**Font-fallback**

Always include a **fallback** like sans-serif, serif, or monospace to ensure your site looks okay even if the main font isn't available. If Roboto is not available, it falls back to any **generic** Helvetica font.

Example:

body {

font-family: "Roboto", "Helvetica", "Arial", sans-serif;

}

---------------------------------------------------------------------------------------------------------------------

**Padding:**

An element's padding controls the amount of space between the element's content and its border

Instead of specifying an element's padding-top, padding-right, padding-bottom, and padding-left properties individually, you can specify them all in one line, like this:

padding: 10px 20px 10px 20px;

These four values work like a clock: top, right, bottom, left, and will produce the exact same result as using the side-specific padding instructions.

**Margin:**

An element's margin controls the amount of space between an element's border and surrounding elements.

**Border:**

A **border** in CSS is the edge or outline that surrounds the padding and content of an element’s box, visually separating it from other elements. It can have a specific width, style, and color, and can be applied to all sides or to individual sides of the element.

---------------------------------------------------------------------------------------------------------------------

**Use Attribute Selectors to Style Elements**

You have been adding id or class attributes to elements that you wish to specifically style. These are known as ID and class selectors. There are other CSS Selectors you can use to select custom groups of elements to style.

This selector matches and styles elements with a specific attribute value. For example, the below code changes the margins of all elements with the attribute type and a corresponding value of radio:

[type='radio'] {

margin: 20px 0px 20px 0px;

}

**Style precedence**

Id takes precedence over class

<style>

  .pink-text {

    color: pink;

  }

   #orange-text{

  color:orange;

  }

  .blue-text {

    color: blue;

  }

</style>

<h1 class="pink-text blue-text" id="orange-text">Hello World!</h1>

**Note:** It doesn't matter whether you declare this CSS above or below pink-text class, since the id attribute will always take precedence.

Note: **Inline styles override Above all ways**

**Note:** **Let's add the keyword !important to your pink-text element's color declaration to make 100% sure that your h1 element will be pink.**

**keyword !important  - Overrides everything**

**Browser default < Element < Class < ID < Inline style < !important**

**Color formats**

**✅ 1. Named Colors**

color: white;

* **Simple** and easy to read.
* Limited to **140 predefined** color names (e.g., red, blue, white, black, etc.).
* Not ideal if you need **precise or custom shades**.

**✅ 2. Hexadecimal (Hex) Codes**

color: #FFFFFF; /\* White \*/

* Starts with #, followed by **6 characters** (hex digits for Red, Green, Blue).
  + Format: #RRGGBB
* Short form also available: #FFF (same as #FFFFFF for white).
* Offers **16 million+ color possibilities**.
* Commonly used in web design.
* Harder to read than named colors unless you’re used to it.

**✅ 3. RGB Function**

color: rgb(255, 255, 255); /\* White \*/

* Stands for **Red, Green, Blue** — each value ranges from 0 to 255.
* Gives **precise control** over color.
* Easier to tweak than hex if you're adjusting brightness or transparency.
* You can also use **rgba** for transparency:

color: rgba(255, 255, 255, 0.5); /\* Semi-transparent white \*/

**:root and Pseudo-Classes in CSS**

**✅ What is a Pseudo-Class?**

A **pseudo-class** is a keyword in CSS that describes a **special state** of an element — something you can’t select just by element name, class, or ID.

**🔹 Common Examples of Pseudo-Classes:**

| **Pseudo-Class** | **Description** |
| --- | --- |
| :hover | When the mouse hovers over an element |
| :focus | When an input is focused |
| :nth-child(n) | Targets the nth child of a parent |
| :first-child | Targets the first child |
| :visited | Applies to visited links |
| :not(selector) | Targets everything *except* the given selector |
| :root | Targets the root (<html>) element of the page |

**✅ What is :root?**

* :root is a **special pseudo-class** that always targets the **topmost element** of the document: the <html> tag.
* Unlike other pseudo-classes (like :hover or :focus), :root is **not state-based** — it’s always active.

**🔍 Why Use :root?**

* It’s the **best place to declare global CSS variables** that can be reused throughout your stylesheet.
* CSS variables declared in :root are **available everywhere** in your CSS.

