🔷 What is CSS?

- **CSS stands for** Cascading Style Sheets.
- CSS is used to **style web pages**—colors, layout, fonts, spacing, etc.

🔷 CSS Syntax Breakdown

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

- **Selector:** Which element(s) to style (e.g., div, span).
- **Property:** The style attribute (e.g., color, background-color).
- **Value:** What you want to apply (e.g., red, yellow).
- You can use **multiple selectors**:

```
div, span {
  color: red;
  background-color: yellow;
}
```

🟧 3 Ways to Add CSS to an HTML Page

1️⃣ Inline CSS

- CSS is written directly inside the **HTML element** using the style attribute.

```
<h1 style="color: yellow; background-color: red;">
     About Me
</h1>
```

- ✅ **Quick and simple** for testing.
- ❌ **Not scalable** – If used across many elements, updating becomes tedious.
- ❌ **Not recommended** for large or professional websites.

2️⃣ Internal CSS

- CSS is written **inside a <style> tag in the <head> section** of the same HTML file.

```
<head>
  <style>
    h1 {
      color: yellow;
      background-color: red;
    }
  </style>
</head>
```

- ✅ Good for **small or medium-sized pages**.
- ✅ You can **bulk edit** styles for the whole page.
- ❌ Not ideal for large projects — HTML file gets bulky.

3️⃣ External CSS

- CSS is written in a separate .css file, and linked to the HTML file using <link>.

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

- ✅ **Most recommended** and scalable approach.
- ✅ Clean separation of content (HTML) and design (CSS).
- ✅ Multiple developers can work in parallel (HTML + CSS separately).

CSS Selectors

🧱 1. Element Selector

- It is used to select an html element by its name.
- Targets all elements of a specific type.

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

🎯 2. Class Selector

- Targets elements with a **specific class** using a dot.

```
<div class="red">I am red</div>
```

```
.red {
  background-color: red;
}
```

🆔 3. ID Selector

- Targets elements with a **unique ID** using #.

```
<div id="green">I am green</div>
```

```
#green {
  background-color: green;
}
```

👶 4. Child Selector

- Selects **all elements** that are a **direct child** of a specific parent.

```
div > p {
  color: blue;
  background-color: brown;
}
```

- This applies styles to <p> only if it's a direct child of .

📌 5. Descendant Selector

- Targets all elements **nested inside** another, **regardless of depth**.

```
div p {
  color: blue;
}
```

- Applies even if <p> is inside other tags inside .

🌐 6. Universal Selector

- Targets **all elements**.

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

```
.boxes *{color: red;}
```
// applies to all the elements inside element having class=”boxes”

7.Group Selector

- It is used to apply the **same style** to **multiple elements**

```
div,p,a{ color: red;}
```

8. [attribute] Selector

- It is used to select elements with a specified attribute.

  ```
  a[target] {background-color: yellow; }
  ```

  selects all the anchor tags with a target attribute.

9. [attribute="value"] Selector

- It is used to select elements with a specified attribute and value.

  ```
  a[target="_blank"] { background-color: yellow; }
  ```

  selects all the anchor tags with a target attribute and value "_blank".

10. Adjacent Sibling Selector

- Selects the next immediate sibling of a specified element.

```
element1 + element2 {
  /* styles */
}
```

- Select element2 **only if it immediately follows** element1.

```
<h2>Heading</h2>
<p>Paragraph 1</p>
<p>Paragraph 2</p>
```

```
h2 + p {
  color: red;
}
```

✅ Only "Paragraph 1" becomes red (because it's immediately after <h2>).

11. General Sibling Selector

- Selects all siblings after a specified element

```
element1 ~ element2 {
  /* styles */
}
```

- "Select **all** element2 that are siblings **after** element1".

```
<h2>Heading</h2>
<p>Paragraph 1</p>
<p>Paragraph 2</p>
```

```
h2 ~ p {
  color: blue;
}
```

✅ Both "Paragraph 1" and "Paragraph 2" become blue (since both come after <h2>).

12. Pseudo class Selectors

- Used for styling based on **state** or **position**.

a. Visited & Link

```
a: visited {
  color: yellow;
}

a: link {
  color: green;
}
```

- :visited → After link is clicked
- :link → Unvisited links

b. :hover

```
a:hover {
  background-color: yellow;
}
```

- Applies style when mouse hovers over an element.

c. :active

```
a:active {
  background-color: red;
}
```

- Style when the link is being **clicked**.

d. :first-child

```
p:first-child {
  background-color: aqua;
}
```

- Applies style **only if p is the first child** of its parent.

e. :last-child

```
p:last-child {
  background-color: aqua;
}
```

- Applies style **only if p is the last child** of its parent.

e. :nth-child(n)

```
p:nth-child(2) {
  background-color: aqua;
}
```

- Applies style **only if p is the second child** of its parent.
- Here value of n>0.
- We can also select only even or odd elements by putting n=even or odd.

13. pseudo element selector:

It is used to style specific parts of an element.

a. ::first-line

- **Description:** Styles the **first line** of a block-level element.
- **Used with:** Only block-level elements (like <p>, , etc.)

Example:

```
<p>This is a long paragraph that demonstrates the use of the first-line pseudo-element. Only this first line will be red.</p>
```

```
<style>
  p::first-line {
    color: red;
```

```
  font-weight: bold;
}
</style>
```

🟢 **Output:** Only the **first line** of the paragraph appears in **red bold**. Rest remains normal.

b. ::first-letter

- **Description:** Styles the **first letter** of the text inside a block-level element.
- **Used with:** Only block-level elements.

Example:

```
<p>This is an example paragraph.</p>

<style>
 p::first-letter {
  color: blue;
  font-size: 2rem;
 }
</style>
```

🟢 **Output:** Only the **letter "T"** in "This" will be **blue and large**.

c. ::before

- **Description:** Inserts content **before** the element's content.
- **Note:** You must use the content property.

Example:

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

<style>
```

```
 p::before {
   content: " 👋 Hello! ";
   color: green;
 }
</style>
```

🟢 **Output:**
👋 Hello! This is a paragraph.
(The green text appears **before** the paragraph content.)

d. ::after

- **Description:** Inserts content **after** the element's content.

Example:

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

<style>
 p::after {
   content: " 🌟";
   color: orange;
 }
</style>
```

🟢 **Output:**
This is a paragraph. 🌟
(The orange star appears **after** the paragraph.)

e. ::marker

- **Description:** Styles the **bullet or numbering** of list items.

Example:

```
<ul>
```

```
  <li>Item One</li>
  <li>Item Two</li>
</ul>

<style>
  ::marker {
    color: red;
    font-size: 1.5rem;
  }
</style>
```

🟢 **Output:**

- (in red) Item One
- (in red) Item Two

f. ::selection

- **Description:** Styles the part of the page that is **selected** (highlighted) by the user.

Example:

```
<p>Select this text to see the effect.</p>

<style>
  ::selection {
    background-color: yellow;
    color: black;
  }
</style>
```

🟢 **Output:** When you select any text on the page, the **background becomes yellow** and **text becomes black**.

🔶 What is the CSS Box Model?

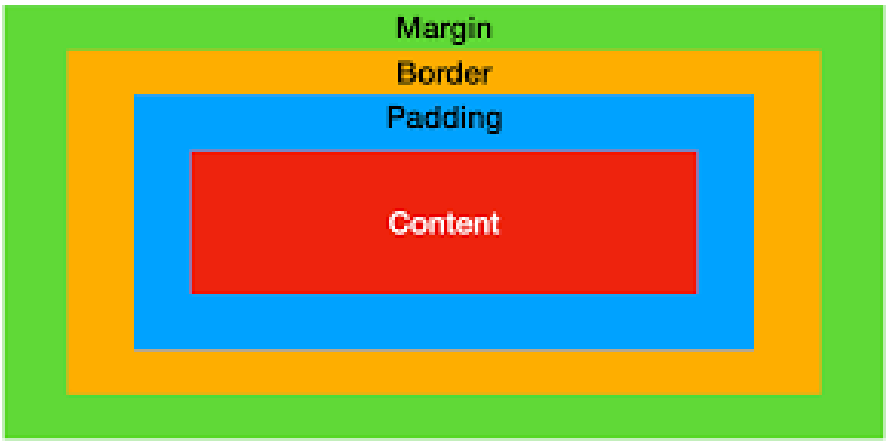
The CSS Box Model is a fundamental concept in CSS which treats every HTML element as a rectangular box consisting of:

1. **Content** – The actual text or image inside the element.
2. **Padding** – The space between the content and the border.

