Introduction to CSS

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

Why CSS?

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

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

How CSS Works with HTML

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

1. Inline CSS

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

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

2. Internal CSS

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

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

3. External CSS (Best Practice)

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

`style.css`

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

`index.html`

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

The “Cascading” in CSS

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

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

Example:

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

Anatomy of a CSS Rule

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

Example:

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

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

What You’ll Learn in CSS

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

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

CSS Syntax and Selectors

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

CSS Syntax

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

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

Example:

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

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

Types of Selectors

1. Element Selector

Selects all elements of a specific type.

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

This targets all `<p>` elements.

---

2. Class Selector

Selects elements with a specific class.

HTML:

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

CSS:

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

Use a period `.` before the class name.

---

3. ID Selector

Selects a single element with a unique ID.

HTML:

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

CSS:

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

Use a hash `#` before the ID name.

---

4. Universal Selector

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

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

This is commonly used for resetting default styles.

---

5. Grouping Selectors

Apply the same styles to multiple selectors at once.

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

This avoids repetition.

---

6. Descendant Selector

Targets elements nested inside other elements.

HTML:

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

CSS:

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

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

---

7. Combining Class and Element Selectors

You can be more specific by combining them.

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

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

Summary

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

Colors in CSS

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

Ways to Define Colors

CSS supports several formats for defining colors:

---

1. Named Colors

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

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

---

2. HEX Codes

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

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

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

- `#ff0000` → red

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

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

---

3. RGB (Red, Green, Blue)

You can define a color using the RGB color model.

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

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

---

4. RGBA (RGB + Alpha)

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

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

This creates a semi-transparent black background.

---

5. HSL (Hue, Saturation, Lightness)

Another way to define colors using:

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

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

---

6. HSLA (HSL + Alpha)

Same as HSL, but with transparency.

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

Applying Colors in CSS

You can use color properties in many different places:

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

Transparent and CurrentColor

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

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

Summary

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

The CSS Box Model

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

What is the Box Model?

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

```
|---------------------------|
|         Margin            |
|  |---------------------|  |
|  |      Border         |  |
|  |  |---------------|  |  |
|  |  |   Padding      |  |  |
|  |  |  |---------|   |  |  |
|  |  |  | Content |   |  |  |
|  |  |  |---------|   |  |  |
|  |  |---------------|  |  |
|  |---------------------|  |
|---------------------------|
```

The Four Parts

1. Content

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

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

---

2. Padding

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

```
padding: 20px;
```

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

---

3. Border

The border around the padding and content.

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

You can control its width, style, and color.

---

4. Margin

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

```
margin: 30px;
```

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

Example

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

The **actual space** this element occupies:

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

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

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

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

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

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

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

Visual Example

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

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

Summary

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

Units in CSS

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

Two Categories of Units

1. Absolute Units

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

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

Example:

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

2. Relative Units

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

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

Commonly Used Units

`px` (Pixels)

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

Fixed spacing that does not scale with screen size.

`%` (Percentage)

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

Useful for making widths or heights relative to parent elements.

`em` vs `rem`

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

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

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

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

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

Use `rem` for consistency in modern responsive design.

`vw` and `vh`

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

These units are powerful for creating fullscreen layouts.

`calc()` Function

Combine different units using `calc()`:

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

Best Practices

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

Summary

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

Typography in CSS

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

Basic Text Properties

1. `font-family`

Sets the typeface for your text.

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

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

Common font stacks:

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

---

2. `font-size`

Controls the size of the text.

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

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

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

---

3. `font-weight`

Defines the boldness of text.

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

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

---

4. `font-style`

Sets text to normal, italic, or oblique.

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

---

5. `text-align`

Aligns text horizontally.

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

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

---

6. `line-height`

Controls the space between lines of text.

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

This improves readability, especially for paragraphs.

---

7. `letter-spacing`

Controls space between characters.

