



HTML & CSS

Basics Guide

Chapters Overview

Chapter 1 – Foundations of HTML and CSS

- 1.1 What is HTML and CSS?
 - 1.2 Evolution of HTML (HTML4, XHTML, HTML5)
 - 1.3 Page Structure and the Role of the DOM
 - 1.4 How CSS Styles Are Applied
 - 1.5 Embedding CSS: Inline, Internal, External
-

Chapter 2 – HTML5 Semantic Structure and Media

- 2.1 Semantic Layout Elements
 - 2.2 Common Text and Structural Elements
 - 2.3 Embedding Media: Images, Audio, Video, Iframes
 - 2.4 Forms and User Inputs in HTML5
-

Chapter 3 – Core CSS Concepts and Layout

- 3.1 CSS Syntax and Selectors
 - 3.2 Specificity and Inheritance
 - 3.3 Typography and Color Styling
 - 3.4 The Box Model Explained
 - 3.5 Display Types and Positioning
 - 3.6 Float and Clear
-

Chapter 4 – Responsive Design and Modern CSS Techniques

- 4.1 Mobile-First and Media Queries
- 4.2 Flexbox: Concepts and Practical Use
- 4.3 Grid Layout: Fundamentals Only
- 4.4 Transitions, Transforms, and Basic Animations
- 4.5 Pseudo-classes and Pseudo-elements
- 4.6 Custom Fonts with @font-face

Chapter 1 – Foundations of HTML and CSS

1.1 What is HTML and CSS?

HTML (HyperText Markup Language) is the foundational language used to create and structure the content on the web. It provides a way to describe elements like text, images, links, and other media so browsers can render them visually. HTML is not a programming language—it's a markup language designed to annotate content.

CSS (Cascading Style Sheets) is a styling language that controls how HTML elements appear on the screen. CSS defines colors, fonts, layout positioning, spacing, and visual transitions. Without CSS, all web pages would look plain and unstyled.

Why are both necessary?

- HTML answers: *What is on the page?* (e.g., a heading, a paragraph, an image)
- CSS answers: *How does it look?* (e.g., bold, centered, blue, with a shadow)

Together, HTML and CSS allow developers to build web pages that are both **structurally sound** and **visually engaging**.

Real-World Analogy:

Imagine a web page as a house:

- **HTML is the structure:** walls, rooms, doors.
- **CSS is the design:** paint, furniture, lighting.

1.2 Evolution of HTML

Early HTML (HTML 1.0 – 3.2)

In the early 1990s, HTML was very basic—limited to formatting text and adding links. Pages were static, and styling was embedded directly using presentational tags like `` and `<center>`.

HTML 4.01 and XHTML

Released in 1999, **HTML 4.01** added structure and flexibility to the language. It separated content from presentation and introduced support for:

- Tables, forms, and scripting (JavaScript)
- Basic styling through CSS
- The concept of standards compliance

At the same time, **XHTML** (a stricter XML-based version of HTML) emerged. It enforced stricter rules like:

- Proper nesting of elements
- Mandatory closing tags
- Lowercase tag names

However, XHTML was less forgiving and harder to work with in browsers, especially for beginners.

HTML5: The Modern Standard

HTML5, finalized around 2014, replaced both HTML 4.01 and XHTML. Its goals were to:

- Simplify the language
- Support multimedia (video, audio, canvas) without plugins
- Introduce **semantic elements** for better structure
- Improve browser error-handling
- Work better across devices (desktop, tablet, mobile)

HTML5 is **backward compatible** and **flexible**. It lets developers write cleaner, more meaningful code while enhancing accessibility and search engine optimization.

HTML Version	Key Features	Status
HTML 3.2	Tables, forms	Obsolete
HTML 4.01	Structure, CSS	Legacy
XHTML	XML syntax, stricter	Obsolete
HTML5	Modern, semantic, multimedia	Current

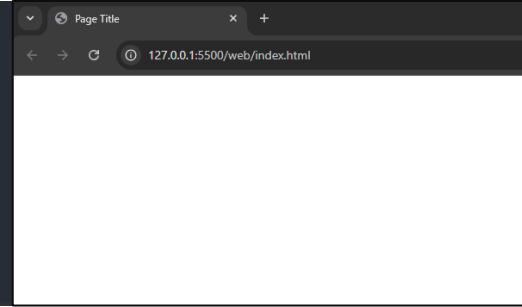
1.3 Page Structure and the Role of the DOM

HTML Document Structure

Every HTML page follows a consistent structure that browsers expect. The basic skeleton includes:

```

1  <!DOCTYPE html>
2  <html>
3  |   <head>
4  |       <title>Page Title</title>
5  |   </head>
6  |   <body>
7  |       <!-- Visible content goes here -->
8  |   </body>
9  </html>
```

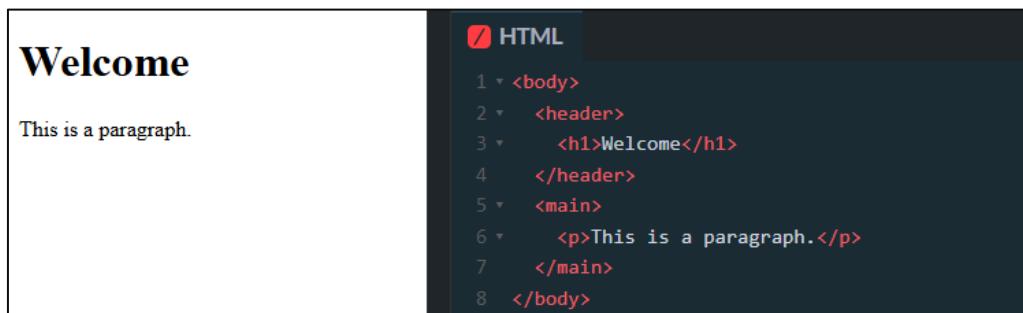


Key sections:

- `<!DOCTYPE html>`: Declares the document type as HTML5.
- `<html>`: Root element wrapping the entire document.
- `<head>`: Contains metadata (like the title, linked CSS, character encoding).
- `<body>`: Holds the visible content—text, images, links, etc.

Content Hierarchy

HTML follows a **tree-like hierarchy**. Example:



The screenshot shows a browser window with a dark theme. On the left, there's a preview of the page with the word "Welcome" and a paragraph of text. On the right, the developer tools are open, showing the DOM structure. The code pane displays the following HTML code:

```
1 <body>
2   <header>
3     <h1>Welcome</h1>
4   </header>
5   <main>
6     <p>This is a paragraph.</p>
7   </main>
8 </body>
```

Here, `<body>` is the parent of `<header>` and `<main>`. Each element can contain other elements, forming a **nested structure**.

What Is the DOM?

DOM (Document Object Model) is the browser's internal representation of the HTML page. It converts the HTML into a **live tree of nodes** that JavaScript and the browser can interact with.

- Elements like `<h1>` or `<p>` become **nodes** in the DOM.
- You can use **JavaScript** to change the DOM dynamically (e.g., hide/show content, change styles, update text).
- CSS also interacts with the DOM, applying styles based on selectors.

Think of the DOM as a **live map of your page** that can be queried, modified, and updated in real time.

Why DOM Matters

- Enables **dynamic interactivity** (buttons, forms, effects)
- Helps browsers **render and update** content efficiently
- Forms the foundation for **JavaScript-driven applications** like single-page apps (SPAs)

1.4 How CSS Styles Are Applied

CSS (Cascading Style Sheets) is used to control the appearance of HTML content. It allows you to define how elements should look—colors, fonts, spacing, layout, and more—separately from the HTML structure.

The Cascade

The "Cascading" part of CSS refers to **how styles are applied when there are multiple conflicting rules**. The browser follows these rules in order:

1. **Source order** – Later styles override earlier ones if they have the same specificity.
 2. **Specificity** – More specific selectors override less specific ones.
 3. **Importance** – Styles marked `!important` override normal rules (though best avoided).
 4. **Origin** – Browser default < Author-defined < User overrides
-

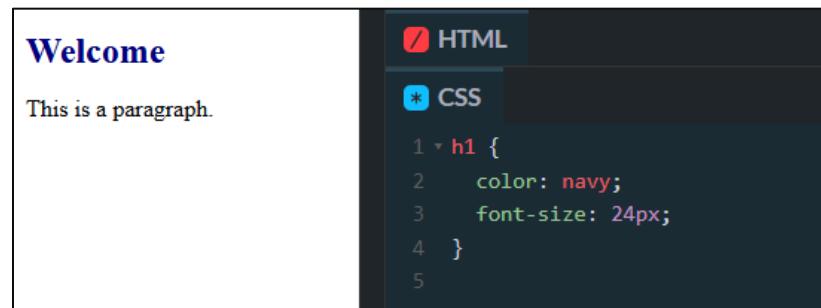
CSS Selectors

Selectors target HTML elements for styling. Examples:

Selector	Description
<code>p</code>	All <code><p></code> tags
<code>.note</code>	All elements with class <code>note</code>
<code>#main</code>	Element with ID <code>main</code>
<code>div > p</code>	<code><p></code> elements directly inside a <code><div></code>
<code>a:hover</code>	Links when hovered by the mouse

Style Properties

Each rule consists of a **property** and a **value**:



The screenshot shows a browser window with developer tools open. On the left, the page content says "Welcome" and "This is a paragraph.". On the right, the developer tools pane has two tabs: "HTML" (selected) and "CSS". Under "CSS", there is a single rule:

```

1 ▶ h1 {
2   color: navy;
3   font-size: 24px;
4 }
5

```

- `color` and `font-size` are **properties**
- `navy` and `24px` are **values**

How Browsers Apply CSS

1. Parse HTML into the DOM tree
 2. Load and apply CSS to elements using selectors
 3. Compute the final visual style using the cascade
 4. Render the page on screen
-

Why CSS Is Powerful

- It keeps content and design **separated**
- Allows **global style changes** by editing one file
- Enables **responsive and dynamic designs**

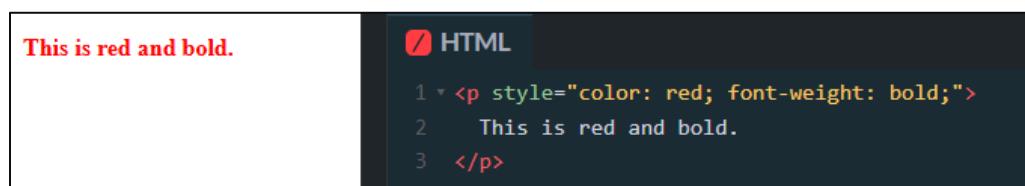
Without CSS, every webpage would look like plain black text on a white screen.

1.5 Embedding CSS: Inline, Internal, External

There are **three main ways** to apply CSS styles to a webpage. Each has its ideal use depending on the situation.

1. Inline CSS

Style is added directly to an HTML element using the `style` attribute.



The screenshot shows a code editor interface with two panes. The left pane displays the rendered HTML content: "This is red and bold." The right pane shows the corresponding HTML code:

```
1 <p style="color: red; font-weight: bold;">
2   This is red and bold.
3 </p>
```

Pros:

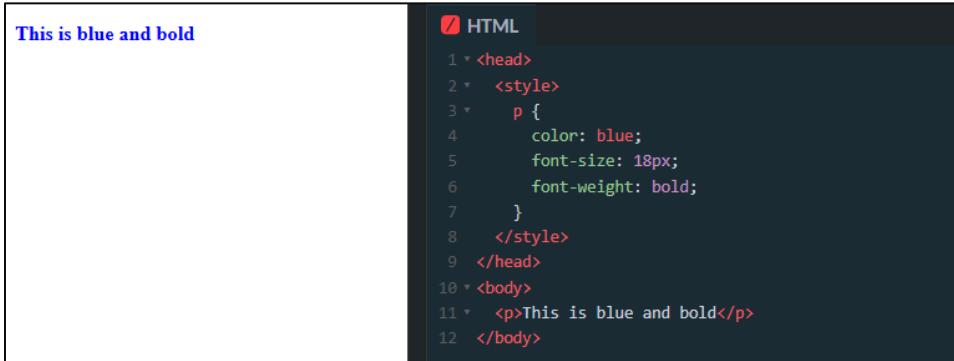
- Quick to apply
- Useful for one-time changes or testing

Cons:

- Breaks separation of content and style
- Harder to maintain
- Repetition across multiple elements

2. Internal CSS

Style rules are written inside a `<style>` block in the `<head>` of the HTML document.



