**HTML Basics**

**HTML**: HTML is the standard markup language for Web pages.

**HTML** alone is not sufficient for a web developer because HTML only defines the structure of the data that will be rendered on the **browser** in a webpage, to make it visually appealing and to make it functional, we will need to use **CSS** and **JavaScript** as well.

HTML itself is not a programming language; it's a markup language. It provides the structure and content of a webpage, but it doesn't have the ability to perform computations or logic like a programming language does.

However, HTML is often used in conjunction with other languages to create dynamic and interactive webpages. The primary languages that work together with HTML in web development are:

1. **CSS (Cascading Style Sheets)**: CSS is used to style the HTML elements, controlling their appearance, layout, and formatting. It defines how elements should be displayed on the screen or other media types.

1. **JavaScript**: JavaScript is a programming language that adds interactivity and dynamic behavior to webpages. It can manipulate the HTML content, respond to user actions, fetch data from servers, and update the webpage without requiring a page reload.

While HTML defines the structure and content of a webpage, CSS defines its presentation, and JavaScript adds functionality and behavior. Together, these three languages form the foundation of modern web development.

**HTML Syntax**

****

**Tags, Attributes and Elements:**

1. **Tags**:
   * Tags are the building blocks of HTML markup.
   * They define the structure and content of an HTML document.
   * Tags are enclosed in angle brackets **< >**.
   * Tags can be either opening tags **<tag>** or closing tags **</tag>**.
   * Opening tags denote the beginning of an element, while closing tags denote the end of an element.
   * Examples of tags: **<h1>**, **<p>**, **<div>**, **<img>**, **<a>**, **<input>**, **<table>**, etc.

**What is HTML?**

HTML is a language for describing Web pages.  
   HTML stands for HyperText Markup Language.  
   HTML is not a programming language, it is a markup language.  
   A markup language is a collection of markup tags.  
   HTML uses markup tags to describe Web pages.

**What are Tags?**  
   HTML markup tags are usually called HTML tags or just tags.  
   HTML tags are keywords surrounded by angle brackets like <html>.  
   HTML tags normally come in pairs, like <b> and </b>.  
   The first tag in a pair is the start tag; the second tag is the end tag.  
   Start and end tags are also called opening tags and closing tags.

**HTML Documents = Web Pages**  
   HTML documents describe Web pages.  
   HTML documents contain HTML tags and plain text.  
   HTML documents are also called Web pages.

The purpose of a Web browser (like Chrome or Mozilla Firefox) is to read  
HTML documents and display them as Web pages. The browser does not display  
the HTML tags, but uses the tags to interpret the content of the page.

**Sample skeleton**  
<html>  
    <head>  
    </head>  
    <body>  
        <h1>My First Heading</h1>  
        <p>My first paragraph</p>  
    </body>

**HTML Images**

**Image tag**

The HTML **<img> tag**is used to embed an image in a web page.

The <img> tag creates a holding space for the referenced image.

The <img> tag is a void tag, it contains attributes only, and does not have a closing tag.

The <img> tag has two required attributes:

* src - The required src attribute specifies the path (URL) to the image.
* alt - The required alt attribute provides an alternate text for an image.

You can resize the images using

1. style attribute to specify the width and height of an image.
2. width and height attributes

**Image Maps (Image tag + map)**

The HTML <map> tag defines an image map. An image map is an image with clickable areas. The areas are defined with one or more <area> tags.

usemap attribute to use to crate clickable areas

The **<map> element**is used to create an image map, and is linked to the image by using the required **name/id**attribute  
**The Areas**

Then, add the clickable areas.

A clickable area is defined using an <area> element.

**Shape**

You must define the shape of the clickable area, and you can choose one of these values:

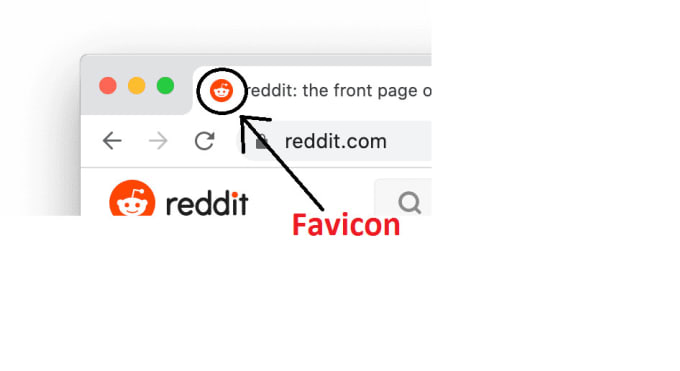
* rect - defines a rectangular region
* circle - defines a circular region
* poly - defines a polygonal region
* default - defines the entire region

**Background Images (any container with Style attribute)**

To add a background image on an HTML element, use the HTML **style attribute**and the CSS background-image property

**Favicon**

To add a favicon to your website, use Link and type icon and provide the path of the file in href, A common name for a favicon image is "favicon.ico".



</html>

**Forms**

HTML forms are used to collect different kinds of user input. A form is an area that  
can contain form elements.  
Form elements are elements that allow the user to enter information in a form (like  
text fields, text area fields, drop-down menus, radio buttons, check boxes, and so  
on).