**📌 Example:**

:root {

--main-bg: #f0f0f0;

--font-color: #333;

}

body {

background-color: var(--main-bg);

color: var(--font-color);

}

**✅ Summary: :root vs Other Pseudo-Classes**

| **Feature** | **:root** | **Other Pseudo-Classes (e.g., :hover, :focus)** |
| --- | --- | --- |
| Targets | <html> element | Elements in a specific **state** |
| Always active? | ✅ Yes | ❌ No (depends on state) |
| Common usage | Defining global variables | Interactivity, conditional styling |
| Specificity | Slightly higher than html selector | Varies |

**✅ Difference Between Pseudo-Class and Pseudo-Element**

| **Feature** | **Pseudo-Class (:)** | **Pseudo-Element (::)** |
| --- | --- | --- |
| Syntax | Uses a single colon : | Uses two colons :: |
| Purpose | Targets a **state** or condition | Creates **virtual elements** before or after content |
| Examples | :hover, :focus, :root | ::before, ::after, ::first-letter |
| Real Element? | No | Yes, acts like an extra (invisible) element |

Pseudo-elements like ::before and ::after **must** have the content property.

eg

h1::before {

content: "★ ";

color: gold;

}

**Use the s Tag to Strikethrough Text**

To strikethrough text, which is when a horizontal line cuts across the characters, you can use the s tag. It shows that a section of text is no longer valid. With the s tag, the browser applies the CSS of text-decoration: line-through; to the element.

**📦 General Syntax of box-shadow**

box-shadow: offset-x offset-y blur-radius spread-radius color;

* offset-x: horizontal shadow position (right if positive, left if negative)
* offset-y: vertical shadow position (down if positive, up if negative)
* blur-radius: how blurry the shadow is
* spread-radius: (optional) how much the shadow grows or shrinks
* color: shadow color (can use rgba for transparency)

**Creating multiple Box shadow**

**Eg:**

**box-shadow**: 0 10px 20px rgba(0, 0, 0, 0.19),

0 6px 6px rgba(0, 0, 0, 0.23);

**1. First Shadow**

css

0 10px 20px rgba(0, 0, 0, 0.19)

* 0 → No horizontal offset (shadow is centered horizontally)
* 10px → 10 pixels below the element
* 20px → Blur radius (soft shadow edges)
* rgba(0,0,0,0.19) → Black with 19% opacity (very light)

➡️ This creates a **large, soft, faint shadow** below the element.

**2. Second Shadow**

css

0 6px 6px rgba(0, 0, 0, 0.23)

* 0 → Centered horizontally
* 6px → 6 pixels below the element
* 6px → Smaller blur
* rgba(0,0,0,0.23) → Slightly darker (23% opacity)

➡️ This creates a **smaller, darker shadow** closer to the element.

**Combined Effect**

By layering these two shadows:

* You get a **realistic depth effect** — soft outer glow + tighter inner shadow.
* Commonly used in **Material Design** buttons/cards for subtle elevation.

**Opacity**

The opacity property in CSS is used to adjust the opacity, or conversely, the transparency for an item.

A value of 1 is opaque, which isn't transparent at all.  
A value of 0.5 is half see-through.  
A value of 0 is completely transparent.

 Valid values are **from 0 to 1**.

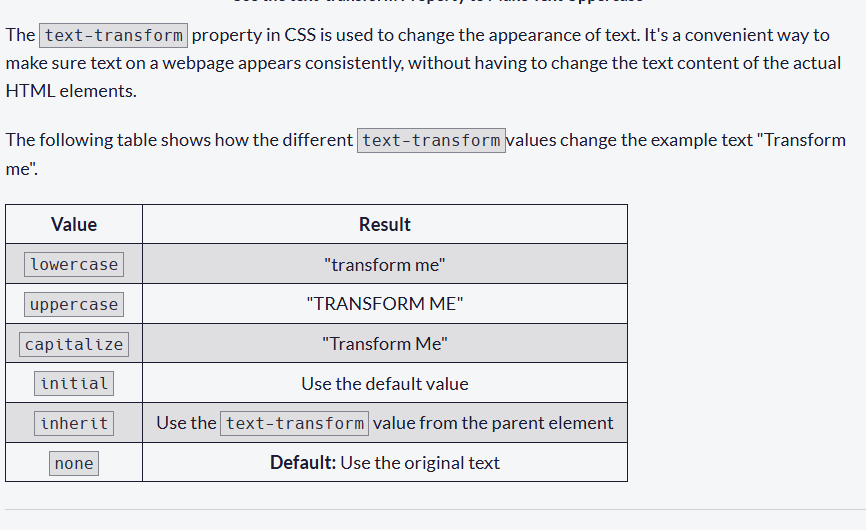
 It accepts **decimal values**, like:

* 0.25 (25% opaque, 75% transparent)
* 0.5 (50/50)
* 0.75 (mostly visible)

**🚫 Units:**

* ❌ **No units** are used. Just the number.
* **Incorrect:** opacity: 50%; ❌
* **Correct:** opacity: 0.5; ✅

**Text-transform**



Here are some common **font-weight** values:

| **Weight** | **Number** | **Keyword** |
| --- | --- | --- |
| Thin | 100 | — |
| Extra Light | 200 | — |
| Light | 300 | — |
| Normal | 400 | normal |
| Medium | 500 | — |
| Semi-Bold | 600 | — |
| Bold | 700 | bold |
| Extra-Bold | 800 | — |
| Black | 900 | — |
|  |  |  |

**Difference Between position and margin:**

* **position** (e.g., relative, absolute) changes **how and where an element is placed** in the layout. With position: relative, the element can be moved visually using top, left, etc., **without affecting surrounding elements**.
* **margin** adds **space outside** an element, pushing it away from other elements. It **does affect layout flow** and spacing between elements.

In short:  
👉 position changes **where** the element appears,  
👉 margin changes **how much space** is around it.

**1. Why give position: relative to a parent without top, left, etc.?**

Even if you **don’t move the parent**, setting position: relative on it **creates a positioning context**.

That means:

**Any absolutely positioned child (position: absolute) will now be positioned *relative to the parent*** — instead of the whole page.

**🔧 Example:**

<div class="box">

<div class="tooltip">I'm inside!</div>

</div>

.box {

position: relative; /\* Not moving it, just setting context \*/

}

.tooltip {

position: absolute;

top: 0;

right: 0;

}

✅ The tooltip appears at the top-right **of the box**, not the whole page.

-------------------------------------------------

**🔗 2. Other things related to position**

There are **five main position values** in CSS:

| **Value** | **Description** |
| --- | --- |
| static | Default. Normal flow. Can’t use top, left, etc. |
| relative | Moves the element visually but **keeps its space**. Creates a reference for absolute children. |
| absolute | Removes element from normal flow. Positions it **relative to the nearest non-static parent**. |
| fixed | Positions element **relative to the viewport**. It stays fixed when scrolling. |
| sticky | Acts like relative until you scroll to a certain point — then it "sticks". |

**🧠 Summary:**

* Use position: relative on a parent to **let absolute children use it as a reference**.
* top, left, etc., **only move the element** if you want it to.
* position controls **how an element is placed** and **what it reacts to**.
* margin controls **spacing** between elements — not placement behavior.

**Difference Between position: relative and position: absolute in CSS**

**1. position: relative**

* The element is positioned **relative to its original position** in the normal document flow.
* It still **occupies space** in the layout as if it wasn't moved.
* The top, left, right, and bottom properties **shift the element from its original position**, without affecting other elements.

**Example:**

css

.relative-box {

position: relative;

top: 20px;

left: 30px;

}

This moves the element **20px down and 30px to the right**, but **other elements still behave as if it's in its original place**.

**2. position: absolute**

* The element is **removed from the normal document flow**.
* It **does not take up space** in the layout, meaning other elements behave as if it doesn't exist.
* It is positioned **relative to the nearest ancestor** with position: relative, absolute, or fixed. If no such ancestor exists, it's positioned relative to the <html> element.

**Example:**

css

.absolute-box {

position: absolute;

top: 10px;

left: 50px;

}

This places the element **exactly 10px from the top and 50px from the left** of its positioned ancestor or the page.

**Key Differences**

| **Feature** | **position: relative** | **position: absolute** |
| --- | --- | --- |
| Positioned relative to | Its original position | Nearest positioned ancestor or the page |
| Takes up layout space | ✅ Yes | ❌ No |
| Affects other elements | ✅ Yes | ❌ No |
| Part of document flow | ✅ Yes | ❌ No |

Note: If you forget to add a position rule to the parent item, (this is typically done using position: relative;), the browser will keep looking up the chain and ultimately default to the body tag.

**Float**

The float property is a **positioning tool** that doesn't rely on the position property. When an element is floated, it is **removed from the normal document flow** and pushed to the **left or right** of its containing element.

Floated elements are often used in layouts to allow text or other elements to **wrap around them**. To control how much space a floated element takes up, it’s commonly paired with the width property.