This is blue and bold

```
✓ HTML
1 ▾ <head>
2 ▾   <style>
3 ▾     p {
4       color: blue;
5       font-size: 18px;
6       font-weight: bold;
7     }
8   </style>
9 </head>
10 ▾ <body>
11 ▾   <p>This is blue and bold</p>
12 </body>
```

Pros:

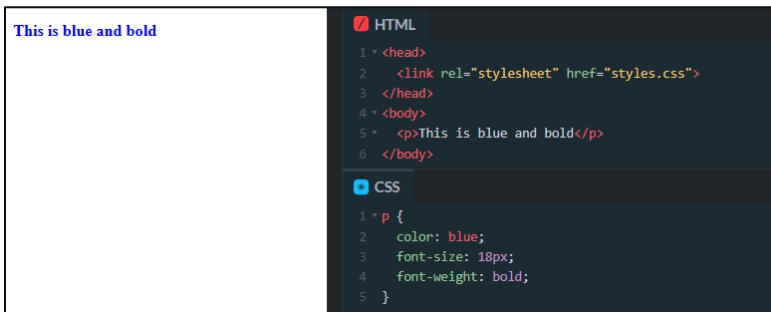
- Better than inline for styling entire pages
- Keeps style centralized within the same file

Cons:

- Doesn't scale well across multiple pages
- Still mixes style with structure

3. External CSS

Style rules are placed in a separate `.css` file and linked to the HTML using a `<link>` tag.



This is blue and bold

```
✓ HTML
1 ▾ <head>
2   <link rel="stylesheet" href="styles.css">
3 </head>
4 ▾ <body>
5 ▾   <p>This is blue and bold</p>
6 </body>
```

```
▢ CSS
1 p {
2   color: blue;
3   font-size: 18px;
4   font-weight: bold;
5 }
```

Pros:

- **Best practice** for large projects
- Clean separation of structure and style
- Reusable across multiple HTML pages
- Easy to maintain and update

Cons:

- Requires extra file load (minimal issue with caching)
-

Summary Table

Method	Use Case	Maintainability
Inline	Small, quick changes	Low
Internal	One-page projects or tests	Medium
External	Full websites, scalable apps	High (preferred)

Best Practice:

Use **external CSS** for long-term projects. Reserve **internal** for quick prototypes and **inline** only for rare exceptions.

Chapter 2 – HTML5 Semantic Structure and Media

2.1 Semantic Layout Elements

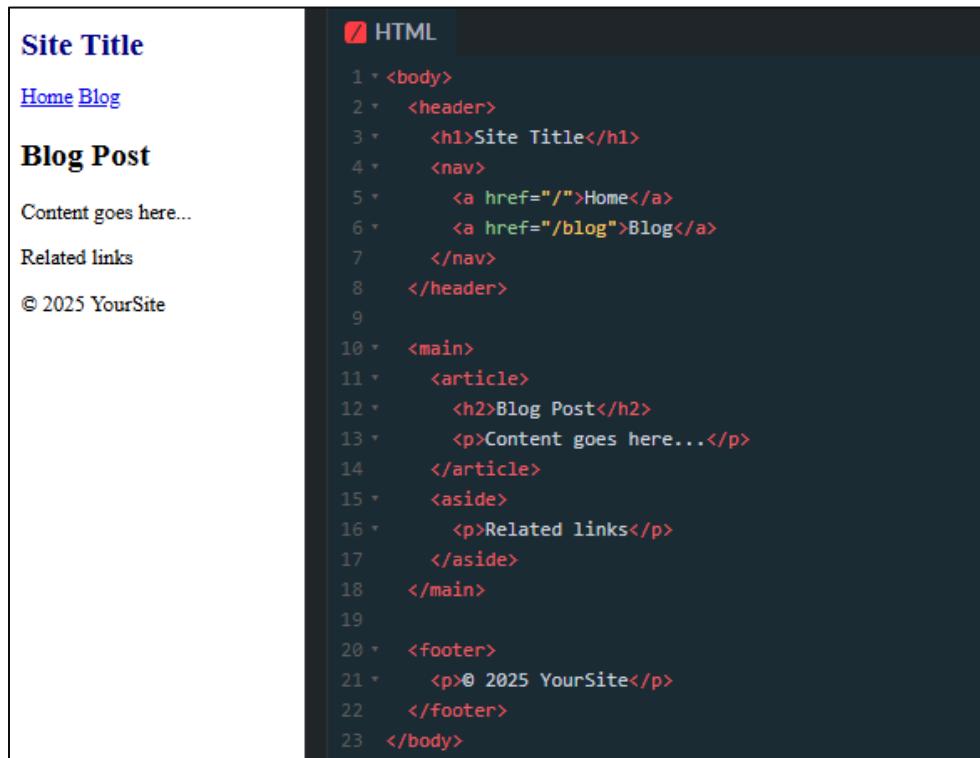
What Are Semantic Elements?

Semantic elements clearly describe their **meaning** both to the browser and the developer. Instead of using generic containers like `<div>`, semantic elements have **descriptive names** that improve readability, accessibility, and SEO.

Common Semantic Layout Elements

Element	Purpose
<code>header</code>	Introductory content or navigation for a section
<code>nav</code>	Major blocks of navigation links
<code>main</code>	Central content unique to the page
<code>section</code>	Thematic grouping of related content
<code>article</code>	Independent, reusable content (e.g., blog post)
<code>aside</code>	Related side content (e.g., ads, sidebars)
<code>footer</code>	Bottom section with closing info or navigation

Sample Semantic Page Layout



The screenshot displays a code editor interface with two panes. The left pane shows a visual representation of a website layout. It includes a header section with a title and navigation links, a main content area with a blog post, and a footer section with copyright information. The right pane shows the semantic HTML code for this layout. The code uses various semantic elements like `<header>`, `<main>`, and `<article>` to structure the content.

```
Site Title
Home Blog
Blog Post
Content goes here...
Related links
© 2025 YourSite

HTML
1 <body>
2   <header>
3     <h1>Site Title</h1>
4     <nav>
5       <a href="/">Home</a>
6       <a href="/blog">Blog</a>
7     </nav>
8   </header>
9
10  <main>
11    <article>
12      <h2>Blog Post</h2>
13      <p>Content goes here...</p>
14    </article>
15    <aside>
16      <p>Related links</p>
17    </aside>
18  </main>
19
20  <footer>
21    <p>© 2025 YourSite</p>
22  </footer>
23 </body>
```

Why Semantic Elements Matter

- **Improves accessibility:** Screen readers can identify sections clearly
- **Boosts SEO:** Search engines better understand content purpose
- **Enhances maintainability:** Code becomes self-descriptive
- **Future-friendly:** Designed to work well across platforms and devices

Before HTML5, developers relied heavily on `<div id="header">`, `<div id="footer">`, etc. Semantic elements now eliminate this need.

2.2 Common Text and Structural Elements

In addition to layout containers, HTML provides a variety of elements to structure and format the content itself—such as headings, paragraphs, lists, quotes, and inline text enhancements.

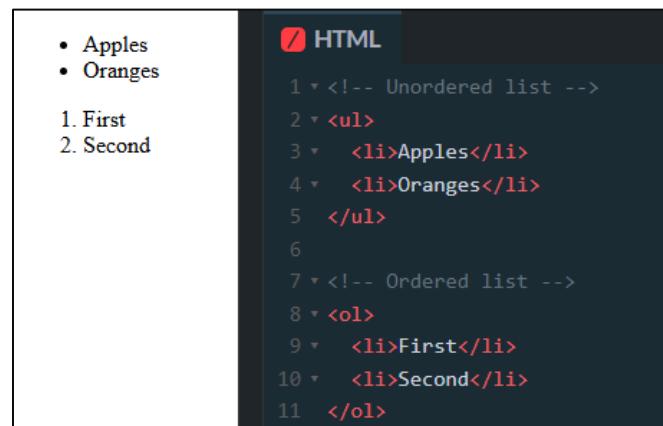
Text Content Elements

Element	Use Case
h1–h6	Headings, from most to least important
p	Paragraphs of text
br	Line breaks (use sparingly)
hr	Thematic breaks (horizontal lines)

Headings should follow a **hierarchical order**. Use one `<h1>` per page, typically for the main title.

Lists

HTML supports two main types of lists:



The screenshot shows a code editor interface with a dark theme. On the left, there is a sidebar containing two lists:

- Apples
- Oranges

1. First
2. Second

On the right, the main editor area displays the corresponding HTML code:

```
1 <!-- Unordered list -->
2 <ul>
3   <li>Apples</li>
4   <li>Oranges</li>
5 </ul>
6
7 <!-- Ordered list -->
8 <ol>
9   <li>First</li>
10  <li>Second</li>
11 </ol>
```

- `ul` = bullet points
 - `ol` = numbered items
 - `li` = list item
-

Blockquotes and Quotes

Use block-level and inline quoting for cited content:

This is a
long
quotation.

He said, “let’s go.”

/ HTML

```
1 <blockquote cite="https://example.com">
2   This is a long quotation.
3 </blockquote>
4
5 <p>He said, <q>let's go.</q></p>
```

- `blockquote`: for long quotes
 - `q`: for inline quotations
 - `cite`: to attribute a source
-

Emphasizing Text

Element	Meaning
<code>strong</code>	Strong importance (usually bold)
<code>em</code>	Emphasis (usually italic)
<code>mark</code>	Highlighted or marked text
<code>code</code>	Inline code or commands

Example:

Press `Ctrl + S` to save.

/ HTML

```
1 <p>Press <code>Ctrl + S</code> to save.</p>
```

Grouping Content

- `div`: Generic block-level container
- `span`: Generic inline container

These are still useful when semantic elements don't apply, often used with classes for styling.

Semantic and properly structured content **improves clarity**, helps **machines understand your content**, and makes your pages more **accessible and responsive**.

2.3 Embedding Media: Images, Audio, Video, Iframes

Modern web pages are no longer just text—they often include **images**, **multimedia**, and **embedded content** from other platforms. HTML5 introduced standard ways to support all of these natively.

Images

The `` tag embeds images.



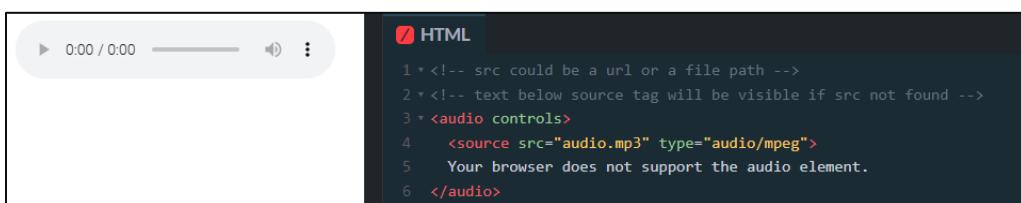
A screenshot of a code editor interface. On the left is a preview window showing a dark image of star trails. On the right is the HTML code:

```
1 <!-- src could be a url or a file path -->
2 
3
```

- `src`: Path to the image file
 - `alt`: Text alternative (important for accessibility and SEO)
 - `width` and `height`: Can be set via CSS or as attributes
-

Audio

HTML5 introduced the `<audio>` tag for playing sound files.



A screenshot of a code editor interface. On the left is a media player showing a play button, a progress bar at 0:00 / 0:00, and volume controls. On the right is the HTML code:

```
1 <!-- src could be a url or a file path -->
2 <!-- text below source tag will be visible if src not found -->
3 <audio controls>
4   <source src="audio.mp3" type="audio/mpeg">
5   Your browser does not support the audio element.
6 </audio>
```

- `controls`: Shows play/pause UI
 - Multiple `<source>` tags allow fallback for different formats
 - Common formats: mp3, ogg, wav
-

Video

The `<video>` tag supports embedded videos without plugins.



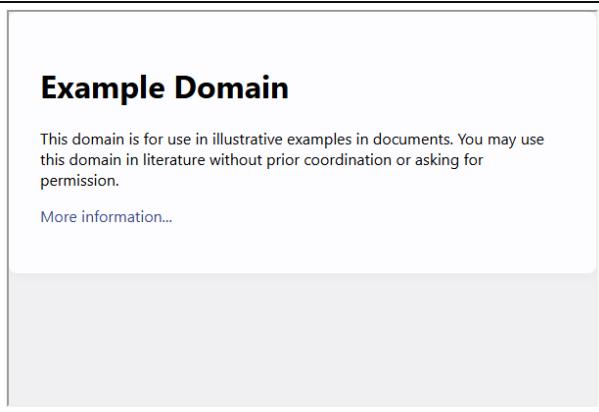
HTML

```
1 * <!-- src could be a url or a file path -->
2 * <!-- text below source tag will be visible if src not found -->
3 * <video controls width="640" height="360">
4     <source src="video.mp4" type="video/mp4">
5     Your browser does not support the video tag.
6 </video>
7
8
9
```

- `controls`: Shows playback controls
 - You can also use `autoplay`, `loop`, and `muted` attributes
 - Common formats: mp4, webm, ogg
-

Iframes

Use `<iframe>` to embed external pages, like maps, videos, or widgets.



