

1. Introduction to CSS

What is CSS?

- **Definition and Purpose:** CSS (Cascading Style Sheets) is a style sheet language used for describing the presentation of a document written in HTML. It controls the layout, colors, fonts, and spacing of web pages.
- **Brief History and Evolution:** CSS was first introduced in 1996 by the W3C and has evolved through multiple versions, including CSS1, CSS2, and the latest CSS3 with advanced features like Flexbox, Grid, and animations.

Why Learn CSS?

- **Importance in Web Design and Development:** CSS is essential for creating visually appealing, structured, and responsive web pages.
- **Enhancing User Experience:** Proper use of CSS improves readability, navigation, and user interaction, making websites more engaging.

How CSS Works

- **Role of the Browser:** Browsers interpret CSS rules to render web pages as specified by the developer.
- **CSS Rendering Process:** The browser applies styles by parsing the CSS, constructing a style tree, and rendering elements accordingly.

2. Purpose of CSS

Separation of Content and Design

- **HTML for Structure, CSS for Style:** CSS allows developers to separate content (HTML) from design, making code cleaner and more maintainable.
- **Easier Maintenance:** Changing the appearance of a website can be done by modifying the CSS without altering the HTML structure.

3. Advantages of Using CSS

Reusability and Consistency

- **Reusable Styles:** CSS rules can be applied across multiple pages, reducing redundancy and saving time.

- **Consistency Across Pages:** A single CSS file can ensure that all web pages share the same look and feel.

Better Accessibility and Performance

- **Improved Readability:** CSS enhances accessibility by allowing better font choices, spacing, and layout adjustments.
- **Faster Loading Times:** External CSS files reduce the size of HTML files, leading to quicker page loads.

4. CSS in Modern Web Development

Integration with JavaScript Frameworks

- **Enhancing Interactivity:** CSS works alongside JavaScript libraries like React, Angular, and Vue.js to create dynamic user interfaces.
- **Component-Based Styling:** Modern frameworks use CSS-in-JS, SCSS, and CSS modules for better styling management.

Responsive and Adaptive Design

- **Media Queries:** CSS enables responsive design by allowing styles to change based on screen size and device type.
- **Flexbox & Grid:** These CSS features help create adaptive layouts that adjust seamlessly to different screen resolutions.

CSS Properties

Definition of CSS Properties

CSS properties define the style and behavior of HTML elements. Each property controls a specific aspect of how an element is displayed on a webpage.

Basic Syntax

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

- **property:** The style attribute you want to modify (e.g., color, font-size).
- **value:** The setting for that property (e.g., red, 16px).

Example

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

This will set the text color of all `<p>` elements to blue and the font size to 16px.

Categories of Properties

1. Text-Related Properties

1. Color

Definition and Purpose: The `color` property sets the color of the text on a webpage. It enhances the visual appearance of content.

Importance in Web Design and Development: Used to improve the readability and aesthetics of text content.

How CSS `color` Works: The browser applies the specified color to the text of the selected HTML element.

Syntax:

```
selector {  
  color: value;  
}
```

Examples:

```
p {  
  color: red; /* Sets the text color to red */  
}  
  
h1 {  
  color: #3498db; /* Sets the text color to a shade of blue */  
}
```

2. Font-Size

Definition and Purpose: The `font-size` property specifies the size of the text. It helps in making text more readable and adaptable.

Importance in Web Design and Development: Adjusts text size for better readability and accessibility.

How CSS `font-size` Works: The browser scales the text based on the value defined.

Syntax:

```
selector {  
  font-size: value;  
}
```

Examples:

```
p {  
  font-size: 16px; /* Sets the font size to 16 pixels */  
}  
  
h2 {  
  font-size: 1.5em; /* Sets the font size to 1.5 times the parent element  
  */  
}
```

3. Font-Family

Definition and Purpose: The `font-family` property defines the typeface for text content. It allows customization of typography.

Importance in Web Design and Development: Ensures consistent and visually appealing typography across the website.

How CSS `font-family` Works: Browsers display the specified font, falling back to alternatives if the first isn't available.

Syntax:

```
selector {  
  font-family: value;  
}
```

Examples:

```
body {
  font-family: Arial, sans-serif; /* Uses Arial, falls back to sans-serif */
}

h1 {
  font-family: 'Times New Roman', serif; /* Uses Times New Roman */
}
```

4. Font-Weight

Definition and Purpose: The `font-weight` property defines the thickness or boldness of text.

Importance in Web Design and Development: Used to highlight important text or headings.

How CSS `font-weight` Works: Browsers apply the defined weight to text, ranging from normal to bold.

Syntax:

```
selector {
  font-weight: value;
}
```

Examples:

```
strong {
  font-weight: bold; /* Makes text bold */
}

p {
  font-weight: 600; /* Applies a semi-bold effect */
}
```

5. Text-Align

Definition and Purpose: The `text-align` property sets the horizontal alignment of text within its container.

Importance in Web Design and Development: Improves content layout and readability.

How CSS `text-align` Works: The browser aligns the text as specified.

Syntax:

```
selector {  
  text-align: value;  
}
```

Examples:

```
div {  
  text-align: center; /* Centers the text */  
}  
p {  
  text-align: justify; /* Justifies the text */  
}
```

6. Line-Height

Definition and Purpose: The **line-height** property sets the spacing between lines of text, improving readability.

Importance in Web Design and Development: Controls text density for better user experience.

How CSS **line-height Works:** The browser adjusts the space between lines based on the defined value.

Syntax:

```
selector {  
  line-height: value;  
}
```

Examples:

```
p {  
  line-height: 1.5; /* 1.5 times the font size */  
}  
  
h2 {  
  line-height: 20px; /* Fixed 20px spacing */  
}
```

7. Text-Transform

Definition and Purpose: The `text-transform` property controls text capitalization.

Importance in Web Design and Development: Improves text formatting and emphasis.

How CSS `text-transform` Works: The browser changes the text case according to the specified value.

Syntax:

```
selector {  
  text-transform: value;  
}
```

Examples:

```
h1 {  
  text-transform: uppercase; /* Converts text to uppercase */  
}  
p {  
  text-transform: capitalize; /* Capitalizes the first letter of each word */  
}
```

Box Model Properties

Introduction Margin

The `margin` property in CSS is used to create space around elements, outside of any defined borders. It helps in positioning elements on a webpage and managing spacing between elements.

Syntax

```
selector {  
  margin: value; /* Can be auto, length, percentage, inherit, or  
multiple values */  
}
```

Different Ways to Define Margins

Margins can be set using different units:

- **Auto:** The browser automatically calculates the margin.
- **Length (px, em, rem, etc.):** Defines a fixed space.
- **Percentage (%):** Sets margin relative to the width of the containing element.
- **Inherit:** Inherits the margin value from its parent element.

Individual Margin Properties

1 margin-top

Defines the top margin of an element.

Syntax:

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

Example:

```
<div style="margin-top: 30px; background: lightblue;">Top Margin  
Applied</div>
```

2 margin-right

Defines the right margin of an element.

Syntax:

```
div {  
    margin-right: 15px;  
}
```

Example:

```
<div style="margin-right: 25px; background: lightgreen;">Right Margin  
Applied</div>
```

3 margin-bottom

Defines the bottom margin of an element.

Syntax:

```
div {  
    margin-bottom: 10px;  
}
```

Example:

```
<div style="margin-bottom: 20px; background: lightcoral;">Bottom Margin  
Applied</div>
```

4 margin-left

Defines the left margin of an element.

Syntax:

```
div {  
    margin-left: 25px;  
}
```

Example:

```
<div style="margin-left: 40px; background: lightsalmon;">Left Margin  
Applied</div>
```

Shorthand Margin Property

Margins can be specified in one declaration using shorthand notation.