**A form is defined with the <form> tag:**  
<form>  
.  
input elements  
.  
</form>

**input Tag and Attributes**  
The most used form tag is the <input> tag. The type of input is specified with  
the type attribute.   
**The following types are the most commonly used input types.**  
    <input type="button">  
    <input type="checkbox">  
    <input type="color">  
    <input type="date">  
    <input type="datetime-local">  
    <input type="email">  
    <input type="file">  
    <input type="hidden">  
    <input type="image">  
    <input type="month">  
    <input type="number">  
    <input type="password">  
    <input type="radio">  
    <input type="range">  
    <input type="reset">  
    <input type="search">  
    <input type="submit">  
    <input type="tel">  
    <input type="text">  
    <input type="time">  
    <input type="url">  
    <input type="week">

**Semantic Elements:**

* + Semantic elements are HTML tags that convey meaningful information about the content they contain to both the browser and the developer.
  + They describe the purpose or meaning of the content they enclose, making the code more descriptive, understandable, and accessible.
  + Examples of semantic elements include **<header>**, **<nav>**, **<section>**, **<article>**, **<aside>**, **<footer>**, **<main>**, **<figure>**, **<figcaption>**, etc.
  + These elements are designed to be used based on the meaning and structure of the content, rather than for purely stylistic or layout purposes.

Using semantic elements wherever appropriate is considered a best practice in web development because it improves the accessibility, search engine optimization (SEO), and maintainability of the code. It also helps assistive technologies like screen readers to better understand and navigate the content of a webpage.

**Void Elements (Self-Closing Tags):**

* + Void elements, also known as self-closing tags, are HTML elements that do not have a closing tag. They stand alone and do not contain any content.
  + Void elements are usually used to insert or embed something into the document, such as images, line breaks, or input fields.
  + Examples of void elements include **<img>**, **<br>**, **<input>**, **<meta>**, **<link>**, etc.

**Block-level elements**:

·       These elements typically start on a new line and take up the full width available to them.

·       Examples of block-level elements include **<div>**, **<p>**, **<h1>** to **<h6>**, **<ul>**, **<ol>**, **<li>**, **<table>**, **<form>**, **<header>**, **<footer>**, **<section>**, **<article>**, **<nav>**, etc.

·       Block-level elements can contain other block-level and inline-level elements.

·       Examples of styling attributes that can be applied to block-level elements include **width**, **height**, **margin**, **padding**, **border**, etc.

**Inline-level elements**:

·       These elements do not start on a new line and only take up as much width as necessary.

·       Examples of inline-level elements include **<span>**, **<a>**, **<strong>**, **<em>**, **<img>**, **<input>**, **<abbr>**, **<cite>**, **<code>**, **<label>**, **<small>**, **<sub>**, **<sup>**, etc.

·       Inline-level elements typically flow within the content of block-level elements or other inline-level elements.

·       Examples of styling attributes that can be applied to inline-level elements include **font-size**, **font-style**, **font-weight**, **color**, **text-decoration**, etc.

**Difference Between div and span Tag in HTML**

Below is a table showcasing major differences between div and span in HTML.

|  |  |  |
| --- | --- | --- |
| **Feature** | **div** | **span** |
| **Type of Element** | Block-level | Inline |
| **Default Display** | Takes up the full width available, with a new line before and after | Only takes up as much width as necessary, without forcing new lines |
| **Primary Use** | Used to group larger blocks of content or other elements | Used to group a small chunk of HTML elements or to apply styling to part of a text |
| **Styling** | It can have width, height, margin, and padding, which affects the layout significantly. | Does not affect the layout with width and height; margin and padding are applied differently. |
| **Impact on Layout** | Significant, as it often creates a "box" for other elements | Minimal, typically used for styling text or small elements within text |
| **Nesting** | It can contain other block-level elements or inline elements | Typically contains only data or other inline elements |
| **Accessibility** | Not inherently accessible and requires additional attributes for accessibility. | Inherently inline but also requires additional attributes for accessibility when used for grouping |

|  |  |
| --- | --- |
| It accepts align attribute. | It does not accept align attribute. |
| This tag should be used to wrap a section, for highlighting that section. | This tag should be used to wrap any specific word that you want to highlight in your webpage. |

**Attributes**

Attributes are used along with the HTML tags to define the characteristics of the element.

**For id, class, style, and data- attributes**:

* These attributes are commonly used with various HTML elements to provide styling and structure to the document. They are not specific to any single tag.

**For src, alt, width, height, and hidden attributes**:

* These attributes are commonly associated with the **<img>** tag for embedding images.

**For href, target, rel, and type attributes**:

* These attributes are commonly associated with the **<a>** tag for creating hyperlinks.

**For disabled, readonly, placeholder, maxlength, min, max, required, and checked attributes**:

* These attributes are commonly associated with various input elements like **<input>**, **<textarea>**, and **<select>** for form controls.

**For title attribute**:

* It can be used with various HTML elements to provide additional information, often displayed as a tooltip.

**Entities**

In HTML some characters are reserved like ‘<’, ‘>’, ‘/’, etc. To use these characters in our webpage we need to use the character entities called HTML Entities.

1.      <: Less than sign (<)

2.     >: Greater than sign (>)

3.     &: Ampersand (&)

4.     ": Double quotation mark (")

5.     ' or ': Single quotation mark (')

6.     : Non-breaking space ( )

7.     ©: Copyright symbol (©)

8.    ®: Registered trademark symbol (®)

9.    ™: Trademark symbol (™)

10.  €: Euro symbol (€)

11.    £: Pound sterling symbol (£)

12.   ¥: Yen symbol (¥)

13.   §: Section symbol (§)

14.  °: Degree symbol (°)

15.   ×: Multiplication sign (×)

16.  ÷: Division sign (÷)

17.   –: En dash (–)

18.  —: Em dash (—)

19.  « and »: Left and right angle quotation marks (« and »)

20.  …: Horizontal ellipsis (…)

**CSS**

CSS is a styling language for web development.

It separates content (HTML) from presentation, enhancing maintainability.

Selectors target HTML elements, while declarations define their styles.

CSS enables responsive design through media queries.

It supports animations and transitions for interactive web experiences.

**BOX Model**

The box model is a fundamental concept in web design and development that describes how elements on a webpage are structured and spaced out. It's crucial for understanding how elements like divs, paragraphs, images, etc., are rendered within the layout of a webpage.