HTML

```
1 * <!-- src could be a url or a file path -->
2 <iframe src="https://example.com" width="600" height="400"></iframe>
```

- Can be styled with CSS

- Use responsibly—`iframes` may affect performance or pose security risks
 - Often used for embedding YouTube, Google Maps, or other third-party tools
-

Accessibility Note

Always include:

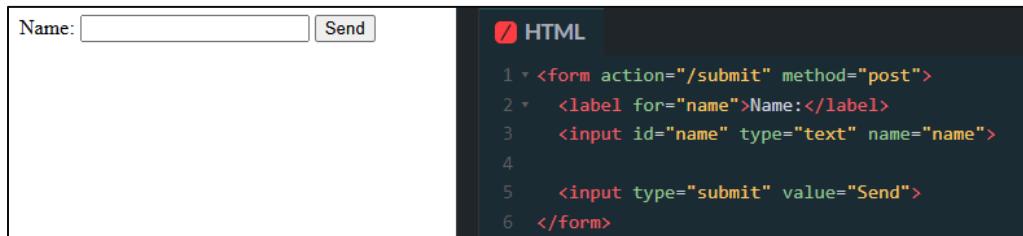
- `alt` text for images
- Text fallback inside `<audio>` or `<video>` tags
- Meaningful content around embedded media

Rich media makes your site more engaging but must be used with care to ensure speed, usability, and compatibility.

2.4 Forms and User Inputs in HTML5

Forms are essential for collecting user input—such as logins, search queries, registrations, and feedback. HTML5 introduced **new input types** and **built-in validation** to improve usability and interactivity.

Basic Form Structure



The screenshot shows a simple web form on the left and its corresponding HTML code on the right. The form has a text input field labeled "Name:" and a "Send" button. The HTML code uses a `<form>` element with `action="/submit"` and `method="post"`. It includes a `<label for="name">` and a `<input id="name" type="text" name="name">` for the text input, and a `<input type="submit" value="Send">` for the button.

```
✓ HTML
1 <form action="/submit" method="post">
2   <label for="name">Name:</label>
3   <input id="name" type="text" name="name">
4
5   <input type="submit" value="Send">
6 </form>
```

- `action`: Where the form data will be sent
 - `method`: `get` (for URL-based queries) or `post` (for secure submission)
 - `label`: Associates text with form controls (important for accessibility)
-

New HTML5 Input Types

HTML5 introduced many **semantic input types** to improve form behavior:

Input Type	Purpose
<code>text</code>	Generic single-line text
<code>email</code>	Validates email format

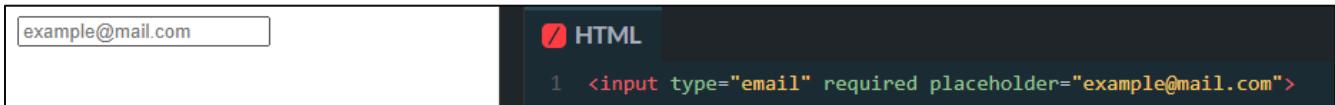
url	Validates URL format
number	Numeric input, with min/max
tel	Phone number (no validation)
date	Date picker
range	Slider input
color	Color picker
search	Optimized for search fields
checkbox	On/off toggles
radio	Multiple-choice options
file	File upload control
password	Obscured text input

Validation and Attributes

HTML5 allows validation without JavaScript:

- **required**: Field must be filled
- **min, max**: Limits for number/date inputs
- **pattern**: Regular expression check
- **placeholder**: Hint text inside fields
- **autocomplete**: Helps browsers remember user input

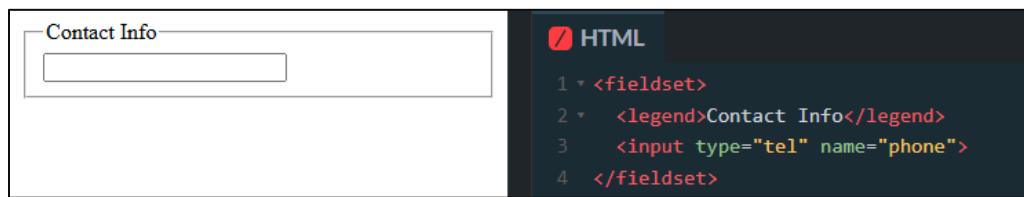
Example:



The screenshot shows a simple HTML page with a single input field. The input field has a placeholder value of "example@mail.com". To the right of the input field is a code editor window showing the corresponding HTML code: <input type="email" required placeholder="example@mail.com">. The code editor has tabs for CSS and JS, with the HTML tab selected.

Grouping Inputs

- **<fieldset>**: Groups related fields visually and semantically
- **<legend>**: Title for the fieldset group



The screenshot shows a form titled "Contact Info" containing a single input field. To the right of the form is a code editor window showing the corresponding HTML code: <fieldset><legend>Contact Info</legend><input type="tel" name="phone"></fieldset>. The code editor has tabs for CSS and JS, with the HTML tab selected.

Accessibility Tips

- Always use **<label>** with **for** attributes
- Use **aria-*** attributes when needed for screen readers

- Test form usability with keyboard navigation

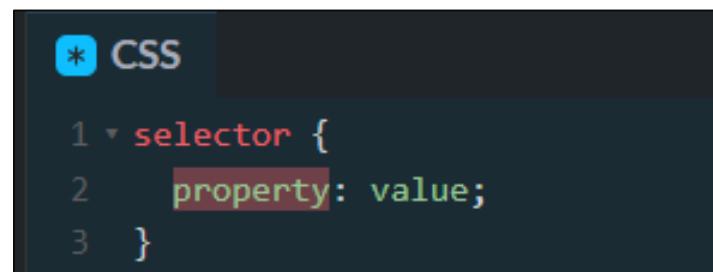
HTML5 makes forms smarter, safer, and easier to build—often **without needing JavaScript** for basic validation.

Chapter 3 – Core CSS Concepts and Layout

3.1 CSS Syntax and Selectors

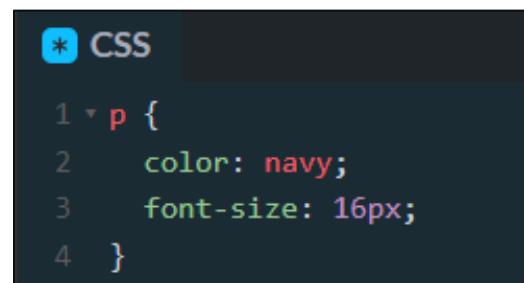
Understanding how to **target elements** with CSS is fundamental to styling a web page. The combination of **selectors**, **properties**, and **values** allows precise control over the visual presentation of content.

Basic CSS Rule Structure



```
* CSS
1 ▶ selector {
2   property: value;
3 }
```

Example:



```
* CSS
1 ▶ p {
2   color: navy;
3   font-size: 16px;
4 }
```

- **Selector:** Targets HTML elements
 - **Property:** The style you want to change (e.g. `color`)
 - **Value:** The setting for that property (e.g. `navy`)
-

Common Selectors

Selector	Targets
p	All <p> elements
.className	All elements with that class
#idName	The element with that ID
div p	All <p> inside <div>
ul > li	Direct children inside
a:hover	Links when hovered

Group Selectors

Apply styles to multiple elements at once:

* CSS

```

1 ▶ h1, h2, h3 {
2   font-family: sans-serif;
3 }
```

Universal Selector

Targets everything:

* CSS

```

1 ▶ * {
2   margin: 0;
3   padding: 0;
4 }
```

Useful for CSS resets.

Attribute Selectors

Target elements based on attribute values:

* CSS

```

1 ▶ input[type="text"] {
2   border: 1px solid #ccc;
3 }
```

Pseudo-classes

Style elements based on their state:

Selector	Use Case
<code>:hover</code>	When hovered
<code>:focus</code>	When focused (e.g., form input)
<code>:first-child</code>	First child of a parent
<code>:nth-child(2)</code>	Second child, etc.

Best Practices

- Keep selectors **simple and clear**
- Use **class selectors** for reusable styles
- Avoid using IDs for styling (too specific)
- Structure CSS from general to specific for easier overrides

3.2 Specificity and Inheritance

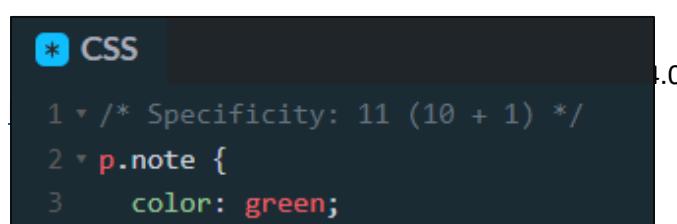
When multiple CSS rules apply to the same element, the browser must decide **which style takes precedence**. This is controlled by **specificity**, the **cascade**, and whether a property is **inherited**.

Understanding Specificity

Specificity is a scoring system based on how a rule is written:

Selector Type	Specificity Score
Inline styles	1000
ID selectors	100
Class selectors	10
Element selectors	1

Example:

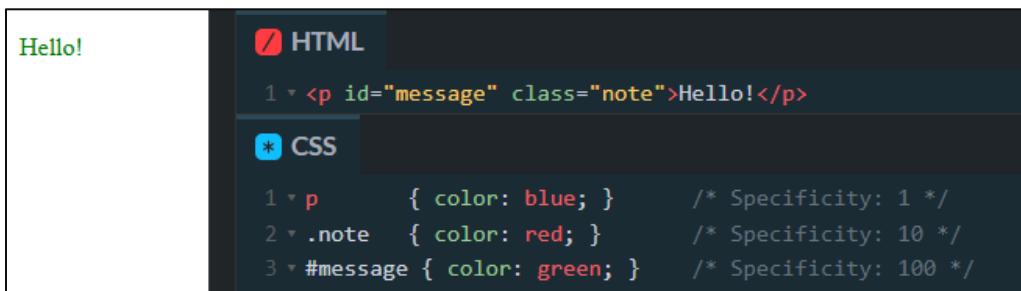


The screenshot shows a code editor with a dark theme. A tooltip or status bar at the top left says "* CSS". The code in the editor is:

```
1 /* Specificity: 11 (10 + 1) */
2 p.note {
3     color: green;
```

If multiple rules conflict, the one with the **higher specificity** wins.

How Specificity Works



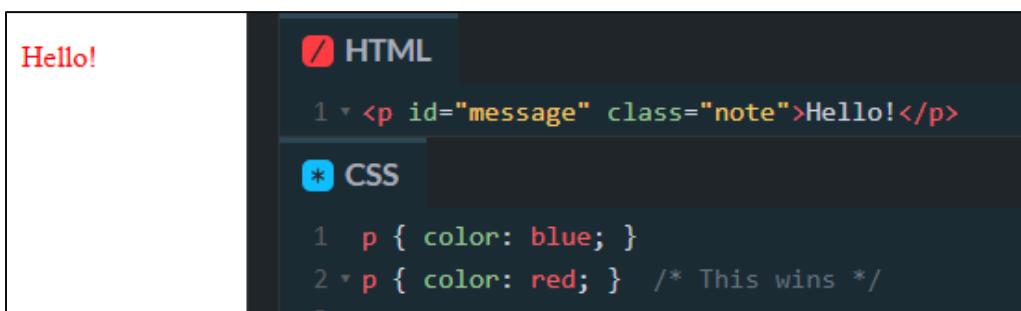
```
HTML
1 <p id="message" class="note">Hello!</p>

CSS
1 p      { color: blue; }      /* Specificity: 1 */
2 .note   { color: red; }      /* Specificity: 10 */
3 #message { color: green; }  /* Specificity: 100 */
```

The final color will be **green**, because `#message` has the highest specificity.

The Cascade

If two rules have **equal specificity**, the **last one defined** wins.