Syntax:

```
selector {  
    margin: top right bottom left;  
}
```

Example:

```
div {  
    margin: 10px 20px 30px 40px;  
}
```

This sets:

- top margin to 10px

- **right** margin to **20px**
- **bottom** margin to **30px**
- **left** margin to **40px**

Variations of Shorthand

- **Two values:** `margin: vertical horizontal;`
- **Three values:** `margin: top horizontal bottom;`
- **One value:** `margin: all-sides;`

Centering Elements Using Margin Auto

To center a block element horizontally, use `margin: auto;`

Example:

```
div {  
  width: 50%;  
  margin: auto;  
  background-color: lightgray;  
}  
  
<div>This div is centered</div>
```

Negative Margins

Negative values can be used to reduce space.

Example:

```
div {  
  margin-left: -10px;  
}
```

Best Practices

- Avoid excessive negative margins to prevent layout issues.
- Use `auto` for centering elements effectively.
- Prefer `rem` or `em` over `px` for responsive designs.

Summary Table

Property	Description	Example
<code>margin</code>	Sets margin on all sides	<code>margin: 10px 20px 30px 40px;</code>
<code>margin-top</code>	Sets top margin	<code>margin-top: 10px;</code>
<code>margin-right</code>	Sets right margin	<code>margin-right: 15px;</code>
<code>margin-bottom</code>	Sets bottom margin	<code>margin-bottom: 20px;</code>
<code>margin-left</code>	Sets left margin	<code>margin-left: 25px;</code>

Padding in CSS

Definition and Purpose

The `padding` property in CSS is used to create space between an element's content and its border. It improves layout structure and content readability by controlling the internal spacing of elements.

Why Learn Padding?

- **Improves Layout:** Helps in spacing content within containers.
- **Enhances Readability:** Prevents content from touching the element's border.

How CSS Padding Works

- **Role of the Browser:** Browsers calculate padding values and apply them inside the element's border.
- **Rendering Process:** Padding affects the box model, impacting the overall size and layout of the element.

Syntax

```
selector {  
  padding: value;  
}
```

Values: Length units ([px](#), [em](#), [rem](#)), percentages ([%](#)), or [inherit](#).

Individual Padding Properties

1. padding-top

Defines the top padding of an element.

Syntax:

```
div {  
  padding-top: 20px;  
}
```

Example:

```
<div style="padding-top: 30px; background: lightblue;">Top Padding  
Applied</div>
```

2. padding-right

Defines the right padding of an element.

Syntax:

```
div {  
  padding-right: 15px;  
}
```

Example:

```
<div style="padding-right: 25px; background: lightgreen;">Right Padding  
Applied</div>
```

3. padding-bottom

Defines the bottom padding of an element.

Syntax:

```
div {  
    padding-bottom: 10px;  
}
```

Example:

```
<div style="padding-bottom: 20px; background: lightcoral;">Bottom Padding  
Applied</div>
```

4. padding-left

Defines the left padding of an element.

Syntax:

```
div {  
    padding-left: 25px;  
}
```

Example:

```
<div style="padding-left: 40px; background: lightsalmon;">Left Padding  
Applied</div>
```

Shorthand Padding Property

You can define all four paddings in a single line.

Syntax:

```
selector {  
    padding: top right bottom left;  
}
```

Example:

```
div {  
    padding: 10px 20px 30px 40px;  
}
```

Responsive Padding with Percentages

Example:

```
div {  
  padding: 5%;  
}
```

This sets the padding to 5% of the element's width.

Best Practices

- Use consistent units (`px`, `em`, `rem`) for uniform design.
- Avoid excessive padding to prevent layout issues.
- Prefer relative units for responsive designs.

Summary Table

Property	Description	Example
<code>padding</code>	Sets padding on all sides	<code>padding: 20px;</code>
<code>padding-top</code>	Sets top padding	<code>padding-top: 10px;</code>
<code>padding-right</code>	Sets right padding	<code>padding-right: 15px;</code>
<code>padding-bottom</code>	Sets bottom padding	<code>padding-bottom: 20px;</code>
<code>padding-left</code>	Sets left padding	<code>padding-left: 25px;</code>

Border in CSS

Definition and Purpose

The `border` property in CSS is used to create a border around an HTML element. It defines the style, width, and color of the border.

Syntax

```
selector {  
  border: width style color;  
}
```

Example:

```
div {  
  border: 2px solid black;  
}
```

Individual Border Properties

- **border-width:** Sets the width of the border.
Example: `border-width: 3px;`
- **border-style:** Sets the style (solid, dashed, dotted, etc.).
Example: `border-style: dashed;`
- **border-color:** Sets the color of the border.
Example: `border-color: blue;`
- **border-radius:** Rounds the corners of the border.
Example: `border-radius: 10px;`

Best Practices

- Use `border-radius` for modern design aesthetics.
- Combine shorthand for efficiency: `border: 1px solid #ccc;`
- Use `rgba` colors for transparent borders.

Summary Table

Property	Description	Example
<code>border</code>	Sets all border properties	<code>border: 1px solid black;</code>
<code>border-width</code>	Sets width of border	<code>border-width: 2px;</code>

`border-style` Sets style of border

`border-style:`
`dashed;`

`border-color` Sets color of border

`border-color: red;`

Width and Height in CSS

Definition and Purpose

- **Width:** Defines the horizontal size of an element.
- **Height:** Defines the vertical size of an element.

These properties control the size of elements and are crucial for creating well-structured layouts.

Syntax

```
selector {  
  width: value;  
  height: value;  
}
```

Values: Length units (`px`, `em`, `rem`), percentages (`%`), `auto`, `max-content`, `min-content`.

Width Property

Syntax:

```
div {  
  width: 300px;  
}
```

Examples:

1. Fixed width:

```
div {  
  width: 400px;  
}
```

2. Percentage width (responsive):

```
div {  
  width: 50%;  
}
```

3. Auto width:

```
div {  
  width: auto;  
}
```

Height Property

Syntax:

```
div {  
  height: 200px;  
}
```

Examples:

1.Fixed height:

```
div {  
  height: 150px;  
}
```

2.Percentage height (responsive):

```
div {
```

```
height: 70%;  
}
```

3. Auto height:

```
div {  
    height: auto;  
}
```

Best Practices

- Use relative units (`%`, `em`, `rem`) for responsive design.
- Avoid setting both fixed width and height for flexible layouts.
- Combine with `max-width` and `max-height` for better responsiveness.

Summary Table

Property	Description	Example
<code>width</code>	Sets element's width	<code>width: 300px;</code>
<code>height</code>	Sets element's height	<code>height: 200px;</code>
<code>max-width</code>	Sets maximum width	<code>max-width:</code> <code>100%;</code>
<code>max-height</code>	Sets maximum height	<code>max-height:</code> <code>500px;</code>

Background-Related Properties

1 Background-Color

- **Definition:** Sets the background color of an element.

Syntax:

```
selector {  
    background-color: color;  
}
```

Example 1:

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

Example 2:

```
section {  
    background-color: #f2f2f2;  
}
```

2 Background-Image

- **Definition:** Sets an image as the background of an element.

Syntax:

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

Example 1:

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

Example 2:

```
header {  
    background-image: url('header-image.png');  
}
```

3 Background-Repeat

- **Definition:** Specifies if and how a background image will repeat.

Syntax:

```
selector {  
    background-repeat: value;  
}
```

Example 1:

```
div {  
    background-repeat: no-repeat;  
}
```

Example 2:

```
footer {  
    background-repeat: repeat-x;  
}
```

1.4 Background-Position

- **Definition:** Specifies the position of the background image.