The box model conceptualizes each HTML element as a rectangular box. This box consists of four main parts:

1.   **Content**: This is the actual content of the HTML element, such as text, images, etc. It's contained within the padding and border areas.

2.   **Padding**: Padding is the space between the content of the element and its border. It provides spacing between the content and the border, essentially giving it some breathing room.

3.   **Border**: The border surrounds the padding and content of the element. It defines the outer edge of the element. Borders can have various styles, colors, and widths.

4.   **Margin**: The margin is the space outside the border of the element. It separates the element from other elements on the page. Margins are used to create spacing between elements.

When you set the width and height of an element in CSS, you're actually setting the width and height of the content area. The padding, border, and margin are added to these dimensions.

For example, if you set the width of a div to 200 pixels and it has 10 pixels of padding and a 1-pixel border, the actual width of the div on the webpage will be 222 pixels (200 pixels for content + 10 pixels for padding on each side + 1 pixel for the left border and 1 pixel for the right border).

In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: content, padding, borders and margins. The image below illustrates the box model:

Explanation of the different parts:

* **Content** - The content of the box, where text and images appear
* **Padding** - Clears an area around the content. The padding is transparent
* **Border** - A border that goes around the padding and content
* **Margin** - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.

Example

Demonstration of the box model:

div {  
  width: 300px;  
  border: 15px solid green;  
  padding: 50px;  
  margin: 20px;  
}

**Key components of the box model**

**1. Content Area**

* The content area is the central part of the CSS box model, containing the [main content](https://www.geeksforgeeks.org/html-main-tag/) (e.g., text, images, videos, or elements like <p> or <span>).
* It can be styled with [CSS properties](https://www.geeksforgeeks.org/css-properties-complete-reference/)like height and width.

The content edge refers to the four edges of the content area

* Left content edge
* Right content edge
* Top content edge
* Bottom content edge

**2. Padding Area**

* The padding area is the space between the content and the border of an element.
* It includes the areas highlighted in light green and skin color in the example.
* The distance between the [content edge](https://www.geeksforgeeks.org/what-does-meta-http-equivx-ua-compatible-contentieedge-do/)and the border is the padding.
* The border marks the end of the padding area.
* The padding area contributes to the element's total dimensions.
* Padding can be adjusted using [CSS properties](https://www.geeksforgeeks.org/css-properties-complete-reference/).
* It works similarly with box-sizing: content-box and box-sizing: border-box, but with slight calculation differences.

**3. Border Area**

* The area that marks the end of an element is called as the[border](https://www.geeksforgeeks.org/css-borders/)it is the outer fencing for the element.
* The default border properties are provided in CSS to control the thickness of this outer fencing.
* The border area also add 's up to the complete[height and width](https://www.geeksforgeeks.org/how-to-set-the-width-and-height-of-an-image-using-html/)of the element.
* The more the border [width](https://www.geeksforgeeks.org/css-width-property/) the more will be the height or width of the element.
* In the above image the area marked with skin color is called the border area.

**4. Margin Area**

* The area outside the border of an element is called the [margin area](https://www.geeksforgeeks.org/css-box-model/).
* Basically this area depends on the parent of the element.
* The distance between the border of the parent element and the border of the child element is called as the margin.
* CSS has provides certain [margin properties](https://www.geeksforgeeks.org/css-margins-padding/)to get control over this scenario.

**Content-Box VSS Border-Box**

The main difference between the content-box and border-box models lies in how they calculate the total width and height of an element, including its content, padding, border, and margin.

1.   **Content-Box**:

·       In the content-box model, the width and height of an element are calculated based solely on the content area's dimensions. Padding, border, and margin are added to this width and height.

·       This means that if you set the width of an element to 200 pixels, that width applies only to the content area, and any padding, border, or margin will be added to the 200 pixels.

**1. Content-Box(default property)**

When the user set's the value of the [box-sizing property](https://www.geeksforgeeks.org/css-box-sizing-property/)for an element as[content-box](https://www.geeksforgeeks.org/how-is-border-box-different-from-content-box/)or even if user do not set's it ,it remains by default as content-box and in the actual height and width of the element the dimensions of the content area as well as the padding area is added to constitute the final dimensions of the element.

**↔**​

<style>

div {

height: 200px;

width: 200px;

box-sizing: content-box;

padding-left: 20px;

padding-right: 20px;

border-left: 2px solid red;

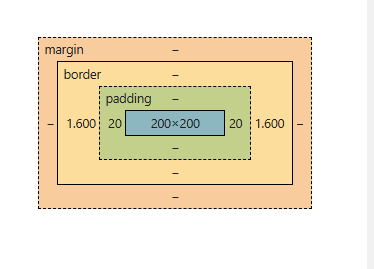
border-right: 2px solid red;

}

</style>

**↔**​

This code will create a box model with a[border line](https://www.geeksforgeeks.org/css-font-border/)width of 0.4px always and border-area of 1.6px and padding area as 20px width on both sides of the content area.



**Content Area (Width**) :The width of the content area is fixed at 200px.

**Padding**

* Padding adds extra space inside the element, around the content.
* Padding Left: 20px
* Padding Right: 20px
* Total padding width: 20px + 20px = 40px

**Border**

* The border, being solid, has a width, but it is calculated differently from the padding.
* Line Width of Border: 0.4px (the width of the line itself)
* Area of Border: 1.6px (the actual space the border occupies visually)
* Border width for both sides: 1.6px (left) + 1.6px (right) = 3.2px

**Total Width**

* Total width of the element can be calculated by adding the padding and border areas to the content area width.
* Formula for Total Width = (Padding-Left + Padding-Right + Border-Area-Left + Border-Area-Right) + Content Area Width
* Total Width = (20px + 20px + 1.6px + 1.6px) + 200px = 243.2px
* The total width of the element becomes 243.2px.