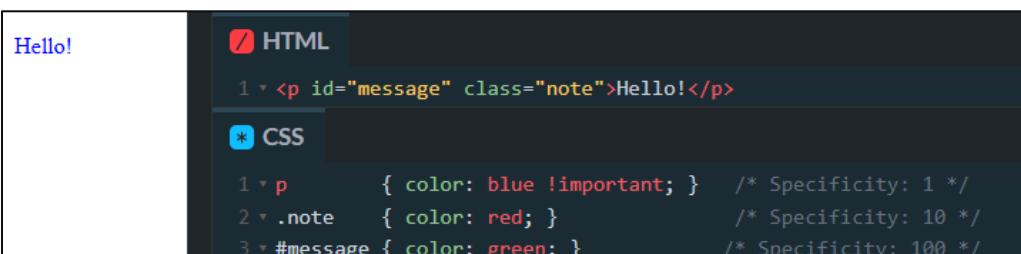


```
HTML
1 <p id="message" class="note">Hello!</p>

CSS
1 p { color: blue; }
2 p { color: red; } /* This wins */
```

The `!important` Declaration

Overrides all specificity rules:



```
HTML
1 <p id="message" class="note">Hello!</p>

CSS
1 p      { color: blue !important; } /* Specificity: 1 */
2 .note   { color: red; }           /* Specificity: 10 */
3 #message { color: green; }       /* Specificity: 100 */
```



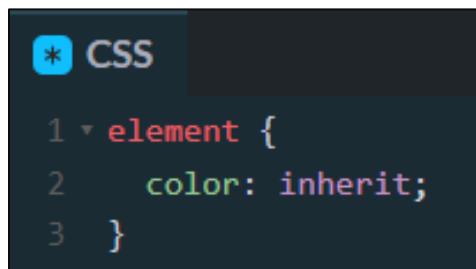
Use sparingly—it can make debugging harder and break expected behavior.

Inheritance in CSS

Some properties **naturally inherit** from parent elements:

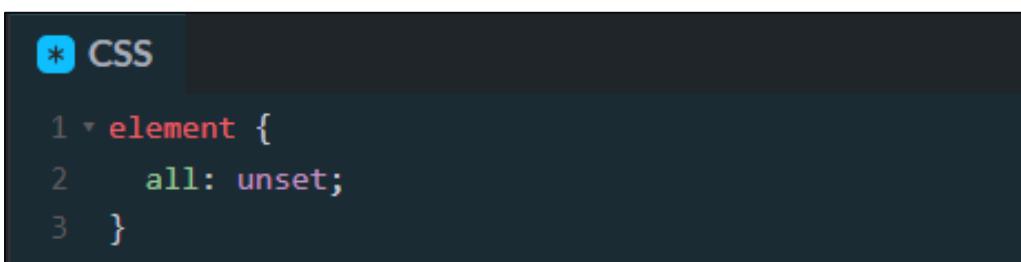
Inherited by default	Not inherited
<code>color</code>	<code>margin</code>
<code>font-family</code>	<code>padding</code>
<code>line-height</code>	<code>border</code>

To force inheritance:



```
* { element {
    color: inherit;
}
```

Or to prevent it:



```
* { element {
    all: unset;
}
```

Best Practices

- Use **classes** for styling instead of IDs
- Minimize the use of `!important`

- Organize styles from **generic → specific**
- Rely on **inheritance** where practical to reduce repetition

3.3 Typography and Color Styling

Typography plays a key role in making web content **readable**, **accessible**, and **visually consistent**. CSS gives full control over fonts, text size, spacing, alignment, and color.

Font Families

Set the typeface using the `font-family` property:

Hello World!

Paragraph Sample

HTML

```
1 <h1>Hello World!</h1>
2 <p id="message" class="note">Paragraph Sample</p>
```

CSS

```
1 body {
2   font-family: "Helvetica", "Arial", sans-serif;
3 }
```

- Always provide **fallback fonts**
- Use **generic families**: serif, sans-serif, monospace, cursive, fantasy

You can also import fonts from services like Google Fonts:

Hello World!

Paragraph Sample

HTML

```
1 <h1>Hello World!</h1>
2 <p id="message" class="note">Paragraph Sample</p>
```

CSS

```
1 @import url('https://fonts.googleapis.com/css2?family=Manufacturing+Consent&family=Roboto:ital,wght@0,100..900;1,100..900&display=swap');
2
3 body {
4   font-family: "Manufacturing Consent", system-ui;
5 }
```

Font Sizes and Units

Paragraph Sample

HTML

```
1 <p id="message" class="note">Paragraph Sample</p>
```

CSS

```
1 p {
2   font-size: 16px;
3 }
```

Common units:

Unit	Description
px	Fixed pixel size
em	Relative to parent font
rem	Relative to root (<code>html</code>) font size
%	Percentage of parent

Use `rem` for scalability across screen sizes.

Text Styles

Property	Function
font-weight	Boldness (normal, bold, 600)
font-style	Italic, normal
text-transform	Uppercase, lowercase, capitalize
text-align	Left, right, center, justify
line-height	Space between lines
letter-spacing	Space between characters

Example:



The screenshot shows a code editor interface with two tabs: HTML and CSS. The HTML tab contains the following code:

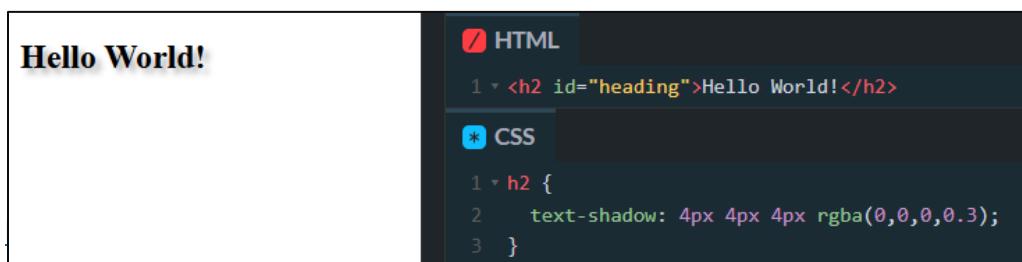
```
<h1 id="heading">Heading</h1>
```

The CSS tab contains the following code:

```
h1 {
  font-weight: bold;
  text-align: center;
  text-transform: uppercase;
}
```

Text Decoration and Shadow

- `text-decoration: underline, none, line-through`
- `text-shadow: Adds visual depth to text`



The screenshot shows a code editor interface with two tabs: HTML and CSS. The HTML tab contains the following code:

```
<h2 id="heading">Hello World!</h2>
```

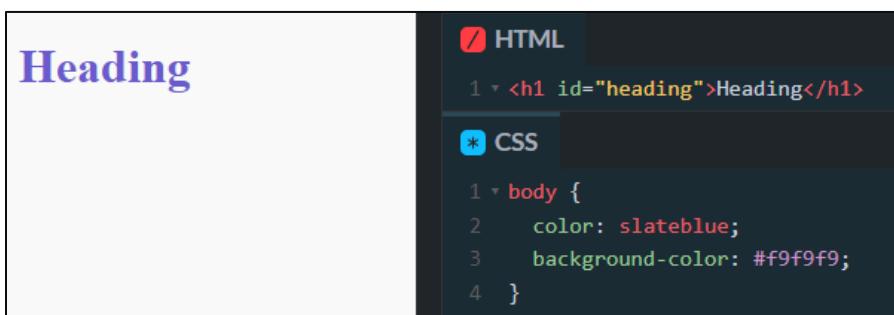
The CSS tab contains the following code:

```
h2 {
  text-shadow: 4px 4px 4px rgba(0,0,0,0.3);
}
```

Color Styling

Set color using:

- Named colors: red, navy, gray
- Hex values: #ff5733
- RGB: rgb(255, 87, 51)
- HSL: hsl(14, 100%, 60%)



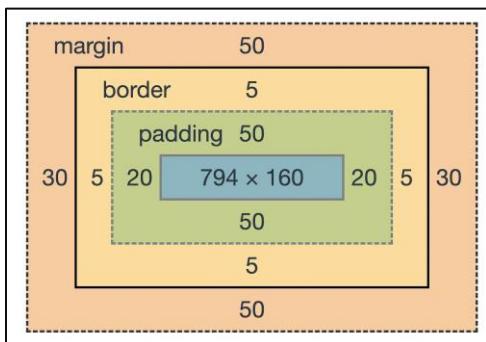
The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains the code: `<h1 id="heading">Heading</h1>`. The 'CSS' tab contains the code: `body { color: slateblue; background-color: #f9f9f9; }`. To the left of the code editor is a preview window showing a blue heading 'Heading' on a light blue background.

Use good contrast for readability and accessibility.

3.4 The Box Model Explained

Every HTML element is rendered as a **box**, and CSS allows control over its dimensions, spacing, and borders. Understanding the **box model** is fundamental to layout and spacing in web design.

Parts of the Box Model



1. **Content:** The actual content (text, image, etc.)
2. **Padding:** Space between content and border

3. **Border:** Visible line around the element
 4. **Margin:** Space between this element and others
-

Controlling Box Dimensions



HTML

```
1 <div class="box">
2   Hello from box!
3 </div>
```

CSS

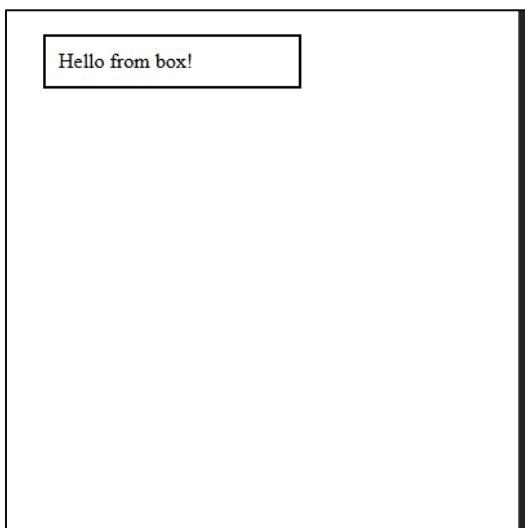
```
1 .box {
2   width: 200px;
3   padding: 10px;
4   border: 2px solid black;
5   margin: 20px;
6 }
```

- **margin:** Outside spacing
 - **border:** Outline thickness and style
-

Box-Sizing Property

By default, total width = **content + padding + border**

To make the element's width include everything, use:



HTML

```
1 <div class="box">
2   Hello from box!
3 </div>
```

CSS

```
1 /* content + padding + border should fit within width */
2 * {
3   box-sizing: border-box;
4 }
5
6 .box {
7   width: 200px;
8   padding: 10px;
9   border: 2px solid black;
10  margin: 20px;
11 }
```

This makes layouts more predictable and easier to manage.

Shorthand Properties

* CSS

```
1 * element {  
2   margin: 10px 20px; /* top/bottom 10px, left/right 20px */  
3   padding: 5px; /* all sides 5px */  
4   border: 1px solid gray; /* width, style, color */  
5 }
```

Margin Collapse

Vertical margins between block elements **may collapse** into a single margin. Horizontal margins do **not** collapse.

Example:

First Paragraph

Second Paragraph

/ HTML

```
1 <p style="margin-bottom: 20px;">First Paragraph</p>  
2 <p style="margin-top: 30px;">Second Paragraph</p>
```

Result: Combined vertical margin = 30px, not 50px.

Mastering the box model is key to precise layout control and spacing across your page.

3.5 Display Types and Positioning

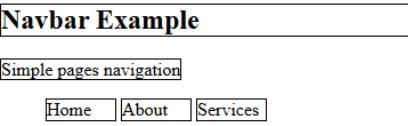
To control how elements flow and stack on a page, you must understand the **display** and **position** properties. These define how elements behave in the layout and how they interact with other content.

Display Property

The `display` property determines how an element is rendered in the document flow.