Syntax:

```
selector {  
    background-position: x-axis y-axis;  
}
```

Example 1:

```
div {  
    background-position: center center;  
}
```

Example 2:

```
section {  
    background-position: top left;  
}
```

5 Background-Size

- **Definition:** Specifies the size of the background image.

Syntax:

```
selector {  
    background-size: value;  
}
```

Example 1:

```
div {  
    background-size: cover;  
}
```

Example 2:

```
header {  
    background-size: 100% 100%;  
}
```

6 Background-Attachment

- **Definition:** Specifies whether the background image scrolls with the content or remains fixed.

Syntax:

```
selector {  
    background-attachment: value;  
}
```

Example 1:

```
div {  
    background-attachment: fixed;  
}
```

Example 2:

```
section {  
    background-attachment: scroll;  
}
```

7 Background (Shorthand)

- **Definition:** A shorthand property for setting multiple background properties in one declaration.

Syntax:

```
selector {  
    background: color image position/size repeat attachment origin clip;  
}
```

Example 1:

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

Example 2:

```
footer {  
    background: #f2f2f2 url('footer-bg.jpg') repeat-x;  
}
```

Border-Related Properties

1 Border

- **Definition:** Sets the width, style, and color of the border of an element.

Syntax:

```
selector {  
    border: width style color;  
}
```

Example 1:

```
div {  
    border: 2px solid black;  
}
```

Example 2:

```
section {  
    border: 3px dashed red;  
}
```

2 Border-Width

- **Definition:** Specifies the width (thickness) of the border of an element.

Syntax:

```
selector {  
    border-width: value;  
}
```

Example 1:

```
div {  
    border-width: 5px;  
}
```

Example 2:

```
header {  
    border-width: 1px 2px;  
}
```

3 Border-Style

- **Definition:** Specifies the style of the border (e.g., solid, dashed).

Syntax:

```
selector {  
    border-style: value;  
}
```

Example 1:

```
div {  
  border-style: solid;  
}
```

Example 2:

```
section {  
  border-style: dotted;  
}
```

4 Border-Color

- **Definition:** Specifies the color of the border.

Syntax:

```
selector {  
  border-color: color;  
}
```

Example 1

```
div {  
  border-color: red;  
}
```

Example 2:

```
footer {  
  border-color: #000;  
}
```

5 Border-Radius

- **Definition:** Rounds the corners of an element's border.

Syntax:

```
selector {  
    border-radius: value;  
}
```

Example 1:

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

Example 2:

```
button {  
    border-radius: 50%;  
}
```

6 Border-Top, Border-Right, Border-Bottom, Border-Left

- **Definition:** Specifies the width, style, and color of each side of an element's border.

Syntax:

```
selector {  
    border-top: width style color;  
    border-right: width style color;  
    border-bottom: width style color;  
    border-left: width style color;  
}
```

Example 1:

```
div {  
    border-top: 2px solid black;  
    border-right: 2px dashed green;  
}
```

Example 2:

```
section {  
  border-left: 3px solid blue;  
  border-bottom: 3px dotted red;  
}
```

2.7 Border-Image

- **Definition:** Allows you to use an image as a border for an element.

Syntax

```
selector {  
  border-image: url('image.png') slice width repeat;  
}
```

Example 1:

```
div {  
  border-image: url('border.png') 30 stretch;  
}
```

Example 2:

```
header {  
  border-image: url('header-border.png') 20 30 20 30 round;  
}
```

Display and Positioning Properties

Display Properties

1 display

Definition:

Specifies the display behavior of an element (e.g., whether it's a block, inline, or another display type).

Syntax:

```
element {  
  display: value;  
}
```

Examples:

1. Set an element to display as a block:

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

Block elements take up the full width available, stacking vertically.

2. Set an element to display as inline:

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

Inline elements take up only as much width as necessary and flow horizontally within a container.

2 display: none**Definition:**

Removes the element from the document layout, making it invisible and not taking up space.

Syntax:

```
element {  
  display: none;  
}
```

Examples:

1. Hide an element without affecting layout:

```
.hide-element {  
  display: none;  
}
```

2. Toggle visibility using JavaScript:

```
document.querySelector(".hide-element").style.display = "none";
```

3 display: flex

Definition:

Enables flexbox layout, where child elements are arranged along a flexible container's main and cross axes.

Syntax:

```
element {  
  display: flex;  
}
```

Examples:

1. Set up a flex container:

```
.container {  
  display: flex;  
  justify-content: space-between;  
}
```

2. Create a horizontal flex layout:

```
.flex-box {  
  display: flex;  
  align-items: center;  
}
```

4 display: grid

Definition:

Enables grid layout, allowing elements to be positioned into rows and columns.

Syntax:

```
element {  
  display: grid;  
}
```

Examples:

1. Set up a basic grid layout:

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

2. Create a 3x3 grid:

```
.grid-box {  
  display: grid;  
  grid-template-columns: repeat(3, 1fr);  
  grid-gap: 10px;  
}
```

Positioning Properties

1. position: static

Definition:

Default position value. Elements are positioned based on the normal document flow.

Syntax:

```
element {  
  position: static;  
}
```

Examples:

Static is the default behavior, so no special declaration is needed:

```
p {  
  position: static;  
}
```

2 position: relative

Definition:

An element is positioned relative to its normal position, allowing for adjustment with **top**, **right**, **bottom**, and **left** properties.

Syntax:

```
element {  
  position: relative;  
  top: value;  
  left: value;  
}
```

Examples:

1. Shift an element 20px down and 30px right from its normal position:

```
div {  
  position: relative;  
  top: 20px;  
  left: 30px;  
}
```

2. Apply relative positioning to a child element

```
.parent {  
  position: relative;  
}  
  
.child {  
  position: relative;  
  top: 10px;  
  left: 10px;  
}
```

3 position: absolute

Definition:

Positions an element relative to its closest positioned ancestor (non-static). If no positioned ancestor exists, it's positioned relative to the initial containing block (usually the `<html>` element).

Syntax:

```
element {  
  position: absolute;  
  top: value;  
  right: value;  
  bottom: value;  
  left: value;  
}
```

Examples:

1. Position an element absolutely inside a parent:

```
.parent {  
  position: relative;  
}  
.child {  
  position: absolute;  
  top: 50px;  
  left: 100px;  
}
```

2. Position an element at the top-left corner:

```
div {  
  position: absolute;  
  top: 0;  
  left: 0;  
}
```

4 position: fixed

Definition:

Positions an element relative to the browser window (viewport), so it stays in the same position even when the page is scrolled.

Syntax:

```
element {  
  position: fixed;  
  top: value;  
  right: value;  
  bottom: value;  
  left: value;  
}
```

Examples:

1. Create a fixed navigation bar:

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

2. Position a button at the bottom-right corner:

```
button {  
  position: fixed;  
  bottom: 10px;  
  right: 10px;  
}
```

5 position: sticky

Definition:

An element is treated as relative until it reaches a defined scroll position, then it "sticks" to the top (or another defined position) of its container.

Syntax:

```
element {  
  position: sticky;  
  top: value;  
}
```

Examples:

1. Create a sticky header:

```
header {  
  position: sticky;  
  top: 0;  
  background-color: #fff;  
}
```

2. Create a sticky sidebar:

```
.sidebar {  
  position: sticky;  
  top: 50px;  
}
```

z-index Property

1 z-index

Definition:

Controls the stacking order of positioned elements that overlap. Elements with a higher **z-index** appear in front of those with a lower **z-index**.