The reason the total width is increased unexpectedly is because[box-sizing: content-box](https://www.geeksforgeeks.org/css-box-sizing-property/)applies the width to the content area only .The padding and border are added outside the content area, leading to an increase in the overall [width and height](https://www.geeksforgeeks.org/html-width-height-attribute-vs-css-width-height-property/) of the element.

2.   **Border-Box**:

·       In the border-box model, the width and height of an element are calculated including the padding and border dimensions. The margin is still added outside of this total width and height.

·       This means that if you set the width of an element to 200 pixels in border-box mode, that 200 pixels includes the content area, padding, and border. The margin is then added outside of this total.

**2. Border-Box**

When the box-sizing property is set as [border-box](https://www.geeksforgeeks.org/what-is-the-use-of-box-sizing-property-in-css/) the actual dimensions of the element's remains same as that of the actual dimensions set by the user. The difference it makes is just that the size of the content area get's altered in a manner so that it could accommodate the[padding area](https://www.geeksforgeeks.org/css-padding/) and the[border area](https://www.geeksforgeeks.org/css-border-property/)so the resultant could be equal to the actual dimensions entered by the user.

**↔**​

<style>

div {

height: 200px;

width: 200px;

box-sizing:border-box;

padding-left: 20px;

padding-right: 20px;

border-left: 2px solid red;

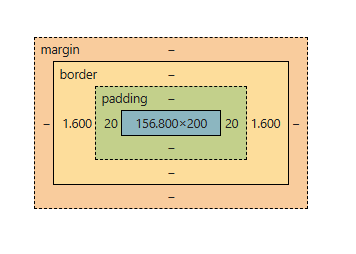
border-right: 2px solid red;

}

</style>

**↔**​

This code will create a box model by altering the dimensions specifically the width of the content area to accomodate the padding and the border area with the border line-width.



* **Width of Border and Padding Border width**: 0.4px (line width) and 1.6px + 1.6px = 3.2px (total border area).
* **Padding width**: 20px + 20px = 40px.
* **User-Entered Width** : The width entered by the user is 200px, which applies to the content area only when box-sizing: content-box is used.
* **Box-Sizing Behavior**:The box-sizing: content-box property adds the padding and border outside the content area, causing the total width to increase.
* **Adjusting Content Area Width**: To ensure the total width remains 200px, the extra width from padding and borders (40px + 3.2px = 43.2px) is subtracted from the total width.
* **New content area width** : 200px - 43.2px = 156.8px.
* **Final Width Calculation** : The final total width is: 156.8px (content area) + 40px (padding) + 3.2px (border) = 200px, ensuring the user’s entered width remains unchanged.

**CSS Units**

CSS units are used to specify the size, position, and other properties of elements in Cascading Style Sheets (CSS). They come in various types, each serving different purposes and providing flexibility in styling web content. Here's an overview of some common CSS units:

1.   **Absolute Length Units**:

·       **px (Pixels)**: A pixel is a single dot on a screen. It's a fixed-size unit that doesn't change with the resolution of the display.

·       **pt (Points)**: A point is a physical unit of measurement commonly used in print. It's approximately 1/72nd of an inch and is less common in web design.

·       **in (Inches)**, **cm (Centimeters)**, **mm (Millimeters)**: Physical units that represent inches, centimeters, and millimeters, respectively. They are less commonly used in web design but can be useful for printing stylesheets.

2.   **Relative Length Units**:

·       **em**: Relative to the font-size of the element itself. For example, if the font-size of an element is 16px, 1em is equal to 16px.

·       **rem**: Similar to em, but it's relative to the font-size of the root element (usually the <html> element), making it more predictable and easier to manage in complex layouts.

·       **% (Percentage)**: Represents a percentage of the parent element's size. For example, if the width of a parent container is 200px and you set a child element's width to 50%, it will be 100px wide.

3.   **Viewport Percentage Lengths**:

·       **vw (Viewport Width)**: Represents a percentage of the viewport's width. 1vw is equal to 1% of the viewport's width.

·       **vh (Viewport Height)**: Represents a percentage of the viewport's height. 1vh is equal to 1% of the viewport's height.

·       **vmin**: Represents the smaller of vw or vh.

·       **vmax**: Represents the larger of vw or vh.

4.   **Font Relative Length Units**:

·       **ex**: Represents the height of the lowercase letter 'x' in the current font. It's a relative unit often used in typography.

·       **ch**: Represents the width of the "0" (zero) character in the current font.

5.   **Others**:

·       **fr (Flexible Length)**: Represents a fraction of the available space in a flex container. It's used in CSS Grid layouts to define flexible widths or heights of grid items.

**CSS Colors**

In CSS, colors can be specified using various methods to achieve the desired visual appearance. Here are some common ways to define colors:

1.   **Keyword Colors**:

·       CSS provides a set of predefined color names such as red, blue, green, etc. These names represent specific colors.

·       Example: color: red;

2.   **Hexadecimal Colors**:

·       Hexadecimal (hex) notation represents colors using a six-digit combination of numbers and letters. Each pair of digits represents the intensity of red, green, and blue (RGB) respectively.

·       Example: color: #ff0000; (red), color: #00ff00; (green)

3.   **RGB Colors**:

·       RGB notation specifies colors using the intensity levels of red, green, and blue channels. Each channel value ranges from 0 to 255.

·       Example: color: rgb(255, 0, 0); (red), color: rgb(0, 255, 0); (green)

4.   **RGBA Colors**:

·       RGBA notation is similar to RGB but includes an additional alpha channel representing opacity. The alpha value ranges from 0 (fully transparent) to 1 (fully opaque).