Value	Description
<code>block</code>	Starts on a new line, takes full width
<code>inline</code>	Flows with text, cannot have width/height
<code>inline-block</code>	Like <code>inline</code> , but supports width and height
<code>none</code>	Removes element from layout

Examples:



HTML
CSS

```

HTML
1 <nav>
2   <h1>Navbar Example</h1>
3   <span>Simple pages navigation</span>
4   <ul>
5     <li>Home</li>
6     <li>About</li>
7     <li>Services</li>
8   </ul>
9 </nav>

CSS
1 h1, span, nav li { border: 1px solid black } /* just for display */
2
3 h1 { display: block; } /* full width by default */
4 span { display: inline; width: 100px } /* width doesn't applied */
5 nav li { display: inline-block; width: 60px } /* width applied */

```

Visibility vs. Display

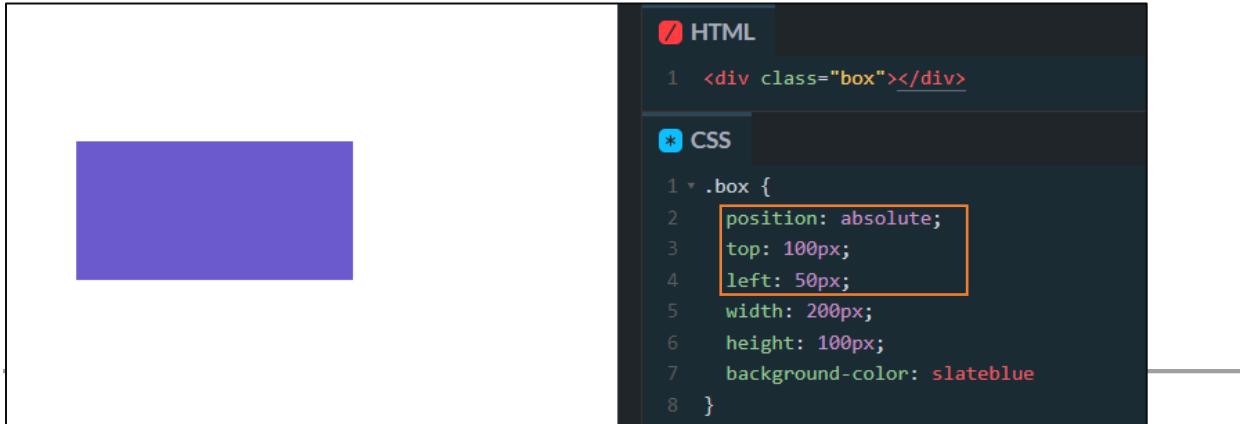
- `display: none;` removes the element from the flow
 - `visibility: hidden;` hides it visually but leaves the space it occupied
-

Position Property

Controls how an element is **placed** on the page.

Value	Behavior
static	Default (in normal flow)
relative	Offset from its normal position
absolute	Positioned relative to nearest non-static ancestor
fixed	Positioned relative to the viewport
sticky	Scrolls until a point, then sticks in place

Example:



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains a single line of code:

```
<div class="box"></div>
```

. The 'CSS' tab contains the following styles:

```

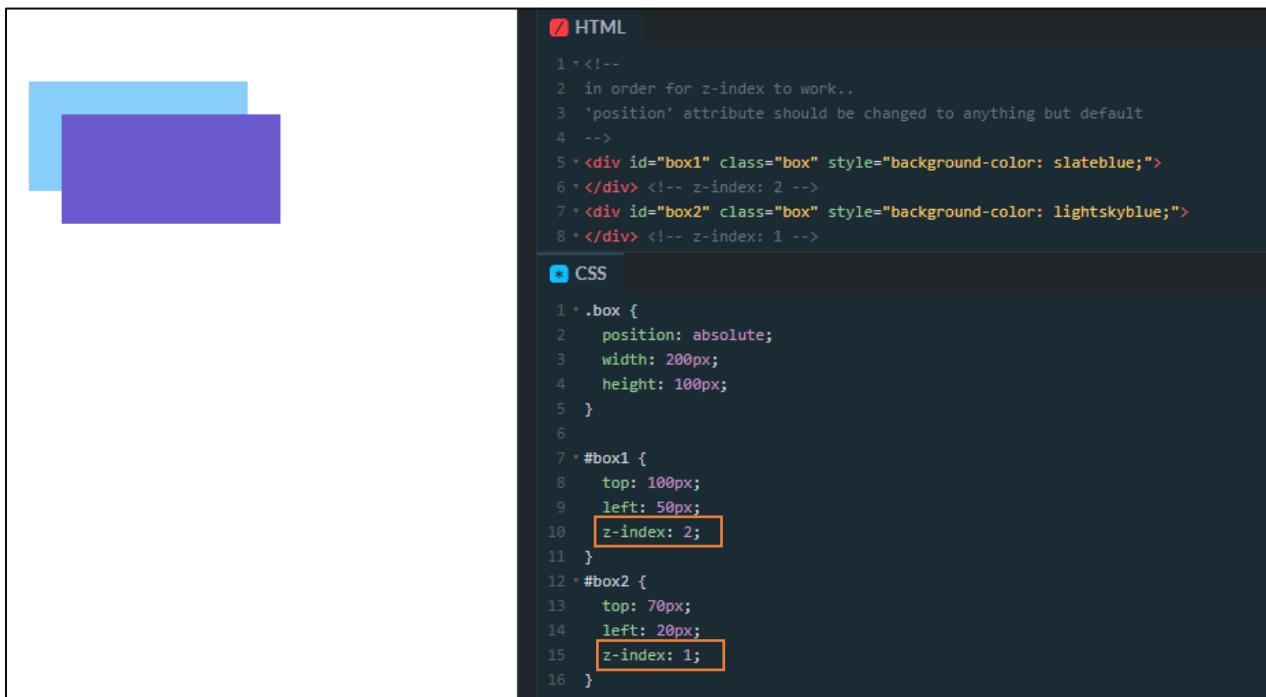
1 .box {
2   position: absolute;
3   top: 100px;
4   left: 50px;
5   width: 200px;
6   height: 100px;
7   background-color: slateblue
8 }

```

A purple rectangular box is displayed on the page at the specified position.

Z-Index

Used with positioned elements to control **stacking order**.



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains the following code:

```

1 <!--
2 in order for z-index to work..
3 "position" attribute should be changed to anything but default
4 -->
5 <div id="box1" class="box" style="background-color: slateblue;">
6 </div> <!-- z-index: 2 -->
7 <div id="box2" class="box" style="background-color: lightskyblue;">
8 </div> <!-- z-index: 1 -->

```

The 'CSS' tab contains the following styles:

```

1 .box {
2   position: absolute;
3   width: 200px;
4   height: 100px;
5 }
6
7 #box1 {
8   top: 100px;
9   left: 50px;
10  [z-index: 2];
11 }
12 #box2 {
13   top: 70px;
14   left: 20px;
15  [z-index: 1];
16 }

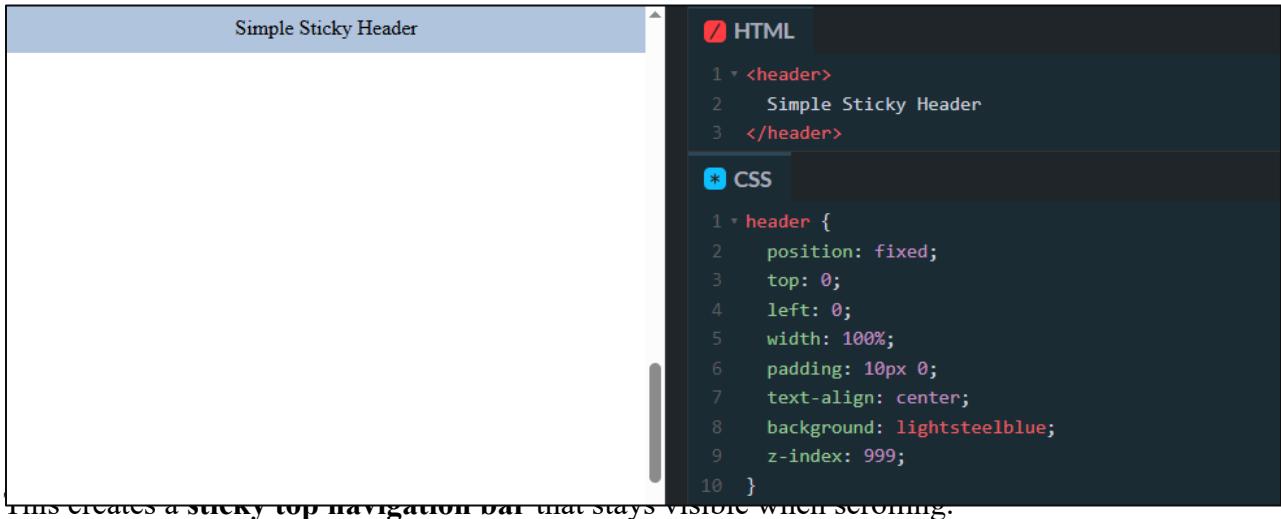
```

The page displays two overlapping boxes: a blue box on top and a purple box below it, demonstrating the stacking order based on their z-index values.

Higher `z-index` means it appears **on top** of lower values.

Combining Position and Display

Example layout snippet:



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains the following code:

```
1 <header>
2   Simple Sticky Header
3 </header>
```

The 'CSS' tab contains the following code:

```
1 header {
2   position: fixed;
3   top: 0;
4   left: 0;
5   width: 100%;
6   padding: 10px 0;
7   text-align: center;
8   background: lightsteelblue;
9   z-index: 999;
10 }
```

A note at the bottom of the code editor says: "This creates a **sticky top navigation bar** that stays visible when scrolling."

Understanding `display` and `position` allows you to:

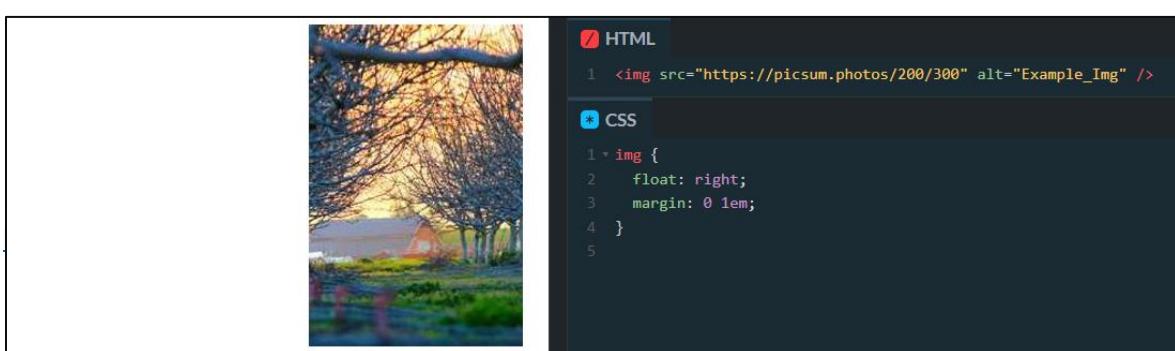
- Build flexible layouts
- Overlay content (modals, dropdowns)
- Handle responsiveness effectively

3.6 Float and Clear

Before modern layout systems like Flexbox and Grid, web developers relied heavily on the `float` property to build multi-column layouts. Though no longer the go-to technique, `float` is still useful in some cases, especially for **wrapping content around images**.

Using Float

The `float` property removes an element from normal flow and positions it to the **left or right** of its container.



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. On the left, there is a preview window showing a landscape image with trees and a house. The 'HTML' tab contains the following code:

```
1 
```

The 'CSS' tab contains the following code:

```
1 * img {
2   float: right;
3   margin: 0 1em;
4 }
```



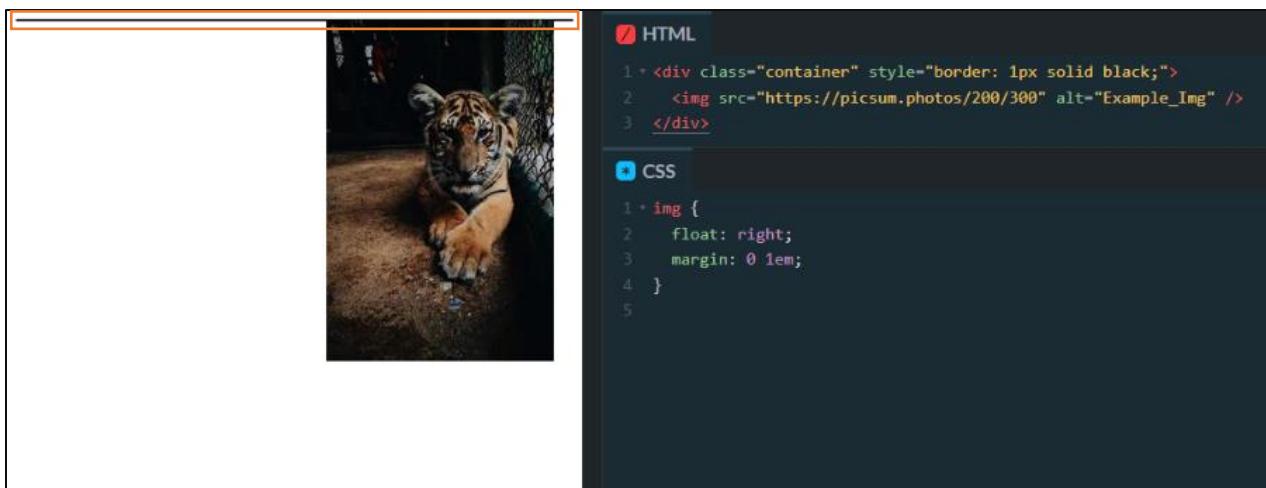
In this example, the image will float to the right, and surrounding text will wrap around it.