Syntax:

```
element {  
  z-index: integer;  
}
```

Examples:

1. Place one element on top of another:

```
.box1 {  
  position: absolute;  
  z-index: 2;  
}  
.box2 {  
  position: absolute;  
  z-index: 1;  
}
```

2. Change the stacking order dynamically:

```
.front {  
  position: absolute;  
  z-index: 10;  
}  
.back {  
  position: absolute;  
  z-index: 1;  
}
```

Best Practices for CSS Properties

- **Readability:** Use indentation and comments for clarity.
- **Consistency:** Stick to consistent units (px, em, rem).
- **Maintainability:** Use external stylesheets for larger projects.
- **Optimization:** Minimize redundant properties to improve performance.

Summary Table

Property	Description	Example
color	Sets text color	color: red;
font-size	Sets the size of the text	font-size: 16px;

margin	Adds space outside an element	<code>margin: 20px;</code>
padding	Adds space inside an element	<code>padding: 10px;</code>
border	Adds a border around an element	<code>border: 1px solid black;</code>
background-color	Sets background color	<code>background-color: yellow;</code>
display	Defines element display type	<code>display: flex;</code>
position	Sets the positioning of elements	<code>position: relative;</code>
z-index	Controls element stacking order	<code>z-index: 1;</code>

Commonly Used Properties

1. Overflow

Definition:

Specifies what happens if content overflows an element's box.

Syntax:

```
element {  
  overflow: value;  
}
```

Examples:

1. Hide overflow content:

```
div {  
  overflow: hidden;  
}
```

2. Allow horizontal scrolling:

```
div {  
  overflow-x: scroll;  
}
```

Practical Use Case:

Use Case: Hiding excess content in a container.

```
.content-wrapper {  
  overflow: hidden;  
}
```

2. Box-shadow

Definition:

Adds a shadow effect to an element's box.

Syntax:

```
element {  
  box-shadow: horizontal-offset vertical-offset blur-radius spread-radius  
  color;  
}
```

Examples:

1. Add a simple shadow:

```
div {  
  box-shadow: 5px 5px 10px rgba(0, 0, 0, 0.5);  
}
```

2. Add an inset shadow:

```
div {  
  box-shadow: inset 0 0 10px rgba(0, 0, 0, 0.5);  
}
```

Practical Use Case:

Use Case: Creating depth for UI components like cards.

```
.card {  
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);  
}
```

3. Visibility

Definition:

Controls the visibility of an element, but it still takes up space in the layout.

Syntax:

```
element {  
  visibility: value;  
}
```

Examples:

1. Make an element invisible (but still takes space):

```
p {  
  visibility: hidden;  
}
```

2. Make an element visible:

```
p {  
  visibility: visible;  
}
```

Practical Use Case:

Use Case: Hiding content without removing it from the layout.

```
.invisible {  
  visibility: hidden;  
}
```

4. Opacity

Definition:

Sets the opacity level of an element, making it transparent or semi-transparent.

Syntax:

```
element {  
  opacity: value;  
}
```

Examples:

1. Set opacity to 50%:

```
div {  
  opacity: 0.5;  
}
```

2. Set opacity to 100% (fully opaque):

```
button {  
  opacity: 1;  
}
```

Practical Use Case:

Use Case: Creating hover effects with transparency.

```
.hover-effect:hover {  
  opacity: 0.8;  
}
```

Structure of a CSS Declaration

1 Selector, Property, and Value

Definition:

A **CSS declaration** consists of a **selector**, a **property**, and a **value**.

- **Selector:** Specifies which HTML element(s) the style will apply to.
- **Property:** Defines what aspect of the element you want to style (e.g., `color`, `font-size`).
- **Value:** Defines the specific setting for the property (e.g., `red`, `16px`).

Syntax:

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

Examples:

1. Styling the background color of all paragraphs:

```
p {  
  background-color: lightgray;  
}
```

2. Changing the font size for a class `header`:

```
.header {  
  font-size: 24px;  
}
```

CSS Syntax Rules

1 Importance of Semicolons and Curly Braces

Definition:

- **Semicolons** are used to separate multiple declarations within a rule set.
- **Curly braces** `{ }` enclose the block of CSS declarations for a specific selector.

Syntax:

```
selector {  
  property: value; /* Semicolon after each declaration */  
  property2: value2;  
}
```

Examples:

1. Semicolons:

Each property-value pair is terminated with a semicolon (except for the last one, although it's recommended to use semicolons consistently).

```
p {  
  color: blue;  
  font-size: 16px; /* Semicolon after each declaration */  
}
```

2. Curly Braces:

Curly braces wrap the block of declarations for the selector.

```
.box {  
  width: 100px;  
  height: 200px;  
}
```

Practical Use Case:

- Always use semicolons at the end of every declaration to avoid errors and improve readability.
- Properly wrap declarations with curly braces to define the scope.

Best Practices

1 Readability and Maintainability

Definition:

Writing clear and understandable CSS code is important for long-term project maintainability. Consistency and structure improve collaboration among developers and ease of updates.

Best Practices:

1. Consistent Naming Conventions:

Use descriptive and consistent class and ID names to improve readability.

```
.main-header { /* Good */  
  font-size: 28px;  
}  
.mh { /* Bad: unclear naming */  
  font-size: 28px;  
}
```

2. Use Short and Clear Properties:

Prefer shorthand properties where possible for cleaner code.

```
/* Shorthand for margin */  
div {  
  margin: 10px 20px;  
}  
  
/* Longer version */  
div {  
  margin-top: 10px;  
  margin-right: 20px;  
  margin-bottom: 10px;  
  margin-left: 20px;  
}
```

3. Organizing Styles:

Group related styles together, such as typography, layout, colors, etc.

```
/* Typography */  
h1, h2, h3 {  
  font-family: Arial, sans-serif;  
}  
  
/* Layout */  
.container {  
  display: flex;
```

```
justify-content: center;
}
```

```
/* Colors */
.button {
  background-color: #3498db;
  color: white;
}
```

2 Using Comments and Indentation

Definition:

- **Comments** are used to explain sections of code and improve understanding.
- **Indentation** organizes code structure to make it easier to read.

Syntax:

1. Comments:

```
/* This is a comment */
selector {
  property: value;
}
```

2. Indentation:

Use consistent indentation (typically 2 or 4 spaces) to organize the code.

Examples:

Comments:

```
/* Header styles */
h1 {
  font-size: 32px;
}

/* Footer styles */
footer {
  background-color: #2c3e50;
}
```

Indentation:

Keep proper indentation to make code readable.

```
/* Correct Indentation */
.container {
  display: flex;
  justify-content: space-between;
}

/* Bad Indentation */
.container {
display: flex;
justify-content: space-between;
}
```

Practical Use Case:

Comments: Provide context to complex styles.

```
/* This section defines the colors used in the theme */
body {
  background-color: #f4f4f4;
}
```

Indentation: Organize nested CSS rules with proper indentation.

```
.wrapper {
  display: flex;
  flex-direction: column;
  .child {
    margin: 10px;
  }
}
```

General Best Practices for CSS

1. Avoid Inline Styles:

Inline styles can clutter HTML and reduce maintainability. Use external or internal stylesheets instead.

```

<!-- Avoid Inline Style -->
<div style="color: red; font-size: 16px;">Hello</div>

<!-- Preferred Approach -->
<style>
  .greeting {
    color: red;
    font-size: 16px;
  }
</style>
<div class="greeting">Hello</div>

```

2.Minimize the Use of !important:

Overuse of `!important` can lead to specificity issues and make your code difficult to manage. Try to use it only when absolutely necessary.

```

/* Avoid overusing !important */
p {
  color: blue !important;
}

```

3. Use Consistent Units:

Consistently use the same units where possible. For instance, avoid mixing `px` with `em` unless necessary.

```

/* Consistent Units */
.text {
  font-size: 1.5em; /* Better for responsiveness */
}

/* Mixing Units */
.text {
  font-size: 16px;
  line-height: 1.5em; /* Mixing px with em */
}

```

4. CSS File Organization:

1. Use separate stylesheets for different parts of the site (e.g., one for layout, one for typography, etc.).
2. Maintain a logical order (e.g., reset/normalize, layout, components, utilities).