**✅ Example:**

css

.image-left {

float: left;

width: 200px;

margin-right: 20px;

}

Html

<img src="photo.jpg" class="image-left">

<p>This text will wrap around the floated image.</p>

**🔁 Key Characteristics of float:**

* **Removes the element from normal flow** (but not entirely like position: absolute).
* **Allows inline content (like text) to flow around it**.
* **Needs to be "cleared"** if you want following elements to stop wrapping around the float.

-------------------------------------

**💡 Tip:**

To prevent layout issues with floated elements, it's common to use **clearfix** techniques or add overflow: hidden; or display: flow-root; to the container.

**HSL Color in CSS3**

CSS3 introduced the hsl(hue,saturation,lightness) function to define colors using:

* **Hue**: The type of color, represented as an angle (0–360°) on a color wheel.
  + 0° = red, 120° = green, 240° = blue.
* **Saturation**: Intensity of the color, given as a percentage.
  + 100% = full color, 0% = gray.
* **Lightness**: Brightness of the color, from 0% (black) to 100% (white).
  + 50% = normal color.

**Examples:**

hsl(0, 100%, 50%) /\* Red \*/

hsl(120, 100%, 50%) /\* Green \*/

hsl(240, 100%, 50%) /\* Blue \*/

**CSS Linear Gradient**

CSS allows smooth color transitions using the linear-gradient() function with the background property.

**Syntax:**

background: linear-gradient(direction, color1, color2, ...);

* **Direction**: e.g., 90deg (left to right), 45deg (diagonal).
* **Colors**: Listed in the order they appear in the gradient.

**Example:**

background: linear-gradient(90deg, red, yellow, green);

This creates a horizontal gradient from red to yellow to green.

 background: linear-gradient(35deg,#CCFFFF,#FFCCCC);

repeating-linear-gradient()

repeating-linear-gradient(angle, color1 position1, color2 position2, color3 position3, color4 position4, ...);

**📝 Explanation:**

* **angle**: Direction of the gradient (e.g., 45deg, to right, etc.).
* **color1 position1**: Start with color1 at position1.
* **color2 position2**: Blend from color1 to color2 until position2.
* **color3 position3**: At this point, change immediately to color3.
* **color4 position4**: Blend from color3 to color4 until position4.

Then the entire pattern **repeats**.

**✅ Example:**

repeating-linear-gradient(45deg, yellow 0px, blue 40px, green 40px, red 80px);

🔸 This means:

* **Yellow → Blue** from 0px to 40px
* **Instant Green** at 40px
* **Green → Red** from 40px to 80px
* Then the same pattern repeats every 80px

**🎨 Add a Subtle Pattern Texture to the Page Background**

To enhance the visual interest of your webpage without distracting from the main content, you can apply a subtle repeating pattern as the background. This is done using the background property with the url() function, which loads an external image.

Here's how to apply a soft texture to the entire page:

body {

background: url("https://cdn-media-1.freecodecamp.org/imgr/MJAkxbh.png");

}

**📝 Explanation:**

* **body** – Targets the whole page.
* **background** – Applies the image as a background.
* **url("...")** – Loads the image from the given web address.
* The image **automatically repeats** to fill the entire page, creating a seamless pattern.

This technique gives your layout a gentle textured feel without overpowering the foreground elements.

**Scale an Element Using the CSS transform Property**

To make the element with id="ball2" grow to 1.5 times its original size, use:

#ball {

transform: scale(1.5);

}

**📝 Explanation:**

* #ball: Targets the element with the id of ball2.
* transform: scale(1.5): Enlarges the element to **150%** of its original width and height.

This change keeps the element centered on its original position unless additional transform-origin or positioning is applied.

**Skewing an Element Along the X-Axis with CSS**

The CSS transform property can change the shape and position of an element in 2D or 3D space.  
Using the skewX() function, you can tilt (skew) an element horizontally by a specified number of degrees.

**Syntax:**

transform: skewX(angle);

* **angle**: The degree to skew along the X-axis. Positive values skew clockwise; negative values skew counterclockwise.

**Example:** Skew the element with the id bottom by 24 degrees along the X-axis.

#bottom {

transform: skewX(24deg);

}

**Effect:**  
This will visually slant the element to the right, creating a dynamic, angled appearance without changing its actual layout in the page flow.

Note : skewY(angle); works on y axis

**CSS Animation**

CSS animations allow you to create smooth transitions between styles without needing JavaScript.  
They work by combining **animation properties** with the @keyframes rule.