Common Float Use Cases

- Wrapping text around an image
 - Building basic multi-column layouts (legacy)
 - Aligning content side-by-side in older designs
-

The Float Problem

When an element is floated, its parent container may **collapse in height** because floated children are removed from the flow.



The screenshot shows a code editor with two tabs: 'HTML' and 'CSS'. The 'HTML' tab contains the following code:

```
<div class="container" style="border: 1px solid black;">
  
</div>
```

The 'CSS' tab contains the following code:

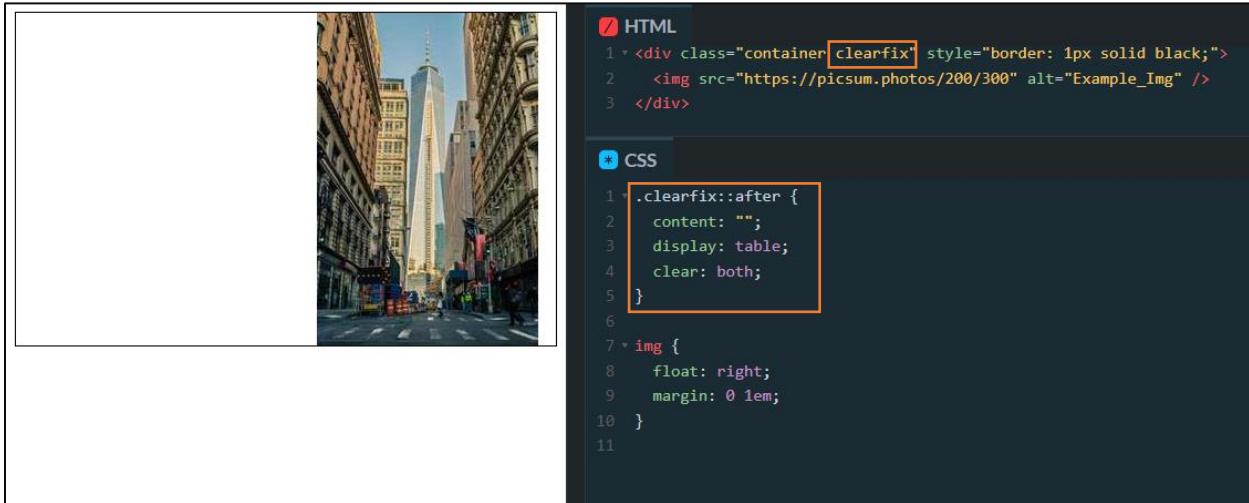
```
.container {
  float: right;
  margin: 0 1em;
}
```

To the left of the code editor is a visual representation of the resulting layout. It shows a large orange-bordered box labeled 'HTML' containing a tiger image. The tiger is positioned to the right of the box, and the text 'The container may appear to have no height unless cleared.' is wrapped around the image. The entire setup is contained within a larger white box with an orange border.

The `.container` may appear to have no height unless cleared.

Clearing Floats

To fix layout collapse, apply a **clearfix** to the container:



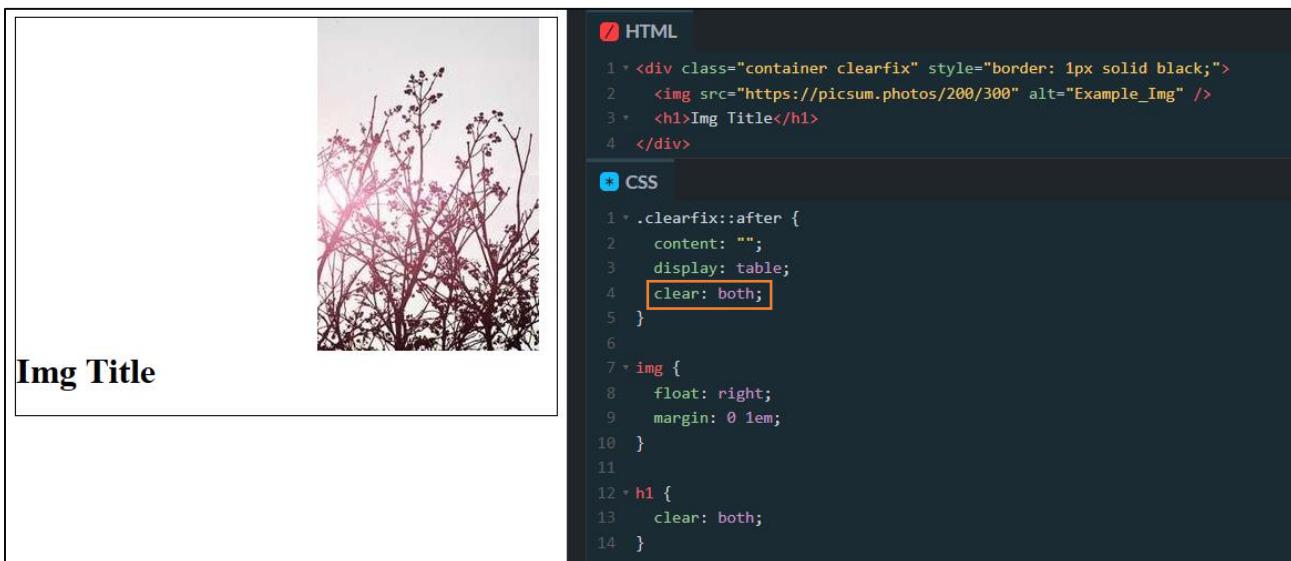
The screenshot shows a code editor with two panes. The left pane displays a photograph of a city street with tall buildings, and the right pane shows the corresponding HTML and CSS code.

```
HTML
1 <div class="container clearfix" style="border: 1px solid black;">
2   
3 </div>

CSS
1 .clearfix::after {
2   content: "";
3   display: table;
4   clear: both;
5 }
6
7 * img {
8   float: right;
9   margin: 0 1em;
10 }
11
```

The **clear** Property

Stops an element from appearing next to floated elements.



The screenshot shows a code editor with two panes. The left pane displays a photograph of a plant, and the right pane shows the corresponding HTML and CSS code.

```
HTML
1 <div class="container clearfix" style="border: 1px solid black;">
2   
3   <h1>Img Title</h1>
4 </div>

CSS
1 * .clearfix::after {
2   content: "";
3   display: table;
4   clear: both;
5 }
6
7 * img {
8   float: right;
9   margin: 0 1em;
10 }
11
12 * h1 {
13   clear: both;
14 }
```

When to Use Float Today

- **Good for:** Simple image wrapping, side labels
- **Avoid for:** General page layout (use Flexbox or Grid instead)

Floats helped build the early web, but they've largely been replaced by more powerful layout systems.

Chapter 4 – Responsive Design and Modern CSS Techniques

4.1 Mobile-First and Media Queries

Modern websites must adapt to many screen sizes—from phones to desktops. **Responsive Design** ensures your layout adjusts gracefully based on the device. This is achieved using **media queries** and a **mobile-first** approach.

What is Mobile-First?

A **mobile-first** design strategy means:

- You design for **small screens first** (mobile)
- Then use media queries to enhance the layout for **larger screens**

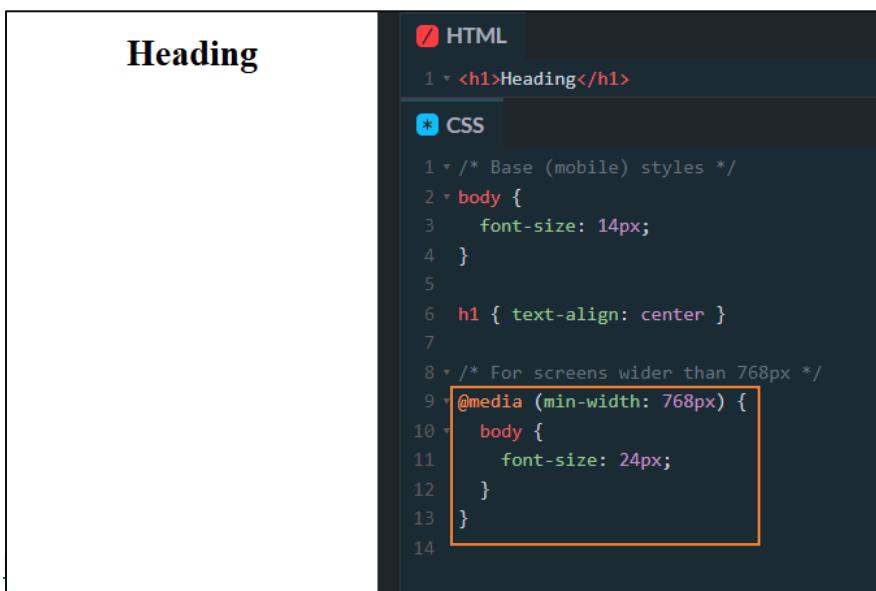
Benefits:

- Faster load times on mobile
 - Better accessibility
 - Progressive enhancement
-

What are Media Queries?

Media queries apply CSS rules conditionally, based on the **screen's size, orientation, resolution**, or other properties.

Example:



The screenshot shows a code editor interface with two tabs: "HTML" and "CSS". The "HTML" tab contains the following code:

```
<h1>Heading</h1>
```

The "CSS" tab contains the following CSS code:

```
/* Base (mobile) styles */
body {
    font-size: 14px;
}

h1 { text-align: center }

/* For screens wider than 768px */
@media (min-width: 768px) {
    body {
        font-size: 24px;
    }
}
```

The media query block is highlighted with an orange rectangle.

Heading

HTML
1 <h1>Heading</h1>

CSS
1 /* Base (mobile) styles */
2 body {
3 font-size: 14px;
4 }
5
6 h1 { text-align: center }
7
8 /* For screens wider than 768px */
9 @media (min-width: 768px) {
10 body {
11 font-size: 24px;
12 }
13 }
14

Media Query Syntax

* CSS

```
1 @media (media-type) and (condition) {
2 /* CSS rules */
3 }
```

Common conditions:

Condition	Use Case
<code>min-width</code>	Style if screen is wider
<code>max-width</code>	Style if screen is narrower
<code>orientation: portrait</code>	Phones in vertical mode
<code>resolution: 300dpi</code>	High-DPI displays

Breakpoints

Breakpoints are screen widths where layout changes. Common examples:

* CSS

```
1 /* Tablet and up */
2 @media (min-width: 768px) { ... }
3
4 /* Desktop and up */
5 @media (min-width: 1024px) { ... }
```

Use breakpoints based on **content needs**, not just device types.

Best Practices

- Start with a **flexible base** layout
 - Avoid hard-coded widths (use %, vw, flex)
 - Use media queries to **enhance**, not duplicate
 - Keep CSS organized (e.g. mobile styles first, then queries)
-

Responsive design isn't optional anymore—it's expected. Media queries give you control over how your site behaves across all devices.

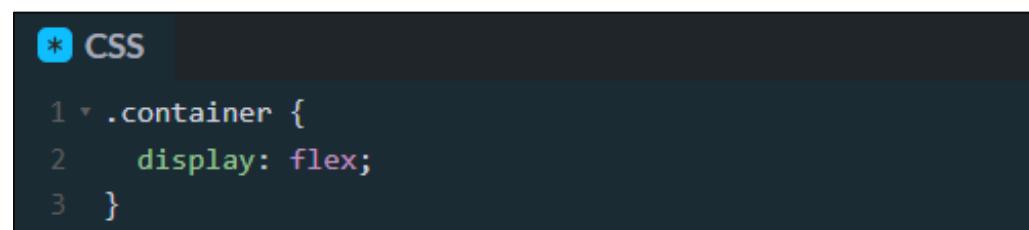
4.2 Flexbox: Concepts and Practical Use

Flexbox (Flexible Box Layout) is a modern CSS layout system designed to arrange items **in one dimension**—either as rows or columns—with automatic resizing, spacing, and alignment.

Why Use Flexbox?