·       Example: color: rgba(255, 0, 0, 0.5); (semi-transparent red)

5.   **HSL Colors**:

·       HSL (Hue, Saturation, Lightness) notation defines colors based on their hue, saturation, and lightness values.

·       Hue: Represents the color itself (0-360 degrees).

·       Saturation: Represents the intensity of the color (0-100%).

·       Lightness: Represents the brightness of the color (0-100%).

·       Example: color: hsl(0, 100%, 50%); (red)

6.   **HSLA Colors**:

·       Similar to HSL, but includes an alpha channel for opacity.

·       Example: color: hsla(0, 100%, 50%, 0.5); (semi-transparent red)

7.   **System Colors**:

·       CSS provides some predefined colors that are based on the user's system preferences, such as ButtonFace, Window, etc. However, these are not widely used due to limited support and inconsistency across platforms.

**Positions in CSS**

1. **Absolute Position**:
   * Elements with **position: absolute;** are positioned relative to their nearest positioned ancestor.
   * If there is no positioned ancestor, it's positioned relative to the initial containing block (which is usually the **<html>** element).
   * The element is taken out of the normal flow of the document, so it won't affect the position of other elements.
   * You can use properties like **top**, **bottom**, **left**, and **right** to specify the exact position of the element relative to its containing block.
2. **Relative Position**:
   * Elements with **position: relative;** are positioned relative to their normal position in the document flow.
   * Unlike absolute positioning, the element still occupies space in the document flow, so it will affect the position of other elements.
   * When you apply **top**, **bottom**, **left**, or **right** properties, they are relative to the element's normal position.
3. **Fixed Position**:
   * Elements with **position: fixed;** are positioned relative to the viewport, which means they always stay in the same place even if the page is scrolled.
   * Similar to absolute positioning, fixed-positioned elements are taken out of the normal flow of the document.
   * Useful for creating elements like headers or navigation bars that should always be visible at the top of the page.
4. **Static Position**:
   * This is the default position for all elements (**position: static;**).
   * Elements with static positioning are positioned according to the normal flow of the document.
   * They ignore properties like **top**, **bottom**, **left**, and **right**, as well as **z-index**.
   * Static-positioned elements are not affected by the **position** property. They just follow the natural order of elements in the HTML.

**5. Stikcy Position**:

* A sticky position in CSS is like a hybrid of relative and fixed positioning. When an element is set to "sticky", it behaves like a relatively positioned element until it reaches a specified scroll position, then it becomes fixed.
* Imagine you have a sticky note on a piece of paper. Initially, it moves with the paper like a relatively positioned element would. However, if you scroll down the paper, eventually the note reaches a point where it sticks in place, as if it's fixed to the paper. That's essentially how "position: sticky" works in CSS.
* It's commonly used for navigation bars that stick to the top of the viewport once you scroll past them, or for table headers that remain visible as you scroll through a long table.

**Z Index**

In CSS, the z-index property controls the stacking order of positioned elements. When elements overlap, the z-index determines which one appears on top of the others. Elements with a higher z-index value will be placed in front of elements with a lower z-index value.

Think of it like a stack of papers on a desk. Each paper represents an element on the webpage. The z-index value determines the order of these papers in the stack. Papers with a higher z-index are placed on top of papers with lower z-index values.

It's important to note that the z-index property only works on positioned elements (i.e., elements with a position value other than static, such as relative, absolute, or fixed). If an element isn't positioned, setting a z-index value will have no effect.

**CSS combinators**

CSS combinators are used to define relationships between different selectors in a stylesheet. They allow you to target elements based on their position in the HTML document relative to other elements. By using combinators, you can apply styles more precisely without needing to add extra classes or IDs. There are four main types of CSS combinators:

**1. Descendant Combinator (space)**

The descendant combinator selects elements that are descendants (children, grandchildren, etc.) of a specified element.

**Syntax:**

A B {

/\* styles \*/

}

* A: The ancestor element (the parent or higher up in the DOM tree).
* B: The descendant element (any element nested within A).

div p {

color: red;

}

This targets all <p> elements that are inside a <div>, no matter how deeply nested they are.

**2. Child Combinator (>)**

The child combinator selects elements that are **direct children** of a specified element, not just descendants.

**Syntax:**

A > B {

/\* styles \*/

}

* A: The parent element.
* B: The direct child element of A.

**Example:**

div > p {

color: blue;

}

This targets only <p> elements that are **directly inside** a <div>. It does not apply to <p> elements that are nested deeper inside other elements.

**3. Adjacent Sibling Combinator (+)**

The adjacent sibling combinator targets an element that immediately follows another element. Both elements must share the same parent.

**Syntax:**

A + B {

/\* styles \*/

}

* A: The first element.
* B: The immediately following sibling element.

**Example:**

h1 + p {

font-size: 18px;

}

This targets a <p> element that comes immediately after an <h1> element within the same parent.

**4. General Sibling Combinator (~)**

The general sibling combinator selects elements that are **siblings** of a specified element, meaning they share the same parent, but are not necessarily adjacent.

**Syntax:**

A ~ B {

/\* styles \*/

}

* A: The first element.
* B: Any sibling element that follows A (not necessarily immediately).

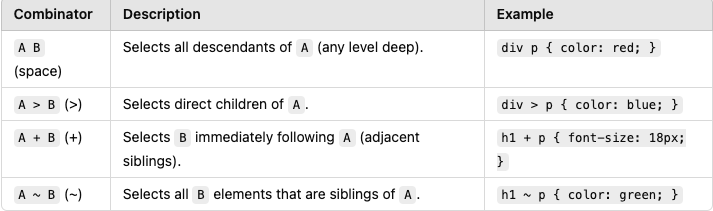
**Example:**

h1 ~ p {

color: green;

}

**This targets all <p> elements that are siblings of an <h1>, regardless of whether they are adjacent or not.**

****

**Pseudo-Selectors in CSS**

**Pseudo-selectors** in CSS are special keywords added to selectors that allow you to style specific parts of an element or an element’s state without having to add extra classes or IDs to the HTML. They enable you to apply styles to elements based on conditions such as user interaction, structural relationships in the HTML, or certain dynamic states of an element.