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

---

8. `word-spacing`

Controls space between words.

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

---

9. `text-transform`

Changes the case of text.

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

.lower {
  text-transform: lowercase;
}

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

---

10. `text-decoration`

Controls underlining, overlining, and line-through.

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

Using Google Fonts

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

HTML

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

CSS

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

Summary

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

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

Backgrounds and Borders in CSS

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

Background Properties

1. `background-color`

Sets a solid color behind an element.

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

---

2. `background-image`

Adds an image as the background.

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

You can use local images or remote URLs.

---

3. `background-repeat`

Controls if the background image repeats.

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

---

4. `background-size`

Sets the size of the background image.

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

---

5. `background-position`

Positions the background image within the element.

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

---

6. `background-attachment`

Controls scroll behavior.

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

7. Shorthand Property: `background`

You can combine all background properties in one line.

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

Border Properties

1. `border-width`

Sets the thickness of the border.

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

2. `border-style`

Defines the style of the border.

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

3. `border-color`

Sets the color of the border.

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

---

4. Shorthand: `border`

You can combine width, style, and color.

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

---

5. Individual Sides

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

---

6. `border-radius`

Rounds the corners of an element.

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

You can also use percentages to make circular shapes:

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

Summary

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

Margin and Padding in CSS

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

Padding vs Margin

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

Padding

Apply space inside the border, around the content.

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

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

Individual sides

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

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

Shorthand

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

Margin

Adds space outside the border of an element.

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

This separates the `.card` from nearby elements.

Individual sides

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

Shorthand

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

Auto Margin (Horizontal Centering)

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

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

Margin Collapse

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

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

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

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

Summary

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

The Display Property in CSS

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

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

Common Display Values

1. `block`

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

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

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

---

2. `inline`

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

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

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

---

3. `inline-block`

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

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

---

4. `none`

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

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

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

---

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

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

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

Visual Example

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

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

Changing Display in Practice

```css
nav {
  display: block;
}

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

Summary

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

Positioning in CSS

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

The `position` Property

There are **five main values**:

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

---

1. `static` (Default)

Every element is positioned statically by default.

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

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

---

2. `relative`

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

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

It stays in the document flow, but shifts slightly.

---

3. `absolute`

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

```
.parent {
  position: relative;
}

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

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

---

4. `fixed`

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

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

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

---

5. `sticky`

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

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

Sticky headers or sidebars often use this behavior.

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

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

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

`z-index`

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

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

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

Summary

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

Flexbox in CSS

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

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

Getting Started

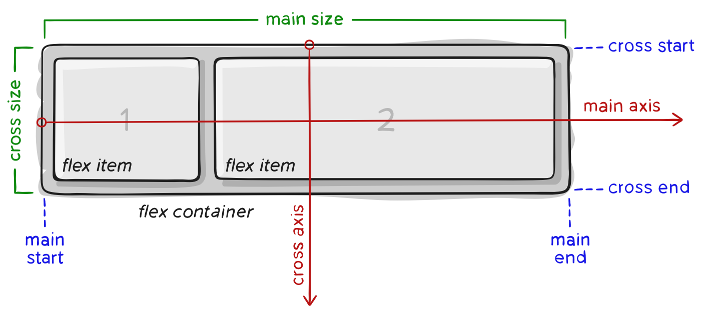
To use Flexbox, set the parent container's `display` to `flex` :