Approach in CSS

1. Inline CSS

i). Syntax and Use Cases

Definition:

Inline CSS is applied directly within the `style` attribute of an HTML element. It is used for styling a single element without the need for external or internal styles.

Syntax:

```
<element style="property: value;">
  Content
</element>
```

Examples:

1. Applying inline styles to a paragraph:

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

2. Applying background color to a div:

```
<div style="background-color: #f0f0f0; padding: 20px;">This is a div with
inline styles.</div>
```

ii). Pros and Cons

Pros:

- **Quick and easy for small, single-use styling:** Inline CSS is quick for styling individual elements.
- **Overrides other styles:** Inline styles have the highest specificity and will override styles in internal and external CSS unless `!important` is used.

Cons:

- **Difficult to maintain:** It's harder to update or change styles across the site because each element has its own style.
- **Reduces reusability:** You can't reuse the styles, making the code repetitive.
- **Separation of concerns is violated:** Inline styles mix content and presentation, making the HTML harder to read and maintain.

Practical Use Case:

- **Use Case:** Inline CSS can be useful for quick prototyping or when you need to apply a specific style to one element temporarily.

```
<div style="font-size: 20px; color: red;">Single element style</div>
```

2. Internal CSS

1 **<style>** Tag in HTML

Definition:

Internal CSS is placed inside the **<style>** tag within the **<head>** section of the HTML document. It applies styles to elements within that specific HTML document.

Syntax:

```
<head>
  <style>
    selector {
      property: value;
    }
  </style>
</head>
```

Examples:

1. Adding styles inside the `<style>` tag:

```
<head>
  <style>
    h1 {
      color: blue;
      font-size: 32px;
    }
    p {
      color: gray;
    }
  </style>
</head>
```

2. Using internal CSS to style a button:

```
<head>
  <style>
    button {
      background-color: green;
      color: white;
      padding: 10px;
    }
  </style>
</head>
```

2 Scope and Applications

Scope:

Internal CSS only applies to the specific HTML document it is written in. If you have multiple pages, you'll need to duplicate the styles in each page or use external CSS for reusability.

Applications:

- **Small websites or single-page applications:** Internal CSS is a good choice for small projects that don't require a separate stylesheet.
- **Testing and quick prototyping:** When you are quickly testing styles in a single page, internal CSS can be a time-saver.

3. External CSS

1 Linking External Stylesheets

Definition:

External CSS is written in a separate `.css` file, and it is linked to HTML documents using the `<link>` tag in the `<head>` section. This method separates the HTML content from styling.

Syntax:

```
<head>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

Examples:

1. Linking an external CSS file to an HTML document:

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

Example `styles.css` file:

```
h1 {
  color: red;
  font-size: 40px;
}

p {
  color: black;
  line-height: 1.5;
}
```

2 Benefits of Modular CSS

Benefits:

- **Separation of concerns:** Keeps HTML content separate from its styling, which makes the code more readable and maintainable.

- **Reusability:** Styles defined in an external CSS file can be applied to multiple HTML documents, reducing redundancy.
- **Performance:** External CSS allows the browser to cache styles, reducing page load times when users navigate between pages of a site.
- **Modular approach:** By using separate CSS files for different sections (layout, typography, themes, etc.), you can keep the code organized and maintainable.

Practical Use Case:

- **Use Case:** External CSS is best for large projects where styles need to be shared across multiple pages or websites.

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

4. Comparison of CSS Approaches

1 When to Use Each Approach

i). Inline CSS:

- **When to use:**
 - For quick styling of one-off elements.
 - For testing styles temporarily.
 - When styles are dynamic and don't need to be reused (e.g., applying styles conditionally via JavaScript).

ii). Internal CSS:

- **When to use:**
 - For styling a single document without the need for separate stylesheets.
 - For small websites or when you don't need to reuse the styles across multiple pages.
 - When prototyping or working on projects with minimal styles.

iii). External CSS:

- **When to use:**
 - For large projects or websites where you need to maintain consistent styling across multiple pages.
 - When styling needs to be reused across multiple pages or websites.
 - When you want to keep the HTML clean and make your project scalable.
 - If performance is a priority (external CSS files are cached by browsers).

Comparison Table:

CSS Approach	Pros	Cons	Best Use Case
Inline CSS	<ul style="list-style-type: none">- Quick to implement.- Highest specificity	<ul style="list-style-type: none">- Hard to maintain.- Violates separation of concerns.	Quick prototyping or testing styles for a single element.
Internal CSS	<ul style="list-style-type: none">- Styles within the document.- No extra file needed.	<ul style="list-style-type: none">- Limited scope (applies to one document).	Small websites or single-page applications.
External CSS	<ul style="list-style-type: none">- Reusable across multiple pages.- Modular and organized.	<ul style="list-style-type: none">- Requires separate file.- Extra HTTP request.	Large websites, scalable projects, shared styles across pages.

Selectors

Overview of CSS Selectors

1 Purpose and Importance

Definition:

CSS selectors are used to target HTML elements and apply styles to them. The selector determines which elements are affected by the given CSS rule.

Importance:

- **Efficient styling:** Selectors help apply styles selectively and efficiently across a webpage.
- **Targeting specific elements:** CSS selectors allow you to target elements with precision, based on IDs, classes, element names, and relationships between elements.
- **Improved readability and maintenance:** Proper use of selectors makes the CSS code more organized and maintainable.

Examples of Selectors:

- Universal selector (*)
- Group selector (**selector1, selector2**)
- ID selector (**#id**)
- Class selector (**.class**)
- Tag/element selector (**tagName**)
- Descendant selector (**ancestor descendant**)
- Child selector (**parent > child**)

2. Universal Selector

1 Syntax: *

Definition:

The **universal selector** targets all elements within the document. It is often used for resetting or applying general styles to all elements.

Syntax:

```
* {  
  property: value;  
}
```

Examples:

1. Resetting margin and padding for all elements:

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

2. Setting a universal font for the entire document:

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

Use Cases:

- Use when you want to apply a universal style, such as resetting margins or applying default font styles.
- Not recommended for complex websites as it can be inefficient.

3. Group Selector

1 Syntax: `selector1, selector2`

Definition:

The **group selector** allows you to apply the same style to multiple elements by grouping them together in a comma-separated list. This helps reduce redundancy in your CSS.

Syntax:

```
selector1, selector2 {  
  property: value;  
}
```

Examples:

1. Applying the same style to multiple elements:

```
h1, h2, h3 {  
  font-family: 'Times New Roman', serif;  
  color: darkblue;  
}
```

2. Grouping selectors for links and buttons:

```
a, button {  
  text-decoration: none;  
  background-color: #3498db;  
  color: white;  
}
```

2. Combining Multiple Selectors:

- You can group selectors of different types (e.g., element, class, ID) to apply shared styles.