   Background colour extends into padding
3. **Border** – The edge around the padding.
4. **Margin** – The space outside the border, separating it from other elements.

   Background colour does **not** affect margin area

Total space taken by an element = content + padding + border + margin



Width / Height Calculation

Total Width = content Width + Left & Right Padding + Left & Right Border

Total Height = content height + top & bottom padding + top & bottom border

📦 box-sizing Property

```
box-sizing: border-box;
```

This changes how width/height is calculated:

- **Without** border-box: height/width applies **only** to content.
- **With** border-box: height/width includes **content + padding + border**.

Margin Collapse

When two vertical margins meet (e.g., between two divs), only the **larger margin is applied**.

> If box A has margin-bottom: 25px,
>
> and box B has margin-top: 35px,
>
> → result: total margin between them = 35px

✅ What is a Font?

- A **font** defines how text looks (style of characters).
- Examples: Arial, Times New Roman, Poppins, Ballu Bhai, etc.

✅ Applying Fonts in CSS

- Use the font-family property to change fonts:

```
p {
  font-family: "Tahoma", "Geneva", "sans-serif";
}
```

- Always provide **fallback fonts** in case the first one isn't available.

✅ Font Properties in CSS

- font-style: for styling text (e.g., italic, normal, oblique)
- font-weight: for boldness (e.g., bold, lighter, 100–900)
- font-size: changes size of text (use px, em, %, etc.)

✅ How to Use Google Fonts

1. Visit: https://fonts.google.com
2. Search for a font (e.g., "Poppins", "Ballu Bhai")
3. Choose styles (e.g., regular, 600, italic)
4. Two ways to include:
    - **Link** (HTML):

      ```
      <link
      href="https://fonts.googleapis.com/css2?family=Poppins&display=swap" rel="stylesheet">
      ```

    - **Import** (CSS):

      ```
      @import
      url('https://fonts.googleapis.com/css2?family=Poppins&display=swap');
      ```

✅ Line Height

- Controls vertical spacing **between lines** of text.
- Syntax:

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

- Default is ~110–120% of font size (e.g., 1.1 to 1.2).
- Helps improve readability.

✅ Letter Spacing

- Adds/removes space **between letters**.
- Syntax:

  ```
  p {
  ```

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

✅ Text Transform

- Controls **capitalization** of text.
- Values:
  - capitalize: Capitalizes first letter of each word.
  - uppercase: Converts all to UPPERCASE.
  - lowercase: Converts all to lowercase.

```
<h2 style="text-transform: capitalize;">about fonts</h2>
<!-- Output: About Fonts -->

<p style="text-transform: uppercase;">about fonts</p>
<!-- Output: ABOUT FONTS -->

<p style="text-transform: lowercase;">ABOUT FONTS</p>
<!-- Output: about fonts -->
```

✅ Text Decoration

Used for underlines and more styling.

```
p {
  text-decoration: underline dotted red 2px;
}
```

- **text-decoration:** Adds underline, overline, line-through.

```
p {
  text-decoration: underline;
```

```
}
```

- **text-decoration-color:** Changes color of underline.

```
p {
  text-decoration-color: aqua;
}
```

- **text-decoration-style:** Sets style of underline.
  - Values: solid, dotted, dashed, wavy, double

- **text-decoration-thickness:** Controls **thickness** of line.

```
p {
  text-decoration-thickness: 4px;
}
```

✅ **Text Indent**

- Indents the **first line** of a paragraph.
- Example:

```
<p style="text-indent: 40px;"> This is an example of a paragraph with an indented first line. The first line starts 40px to the right</p>
```

Before:

| This is an example of a paragraph with...

After text-indent: 40px:

| This is an example of a paragraph with...

✅ **Text Overflow**

- Handles content that overflows its container.

- Needs these three together:

```
.lorem {
  width: 45px;
  overflow: hidden;
  text-overflow: ellipsis; /* or "clip" */
}
```

ellipsis → adds "..."

clip → cuts text sharply without dots

✅ Word Break

- Forces line breaks **within** words.
- Example:

```
p {
  word-break: break-all;
}
```

Without break-all → the whole word might overflow

With break-all → word breaks at any character

✅ Text Align

- Aligns text inside its container.
    - Values: left, right, center, justify
- Syntax:

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

What are Colors in CSS?

Colors are used in CSS to style text, backgrounds, borders, etc.

Ways to Apply Colors in CSS

a) Color Keywords: Use predefined CSS color names.

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

b) Hex Color Codes:

- A hexadecimal way to define colors, starts with #.
- The hex code consists of a hash(#) symbol followed by six characters. These six characters are arranged into a set of three pairs (RR, GG, and BB). Each character pair defines the intensity level of the colour, where R stands for red, G stands for green, and B stands for blue. The intensity value lies between 00 (no intensity) and ff (maximum intensity).

```
p {
  color: #34db76; /* Light greenish */
}
```

Format: #RRGGBB or shorthand #RGB

c) RGB (Red, Green, Blue): Each color component ranges from 0 to 255.

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

d) RGBA (RGB + Alpha): Alpha controls opacity (0 = transparent, 1 = fully visible).

```
p {
```

```
  color: rgba(0, 0, 255, 0.5); /* Semi-transparent blue */
}
```

e) HSL (Hue, Saturation, Lightness):

Hue:

- Hue represents the type of color.
- It is measured in degrees, and its value lies between 0 to 360.
- 0 degree represents black, 120 degree is for green, and 360 degree is for blue.

Saturation:

- Saturation controls the intensity or purity of the color.
- It is measured in percentage, and its value lies between 0% and 100%.
- 0% saturation is no color (grayscale), and 100% saturation is the most intense colour.

Lightness:

- Lightness determines how light or dark the colour is.
- It is measured in percentage, and its value lies between 0% and 100%.
- 0% lightness represents pure black, 50% lightness represents normal colour, and 100% lightness is pure white.

```
p {
  color: hsl(120, 100%, 50%); /* Pure green */
}
```

f) HSLA (HSL + Alpha): HSL with opacity

```
p {
  color: hsla(240, 100%, 50%, 0.3); /* Semi-transparent blue */
}
```

Why Use Different Formats?

- **Keywords:** Easy for basic usage.
- **Hex/RGB:** Designers often provide color specs this way.

- **RGBA/HSLA:** When opacity or advanced customization is needed.

✅ What is Specificity in CSS?

Specificity is a method used by the CSS Cascade Algorithm to determine which style rule wins when multiple styles target the same element.

The Cascade Algorithm:

The cascade is the algorithm for solving conflicts where multiple CSS rules apply to an HTML element.

🧠 Cascade Algorithm Basics (4 Factors):

1. Position (Order of Appearance)

- If selectors have **same specificity**, the **last one written** wins.
  - 🟢 *Example:*

```
.cRed { color: red; }

.cPurple { color: purple; } /* This wins if applied last */
```

2. Specificity Value

Each selector type has a **weight**. The higher the specificity, the higher the priority.

Selector Type	Specificity Score
Inline Styles	1000
ID Selectors (#id)	100
Class, Attribute, Pseudo-class (.class, [type], :hover)	10
Element & Pseudo-elements (h1, ::before)	1
Universal (*)	0

3. Origin

Where the style comes from:

- Browser default (User Agent) styles have the lowest priority.
- Author-defined CSS wins over browser defaults.
- External, internal or inline author CSS follows the specificity rules.

4. !important

- Overrides everything — **even inline styles**.