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

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

Main Concepts

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



Flex Direction

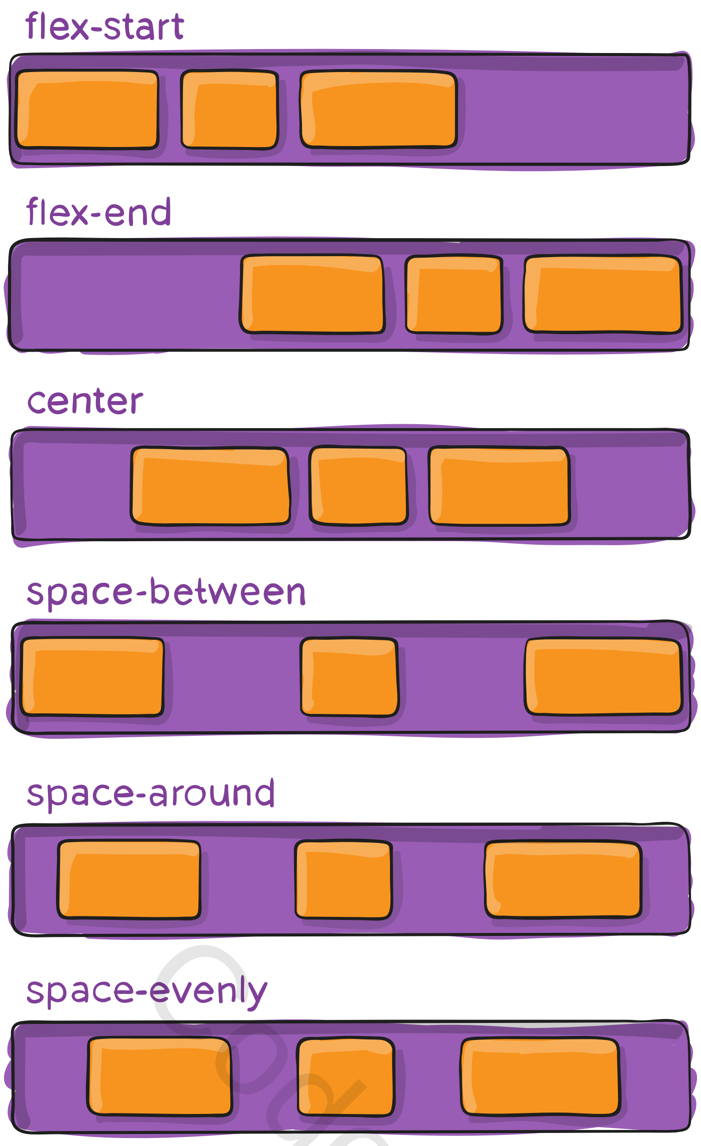
Controls the direction of flex items.

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

Justify Content (Main Axis Alignment)

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

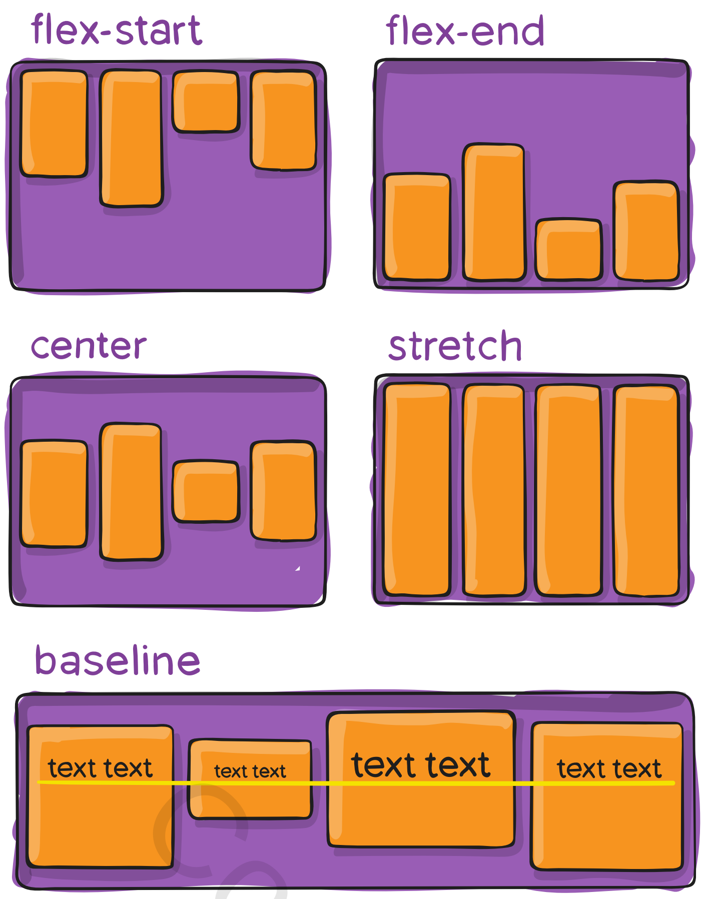
```
.container {
  justify-content: flex-start;    /* default */
  justify-content: flex-end;
  justify-content: center;
  justify-content: space-between;
  justify-content: space-around;
  justify-content: space-evenly;
}
```