```
h1, .header, #main-title {  
  font-size: 24px;  
}
```

Use Cases:

- When multiple elements need the same styling.
- Helps reduce repetition in CSS, keeping the code concise.

4. ID Selector

1 Syntax: `#id`

Definition:

The **ID selector** targets an element with a specific `id` attribute. IDs must be unique within a document, meaning only one element should have a particular `id`.

Syntax:

```
#id {  
  property: value;  
}
```

Examples:

1. Styling a specific element with an ID:

```
#header {  
  background-color: #2c3e50;  
  color: white;  
}
```

2. Applying style to an element with a unique ID:

```
#footer {  
  padding: 20px;  
  text-align: center;  
}
```

Specificity and Uniqueness:

- IDs have a higher specificity than classes and tag selectors.
- An ID must be unique within a page to avoid conflicting styles.

Use Cases:

- Best for styling specific elements that are unique within a page, such as a header or footer.
- Useful for JavaScript targeting (e.g., `document.getElementById()`).

5. Class Selector

1 Syntax: `.class`

Definition:

The **class selector** targets elements that share the same class attribute. A class can be used on multiple elements, allowing for reusable styles.

Syntax:

```
.class {  
  property: value;  
}
```

Examples:

1. Styling elements with a specific class:

```
.btn {  
  background-color: green;  
  color: white;  
  padding: 10px 20px;  
  border-radius: 5px;  
}
```

2. Applying styles to multiple elements with the same class:

```
.container {  
  display: flex;  
  justify-content: space-between;  
}  
  
.container div {  
  margin: 10px;  
}
```

Reusability Across Elements:

- Classes are reusable, so you can apply the same styles to multiple elements, enhancing consistency and flexibility.

Use Cases:

- Use classes for elements that share similar styling or behavior (e.g., buttons, form inputs, containers).
- Commonly used in large-scale websites for modular and maintainable styling.

6. Tag/Element Selector

1 Syntax: **tagName**

Definition:

The **tag (element) selector** targets HTML elements based on their tag name (e.g., **div**, **p**, **a**).

Syntax:

```
tagName {  
  property: value;  
}
```

Examples:

1. Styling all paragraphs on a page:

```
p {  
  font-size: 16px;  
  line-height: 1.5;  
}
```

2. Changing the color of all links:

```
a {  
  color: #3498db;  
}
```

Applying Styles to Specific HTML Tags:

- Use tag selectors when you want to style all instances of a particular HTML element.

Use Cases:

- Use when applying styles to specific elements (like all paragraphs, images, etc.) without affecting other types.

7. Descendant Selector

1 Syntax: **ancestor descendant**

Definition:

The **descendant selector** targets elements that are nested within other elements. It selects an element that is a descendant (any level deep) of a specified ancestor.

Syntax:

```
ancestor descendant {  
  property: value;  
}
```

Examples:

1. Targeting paragraphs inside a container:

```
.container p {  
  font-size: 18px;  
  color: gray;  
}
```

2. Applying styles to nested links:

```
.sidebar a {  
  color: white;  
}
```

Use Case:

- Use when you want to style elements that are nested within specific parent elements, ensuring more specificity.

8. Child Selector

1. Syntax: **parent > child**

Definition:

The **child selector** targets elements that are **direct children** of a specific parent. Unlike the descendant selector, which targets all nested elements, the child selector only affects direct child elements.

Syntax:

```
parent > child {  
  property: value;  
}
```

Examples:

1. Targeting direct child paragraphs within a container:

```
.container > p {  
  font-size: 16px;  
}
```

2. Styling direct child divs within a section:

```
section > div {  
  margin-top: 20px;  
}
```

Use Case:

- Use when you need to target only direct children, rather than any nested elements.

9. Examples and Exercises

Here are some practical exercises for each selector:

1. Universal Selector:

Apply a universal box-sizing rule across all elements.

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

2. Group Selector:

Group multiple headings (h1, h2, h3) and apply a common font and color.

```
h1, h2, h3 {  
  font-family: Arial, sans-serif;  
  color: #333;  
}
```

3. ID Selector:

Style a section with the `id="hero"`.

```
#hero {  
  background-color: #f5f5f5;  
  padding: 50px;  
}
```

4. Class Selector:

Create a class `.highlight` and use it to highlight important text.

```
.highlight {  
  background-color: yellow;  
  font-weight: bold;  
}
```

5. Tag Selector:

Change the color of all paragraph text.

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

6. Descendant Selector:

Style paragraphs within `.content` divs.

```
.content p {  
  line-height: 1.8;  
}
```

7. Child Selector:

Apply a margin to direct child `div` elements within `.wrapper`.

```
.wrapper > div {  
  margin-bottom: 20px;  
}
```

CSS Specificity Flow Notes

Definition:

CSS specificity determines which style rule is applied to an element when multiple rules could apply. It is a system that calculates the weight of different selectors to resolve conflicts.

1. Specificity Hierarchy (Highest to Lowest)

1. **Inline Styles** (Highest) → `style` attribute
 - **Example:** `<p style="color: red;">Text</p>`
 - **Specificity:** 1000
2. **ID Selector** → `#id`
 - **Example:** `#header { color: blue; }`
 - **Specificity:** 0100
3. **Class, Attribute, and Pseudo-Class Selectors** → `.class`, `[type="text"]`, `:hover`
 - **Example:** `.btn { color: green; }`
 - **Specificity:** 0010

4. **Element and Pseudo-Element Selectors** → `div, h1, ::before`
 - **Example:** `p { font-size: 16px; }`
 - **Specificity:** 0001
5. **Universal Selector and Inherited Styles** → `*`, inherited properties
 - **Example:** `* { margin: 0; }`
 - **Specificity:** 0000

2. Specificity Calculation

Specificity is calculated using a 4-part value: (a, b, c, d)

- **a** → Inline styles
- **b** → ID selectors
- **c** → Class, attribute, and pseudo-class selectors
- **d** → Element and pseudo-element selectors

Example Calculation:

CSS

CopyEdit

```
/* Selector: #header .nav li a:hover */
#header .nav li a:hover
```

- **ID Selector (#header)** → **b = 1**
- **Class Selector (.nav)** → **c = 1**
- **Element Selectors (li, a)** → **d = 2**
- **Specificity:** 0,1,1,2

3. Specificity Examples

i). Inline vs. ID Selector

```
<p id="text" style="color: red;">Hello</p>
```

```
#text {
  color: blue;
}
```

Inline style wins → color: red;

ii). ID vs. Class Selector

```
#main {  
  color: blue;  
}
```

```
.content {  
  color: green;  
}
```

```
<div id="main" class="content">Text</div>
```

ID wins → color: blue;

iii). Class vs. Element Selector

```
p {  
  color: black;  
}
```

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

```
<p class="highlight">Important Text</p>
```

Class wins → color: yellow;

4. Important Rule **!important**

- Overrides all specificity.
- Used cautiously as it can make debugging harder.

Example:

```
p {  
  color: blue !important;  
}  
p {  
  color: red;  
}
```

Result: color: blue;

5. Specificity Best Practices

1. **Avoid Overusing ID Selectors** – Use class selectors for reusability.
2. **Minimize !important Usage** – It can create maintenance issues.
3. **Use Simple and Clear Selectors** – Improves readability and maintainability.
4. **Be Mindful of Nesting** – Deep nesting increases specificity.
5. **Follow a Consistent Naming Convention** – E.g., BEM (Block Element Modifier).

6. Specificity Cheatsheet

Selector Type	Specificity Value
Inline Styles (<code>style=""</code>)	1000
ID Selector (<code>#id</code>)	0100
Class Selector (<code>.class</code>)	0010
Attribute Selector (<code>[type]</code>)	0010
Pseudo-class (<code>:hover</code>)	0010
Element Selector (<code>div</code> , <code>h1</code>)	0001
Pseudo-element (<code>::before</code>)	0001
Universal Selector (<code>*</code>)	0000
Inherited Styles	0000

CSS Flexbox Concept Notes

1. Display

Definition:

The `display` property with the value `flex` enables Flexbox on a container, allowing its child elements to be laid out according to Flexbox rules.

Syntax:

```
display: flex;
```

Examples:

1. Basic Flex Container:

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

2. Inline Flex Container:

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

Use Case:

- Creates flexible and responsive layouts without floats or positioning.

2. Flex Direction

Definition:

The `flex-direction` property specifies the direction of the flex items within the container.

Syntax:

`flex-direction: row | row-reverse | column | column-reverse;`

Values and Examples:

1. Row (Default): Left to right in LTR languages.