- Align items horizontally or vertically
 - Distribute space between elements
 - Easily create fluid, responsive layouts
 - Replace floats, inline-block, and complex margin hacks
-

Basic Flex Container Setup



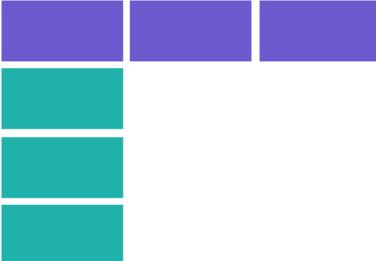
```
* CSS
1 * .container {
2   display: flex;
3 }
```

All direct children of `.container` become **flex items**.

Main Concepts

Term	Definition
Main axis	Direction of content flow (<code>row</code> by default)
Cross axis	Perpendicular to main axis (usually vertical)
Flex container	Parent element with <code>display: flex</code>
Flex item	Direct child of the flex container

Flex Direction



HTML

```
1 <div class="container dir-row">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>
6
7 <div class="container dir-col">
8   <div class="box"></div>
9   <div class="box"></div>
10  <div class="box"></div>
11 </div>
```

CSS

```
1 .container { display: flex; }
2 /* Horizontal (Default) */
3 .dir-row { flex-direction: row; }
4 /* Vertical */
5 .dir-col { flex-direction: column; }
6 /* box style */
7 .box { width: 100px; height: 50px; margin: 3px; }
8 .dir-row .box { background-color: slateblue; }
9 .dir-col .box { background-color: lightseagreen; }
```

Aligning Items



HTML

```
1 <div class="container dir-row">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>
```

CSS

```
1 .container { display: flex; }
2 /* Horizontal (Default) */
3 .dir-row {
4   flex-direction: row;
5   justify-content: space-between; /* Main axis (horizontal) */
6   align-items: center;           /* Cross axis (vertical) */
7 }
8
9 .box { width: 100px; height: 50px; margin: 3px; }
10 .dir-row .box { background-color: slateblue; }
```



Common values:

- justify-content: flex-start, center, space-between, space-around, space-evenly
- align-items: stretch, center, flex-start, flex-end

Wrapping Items

<div

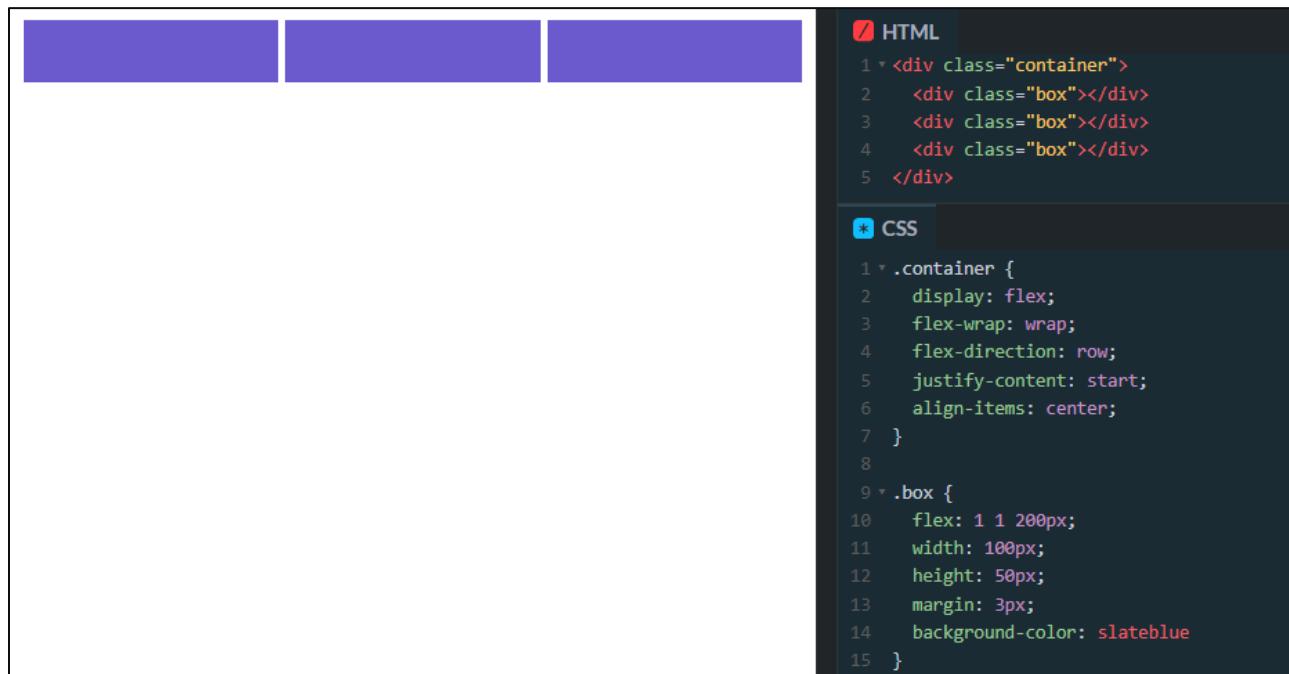
Controlling Flex Items

✖ CSS

```
1 ▶ .box {  
2   flex-grow: 1;  
3   flex-shrink: 1;  
4   flex-basis: 200px;  
5 }
```

- `flex-grow`: how much it can expand
 - `flex-shrink`: how much it can shrink
 - `flex-basis`: initial size before space distribution
 - `flex`: shorthand for above three
-

Real-World Example

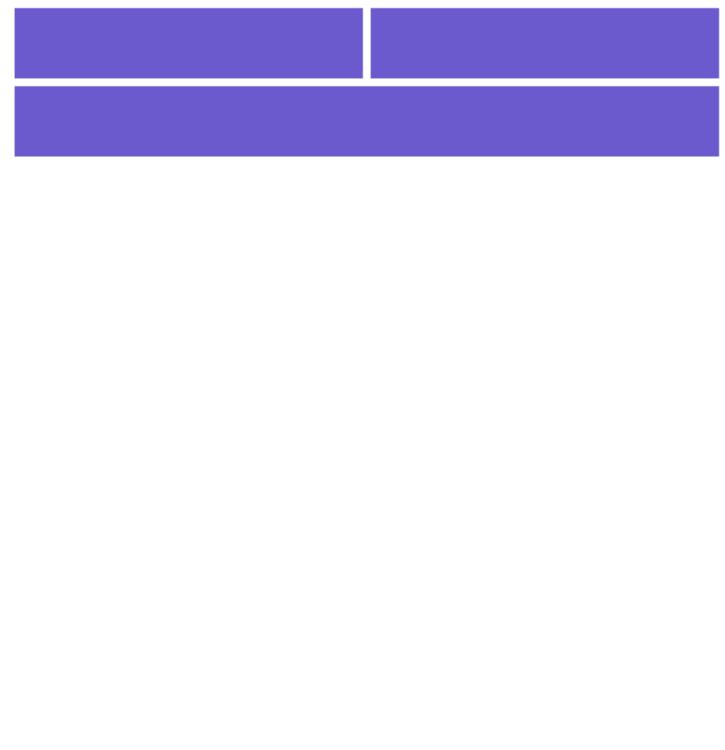


✖ HTML

```
1 <div class="container">  
2   <div class="box"></div>  
3   <div class="box"></div>  
4   <div class="box"></div>  
5 </div>
```

✖ CSS

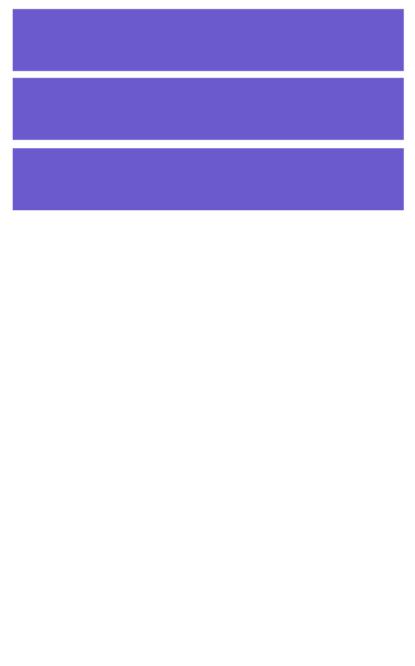
```
1 ▶ .container {  
2   display: flex;  
3   flex-wrap: wrap;  
4   flex-direction: row;  
5   justify-content: start;  
6   align-items: center;  
7 }  
8  
9 ▶ .box {  
10  flex: 1 1 200px;  
11  width: 100px;  
12  height: 50px;  
13  margin: 3px;  
14  background-color: slateblue  
15 }
```



The screenshot shows a web page with a white background. On the left side, there is a vertical sidebar with a light gray background. In the main content area, there are three rectangular boxes arranged horizontally. Each box has a solid slateblue background color. The boxes are evenly spaced and aligned horizontally.

```
✓ HTML
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>

✗ CSS
1 .container {
2   display: flex;
3   flex-wrap: wrap;
4   flex-direction: row;
5   justify-content: start;
6   align-items: center;
7 }
8
9 .box {
10  flex: 1 1 200px;
11  width: 100px;
12  height: 50px;
13  margin: 3px;
14  background-color: slateblue
15 }
```



The screenshot shows a web page with a white background. On the left side, there is a vertical sidebar with a light gray background. In the main content area, there are three rectangular boxes arranged vertically. Each box has a solid slateblue background color. The boxes are evenly spaced and aligned vertically.

```
✓ HTML
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>

✗ CSS
1 .container {
2   display: flex;
3   flex-wrap: wrap;
4   flex-direction: row;
5   justify-content: start;
6   align-items: center;
7 }
8
9 .box {
10  flex: 1 1 200px;
11  width: 100px;
12  height: 50px;
13  margin: 3px;
14  background-color: slateblue
15 }
```

Flexbox is ideal for **menus, toolbars, cards, and centering elements**—especially when building responsive UIs.

4.3 Grid Layout: 2D Layout Made Easy

CSS Grid is a powerful layout system designed for creating **two-dimensional layouts**—arranging elements in rows and columns simultaneously. It complements Flexbox (which is 1D) by handling complex page structures more efficiently.

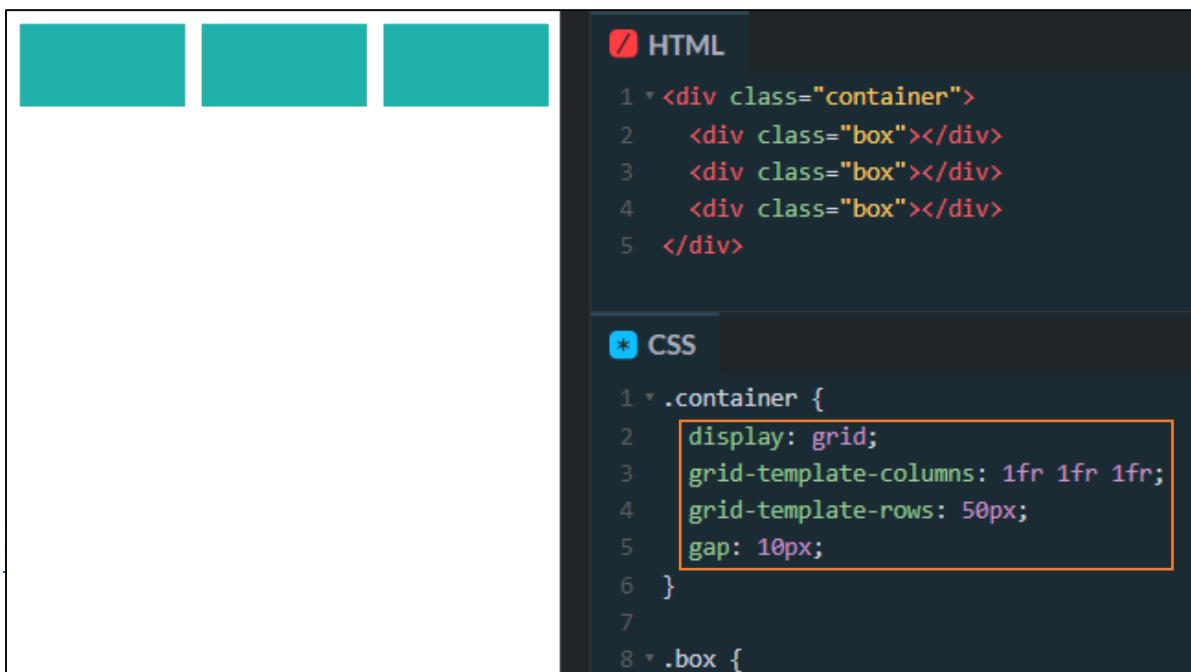
Why Use Grid?

- Create **complex page layouts** with minimal code
 - Control both **rows and columns**
 - Place items anywhere within the layout
 - Build responsive templates and UI sections
-

This creates a grid with **3 equal-width columns**.