**1.Essential Animation Properties**

| **Property** | **Purpose** |
| --- | --- |
| animation-name | The name of the animation (must match the name in @keyframes). |
| animation-duration | How long the animation runs (e.g., 2s, 500ms). |

**Example:**

css

.box {

animation-name: fadeIn;

animation-duration: 2s;

}

**2. The @keyframes Rule**

* Defines *what happens* during the animation.
* You specify styles for different **frames** of the animation, usually at certain percentages (from 0% to 100%).
* You can set multiple stops, not just start and end.

**Example:**

css

@keyframes fadeIn {

0% {

opacity: 0;

}

100% {

opacity: 1;

}

}

In @keyframes, the percentages are **relative to the total animation duration** you set with animation-duration.

So:

* 0% = the very start (time = 0s)
* 100% = the very end (time = animation-duration)
* 32% = 32% of the total duration (time = 0.32 × animation-duration)

Here, the animation starts invisible (opacity: 0) and ends fully visible (opacity: 1).

**3. Additional Animation Properties**

1. **animation-delay**
   * How long to wait before starting the animation.
   * Example: animation-delay: 1s;
2. **animation-iteration-count**
   * Number of times the animation repeats.
   * Values: a number (e.g., 3) or infinite.
   * Example: animation-iteration-count: infinite;
3. **animation-direction**
   * Direction the animation plays:
     + normal (default) — runs from 0% to 100%.
     + reverse — runs from 100% to 0%.
     + alternate — runs forward, then backward.
     + alternate-reverse — runs backward, then forward.
4. **animation-timing-function**
   * Controls speed over time.
   * Common values:
     + linear — constant speed.
     + ease — slow start, fast middle, slow end (default).
     + ease-in — slow start, faster end.
     + ease-out — fast start, slow end.
     + ease-in-out — slow start & slow end.
     + Custom: cubic-bezier(x1, y1, x2, y2).
5. **animation-fill-mode**
   * Defines how styles apply before/after animation:
     + none — no style outside animation time.
     + forwards — keeps final style after animation ends.
     + backwards — applies starting style before animation starts.
     + both — applies both before and after.
6. **animation-play-state**
   * Controls whether animation is running or paused.
   * Values: running (default), paused.

**4. Shorthand Syntax**

You can combine multiple animation properties into one line:

.box {

animation: fadeIn 2s ease-in-out 1s infinite alternate both;

}

This covers:

* Name: fadeIn
* Duration: 2s
* Timing function: ease-in-out
* Delay: 1s
* Iteration count: infinite
* Direction: alternate
* Fill mode: both

**Use tabindex to Add Keyboard Focus**

The HTML tabindex attribute controls whether and how an element receives **keyboard focus**.

* **Positive, zero, or negative integers** determine the focus behavior.
* Many elements (links, buttons, inputs) automatically receive focus in the **order they appear** in the HTML.
* You can make non-focusable elements (like <div>, <span>, <p>) focusable by giving them tabindex="0".

**Example:**

html

<p tabindex="0">

Please read these instructions carefully before filling out the survey.

</p>

📝 **Key Points:**

* tabindex="0" — element becomes keyboard-focusable in normal tab order.
* tabindex="-1" — element is focusable but **not** reachable via keyboard tabbing; focus must be set programmatically (e.g., for modals).
* Once focusable, the element supports the CSS :focus pseudo-class for styling.

**Create a Media Query**

Media Queries in CSS allow you to change styles depending on the device’s **viewport size**. They use conditions like max-width, min-width, max-height, or min-height to apply styles only when those conditions are met.

**Example — set <p> font size to 10px when device height ≤ 800px:**

css

@media (max-height: 800px) {

p {

font-size: 10px;

}

}

📝 **Key Points:**

* @media (max-height: 800px) — targets devices where the visible height is **800px or smaller**.
* You can place any number of CSS selectors and rules inside the media query.
* Styles inside a media query override normal styles **only** when the condition is true.

**Make Typography Responsive**

Instead of using fixed units like px or em for text size, you can use **viewport units** so the size changes with the screen.

* **vw** — percent of the screen’s width.
* **vh** — percent of the screen’s height.
* **vmin** — percent of the smaller side of the screen (width or height).
* **vmax** — percent of the bigger side of the screen (width or height).

**Example:**

css

h2 {

width: 80vw; /\* 80% of the screen’s width \*/

}

p {

width: 75vmin; /\* 75% of the smaller side of the screen \*/

}

Note: It is very useful to define overall outside container width or height

This way, your text stays easy to read on phones, tablets, and computers.