```
.container {  
  flex-direction: row;  
}
```

2. Column: Top to bottom.

```
.container {  
  flex-direction: column;  
}
```

3. Row-Reverse: Right to left.

```
.container {  
  flex-direction: row-reverse;  
}
```

4. Column-Reverse: Bottom to top.

```
.container {  
  flex-direction: column-reverse;  
}
```

Use Case:

- Adjusts the orientation of items in a flex container.

3. Flex Wrap

Definition:

The `flex-wrap` property controls whether the flex items should wrap onto multiple lines.

Syntax:

```
flex-wrap: nowrap | wrap | wrap-reverse;
```

Examples:

1. No Wrap (Default):

```
.container {  
  flex-wrap: nowrap;  
}
```

2. Wrap Items to Next Line:

```
.container {  
  flex-wrap: wrap;  
}
```

3. Reverse Wrap:

```
.container {  
  flex-wrap: wrap-reverse;  
}
```

Use Case:

- Allows content to adjust dynamically to different screen sizes.

4. Justify Content

Definition:

The `justify-content` property aligns items along the main axis (horizontal in row direction).

Syntax:

```
justify-content: flex-start | flex-end | center | space-between |  
space-around | space-evenly;
```

Values and Examples:

1. Flex-Start (Default): Items align to the left.

```
.container {  
  justify-content: flex-start;  
}
```

2. Center: Items are centered.

```
.container {  
  justify-content: center;  
}
```

3. Flex-End: Items align to the right.

```
.container {  
  justify-content: flex-end;  
}
```

4. Space-Between: Equal space between items.

```
.container {  
  justify-content: space-between;  
}
```

5. Space-Around: Equal space around items.

```
.container {  
  justify-content: space-around;  
}
```

6. Space-Evenly: Equal space between all items.

```
.container {  
  justify-content: space-evenly;  
}
```

Use Case:

- Controls the horizontal alignment of flex items.

5. Align Items

Definition:

The `align-items` property aligns items along the cross-axis (vertical in row direction).

Syntax:

```
align-items: flex-start | flex-end | center | baseline | stretch;
```

Values and Examples:

1. **Flex-Start:** Align items to the top.

```
.container {  
  align-items: flex-start;  
}
```

2. **Center:** Vertically center items.

```
.container {  
  align-items: center;  
}
```

3. **Flex-End:** Align items to the bottom.

```
.container {  
  align-items: flex-end;  
}
```

Use Case:

- Aligns items vertically in a row or horizontally in a column.

6. Order

Definition:

The **order** property defines the order of flex items.

Syntax:

```
order: number;
```

Examples:

1. Reordering Items:

```
.item1 {  
  order: 2;  
}  
.item2 {  
  order: 1;  
}
```

Use Case:

- Rearranges items without changing the HTML structure.

7. Flex-Basis

Definition:

The `flex-basis` property sets the initial size of a flex item before it grows or shrinks.

Syntax:

```
flex-basis: auto | length | percentage;
```

Examples:

1. Fixed Item Size:

```
.item {  
  flex-basis: 200px;  
}
```

2. Percentage-Based Size:

```
.item {  
  flex-basis: 50%;  
}
```

Use Case:

- Defines the starting size of items in a flexible layout.

CSS Grid Concept

1. Display

Definition:

The `display` property with the value `grid` enables Grid Layout on a container, allowing child elements to be organized into rows and columns.

Syntax:

```
display: grid;
```

Examples:

1. Basic Grid Container:

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

2. Inline Grid Container:

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

Use Case:

- Enables two-dimensional layout control for rows and columns.

2. Grid Template Columns

Definition:

Defines the number and size of columns in the grid.

Syntax:

```
grid-template-columns: value value ...;
```

Examples:

1. Fixed Column Sizes:

```
.container {  
  grid-template-columns: 100px 200px;  
}
```

2. Fractional Units:

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

Use Case:

- Defines responsive column sizing.

3. Grid Template Rows

Definition:

Defines the number and size of rows in the grid.

Syntax:

```
grid-template-rows: value value ...;
```

Examples:

1. Fixed Row Sizes:

```
.container {  
  grid-template-rows: 50px 100px;  
}
```

2. Fractional Units:

```
.container {  
  grid-template-rows: 1fr 3fr;  
}
```

Use Case:

- Controls vertical space distribution.

4. Gap

Definition:

The `gap` property sets the space between grid items.

Syntax:

```
gap: row-gap column-gap;
```

Examples:

1. Uniform Gap:

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

2. Different Row and Column Gaps:

```
.container {  
  gap: 10px 20px;  
}
```

Use Case:

- Adds spacing between items without margins.

5. Grid Lines

Definition:

Grid lines are the lines that divide the grid into columns and rows.

Use Case:

- Used for precise placement of items.

6. Grid Column

Definition:

Specifies how many columns an item spans.

Syntax:

```
grid-column: start / end;
```

Examples:

1. Span Across Columns:

```
.item {  
  grid-column: 1 / 3;  
}
```

Use Case:

- Expands items horizontally.

7. Grid Row

Definition:

Specifies how many rows an item spans.

Syntax:

```
grid-row: start / end;
```

Examples:

1. Span Across Rows:

```
.item {  
  grid-row: 2 / 4;  
}
```

Use Case:

- Expands items vertically.

CSS Position

1. Static

Definition:

The default positioning of HTML elements. Elements appear in the normal document flow without any special positioning.

Syntax:

```
position: static;
```

Examples:

1. Default Behavior:

```
.box {  
  position: static;  
}
```

2. Implicit Static Position:

```
.box {  
  /* No position set; behaves as static */  
}
```

Use Case:

- For elements that don't need special positioning.

2. Relative

Definition:

Positions an element relative to its normal position without affecting other elements.

Syntax:

```
position: relative;
top: value;
left: value;
```

Examples:**1. Move Down:**

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

2. Shift Right:

```
.box {
  position: relative;
  left: 30px;
}
```

Use Case:

- Adjust placement slightly while keeping space in the flow.

3. Absolute**Definition:**

Removes an element from the document flow and positions it relative to its nearest positioned ancestor.

Syntax:

```
position: absolute;
top: value;
left: value;
```

Examples:

1. Position to Parent:

```
.parent {  
  position: relative;  
}  
.child {  
  position: absolute;  
  top: 0;  
  left: 0;  
}
```

2. Fixed Placement:

```
.box {  
  position: absolute;  
  right: 20px;  
  bottom: 10px;  
}
```

Use Case:

- For overlays or dropdowns needing exact placement.

4. Fixed

Definition:

Positions an element relative to the viewport, staying fixed during scrolling.

Syntax:

```
position: fixed;  
top: value;  
left: value;
```

Examples:

1. Sticky Navbar:

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

2. Fixed Button:

```
.button {  
  position: fixed;  
  bottom: 20px;  
  right: 20px;  
}
```

Use Case:

- For persistent elements like headers or floating buttons.

5. Sticky

Definition:

Combines **relative** and **fixed**. The element scrolls with the page but sticks when reaching a defined position.

Syntax:

```
position: sticky;  
top: value;
```

Examples:

1. Sticky Header:

```
.header {  
  position: sticky;  
  top: 0;  
  background-color: white;  
}
```

2. Sticky Sidebar:

```
.sidebar {  
  position: sticky;  
  top: 50px;  
}
```

Use Case:

- For headers or menus that should stay visible while scrolling.

CSS Media Queries

1. Components of Media Queries

Media queries consist of media types, media features, and logical operators to apply styles based on different screen conditions.

2. Media Types

Definition:

Media types define the type of device the styles should apply to.

Types and Syntax:

1. **All:** Matches all devices.