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

💉 Example of Specificity Calculation

```
a.harry.rohan[href]:hover {
  color: blueviolet;
}
```

- 🔹 Breakdown:
  - a → element selector → 1 point
  - .harry, .rohan → 2 class selectors → 2 × 10 = 20 points
  - [href] → attribute selector → 10 points
  - :hover → pseudo-class → 10 points

Total Specificity: 1 + 10 + 10 + 10 + 10 = 41

Priority (Highest to Lowest):

1. !important
2. Inline styles (style="...")

3. ID selectors (#id)
4. Classes, attributes, pseudo-classes (.class, [type], :hover)
5. Elements and pseudo-elements (h1, ::before)
6. Universal selector (*)
7. Browser default styles (User Agent stylesheet)

🎯 Units in CSS

CSS has many units for measuring width, height, font size, etc.

There are two types of length units:

1. Absolute units:

- The absolute length units are fixed and a length expressed in any of these will appear as exactly that size.
- Absolute units are not recommended for use on screen, because screen sizes vary so much.

cm - centimetres (1 cm = 37.8 px)

mm - millimetres

in - inches (1 in = 96 px = 2.54cm)

px - pixel (1px = 1/96th of an inch)

pt - points (1pt = 1/72th of 1 inch)

pc - picas (1pc = 12 pt)

---

2. Relative units:

2.1 🖥️ Viewport Units

a. Viewport Width (vw)

- 1vw = 1% of the **viewport's width** (browser window width).
- width: 80vw; → 80% of screen width.

- ✅ **Responsive**: Automatically adjusts to screen size.
- Useful for full-width layouts or responsive boxes.

b. Viewport Height (vh)

- 1vh = 1% of the **viewport's height**.
- height: 80vh; → 80% of screen height.
- Used for vertical centering or full-page sections.

c. vmin & vmax

- vmin: 1% of viewport's smaller dimension.
- vmax: 1% of viewport's larger dimension.
- ✅ Useful in responsive typography or layout adjustments across devices.

📱 **Example:**

- On **mobile** (portrait): height > width → vmin = vw
- On **desktop**: width > height → vmin = vh

2.2 🔠 Font-relative Units

a. em (Relative to parent element)

- 1em = 100% of the font-size of the **parent**.
- *Example:*

```
.parent {
  font-size: 16px;
}
.child {
  font-size: 1.5em; /* = 24px */
}
```

b. rem (Relative to root element)

- 1rem = 100% of the font-size of the **<html>** (root).

- Default browser root size: 16px.
- *Example:*

```
html { font-size: 12px; }

p { font-size: 2rem; } /* = 24px */
```

---

3. Percentage Units (%)

% values are relative to the parent element's dimension (width or height).

```
<div class="parent">
  <div class="child">50% Width</div>
</div>


.parent {
  width: 600px;
  background: gray;
}
.child {
  width: 50%; /* 50% of 600px = 300px */
  background: red;
}
```

⚠️ **Note:** If the parent's height/width is not defined, % may not work as expected.

---

4. min-width / max-width / min-height / max-height

Used to set bounds on element sizes:

```
.box {
  min-height: 80vh;   /* at least 80% of viewport height */
  max-width: 90vw;    /* maximum 90% of viewport width */
```

```
}
```

📌 When you add more content, height increases beyond min-height.

---

5. Centering with margin: auto

```
.box {
  width: 400px;
  margin: auto; /* Horizontal centering */
  background-color: lightblue;
}
```

⚠️ Works only if width is set and the element is **block-level**.

Using Units in Nesting

If you nest divs and give them percentage widths:

```
<div class="outer">
  <div class="inner"></div>
</div>


.outer {
  width: 80vw;
  background: black;
}
.inner {
  width: 50%;
  background: red;
}
```

Here, .inner gets **50% of .outer**, which is 80vw → .inner = 40vw.

Avoiding Common Mistakes

❌ **Mistake:**

Using % height without setting parent height.

```
.parent {
  /* height not set */
}
.child {
  height: 100%; /* This won't work */
}
```

✅ **Solution:**

Use vh units instead, like height: 80vh;.

🧠 **Key Notes:**

- Always use margin: auto; to horizontally center a **block-level** element.
- Inline elements **don't respond** to width/margin settings like block elements do.
- Use box-sizing: border-box; to include padding and borders inside the total width/height (avoids unexpected scroll).
- Use %, vw, vh, em, rem for **responsive design**.
- px is fixed → combine it with relative units for balanced responsiveness.

Block & Inline Elements:

- Block elements (like) take full width of their container, even if the content is small.

- Inline elements (like) only take the space of their content and sit side by side.
- Cannot set width, height, vertical padding & margin of inline elements.
- Can set width, height, vertical padding & margin of block elements.

Display Property

- You can change the display of any element using the display CSS property:
    - **display: inline;** → Makes block element behave like inline (no new line).
    - **display: block;** → Makes in-line element behave like block.
    - **display: inline-block;** → Inline behaviour with ability to set width, height, padding, and margin.
    - **display: none** → Hides the element and removes it from layout (no space taken).
    - **visibility: hidden** → Hides the element **but keeps the space** it would have taken.
    - **display: flex;** is used to align and position elements easily. By-default it behaves like a block element.

        Example: **justify-content: center;** → Centers items horizontally.
    - **display: inline-flex;** → Similar to flex but behaves inline.

Practical Tips

Always reset margin and padding with a universal selector (* { margin: 0; padding: 0; }) to avoid unexpected layout issues.

Box Shadow (box-shadow): Adds shadow to elements.

🔸 **Syntax:**

box-shadow: <x-offset> <y-offset> <blur-radius> <spread-radius> <color> [inset];

Part	Meaning
`x-offset`	Horizontal shift (right if +ve, left if -ve)
`y-offset`	Vertical shift (down if +ve, up if -ve)
`blur-radius`	Blurriness of the shadow
`spread-radius`	Size expansion or contraction
`color`	Shadow color
`inset` *(optional)*	Shadow appears **inside** the element instead of outside

- **Example:**

```
box-shadow: 5px 15px 5px 0px green;
```

Text Shadow (text-shadow): Adds a shadow behind text

- **Syntax:**

```
text-shadow: <x-offset> <y-offset> <blur-radius> <color>;
```

- **Example:**

```
text-shadow: 3px 3px 5px purple;
```

Outline (outline): Adds outline outside element border.

- **Syntax:**

```
outline: <width> <style> <color>;
```

- **Example:**

```
outline: 2px solid black;
```

- **How it's different from border:**

Feature	border	outline
Part of box model	✅ Yes	❌ No

Feature	border	outline
Affects layout	✅ Yes	❌ No
Supports border-radius	✅ Yes	❌ But follows border curve
Can be offset	❌ No	✅ Yes (using outline-offset)
Multiple sides control	✅ Yes	❌ No (uniform only)

🔸 **Outline Offset:**

- Pushes the outline **away from the edge** of the element that is border.

```
outline-offset: 10px;
```

Styling Lists in CSS

Styling lists is crucial for building elements like navigation bars, menus, and custom bullet points.

Default List Rendering

- HTML Lists (<ul> and <ol>) render with:
    - Bullets (unordered) or numbers (ordered)
    - Default styles like display: list-item
    - A marker (the bullet/number) and the list content

CSS Properties for List Styling

✅ **list-style-type**

- Changes the marker style (bullet, number type, etc.)

Examples:

```
ul {
```

```
  list-style-type: disc;      /* default */
  list-style-type: circle;
  list-style-type: square;
  list-style-type: decimal;
  list-style-type: devanagari;
  list-style-type: '🔥';      /* emoji or custom symbol */
}
```

✅ list-style-position

- Controls where the marker appears:
  - **outside (default)** → marker outside the element box
  - **inside** → marker comes inside the content box

Example:

```
ul {
  list-style-position: inside;
}
```

This helps apply background or border cleanly without clipping the marker.

🧠 Visual Differences

Feature	list-style-position: outside	inside
Marker Location	Outside box	Inside box
Background on	Only text area	Includes marker
Border on	Doesn't wrap marker	Wraps everything

✅ list-style-image

- Replace bullet with a custom image

Example:

```
ul {
  list-style-image: url('star.png');
}
```

⚠️ If the image link is broken or forbidden, it won't render.