There are two main types of pseudo-selectors in CSS:

1. **Pseudo-classes**  
   These target elements based on their state or position in the document structure.
2. **Pseudo-elements**  
   These allow you to style specific parts of an element, such as before or after its content.

**1. Pseudo-Classes**

Pseudo-classes are used to define the special states of an element. They are preceded by a colon (:).

**Common Pseudo-Classes:**

* **:hover**  
  Applies styles when an element is hovered over by the mouse.

a:hover {

color: red;

}

* This will change the color of a link when the user hovers over it.
* **:active**  
  Applied when an element (usually a link or button) is in an active state, like being clicked.
* button:active {
* background-color: blue;
* }

**:focus**  
Applied when an element (such as an input field) gains focus, like when the user clicks into a text input or navigates with the keyboard.

input:focus {

border: 2px solid green;

}

**:first-child**  
Targets the first child element of a specified parent element.

p:first-child {

font-weight: bold;

}

**:nth-child()**  
Selects elements based on their position in a parent’s child list. It can take a number, keyword, or formula to match specific children.

li:nth-child(odd) {

background-color: lightgray;

}

* This will apply a background color to odd-numbered list items.
* **:not()**  
  Selects elements that do **not** match the given selector.

div:not(.special) {

background-color: gray;

}

* This will style all <div> elements except those with the class special.
* **:disabled**  
  Targets disabled form elements, such as buttons or inputs.

button:disabled {

background-color: lightgray;

}

**:checked**  
Applied to input elements like checkboxes or radio buttons when they are checked.

input:checked {

background-color: yellow;

}

**2. Pseudo-Elements**

Pseudo-elements allow you to style a specific part of an element, like its first letter, first line, or content added before or after the element.

**Common Pseudo-Elements:**

* **::before**  
  Inserts content before an element's actual content.
* This will insert the text Note: before the content of every <p> element.
* **::after**  
  Inserts content after an element's actual content.
* **::first-letter**  
  Targets the first letter of an element (usually used for styling a first letter in paragraphs).
* **::first-line**  
  Targets the first line of an element, which can be useful for styling the opening line of paragraphs.
* **::selection**  
  Applies styles to the portion of an element that is selected by the user (such as when text is highlighted).

p::before {

content: "Note: ";

font-weight: bold;

}

p::first-letter {

font-size: 200%;

color: red;

}

p::first-line {

font-weight: bold;

}

::selection {

background-color: yellow;

color: black;

}

By using pseudo-selectors, you can create more interactive, dynamic, and precise styling for your web pages without needing to modify your HTML structure or add extra classes and IDs

**CSS Dropdowns**

A **CSS dropdown** is a menu that appears when a user interacts with a specific element, such as hovering or clicking. Dropdowns are commonly used for navigation menus and are a key feature of web design for organizing and displaying content in a compact and user-friendly way.

**Key Concepts in CSS Dropdowns**

**1. Structure of a Basic Dropdown Menu**

A typical dropdown menu consists of two main parts:

* A **container** element (often a list item or div) that holds the dropdown.
* A **sub-menu** (another list or div) that is hidden by default but revealed when triggered.

<ul class="menu">

<li><a href="#">Home</a></li>

<li class="dropdown">

<a href="#">Services</a>

<ul class="submenu">

<li><a href="#">Web Design</a></li>

<li><a href="#">Web Development</a></li>

<li><a href="#">SEO</a></li>

</ul>

</li>

<li><a href="#">About</a></li>

<li><a href="#">Contact</a></li>

</ul>

In this example:

* The li with the class .dropdown is the main trigger for the dropdown.
* The submenu (.submenu) contains the items that will appear when the dropdown is activated.

**2. Basic CSS for Dropdown**

To hide and show the submenu, we use CSS properties like display and visibility, combined with :hover pseudo-class (or :focus in some cases for accessibility).

.menu {

list-style-type: none;

padding: 0;

}

.menu li {

display: inline-block;

position: relative;

}

/\* Hide the submenu by default \*/

.submenu {

display: none;

position: absolute;

top: 100%;

left: 0;

background-color: #333;

color: white;

list-style-type: none;

padding: 0;

margin: 0;

}

/\* Show submenu when hovering over the parent li \*/

.dropdown:hover .submenu {

display: block;

}

* The .submenu is hidden by default using display: none.
* When the parent .dropdown is hovered, the .submenu becomes visible (display: block).
* position: absolute positions the submenu relative to the parent.

**3. Hover-Activated Dropdown**

A simple way to activate the dropdown is by using the :hover pseudo-class. This method is common in desktop navigation bars.

/\* Hover over the parent to display the submenu \*/

.dropdown:hover .submenu {

display: block;

}

**Key Points for CSS Dropdowns**

* **Visibility**: You typically toggle visibility using display, opacity, or visibility combined with CSS transitions for smooth effects.
* **Positioning**: Dropdowns often rely on absolute positioning (position: absolute) for the submenu to appear correctly below or next to the parent item.
* **Accessibility**: For accessibility, ensure that dropdowns are navigable by keyboard (e.g., via tabindex) and consider adding ARIA attributes like aria-haspopup="true" and aria-expanded for screen readers.
* **Mobile Compatibility**: On mobile devices, dropdown menus may need additional considerations, like being activated by clicks or touch events, and sometimes requiring more space for legibility.