```
@media all {  
  body {  
    font-size: 16px;  
  }  
}
```

2. **Screen:** Targets devices with screens (e.g., desktop, mobile).

```
@media screen {  
  body {  
    background-color: lightgray;  
  }  
}
```

3. Print: Used for print preview and physical printing.

```
@media print {  
  body {  
    font-size: 12px;  
  }  
}
```

Use Case:

- Adapt styles for different devices like screens and printers.

3. Media Features

Definition:

Media features target specific characteristics of the device.

Common Features and Syntax:

1. Width:

```
@media (width: 600px) {  
  div {  
    display: none;  
  }  
}
```

2. Height:

```
@media (height: 800px) {  
  p {  
    font-size: 18px;  
  }  
}
```

3. Max-width:

```
@media (max-width: 768px) {  
  .container {  
    flex-direction: column;  
  }  
}
```

4. Max-height:

```
@media (max-height: 500px) {  
  img {  
    display: none;  
  }  
}
```

5. Min-width:

```
@media (min-width: 1024px) {  
  .sidebar {  
    display: block;  
  }  
}
```

6. Min-height:

```
@media (min-height: 600px) {  
  footer {  
    position: fixed;  
  }  
}
```

7. Orientation:

```
@media (orientation: landscape) {  
  body {  
    background-color: blue;  
  }  
}
```

Use Case:

- To adjust layouts for different screen sizes and orientations.

4. Logical Operators

Definition:

Logical operators allow combining multiple conditions in media queries.

Operators and Syntax:

1. And: Combines multiple conditions.

```
@media screen and (min-width: 600px) and (max-width: 1200px) {  
  body {  
    font-size: 18px;  
  }  
}
```

2. Not: Excludes a specific condition.

```
@media not screen and (max-width: 600px) {  
  body {  
    font-size: 20px;  
  }  
}
```

3. Only: Targets specific media types with stricter conditions.

```
@media only screen and (min-width: 1024px) {  
  nav {  
    display: flex;  
  }  
}
```

4. Or (,): Provides multiple alternative conditions.

```
@media (max-width: 768px), (orientation: portrait) {  
  .menu {  
    display: none;  
  }  
}
```

Use Case:

- Combining conditions for fine-grained control over styles.

CSS Backgrounds

1. Background-Color

Definition: Sets the background color of an element.

Syntax:

```
selector {  
    background-color: value;  
}
```

Examples:

1. Set background to blue:

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

2. Semi-transparent red:

```
div {  
    background-color: rgba(255, 0, 0, 0.5);  
}
```

2. Background-Image

Definition: Specifies an image to be used as the background of an element.

Syntax:

```
selector {  
    background-image: url('image-path');  
}
```

Examples:

1. Set a local background image:

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

2. Set an external image:

```
div {  
    background-image: url('https://example.com/bg.png');  
}
```

3. Background-Repeat

Definition: Specifies if and how a background image is repeated.

Syntax:

```
selector {  
    background-repeat: value;  
}
```

Examples:

1. No repeat:

```
div {  
    background-repeat: no-repeat;  
}
```

2. Repeat horizontally:

```
div {  
    background-repeat: repeat-x;  
}
```

4. Background-Position

Definition: Specifies the starting position of a background image.

Syntax:

```
selector {  
    background-position: value;  
}
```

Examples:

1. Position at top right:

```
div {  
  background-position: top right;  
}
```

2. Centered position:

```
div {  
  background-position: 50% 50%;  
}
```

5. Background

Definition: A shorthand property for setting multiple background properties at once.

Syntax:

```
selector {  
  background: color image repeat position/size;  
}
```

Examples:

1. Multiple background properties:

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

2. Simple solid background:

```
div {  
  background: #ffcc00;  
}
```

CSS Colors

1. Color Name

Definition: Specifies a predefined color name for an element.

Syntax:

```
selector {  
    color: color-name;  
}
```

Examples:

1. Using a named color:

```
p {  
    color: red;  
}
```

2. Setting background color:

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

2. RGB

Definition: Defines a color using the Red, Green, and Blue components.

Syntax:

```
selector {  
    color: rgb(red, green, blue);  
}
```

Examples:

1. Pure red color:

```
h1 {  
  color: rgb(255, 0, 0);  
}
```

2. Gray color:

```
div {  
  color: rgb(128, 128, 128);  
}
```

3. RGBA

Definition: Similar to RGB but includes an alpha (opacity) value.

Syntax:

```
selector {  
  color: rgba(red, green, blue, alpha);  
}
```

Examples:

1. Semi-transparent red:

```
p {  
  color: rgba(255, 0, 0, 0.5);  
}
```

2. Fully opaque blue:

```
div {  
  color: rgba(0, 0, 255, 1);  
}
```

4. HEX

Definition: Defines a color using hexadecimal notation.

Syntax:

```
selector {  
    color: #RRGGBB;  
}
```

Examples:

1. Black color:

```
h2 {  
    color: #000000;  
}
```

2. White color:

```
span {  
    color: #FFFFFF;  
}
```

CSS Transform

1. Translate

Definition: Moves an element from its current position.

Syntax:

```
selector {  
    transform: translate(x, y);  
}
```

Examples:

1. Move element 50px to the right and 20px down:

```
div {  
  transform: translate(50px, 20px);  
}
```

2. Move element 100px left:

```
div {  
  transform: translate(-100px, 0);  
}
```

2. Rotate

Definition: Rotates an element clockwise or counterclockwise.

Syntax:

```
selector {  
  transform: rotate(angle);  
}
```

Examples:

1. Rotate 45 degrees clockwise:

```
img {  
  transform: rotate(45deg);  
}
```

2. Rotate 90 degrees counterclockwise:

```
div {  
  transform: rotate(-90deg);  
}
```

3. Scale

Definition: Scales an element by increasing or decreasing its size.

Syntax:

```
selector {  
    transform: scale(x, y);  
}
```

Examples:

1. Double the size of the element:

```
div {  
    transform: scale(2, 2);  
}
```

2. Scale width to 50% and height to 150%:

```
div {  
    transform: scale(0.5, 1.5);  
}
```

CSS Transition

1. Transition Delay

Definition: Specifies a delay before the transition effect starts.

Syntax:

```
selector {  
    transition-delay: time;  
}
```

Examples:

1. Delay transition by 2 seconds:

```
div {  
    transition-delay: 2s;  
}
```

2. No delay:

```
button {  
    transition-delay: 0s;  
}
```

2. Transition Duration

Definition: Specifies how long the transition effect takes to complete.

Syntax:

```
selector {  
    transition-duration: time;  
}
```

Examples:

1. Transition duration of 1 second:

```
div {  
    transition-duration: 1s;  
}
```

2. Transition duration of 500 milliseconds:

```
div {  
    transition-duration: 500ms;  
}
```

3. Transition Property

Definition: Specifies the CSS property to which the transition effect is applied.

Syntax:

```
selector {  
    transition-property: property-name;  
}
```

Examples:

1. Apply transition to background-color:

```
div {  
  transition-property: background-color;  
}
```

2. Apply transition to all properties:

```
div {  
  transition-property: all;  
}
```