✅ list-style (Shorthand)

- Combines type, position, and image into a single line

Example:

```
ul {
  list-style: disc inside url('dot.png');
}
```

Only one marker will show at a time: **either image or type**.

What is the overflow property?

The overflow property in CSS controls what happens when content inside a box (div or any block element) **exceeds the size of the box**.

📘 CSS overflow Property Values:

1. **overflow: visible** ***(default)***
    - Content overflows and is not clipped.
    - No scrollbars appear.
2. **overflow: hidden**
    - Content is clipped and anything that overflows is not visible.
    - No scrollbars.
3. **overflow: scroll**

- Scrollbars always appear, both horizontal and vertical, whether needed or not.

4. **overflow: auto**
    - Scrollbars only appear if needed (if content overflows).
    - Recommended for most use cases.

🎯 Text Specific Overflow:

To handle text overflow neatly:

```
white-space: nowrap;
overflow: hidden;
text-overflow: ellipsis;
```

- **white-space: nowrap** for printing text only in one-line.
- **text-overflow: ellipsis** shows "..." when text overflows the container.

🧩 Advanced: overflow-x and overflow-y

You can control scroll separately on horizontal and vertical axes:

```
overflow-x: auto;
overflow-y: hidden;
```

Or use shorthand:

```
overflow: auto hidden; /* X: auto, Y: hidden */
```

- The **first value is for X**, second for Y.

🆚 overflow: hidden vs overflow: clip

- Both hide overflowing content.
- But **overflow: clip** also disables programmatic scrolling (like using JavaScript .scrollTo()).

🔧 What is the position property in CSS?

The position property in CSS defines how an element is positioned in the document. It affects how the element behaves with top, left, right, and bottom values.

🅰️ 1. static (Default)

- **Behavior**: Element is positioned as per normal document flow.
- **Cannot use**: top, left, right, bottom, z-index.
- **Example**:

```
.box {
  position: static;
  top: 10px; /* ❌ No effect */
}
```

🅱️ 2. relative

- **Behavior**: Element stays in its place **but can be moved** using top, left, etc.
- **Doesn't affect** other elements' positions.
- **Useful for** offsetting elements while preserving their original space.
- **Example**:

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

🅾️ 3. absolute

- **Behavior**: Element is **removed** from the normal document flow.

- **Positioned relative to** the **nearest positioned ancestor** (relative, absolute, fixed, or sticky).
- **If no ancestor is positioned**, it uses <html> or <body>.
- **Example**:

```
.parent {
  position: relative;
}

.child {
  position: absolute;
  top: 10px;   /* 10px from the top of parent */
  left: 10px;  /* 10px from the left of parent */
}
```

⛔ **4. fixed**

- **Behavior**: Always **fixed** to the viewport.
- **Doesn't move** when you scroll the page.
- **Common use**: Sticky headers, sidebars, floating buttons.
- **Example**:

```
.box {
  position: fixed;
  top: 0;
  left: 0;
}
```

📌 **5. sticky**

- The element acts like relative until you scroll past a point, then becomes fixed.

- Needs a top/left/etc. value to work.
- Commonly used for sticky navbars.
- **Example**:

  ```
  .navbar {
    position: sticky;
    top: 0;
  }
  ```

  Sticks to the top once it's reached there during scroll.

⚠️ **Special Note: CSS properties that make an element behave like it's positioned (even without position being set):**

1. transform
2. filter
3. perspective

- If any of these are applied to a parent, its child with **position: absolute** will treat it like a positioned ancestor.
- Example:

  ```
  .parent {
    transform: translate(0); /* Acts like a positioned parent */
    filter : invert(0);      /* Acts like a positioned parent */
    perspective: 0em;   /* Acts like a positioned parent */
  }


  .child {
    position: absolute;
    top: 0;
    left: 0;
  ```

🔁 **Comparison Table:**

Position	Affects Flow?	Uses top/left?	Relative To
static	Yes	❌ No	Normal flow
relative	No	✅ Yes	Itself
absolute	No	✅ Yes	Nearest positioned ancestor
fixed	No	✅ Yes	Viewport
sticky	Yes/No	✅ Yes	Scroll position until stuck

📌 **z-index Note:** for appearing any element over another element.

- You **must** set a position (other than static) for z-index to work.
- Higher z-index = appears on top.

🎯 **What are CSS Variables?**

CSS Variables (also called *custom properties*) are like variables in programming—they store reusable values, such as colors, font sizes, margins, paddings, etc.

✅ **Why Use CSS Variables?**

- Makes your CSS **easier to manage and update**.
- Helps in applying **themes** (e.g., dark mode, red theme, etc.).
- **Change one value**, and it updates everywhere that variable is used.

🔧 **How to Define CSS Variables:**

🔹 **Global Variable (accessible anywhere):**

Defined in the **:root** selector:

```
:root {
  --primary-color: #0f0;
  --secondary-color: #afa;
  --def-pad: 23px;
  --def-op: 0.3;
}
```

🔹 **Local Variable (limited to a specific element):**

Defined inside a specific CSS rule:

```
.nav li:first-child {
  --primary-color: orange;
}
```

🔍 **How to Use a CSS Variable:**

Use the var() function:

```
.nav {
  background-color: var(--primary-color);
  padding: var(--def-pad);
  opacity: var(--def-op);
}
```

💡 **Fallback Values in CSS Variables:**

If the variable is not found, you can provide a fallback:

```
color: var(--unknown-color, blue); /* uses blue if --unknown-color isn't defined */
```

🌈 **Can Variables Hold Only Colors?**

❌ **No!** CSS variables can hold:

- Colors (#fff, rgb(0,0,0))
- Numbers (10px, 2rem)
- Strings ("Hello", url('image.png'))
- Opacity values
- Any CSS value you might reuse

📌 **Key Takeaways:**

- Use “--" to define a variable.
- Use var(--name) to access it.
- Define in :root for global use.
- Variables reduce repetition and improve maintainability.
- They’re extremely useful in **themes and large projects**.

📱 **What is a Media Query?**

A **media query** is a CSS technique used to make your website **responsive**, i.e., it adapts its style based on **device characteristics** like:

- Screen width
- Screen type (e.g., print, screen, speech)
- Orientation (portrait or landscape)
- Resolution

✅ **Why Use Media Queries?**

- To **change layout/style** for different screen sizes (mobile, tablet, desktop).
- To create **responsive designs** that work across devices.
- To apply styles **conditionally** depending on device capability.

💡 **Basic Syntax:**

```
@media only screen and (max-width: 600px) {
  /* CSS rules here will only apply when screen width ≤ 600px */
  body {
    background-color: lightblue;
  }
}
```

Breakdown:

- **@media:** starts the query
- **only:** optional, used to prevent older browsers from applying
- **screen:** type of device (screen, print, speech, all)
- **(max-width: 600px):** condition (can also use min-width, orientation, etc.)

📐 **Media Types:**

- **all:** for all devices (default)
- **screen:** for computer screens, tablets, smartphones
- **print:** for print preview or printing documents
- **speech:** for screen readers

🧪 **Common Media Features:**

Feature	Description
max-width	Apply styles **up to** this width
min-width	Apply styles **from** this width and larger
orientation	portrait or landscape

Feature	Description
resolution	Based on pixel density
hover, pointer, aspect-ratio, etc.	Advanced conditions

🧱 **Example - Responsive Background:**

```
body {
  background-color: red;
}

@media only screen and (max-width: 455px) {
  body {
    background-color: blue;
  }
}
```

🔁 Result: If the screen is narrower than 455px, background turns blue.