Align Items (Cross Axis Alignment)

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

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



Align Self

Allows individual items to override `align-items` .

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

Flex Wrap

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

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

Gap (Spacing Between Items)

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

This replaces the need for margins between flex items.

Flex Grow, Shrink, Basis

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

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

Shorthand:

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

Example Layout

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

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

Summary

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

CSS Grid

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

Enabling Grid

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

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

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

Defining Rows and Columns

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

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

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

Repeat Syntax

To avoid repeating values:

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

Creates 3 equal-width columns.

Grid Gap

Adds spacing between rows and columns:

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

Placing Items

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

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

You can also use `span`:

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

Named Areas (Optional but Powerful)

Define areas using `grid-template-areas` :

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

Then assign each item:

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

Auto-Placement

Grid can automatically place items:

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

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

Complete Example

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

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

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

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

Summary

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

CSS Media Queries

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

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

Basic Syntax

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

Example: Target screens smaller than 768px

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

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

Common Conditions

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

Responsive Layout Example

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

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

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

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

Combining Multiple Conditions

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

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

Mobile-First Approach

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

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

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

/* Desktop and up */
@media (min-width: 1024px) {
  .card {
    font-size: 18px;
  }
}
```

Media Queries for Print

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

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

Used to style web pages when printed.

Summary

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

CSS Variables and Custom Properties

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

They follow the pattern of:

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

Declaring a CSS Variable

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

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

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

Using a CSS Variable

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

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

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

Why Use CSS Variables?

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

Example: Theming with CSS Variables

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

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

You can override these in a different class for themes:

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

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

Fallback Values

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

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

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

Scoped Variables

Variables can also be scoped to a class or element:

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

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

Real-World Example

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

.button {
  padding: var(--btn-padding);
  color: var(--btn-color);
  background-color: var(--btn-bg);
  border: none;
  border-radius: 6px;
  cursor: pointer;
}
```

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

Summary

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

CSS Transitions and Animations

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

---

1. CSS Transitions

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

Basic Syntax

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

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

Example

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

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

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

Shorthand vs Longhand

Shorthand:

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

Longhand:

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

---

2. CSS Animations

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

Basic Syntax

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

Keyframes

Define how the animation should behave at different points:

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

Example

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

Animation Properties

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

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

Looping Animations

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

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

Combining Transitions and Animations

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

Example using both:

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

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

@keyframes fadeIn {
  from { opacity: 0; }
  to { opacity: 1; }
}

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

Summary

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

CSS Transformations

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

1. `transform` Property

Syntax:

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

Multiple functions can be combined:

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

2. Types of Transformations

a. `translate()`

Moves an element from its current position.

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

Other variations:

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

---

b. `rotate()`

Rotates the element clockwise by default.

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

Use negative values to rotate counter-clockwise:

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

---

c. `scale()`

Scales the size of an element.

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

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

---

d. `skew()`

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

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

Individual axis:

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

---

e. `matrix()`

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

---

3. Transform Origin

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

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

---

4. Combining Multiple Transforms

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

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

---

5. 3D Transforms (Intro Only)

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

Example:

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

Summary

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

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