**Fonts in CSS:**

* CSS provides several properties for styling fonts.
* font-family specifies the font family for an element, allowing you to choose from various font options. font-size sets the size of the font.
* font-weight controls the boldness of the text. font-style allows you to make text italic or oblique. Additionally,
* font-variant, line-height, and text-transform are other properties that can be used to manipulate text appearance.

**Transition Properties:**

* CSS transitions allow you to smoothly change property values over a specified duration.
* The transition-property property determines which properties will transition.
* transition-duration sets the time taken for the transition.
* transition-timing-function specifies the acceleration curve of the transition.
* transition-delay sets a delay before the transition starts.

**Transformation Properties:**

* CSS transformations allow you to modify the appearance of elements in 2D or 3D space.
* transform is the main property used for this purpose.
* It can apply various transformations like translations (translate()), rotations (rotate()), scalings (scale()), skewing (skew()), and more.
* Additionally, transform-origin specifies the point around which transformations are applied, and transform-style controls how nested elements are rendered in 3D space.

**Responsive web design (RWD)**

Responsive web design (RWD) is an approach to web design that aims to create web pages that adjust and adapt their layout and content based on the size of the user's screen or viewport. The goal is to ensure optimal viewing and interaction experience across a wide range of devices, from desktop computers to smartphones and tablets.

Media queries are a key component of responsive web design. They are CSS3 modules that allow web developers to apply different styles to a web page based on various factors, such as screen size, device orientation, and resolution. Media queries enable developers to create rules that specify how the layout and presentation of a web page should change in response to different conditions.

For example, a media query might be used to adjust the layout of a webpage when viewed on a smaller screen, such as a smartphone. This could involve changing the size and positioning of elements, hiding certain content, or reorganizing the page's structure to ensure readability and usability on smaller devices.

**Specificity**

Specificity in the context of web development refers to the set of rules that determine which CSS styles are applied to an HTML element when multiple conflicting styles are present. It's essentially a measure of how specific a CSS rule is in targeting an element.

CSS rules can target elements in different ways: by element type, class, ID, or inline styles. When multiple CSS rules conflict in their attempts to style the same element, the browser uses specificity to decide which style to apply.

1. **Inline Styles**: Styles defined directly within an HTML element using the style attribute have the highest specificity. This means if you define a style directly in the HTML element like <div style="color: blue;">, that style will take precedence over any other styles applied to that element.
2. **ID Selectors**: Selectors targeting elements by their ID have higher specificity than other types of selectors. IDs are unique identifiers for elements, and CSS rules targeting IDs are more specific than those targeting classes or element types. For example, a style defined with #unique-id { color: red; } will override a style defined for the same element with a class or element type.
3. **Class Selectors, Attributes, and Pseudo-Classes**: Selectors targeting elements by class, attributes, or pseudo-classes have medium specificity. These selectors apply to multiple elements but are more specific than element type selectors. For example, a style defined with .my-class { color: green; } will override a style defined for the same element type.
4. **Element Type Selectors**: Selectors targeting elements by their HTML tag name have the lowest specificity. These selectors apply to all elements of that type on the page. For example, a style defined with p { color: black; } will be overridden by more specific styles targeting IDs, classes, or inline styles.

If there's a conflict between two or more styles with the same specificity, the style declared later in the CSS file or inline will take precedence. However, if there's a conflict between styles with different specificities, the browser applies the style with the higher specificity.

If there are two or more CSS rules that point to the same element, the selector with the highest specificity will "win", and its style declaration will be applied to that HTML element.

Think of specificity as a hierarchy that determines which style declaration is ultimately applied to an element.

Look at the following examples:

Example 1

Here, we have used the "p" element as selector, and specified a red color for this element. **Result:** The text will be red:

<html>  
<head>  
  <style>  
    p {color: red;}  
  </style>  
</head>  
<body>  
  
<p>Hello World!</p>  
  
</body>  
</html>

Example 2

Here, we have added a class selector (named "test"), and specified a green color for this class. **Result:** The text will be green (even though we have specified a red color for the element selector "p"). This is because the class selector has higher priority:

<html>  
<head>  
  <style>  
    .test {color: green;}  
    p {color: red;}  
  </style>  
</head>  
<body>  
  
<p class="test">Hello World!</p>  
  
</body>  
</html>

Specificity Hierarchy

Every CSS selector has a position in the specificity hierarchy.

|  |  |  |
| --- | --- | --- |
| Priority | Example | Description |
| Inline style | <h1 style="color: pink;"> | Highest priority, directly applied with the style attribute |
| Id selectors | #navbar | Second highest priority, identified by the unique id attribute of an element |
| Classes and pseudo-classes | .test, :hover | Third highest priority, targeted using class names |
| Attributes | [type="text"] | Low priority, applies to attributes |
| Elements and pseudo-elements | h1, ::before, ::after | Lowest priority, applies to HTML elements and pseudo-elements |

**Grid and Flexbox**

Grid and Flexbox (short for Flexible Box) are two modern CSS layout systems used for creating complex web layouts. They both aim to make it easier to design responsive and flexible web pages, but they have different approaches and use cases.

1. **Flexbox**:
   * Flexbox is a one-dimensional layout model, meaning it deals with layout in one direction at a time—either as a row or a column.
   * It's particularly useful for aligning items within a container along a single axis, distributing space among items, and controlling their alignment, order, and size.
   * Flexbox is great for creating navigation bars, vertical or horizontal lists, and centering elements both horizontally and vertically.
2. **Grid**:
   * CSS Grid is a two-dimensional layout system, allowing you to define both rows and columns in a grid-like structure.
   * It's ideal for creating complex layouts where you need precise control over both rows and columns simultaneously.
   * With Grid, you can easily create grid tracks (rows and columns), place items within specific grid areas, and control the sizing and alignment of those items.
   * It's well-suited for creating entire page layouts, such as headers, footers, sidebars, and content areas, and arranging elements in a more grid-like pattern.