📦 **Responsive Boxes (Flex Example):**

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

.box {
  width: 344px;
  height: 344px;
  background-color: steelblue;
  margin: 3px;
```

```
  padding: 3px;
}

.boxes {
  display: flex;
}

@media only screen and (max-width: 455px) {
  .boxes {
    flex-direction: column;
  }
}
```

💡 The layout of boxes changes to vertical (column) when screen ≤ 455px.

🔄 **Orientation-Based Styling:**

```
@media only screen and (orientation: landscape) {
  body {
    border: 2px solid purple;
  }
}
```

📱 Applies styles **only when** device is in **landscape mode** (width > height).

📊 **What Are Breakpoints?**

Breakpoints are screen widths where your design should change:

- <= 455px: small devices (phones)
- 456px – 768px: tablets
- 769px – 1080px: laptops/desktops
- > 1080px: large screens/monitors

⚠️ **Bonus Tips:**

- When using frameworks (like **Bootstrap** or **Tailwind CSS**), media queries are mostly **abstracted** (already included).
- When writing **raw CSS**, media queries are essential for responsiveness.

🧠 **Topic: Understanding float and clear in CSS**

📌 **Why Learn This?**

- Although Flexbox and Grid are now preferred for layouts, older sites still use float and clear.
- Knowing how they work helps with **maintenance** and understanding legacy code.

🔹 **float Property**

✅ **What it does:**

- It **pulls an element** to the left or right.
- Allows **inline content** (like text) to wrap around it.

💉 **Example:**

```
<img src="image.jpg" style="float: left; width: 230px;">
<p>Lorem ipsum dolor sit amet...</p>
```

🔍 **Result:**

- Image floats to the **left**.
- Paragraph text wraps around the image on the **right**.

🧠 **Note:**

- float: right (makes the image float to the **right**, text wraps to the **left**).

🔹 clear Property

✅ What it does:

- Tells an element to not allow floated elements on a specific side (left/right/both).

🧪 Example:

```
p{
  clear: right;
}
```

🔍 Result:

- This card will **move below** the floated element on the right.
- You can also use:
  - clear: left → Blocks float on left
  - clear: both → Blocks float on both sides

⚠️ Common Float Problem: Overflowing Container

- When children are floated, **the parent's height collapses**.
- The border of the parent container does **not wrap** around its children.

✅ Solution:

Use one of the following:

```
.cards {
  display: flow-root;
}
```

❌ Why Not Use Float for Layouts Today?

- **Not meant** for layout design (originally used for wrapping text around images).

- Complex and unintuitive for modern layouts.
- **Flexbox** and **CSS Grid** are better alternatives:
    - Easier
    - More powerful
    - Responsive by default

✅ Final Advice:

Avoid using float and clear for modern layout designs.
Use Flexbox or Grid instead, but know float/clear in case you encounter them in legacy projects.

⭐ CSS Flexbox –

Flexbox (Flexible Box Layout) is a modern CSS layout system that helps you arrange items in a row or column easily. It automatically adjusts the layout depending on screen size, available space, and your design.

🔹 1. What is Flexbox?

- Flexbox is used to create **flexible**, **responsive**, and **easier layouts**.
- It focuses on placing items properly even when the screen size changes.
- It works on **one-dimensional layout** (only row OR column at a time).

To use Flexbox:

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

🔹 2. Main Axis & Cross Axis

Understanding axes is very important.

➤ **Main Axis**

- Direction in which flex items are placed.
- Depends on **flex-direction** (row or column).

➤ **Cross Axis**

- The axis perpendicular to the main axis.

Example:

- **flex-direction: row** → main axis = horizontal, cross axis = vertical
- **flex-direction: column** → main axis = vertical, cross axis = horizontal

---

🔹 3. flex-direction

This property decides how items are placed inside the container.

Values:

Value	Meaning
row	Default. Items go left → right
row-reverse	Right → left
column	Top → bottom
column-reverse	Bottom → top

Example:

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

🔹 4. justify-content (Align along Main Axis)

Used to adjust **horizontal alignment** when flex-direction is row,
or **vertical alignment** when flex-direction is column.

Values:

- flex-start → items at the beginning
- flex-end → items at the end
- center → items at the center
- space-between → equal space *between* items
- space-around → equal space around items
- space-evenly → equal spacing everywhere

🔹 5. align-items (Align along Cross Axis)

Controls how items align vertically (if row) or horizontally (if column). For a single row only.

Values:

- flex-start
- flex-end
- center
- stretch (default; items grow to fill)
- baseline

Example:

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

🔹 5. align-content (Align along Cross Axis)

Used only when items wrap onto multiple lines.

Value	Meaning
flex-start	All lines packed at the top (start of cross axis)
flex-end	All lines packed at the bottom (end of cross axis)
center	Lines grouped in center
stretch	Lines stretch to fill available space
space-between	Equal space **between** the lines
space-around	Equal space **around** each line
space-evenly	Perfectly even spacing everywhere

🔹 6. flex-wrap

By default, flexbox tries to keep all items in **one line**.

If items don't fit, they shrink.

To allow wrapping:

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

Values:

- nowrap (default)
- wrap (moves items to next line)
- wrap-reverse

🔹 7. flex-flow (Shorthand)

Combines flex-direction + flex-wrap.

Example:

```
flex-flow: row wrap;
```

🔹 8. gap (Spacing between items)

Adds empty space **between** flex items (like margin but cleaner).

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

You can also use:

- row-gap
- column-gap

🔹 9. Item-Level Properties (Applied to individual flex items)

▶️ A. order

Controls **sequence** of items.

```
.item {
  order: 2;
}
```

Higher order means item appears later. Default order is 0.

▶️ B. flex-grow

Decides how much an item should **expand** when extra space is available. Default value of is 0.

```
.item {
  flex-grow: 1; /* grows to fill space */
}
```

If one item has grow 2, another has grow 1 →
the first item gets **double space**.

C. flex-shrink

Decides how items **shrink** when space is less.

```
.item {
  flex-shrink: 1;
}
```

Higher shrink value = shrinks more.

D. flex-basis

Defines initial size of the item (before grow/shrink).

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

E. flex (Shorthand)

Combines grow, shrink, and basis.

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

F. align-self

Overrides align-items **for only one item**.

