use **Promises** to handle asynchronous operations cleanly:

function fetchData() {

return new Promise((resolve) => {

setTimeout(() => {

console.log("Data fetched!");

resolve();

}, 2000);

});

}

fetchData().then(() => {

console.log("Processing data...");

});

## Promises in JavaScript

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation. It provides a way to handle asynchronous code more efficiently and avoids **callback hell**.

#### **Why Use Promises?**

* Helps manage **asynchronous** operations like API calls, file reading, or database access.
* Provides better **readability** compared to nested callbacks.
* Supports **chaining** (.then()) and **error handling** (.catch()).

#### **States of a Promise**

A Promise can be in one of the following states:

1. **Pending** – The initial state, before the operation completes.
2. **Fulfilled** – The operation completed successfully.
3. **Rejected** – The operation failed.

**Creating a Promise**

let myPromise = new Promise((resolve, reject) => {

let success = true; // Simulating success or failure

setTimeout(() => {

if (success) {

resolve("Data fetched successfully!");

} else {

reject("Error fetching data!");

}

}, 2000);

});

myPromise

.then((result) => console.log(result)) // Runs if resolved

.catch((error) => console.log(error)) // Runs if rejected

.finally(() => console.log("Operation complete!")); // Always runs

**Chaining Promises**

function step1() {

return new Promise((resolve) => {

setTimeout(() => {

console.log("Step 1 completed");

resolve();

}, 1000);

});

}

function step2() {

return new Promise((resolve) => {

setTimeout(() => {

console.log("Step 2 completed");

resolve();

}, 1000);

});

}

step1().then(step2).then(() => console.log("All steps done!"));

**Handling Errors with .catch()**

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

let success = false;

if (success) {

resolve("Data fetched!");

} else {

reject("Failed to fetch data!");

}

}, 2000);

});

}

fetchData()

.then((data) => console.log(data))

.catch((error) => console.log("Error:", error));

#### **Using Promise.all() for Parallel Execution**

* Promise.all() runs multiple promises in parallel and waits for all to resolve.
* If **any** promise fails, it rejects the entire operation.

let promise1 = new Promise((resolve) => setTimeout(() => resolve("Data 1"), 2000));

let promise2 = new Promise((resolve) => setTimeout(() => resolve("Data 2"), 1000));

Promise.all([promise1, promise2])

.then((results) => console.log(results)) // ["Data 1", "Data 2"]

.catch((error) => console.log(error));

#### **Using Promise.race()**

* Promise.race() resolves/rejects as soon as **one** of the promises settles.

Promise.race([

new Promise((resolve) => setTimeout(() => resolve("Fast response"), 1000)),

new Promise((resolve) => setTimeout(() => resolve("Slow response"), 3000))

])

.then((result) => console.log(result)); // Output: "Fast response"

#### **Converting Callbacks to Promises**

function asyncOperation(callback) {

setTimeout(() => {

callback("Task completed!");

}, 2000);

}

// Converting to a Promise-based function

function asyncOperationPromise() {

return new Promise((resolve) => {

setTimeout(() => resolve("Task completed!"), 2000);

});

}

asyncOperationPromise().then(console.log);

## Async/Await in JavaScript

async/await is a modern way to handle asynchronous code in JavaScript, making it easier to read and write than using Promises with .then() and .catch().

* **async function**: Always returns a promise.
* **await keyword**: Pauses execution until the promise resolves

#### **Why Use async/await?**

* Improves **readability** (looks like synchronous code).
* Reduces **callback hell** and **promise chaining**.
* Handles **asynchronous** code efficiently.

**Using async and await**

async function fetchData() {

return "Data received!";

}

fetchData().then(console.log); // Output: "Data received!"

Since async always returns a promise, we use .then() to get the result.

**Using await Inside async**

async function getData() {

let result = await new Promise((resolve) => {

setTimeout(() => resolve("Data fetched!"), 2000);

});

console.log(result); // Output after 2 sec: "Data fetched!"

}

getData();

The await keyword **pauses** the function until the promise resolves.

This makes the code look synchronous while still being asynchronous.

async function fetchData() {

try {

let response = await new Promise((resolve, reject) => {

setTimeout(() => reject("Error fetching data!"), 2000);

});

console.log(response);

} catch (error) {

console.log("Caught error:", error);

}

}

fetchData();

#### **Using async/await with API Calls (Fetch Example)**

async function getUser() {

try {

let response = await fetch("https://jsonplaceholder.typicode.com/users/1");

let data = await response.json(); // Convert response to JSON

console.log(data);

} catch (error) {

console.log("API Error:", error);

}

}

getUser();

fetch() returns a promise, so await ensures we get the result before moving ahead.

await response.json() ensures JSON parsing is complete before logging.