Creating a Grid Container

Use `display: grid;` to create a grid container, the use `grid-template-columns` and `grid-template-rows` to define the number and size of grid tracks:



The screenshot shows a code editor with two tabs: "HTML" and "CSS".

HTML:

```
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>
```

CSS:

```
1 .container {
2   display: grid;
3   grid-template-columns: 1fr 1fr 1fr;
4   grid-template-rows: 50px;
5   gap: 10px;
6 }
7
8 .box {
```

The CSS code includes a box selection around the `grid-template-columns`, `grid-template-rows`, and `gap` properties.

Property	Description	Example
grid-template-columns	Defines the columns in the grid (left to right)	1fr 2fr 1fr = 3 columns: 25%, 50%, 25%
grid-template-rows	Defines the row heights (top to bottom)	auto 100px = 1st row auto height, 2nd is fixed

- `fr` = fraction of available space
 - `auto` = adjusts to content
 - `px, %` = fixed units
-

Grid Gaps

When using `gap`, Add space between rows and columns:

Property	Description
gap	Shorthand for row & column gaps
row-gap	Vertical spacing between rows
column-gap	Horizontal spacing between columns

Placing Items

Each direct child of a grid container becomes a **grid item**. You can place them in specific rows/columns:



HTML

```
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5 </div>
```

CSS

```
1 .container {
2   display: grid;
3   grid-template-columns: 1fr 1fr 1fr;
4   gap: 20px;
5 }
6
7 .box {
8   width: 100px;
9   height: 50px;
10}
```

Property	Description
grid-column	Starts at column 2, ends before column 4 (spans 2)
grid-row	Starts at row 1, ends before row 2 (spans 1)

Format: start / end

You can also **name lines** or use **span**:

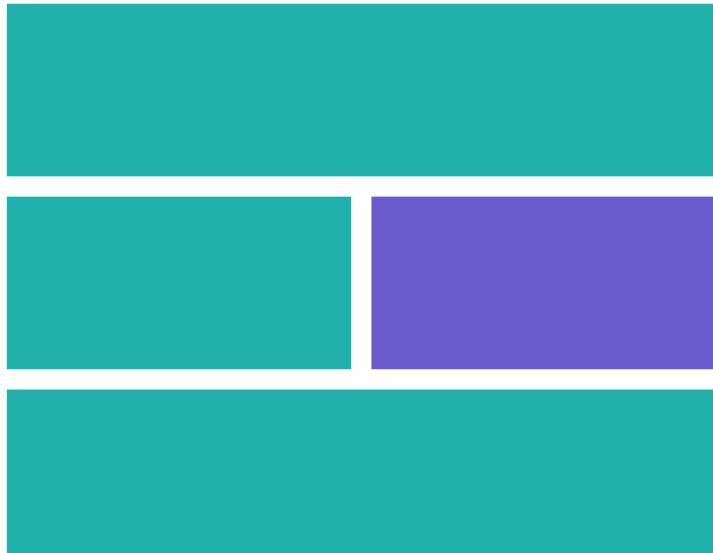
css

CopyEdit

```
grid-column: 1 / span 2; /* span across 2 columns starting from 1 */
```

Grid Areas (Named Layouts)

Define named areas to make layout more readable, then assign areas to items:



```

# HTML
1 <div class="container">
2   <header class="box"></header>
3   <sidebar class="box"></sidebar>
4   <main class="box"></main>
5   <footer class="box"></footer>
6 </div>

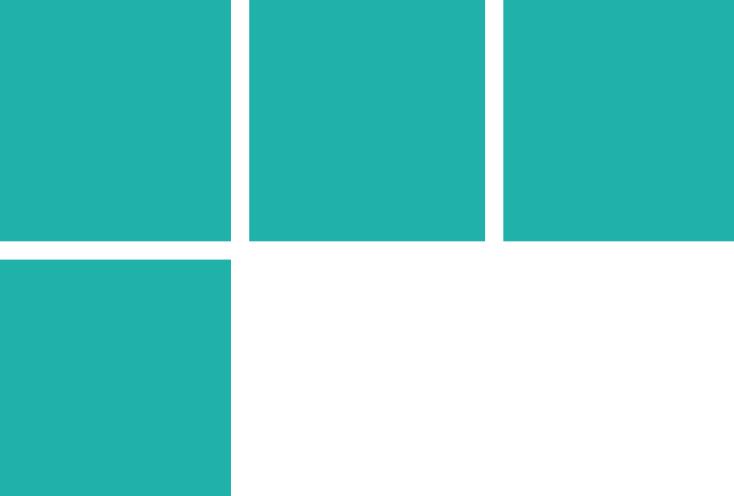
+ CSS
1 .container {
2   width: 100%;
3   height: 550px;
4   display: grid;
5   grid-template-areas:
6     "header header"
7     "sidebar content"
8     "footer footer";
9   gap: 20px;
10 }
11
12 header { grid-area: header; }
13 sidebar { grid-area: sidebar; }
14 main { grid-area: content; }
15 footer { grid-area: footer; }
```

Term	Meaning
grid-area	Refers to named layout zone

template-areas Declares overall grid map

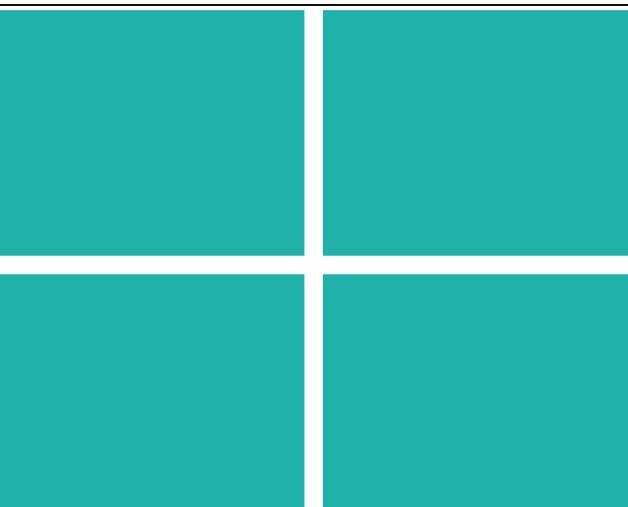
This is **very useful** for responsive page templates.

6. Responsive Grid with Auto-Fit



```
✓ HTML
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5   <div class="box"></div>
6 </div>
```

```
✖ CSS
1 *.container {
2   width: 100%;
3   height: 550px;
4   display: grid;
5   grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
6   gap: 20px;
7 }
8
9 *.box {
10  background-color: lightseagreen
11 }
```



```
✓ HTML
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5   <div class="box"></div>
6 </div>
```

```
✖ CSS
1 *.container {
2   width: 100%;
3   height: 550px;
4   display: grid;
5   grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
6   gap: 20px;
7 }
8
9 *.box {
10  background-color: lightseagreen
11 }
```



```
✓ HTML
1 <div class="container">
2   <div class="box"></div>
3   <div class="box"></div>
4   <div class="box"></div>
5   <div class="box"></div>
6 </div>
```

```
✖ CSS
1 *.container {
2   width: 100%;
3   height: 550px;
4   display: grid;
5   grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
6   gap: 20px;
7 }
8
9 *.box {
10  background-color: lightseagreen
11 }
```



Function	Description
<code>repeat(n, val)</code>	Repeats a column n times
<code>auto-fit</code>	Fills available space by adjusting column count automatically
<code>minmax(a, b)</code>	Each column must be at least a, and can grow up to b

This layout will:

- Create as many 250px-wide columns as can fit
 - Make them responsive by stretching (`1fr`)
-

Summary

Feature	Best For
<code>grid-template-columns/rows</code>	Defining fixed or fluid layout tracks
<code>grid-column/row</code>	Precise placement of items
<code>grid-template-areas</code>	Semantic and readable layouts
<code>auto-fit/minmax</code>	Fully responsive grid systems

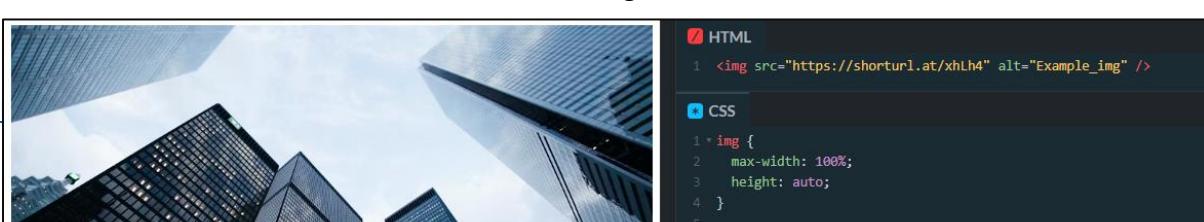
4.4 Responsive Images and Viewport Units

To build fully responsive websites, media like **images** and units like `vw` and `vh` must scale with the screen. This section covers how to handle responsive imagery and use **viewport-based sizing**.

Responsive Images

Basic Technique

Use `max-width` and `height: auto` to make images scale down but not stretch:



The image shows a code editor interface with two tabs: "HTML" and "CSS".

HTML:

```

```

CSS:

```
img {
  max-width: 100%;
  height: auto;
}
```

Property	Purpose
<code>max-width</code>	Prevents image from exceeding container
<code>height: auto</code>	Maintains aspect ratio

This is the **most common method** for responsive images.

<picture> Element

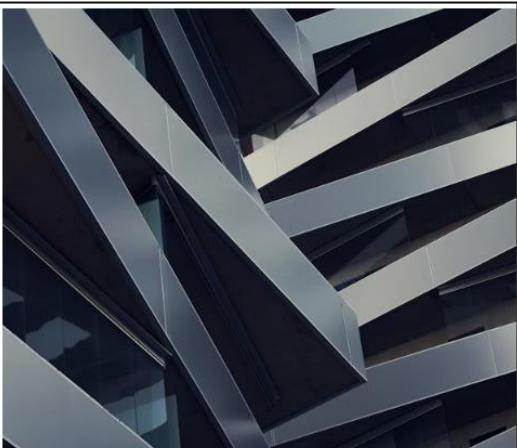
Serves **different images** based on screen size or format:



```

# HTML
1 * <!-- screen size is '500px' ( < '600px' ), so first source is applied -->
2 * <picture>
3   <source media="(max-width: 600px)" srcset="https://shorturl.at/IcJKp">
4   <source media="(min-width: 601px)" srcset="https://shorturl.at/3OY8v">
5   
6 </picture>
7

```



```

# HTML
1 * <!-- screen size is '601px', so second source ('min-width: 601px') is applied -->
2 * <picture>
3   <source media="(max-width: 600px)" srcset="https://shorturl.at/IcJKp">
4   <source media="(min-width: 601px)" srcset="https://shorturl.at/3OY8v">
5   
6 </picture>
7

```

- The browser picks the best matching image
 - Use for high-performance, responsive image loading
-

srcset and sizes

Use these attributes to provide **multiple resolutions** of the same image:

HTML

```
1 
```

- **srcset**: list of image files with width descriptors
 - **sizes**: tells the browser how wide the image will display
 - Best for **retina screens** and varying screen widths
-

Viewport Units

What Are Viewport Units?

Unit	Meaning
vw	1% of the viewport width
vh	1% of the viewport height
vmin	1% of the smaller dimension
vmax	1% of the larger dimension

Example:

CSS

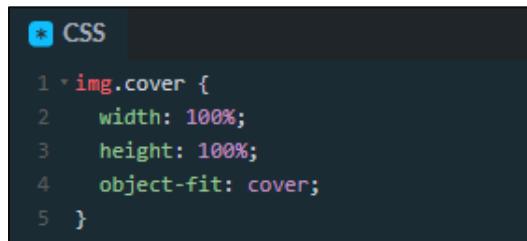
```
1 *#hero {  
2   height: 100vh; /* full screen height */  
3   font-size: 4vw; /* scales with screen width */  
4 }
```

When to Use Viewport Units

- Full-screen sections (100vh, 100vw)
 - Large responsive headings
 - Mobile-first sections that fill the screen
 - Custom breakpoints (instead of px)
-

Tips

- Combine `vw/vh` with **media queries** for best results
- Always test **text in vw**—can be too small on mobile
- Don't forget `object-fit: cover` for background-like images:



```
 1 * img.cover {  
 2   width: 100%;  
 3   height: 100%;  
 4   object-fit: cover;  
 5 }
```

Responsive images save bandwidth and improve loading speed, while viewport units help build scalable, flexible designs.