```
.item {
  align-self: center;
}
```

Values same as align-items:

- flex-start
- flex-end
- center
- stretch

🔹 10. Common Flexbox Mistakes

The video warns about these:

- Using too many margins instead of using gap.
- Forgetting that flexbox works in **one dimension only**.
- Confusing justify-content and align-items.
- Not understanding **main axis vs cross axis**.
- Using fixed widths inside a flexible layout.

🔹 11. Why Flexbox is Useful

- Best for responsive UI elements.
- Makes layout simple (no float or table layout needed).
- Used in navbars, forms, cards, footers, galleries, etc.
- Automatically adjusts layout for different screens.

CSS Grid Layout Module

The Grid Layout Module offers a grid-based layout system, with rows and columns.

The Grid Layout Module allows developers to easily create complex web layouts.

My Header

Link 1
Link 2
Link 3

Lorem Ipsum

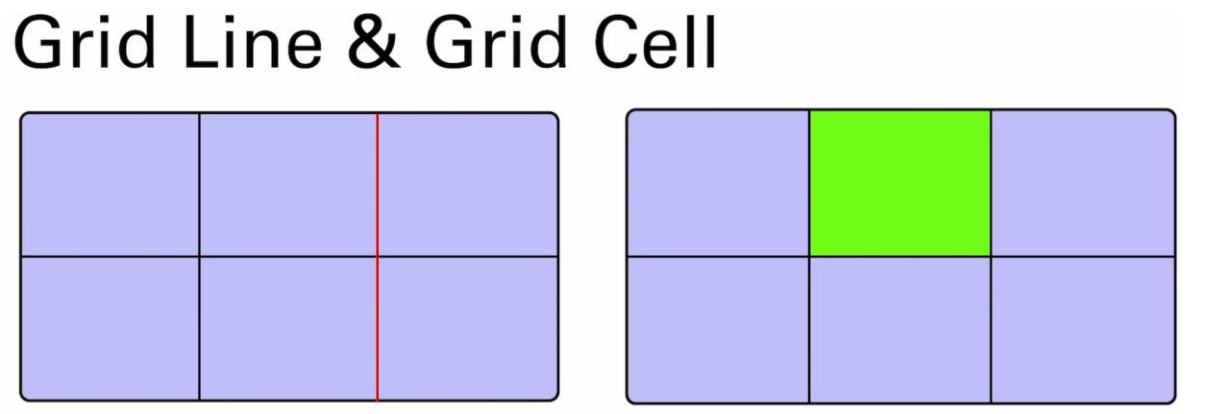
Lorem ipsum odor amet, consectetuer adipiscing elit. Ridiculus sit nisl laoreet facilisis aliquet. Potenti dignissim litora eget montes rhoncus sapien neque urna. Cursus libero sapien integer magnis ligula lobortis quam ut.

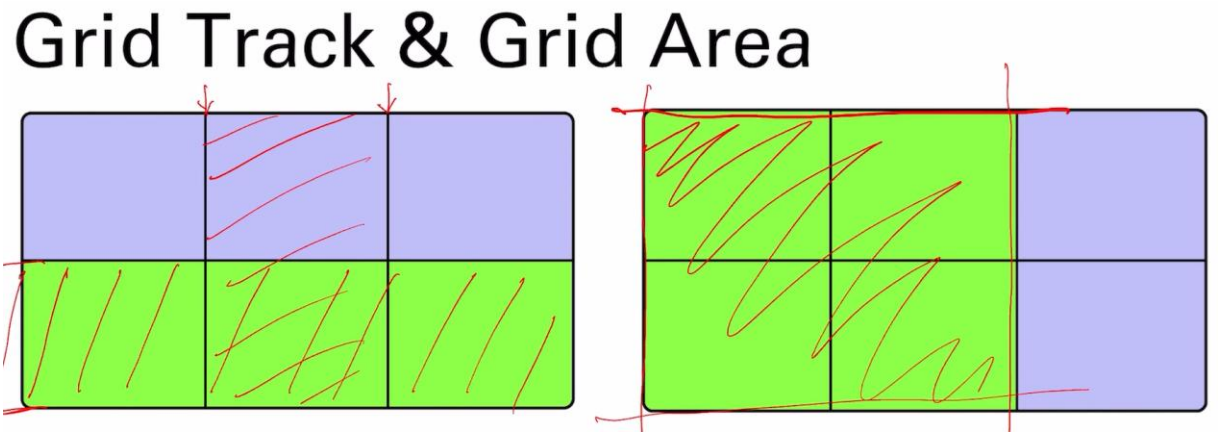
Footer

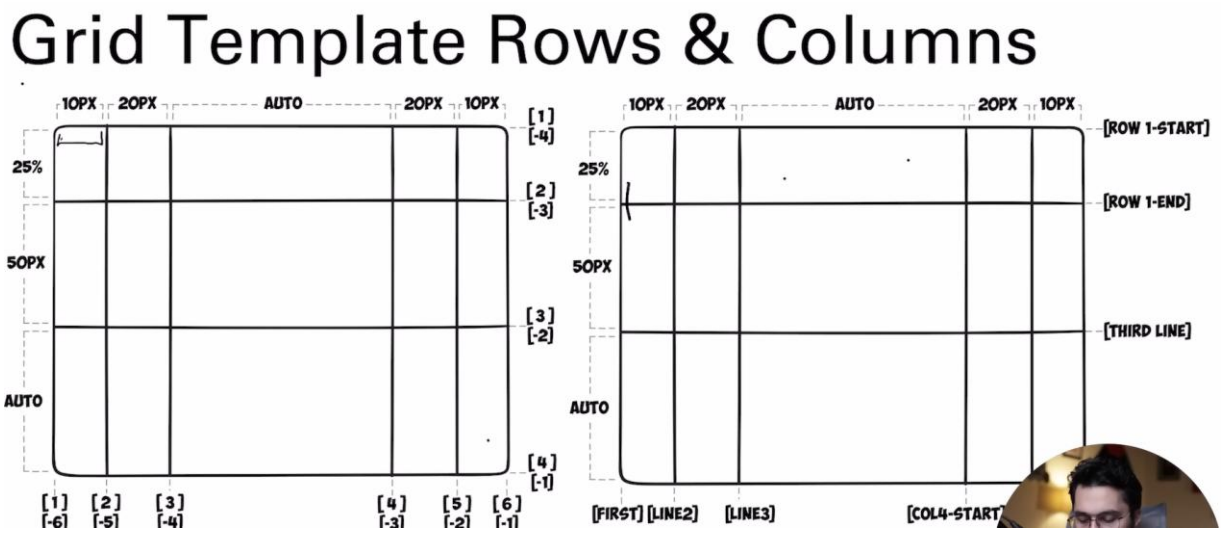
CSS Grid Components

A grid always consists of:

- **A Grid Container** - The parent (container) element, where the display property is set to grid or inline-grid
- **One or more Grid Items** - The direct children of the grid container automatically becomes grid items







All CSS Grid Properties:

1. **grid-template-columns:** specifies the number of columns in a grid layout. We can also give some name to grid-columns or divide it by giving size in any unit.

```
.grid-container {
  display: grid;
  grid-template-columns: [pehla] 120px [doosara] 100px [thishara];
}
```

2. **grid-template-rows:** specifies the number of the rows in a grid layout. We can divide it by giving size in any unit.

```
.grid-container {
  display: grid;
  grid-template-rows: 100px 300px;
}
```

3. **grid-row:** specifies a grid item's size and location in a grid layout, and is a shorthand property for the following properties:

```
.item1 {
  grid-row: 1 / 2;
}
```

   a. **grid-row-start:** Specifies where to start the grid item
   b. **grid-row-end:** Specifies where to end the grid item

```
.item1 {
  grid-row-start: 1;
```

```
  grid-row-end: 3;;
}
```

This means the item covers row 1 to 2.

4. **grid-column:** specifies a grid item's size and location in a grid layout, and is a shorthand property for the following properties:

```
.item1 {
  grid-column: 1 / 2;
}
```

   c. **grid-column-start:** Specifies where to start the grid item
   d. **grid-column-end:** Specifies where to end the grid item

```
.item1 {
  grid-column-start: 1;
  grid-column-end: 2;
}
```

This means the item covers column 1to 2

5. **grid-template-areas:** Specifies how to display columns and rows, using named grid items

```
.item1 {
  grid-area: myArea;
}
.grid-container {
  display: grid;
  grid-template-areas: "myArea myArea  col3 col4 col5";
}
```

Make the named item "myArea" span two columns in a five columns grid layout:

6. **gap:** defines the size of the gap between the rows and between the columns. It is a shorthand for the following properties:

```
.grid-container {
  gap: 50px;
}
```

Set the gap between rows and between columns to 50px:

   a. **row-gap:**
   Specifies the gap between the grid rows
   b. **column-gap:**
   Specifies the gap between the grid columns.

7. **justify-items:** property is set on the grid container to give child elements (grid items) alignment in the inline direction. Inline direction is left to right and block direction is downward. For this property to have any alignment effect, the grid items need available space around themselves in the inline direction.

```
#container {
  display: grid;
  justify-items: end;
}
```

Align each grid item at the end of their grid cell, in the inline direction:

8. **align-items:** property is set on the grid container to give child elements (grid items) alignment in the block direction.For this property to have any alignment effect, the grid items need available space around themselves in the blockdirection.

```
#container {
  display: grid;
  align-items: end;
}
```

Align each grid item at the end of their grid cell, in the block direction:

9. **justify-self & align-self:** same as justify-self & align-self but it only works with a single cell.
10. **place-self:** to set both align & justify-self similar in one line.
11. **justify-content:** Horizontally aligns the whole grid inside the container (when total grid size is smaller than container).
12. **align-content:** Vertically aligns the whole grid inside the container (when total grid size is smaller than container)
13. **place-content:** to set both align & justify-content similar in one line.

Units in grid:

fr: it will divide the width of container in given fraction. We can use it with both grid-template-rows & columns.

1fr means it will divide container width in 1 fraction.

Transform: The transform property applies a 2D or 3D transformation to an element. This property allows you to rotate, scale, move, skew, etc., elements.

CSS 2D Transforms Functions

With the CSS transform property you can use the following 2D transformation functions:

1. rotate(): rotates an element clockwise or counter-clockwise according to a given degree. Using negative values will rotate the element counter-clockwise.