In summary, Flexbox is best for simpler layouts along one axis, while CSS Grid is better suited for more complex layouts requiring control over both rows and columns. In practice, they are often used together to achieve more sophisticated designs, with Flexbox handling alignment and distribution within individual components, and Grid handling the overall layout structure.

**Difference Between CSS Grid and Flexbox**

| **Property** | **Grid** | **Flexbox** |
| --- | --- | --- |
| Dimension | Two - Dimensional | One - Dimensional |
| Features | Can flex combination of items through space-occupying Features | Can push content element to extreme alignment |
| Support Type | Layout First | Content First |
| Primary Use Case | Creating complex layouts with rows and columns | Aligning items in a row or column |
| Performance | Can be less in performance due to very complex grids | Generally faster for simple layouts |

**Bootstrap5 Tutorial**

**What is Bootstrap?**

* Bootstrap is an open-source front-end framework that allows developers to build websites quickly without having to write a lot of custom CSS and JavaScript from scratch.
* It includes a set of ready-made design patterns, components like buttons, forms, navigation bars, and utilities that you can use directly in your projects.
* **Mobile-first**: Bootstrap uses a mobile-first approach, meaning the framework is built with mobile devices in mind first, and then adapts for larger screens (tablets, desktops).

**Why Should We Use Bootstrap?**

1. **Saves Time**: You don’t need to reinvent the wheel for every project. Bootstrap provides ready-to-use components like forms, buttons, and navigation that are already optimized for responsiveness.
2. **Responsive Design**: This is a big one. Bootstrap 5 makes it easy to create layouts that adapt to different screen sizes automatically, which is crucial since most people browse the web on mobile devices today.
3. **Consistent UI**: With Bootstrap, you can ensure a consistent and clean user interface across your website or app. The components are styled in a standard way, so you don’t need to worry about the small details of styling each element.
4. **Customizable**: While Bootstrap provides default styles, you can customize them to fit your brand or project needs, whether it's changing the colors, fonts, or spacing.

**Main Features of Bootstrap 5**

1. **Grid System**:
   * The grid system in Bootstrap helps create layouts with rows and columns. It is built on a 12-column layout, meaning you can divide your page into 12 equal-width columns or use different combinations to build complex layouts.
   * This is crucial for making sure your site looks good on any screen size.
2. **Responsive Utilities**:
   * With classes like d-none, d-sm-block, and others, Bootstrap allows you to control the visibility and display of elements based on screen size.
   * For example, you can hide elements on mobile devices but show them on desktops.
3. **Pre-built Components**:
   * Bootstrap includes a wide variety of UI components that you can drop into your page. These include buttons, navigation bars, modals (pop-up windows), carousels (image sliders), forms, and much more.
4. **Custom Forms**:
   * Bootstrap 5 has a set of customizable form components. You can create text inputs, checkboxes, radio buttons, dropdowns, and other form elements with minimal effort.
   * The forms are also designed to be responsive, so they will adjust to different screen sizes automatically.
5. **Utilities**:
   * Bootstrap comes with many utility classes for spacing, alignment, colors, typography, borders, shadows, and more. These utilities help you tweak your design on the fly without writing custom CSS.

**How Do We Get Started with Bootstrap?**

* To use Bootstrap, you simply link to its CDN (Content Delivery Network) in your HTML file, or you can download it and include it in your project.

The CDN links and Bootstrap5 Official Documentation can be found here

<https://getbootstrap.com/docs/5.3/getting-started/introduction/>

* Once Bootstrap is included, you can start using its pre-built classes and components. For example, to create a button, you just use the class btn btn-primary, and Bootstrap takes care of the rest, like styling and responsiveness.

Example Code

<https://getbootstrap.com/docs/5.3/examples/>

**GIT**

**What is Git?:** Git is like a super-smart assistant for our code. It helps us keep track of every change we make to our project, kind of like how a historian keeps track of events in history. But instead of events, Git keeps track of lines of code!

**How Does Git Work?:** When we start a project, Git creates a special place called a repository, or repo for short. This is like a folder where Git stores all the versions of our code. Every time we make a change to our code and we're happy with it, we tell Git to save that change. We call this saving a commit. Think of a commit like saving a checkpoint in a video game.

**Why Do We Need Git?:** So, why do we need Git? Well, imagine you're working on a project and you accidentally delete something important. Instead of panicking, Git lets you go back to a previous version of your code, like rewinding time! Plus, if you're working on a project with other people, Git helps you all work together smoothly without stepping on each other's toes.

**Basic Git Commands:** Let's talk about some basic Git commands:

1. **git** **init**: Initialize a new repository.
2. **git** **add**: Stage changes for the next commit.
3. **git** **commit**: Commit staged changes.
4. **git push:** Push commits to the remote repository.
5. **git** **fetch**: Fetch updates from the remote without merging them.
6. **git pull**: Fetch updates from the remote and merge them into your branch.

**1. Adding a Remote Repository**

To add a remote repository (usually a URL pointing to a repository on a hosting service like GitHub, GitLab, or Bitbucket), you can use the following command:

git remote add <name> <url>

* <name>: The name you want to give to the remote (commonly origin for the main remote).
* <url>: The URL of the remote repository. This can be an HTTPS or SSH URL.

**2. Viewing the Current Remotes**

You can see the list of all remotes associated with the repository by using:

git remote -v

**3. Modifying a Remote Repository URL**

If you need to change the URL of an existing remote (e.g., if the repository URL changes or you want to switch from HTTPS to SSH), you can use:

git remote set-url origin git@github.com:username/repository.git