```
div {
  transform: rotate(20deg);
}
```

The following example rotates the element clockwise with 20 degrees:

We can also specify the value inside rotate() in turn like 0.25turn. Here 1turn=360deg.

We can also rotate in particular axis by using function like **rotateX(), rotateY() & rotateZ().**

2. scale(): increases or decreases the size(height & width) of an element. By providing value>1 to increase & value<1 to decrease.

We can provide arguments to the scale() in two ways:

a) **scale(1.6):** for scaling same in both direction.
b) **scale(x,y):** for scaling different both height & width.

```
div {
  transform: scale(2, 3);
}
```

The following example increases the element to two times of its original width, and three times of its original height.

Functions provided by scale():

a) **scaleX():** increases or decreases the width of an element. Takes single argument.
b) **scaleY():** increases or decreases the height of an element. Takes single argument.

3. skew(): skews an element along the X and Y-axis by the given angles.

```
div {
  transform: skew(20deg, 10deg);
}
```

following example skews the element 20 degrees along the X-axis, and 10 degrees along the Y-axis:

If the second parameter is not specified, it has a zero value. So, the following example skews the element 20 degrees along the X-axis:

```
div {
  transform: skew(20deg);
}
```

Functions provided by skew():

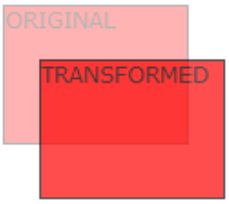
a) skewX(): skews an element along the X-axis by the given angle. Takes single argument.

b) skewY(): skews an element along the Y-axis by the given angle. Takes single argument.

4. translate(): moves an element from its current position

```
div {
  transform: translate(50px, 100px);
}
```

moves the element 50 pixels to the right, and 100 pixels down from its current position.



If we just provide single argument to the translate() then by-default it will move the element in X-axis.

Functions provided by translate():

a) translateX(): for moving position in X-axis. Takes single argument.

b) translateY(): for moving position in Y-axis. Takes single argument.

5. matrix(): combines all the 2D transform functions into one. It takes six parameters: matrix(scaleX(), skewY(), skewX(), scaleY(), translateX(), translateY()).

```
div {
  transform: matrix(1, -0.3, 0, 1, 0, 0);
}
```

6. transform-origin(): allows you to change the position of transformed elements. 2D transformations can change the x- and y-axis of an element. 3D transformations can also change the z-axis of an element. For 2D default value is -50%(in x-axis) & -50%(y-axis).

CSS Transitions: allows you to change property values smoothly, over a given duration.

CSS transition Property: It is a shorthand property for:

- **transition-property (Required) :** Used to specify the property name.
- **transition-duration (Required):** Used to specify the transition duration in sec or mills.
- **transition-timing-function:** specifies the speed curve of the transition effect.
- **transition-delay:** Used to specify after how much time transition would take place in starting.

```
div {
 width: 100px;
 height: 100px;
 background-color: red;
 transition: width 2s linear 1s;
}
```

The following example shows a 100px * 100px element. The element has specified a transition effect for the width property, with a duration of 2 seconds:

How to Trigger the Transition?

The transition is triggered when there is a change in the element's properties. This often happens within pseudo-classes (:hover, :active, :focus, or :checked).

So, from the code above, the transition effect will start when the width property changes value.

Now, we add a div:hover class that specifies a new value for the width property when a user mouses over the element:

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

Notice that when the cursor mouses out of the element, it will gradually change back to its original style.

Change Multiple Property Values

You can change multiple properties by separating them by commas.

The following example adds a transition effect for the width, height, and background-color properties, with a duration of 2 seconds for the width, 4 seconds for the height, and 3 seconds for the background-color:

```
div {
 transition: width 2s, height 4s, background-color 3s;
}
```

Add a transition effect for the width, height, and background-color properties:

transition-timing-function: specifies the speed curve of the transition effect.

This property can have one of the following values:

- **ease** - transition will start slow, then go fast, and end slow (this is default)
- **linear** - transition will keep the same speed from start to end
- **ease-in** - transition will start slow
- **ease-out** - transition will end slow
- **ease-in-out** - transition will have a slow start and end
- **cubic-bezier(n,n,n,n) -** lets you define your own values in a cubic-bezier function

transition-delay: specifies a delay before the transition starts. It is defined in seconds (s) or milliseconds (ms).

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

Add a 1 second delay before starting:

Transition + Transform

```
div {
  transition: width 2s, height 2s, background-color 2s, transform 2s;
}
```

CSS Animations: allows animation of HTML elements without using JavaScript!

An animation lets an element gradually change from one style to another.

You can change as many CSS properties you want, as many times as you want.

To use CSS animation, you must specify some keyframes for the animation.

Keyframes hold what styles the element will have at certain times.

CSS animation-name and animation-duration

animation-name: property specifies a name for the animation.

animation-duration: property defines how long an animation should take to complete. If this property is not specified, no animation will occur, because the default value is 0s (0 seconds).

CSS @keyframes Rule

When you specify CSS styles inside the @keyframes rule, the animation will gradually change from the current style to the new style at certain times. To get an animation to work, you must bind the animation to an element.

```
/* The animation code */
@keyframes myAnimation {
 from {background-color: red;}
 to {background-color: yellow;}
}

/* The element to apply the animation to */
div {
 width: 100px;
 height: 100px;
 background-color: red;
 animation-name: myAnimation;
 animation-duration: 4s;
}
```

The following example binds the "myAnimation" animation to the element. The animation will last for 4 seconds, and it will gradually change the background-color of the element from "red" to "yellow":

In the example above we have used the keywords "from" and "to" in the @keyframes rule, which represents 0% (start) and 100% (complete). It is also possible to use percent. By using percent, you can add as many style changes as you like.

```
@keyframes myAnimation {
 0%   {background-color: red;}
 25%  {background-color: yellow;}
 50%  {background-color: blue;}
 100% {background-color: green;}
}

div {
 width: 100px;
 height: 100px;
 background-color: red;
 animation-name: myAnimation;
```

```
  animation-duration: 4s;
}
```

The following example will change the background-color of the element when the animation is 25% complete, 50% complete, and again when the animation is 100% complete:

animation-delay: property specifies a delay for the start of an animation.

```
div {
  width: 100px;
  height: 100px;
  position: relative;
  background-color: red;
  animation-name: myAnimation;
  animation-duration: 4s;
  animation-delay: 2s;
}
```

The following example has a 2 seconds delay before starting the animation:

Negative values are also allowed. If using negative values, the animation will start as if it had already been playing for *N* seconds.

```
div {
  width: 100px;
  height: 100px;
  position: relative;
  background-color: red;
  animation-name: myAnimation;
  animation-duration: 4s;
  animation-delay: -2s;
}
```

In the following example, the animation will start as if it had already been playing for 2 seconds:

animation-iteration-count: property specifies the number of times an animation should run.

We can also specify it as **infinite.**

```
div {
  width: 100px;
```

```
  height: 100px;
  position: relative;
  background-color: red;
  animation-name: myAnimation;
  animation-duration: 4s;
  animation-iteration-count: 3;
}
```

The following example will run the animation 3 times before it stops:

```
div {
  width: 100px;
  height: 100px;
  position: relative;
  background-color: red;
  animation-name: myAnimation;
  animation-duration: 4s;
  animation-iteration-count: infinite;
}
```

The following example uses the value "infinite" to make the animation continue for ever:

animation-direction: property specifies whether an animation should be played forwards, backwards or in alternate cycles.

The animation-direction property can have the following values:

- **normal** - The animation is played as normal (forwards). This is default
- **reverse** - The animation is played in reverse direction (backwards)
- **alternate** - The animation is played forwards first, then backwards
- **alternate-reverse** - The animation is played backwards first, then forwards

animation-timing-function: property specifies the speed curve of the animation.

The animation-timing-function property can have the following values:

- **ease** - Specifies an animation with a slow start, then fast, then end slowly (this is default)
- **linear** - Specifies an animation with the same speed from start to end
- **ease-in** - Specifies an animation with a slow start

- **ease-out** - Specifies an animation with a slow end
- **ease-in-out** - Specifies an animation with a slow start and end
- **cubic-bezier(n,n,n,n)** - Lets you define your own values in a cubic-bezier function

```
#div1 {animation-timing-function: linear;}
#div2 {animation-timing-function: ease;}
#div3 {animation-timing-function: ease-in;}
#div4 {animation-timing-function: ease-out;}
#div5 {animation-timing-function: ease-in-out;}
```

animation-fill-mode: property specifies a style for the target element when the animation is not playing (before it starts, after it ends, or both).

The **animation-fill-mode** property can have the following values:

- **none** - Default value. Animation will not apply any styles to the element before or after it is executing
- **forwards** - The element will retain the style values that is set by the last keyframe (depends on animation-direction and animation-iteration-count)

  ```
  div {
    width: 100px;
    height: 100px;
    background: red;
    position: relative;
    animation-name: myAnimation;
    animation-duration: 3s;
    animation-fill-mode: forwards;
  }
  ```

  The following example lets the element retain the style values from the last keyframe when the animation ends:

- **backwards** - The element will get the style values that is set by the first keyframe (depends on animation-direction), and retain this during the animation-delay period

  ```
  div {
    width: 100px;
    height: 100px;
    background: red;
    position: relative;
    animation-name: myAnimation;
    animation-duration: 3s;
  ```

```
 animation-delay: 2s;
 animation-fill-mode: backwards;
}
```

The following example lets the element get the style values set by the first keyframe before the animation starts (during the animation-delay period):

- **both** - The animation will follow the rules for both forwards and backwards, extending the animation properties in both directions

```
div {
 width: 100px;
 height: 100px;
 background: red;
 position: relative;
 animation-name: myAnimation;
 animation-duration: 3s;
 animation-delay: 2s;
 animation-fill-mode: both;
}
```

The following example lets the element get the style values set by the first keyframe before the animation starts, and retain the style values from the last keyframe when the animation ends:

CSS Animation Shorthand Property

animation: animation-name animation-duration animation-timing-function animation-delay animation-iteration-count animation-direction

```
div {
 animation: myAnimation 5s linear 2s infinite alternate;
}
```

animation-play-state: property specifies whether the animation is running or paused.

Note: Use this property in a JavaScript to pause an animation in the middle of a cycle.

CSS object-fit Property: It is used to specify how an <img> or <video> should be resized to fit its container.

This property can take one of the following values:

- fill - This is default. Does not preserve the aspect ratio. The image is resized to fill the container (the image will be stretched or squeezed to fit).
- cover - Preserves the aspect ratio, and the image fills the container. Cuts overflowing content if needed.
- contain - Preserves the aspect ratio, and fits the image inside the container, without cutting - leaves empty space if needed.
- none - The image is not resized.
- scale-down - the image is scaled down to the smallest version of none or contain.

object-fit: fill: value does not preserve the aspect ratio, and the image is resized to fill the container (the image will be stretched or squeezed to fit):

```
.image-container {
 width: 200px;
 height: 300px;
 border: 1px solid black;
 margin-bottom: 25px;
}

.image-container img {
 width: 100%;
 height: 100%;
 object-fit: fill;
}
```

object-fit: cover; value preserves the aspect ratio, and the image fills the container. The image will be clipped to fit:

```
.image-container {
 width: 200px;
 height: 300px;
 border: 1px solid black;
 margin-bottom: 25px;
}

.image-container img {
 width: 100%;
 height: 100%;
 object-fit: cover;
}
```

object-fit: contain; value preserves the aspect ratio, and fits the image inside the container, without cutting - will leave empty space if needed:

```
.image-container {
 width: 200px;
 height: 300px;
 border: 1px solid black;
 margin-bottom: 25px;
}
```

```
.image-container img {
  width: 100%;
  height: 100%;
  object-fit: contain;
}
```

object-fit: none; value does not resize or scale the image:

```
.image-container {
  width: 200px;
  height: 300px;
  border: 1px solid black;
  margin-bottom: 25px;
}

.image-container img {
  width: 100%;
  height: 100%;
  object-fit: none;
}
```

object-fit: scale-down; value scales the image down to the smallest version of none or contain:

```
.image-container {
  width: 200px;
  height: 300px;
  border: 1px solid black;
  margin-bottom: 25px;
}

.image-container img {
  width: 100%;
  height: 100%;
  object-fit: scale-down;
}
```

CSS object-position Property:

It is used together with the **object-fit** property to specify how an <img> or <video> should be positioned with x/y coordinates within its container.

The first value controls the x-axis and the second value controls the y-axis. The value can be a string (top,bottom,left, center or right), or a number (in px or %). Negative values are also allowed.

Using the object-position Property:

Let's say that the part of the image that is shown, is not the part that we want. To position the image, we will use the object-position property.

Here we position the image so that the great old building is in center:

```
.image-container {
 width: 200px;
 height: 300px;
 border: 1px solid black;
}

.image-container img {
 width: 100%;
```

```
 height: 100%;
 object-fit: cover;
 object-position: 80% 100%;
}
```

```
object fit: cover;

object-position: top right;
```

To just show the top-right portion of the image by fitting it inside the box.

CSS background-image Property:

background-image property sets one or more background images for an element.

By default, a background-image is placed at the top-left corner of an element, and repeated both vertically and horizontally.

Tip: The background of an element is the total size of the element, including padding and border (but not the margin).

Tip: Always set a background-color to be used if the image is unavailable.

```
body {
 background-image: url("paper.gif");
```

```
 background-color: #cccccc;
}
```

background-position: property sets the starting position of a background image.

By default, a background-image is placed at the top-left corner of an element

Value	Description
left top left center left bottom right top right center right bottom center top center center center bottom	If you only specify one keyword, the other value will be "center"
x% y%	The first value is the horizontal position and the second value is the vertical. The top left corner is 0% 0%. The right bottom corner is 100% 100%. If you only specify one value, the other value will be 50%. Default value is: 0% 0%

background-repeat: property sets if/how a background image will be repeated.

By default, a background-image is repeated both vertically and horizontally.

Value	Description
repeat	The background image is repeated both vertically and horizontally. The last image will be clipped if it does not fit. This is default
repeat-x	The background image is repeated only horizontally
repeat-y	The background image is repeated only vertically

no-repeat	The background-image is not repeated. The image will only be shown once
space	The background-image is repeated as much as possible without clipping. The first and last image is pinned to either side of the element, and whitespace is distributed evenly between the images
round	The background-image is repeated and squished or stretched to fill the space (no gaps)

background-clip: property defines how far the background (color, image, or gradient) should extend within an element.

border-box: Default value. The background extends behind the border

padding-box: The background extends to the inside edge of the border

content-box: The background extends to the edge of the content box

CSS filter property: It defines visual effects (like blur and saturation) to an element (often <img>).

Syntax:

filter: none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | saturate() | sepia() | url();

none: Default value. Specifies no effects

blur(*px*): Applies a blur effect to the image. A larger value will create more blur. If no value is specified, 0 is used.

brightness(*%*): Adjusts the brightness of the image. 0% will make the image completely black. 100% (1) is default and represents the original image. Values over 100% will provide brighter results. Values under 100% will provide darker results.

contrast(*%*): Adjusts the contrast of the image. 0% will make the image completely gray. 100% (1) is default, and represents the original image. Values over 100% increases the contrast. Values under 100% decreases the contrast.

grayscale(*%*):Converts the image to grayscale.0% (0) is default and represents the original image.100% will make the image completely grayscale

opacity(*%*):Sets the opacity level for the image. The opacity-level describes the transparency-level, where: 0% is completely transparent. 100% (1) is default and represents the original image (no transparency).

hue-rotate(*deg*): Applies a hue rotation on the image. The value defines the number of degrees around the color circle the image samples will be adjusted. 0deg is default, and represents the original image.

invert(%): Inverts the samples in the image.0% (0) is default and represents the original image.100% will make the image completely inverted.

saturate(%): Saturates the image.0% (0) will make the image completely un-saturated. 100% is default and represents the original image. Values over 100% provides super-saturated results.

sepia(%): Converts the image to sepia. 0% (0) is default and represents the original image.100% will make the image completely sepia.