CSS (Cascading Style Sheets) CSS is a stylesheet language used to describe the presentation of HTML (or XML) documents. It controls the layout, colors, fonts, spacing, and overall visual appearance of web pages. By separating content (HTML) from presentation (CSS), you achieve cleaner markup, easier maintenance, and more flexible designs.

---

1. Why Use CSS?

- **Separation of Concerns** Keeps HTML focused on structure and semantics, while CSS handles styling.
- **Reusability** One stylesheet can style an entire website, reducing duplication.
- **Maintainability** Changes in design require edits in one place rather than every HTML file.
- **Performance** Browsers cache CSS files, speeding up page loads.
- **Responsiveness** Media queries and flexible units allow designs to adapt to different screen sizes.

---

2. Types of CSS Placement

1. **Inline CSS**

```
<p style="color: red; font-size: 16px;">Hello World</p>
```

   - Styles apply to a single element.
   - High specificity but poor maintainability.

2. **Internal (Embedded) CSS**

```
<head>
  <style>
    p { color: blue; }
  </style>
</head>
```

   - Defined within `<style>` tags in the document's `<head>`.
   - Good for single-page styles, but not across multiple pages.

3. **External CSS**

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

   - Separate `.css` file linked to HTML.
   - Best practice for large sites; allows central control.

---

3. CSS Syntax

```
selector {
  property: value;
  /* more declarations… */
}
```

- **Selector**: targets HTML elements (e.g., `h1`, `.nav`, `#main`)
- **Property**: the style attribute to set (e.g., `color`, `margin`)
- **Value**: the setting for the property (e.g., `#333`, `20px`)

---

4. Selectors

4.1 Basic Selectors

- **Element Selector**

```
p { /* targets all <p> elements */ }
```

- **Class Selector**

```
.highlight { /* targets elements with class="highlight" */ }
```

- **ID Selector**

```
#header { /* targets element with id="header" */ }
```

4.2 Grouping Selectors

Apply the same rules to multiple selectors by separating with commas:

```
h1, h2, h3 { margin-bottom: 10px; }
```

4.3 Combinators

- **Descendant** (space):

```
.nav li a { /* <a> inside an <li> inside .nav */ }
```

- **Child** (>):

```
.container > p { /* only direct children <p> of .container */ }
```

- **Adjacent Sibling** (+):

```
h2 + p { /* <p> immediately after an <h2> */ }
```

- **General Sibling** (~):

```
h2 ~ p { /* all <p> siblings after an <h2> */ }
```

4.4 Other Selectors

- **Universal Selector**

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

- **Attribute Selector**

```
input[type="text"] { width: 100%; }
```

---

5. Properties & Values

5.1 Color Formats

- **Named Colors**:

```
color: crimson;
```

- **Hexadecimal**:

```
color: #ff0000;    /* red */
color: #f00;       /* shorthand */
```

- **RGB / RGBA**:

```
color: rgb(255, 0, 0);
background: rgba(0, 0, 0, 0.5); /* 50% opacity */
```

- **HSL / HSLA**:

```
color: hsl(120, 50%, 50%);      /* green */
color: hsla(120, 50%, 50%, 0.3); /* translucent */
```

5.2 Units

- **Absolute**
  - `px` (pixels)
  - `cm`, `mm`, `in` (less common)
- **Relative**
  - **Font-relative:**
    - `em` (relative to parent’s font-size)

        - `rem` (relative to root's font-size)
    - **Viewport-relative:**
        - `vw` (1% of viewport width)
        - `vh` (1% of viewport height)
    - **Percentage (%)**: relative to parent container.

---

6. Text Styling

```
/* Font family, size, weight, and line height */
body {
  font-family: 'Helvetica Neue', Arial, sans-serif;
  font-size: 16px;       /* base font size */
  font-weight: 400;      /* normal */
  line-height: 1.5;      /* spacing between lines */
}

/* Text alignment and transformation */
h1 {
  text-align: center;    /* left, center, right, justify */
  text-transform: uppercase; /* none, capitalize, uppercase, lowercase */
  letter-spacing: 0.05em;    /* spacing between letters */
}

/* Text decoration */
a {
  text-decoration: none;     /* none, underline, overline, line-through */
}
a:hover {
  text-decoration: underline;
}
```

---

7. Best Practices & Tips

- **DRY Principle**: Don't Repeat Yourself—reuse classes and variables.
- **CSS Variables (Custom Properties)**

  ```
  :root {
    --primary-color: #3498db;
    --spacing-unit: 8px;
  }
  .button {
    background: var(--primary-color);
    padding: calc(var(--spacing-unit) * 2);
  }
  ```

- **Normalize or Reset** styles to achieve consistency across browsers.
- **Mobile-First** approach: write base styles for small screens, then use media queries for larger breakpoints.

```
@media (min-width: 768px) {
  .sidebar { display: block; }
}
```

- **Organize** your stylesheet with comments and logical sections (e.g., Base, Layout, Components).

---

1. The CSS Box Model

Every HTML element is a rectangular box composed of four areas, from inside out:

```
[ margin ]
[ border ]
[ padding ]
[ content ]
```

- **Content**: The actual text, image, or other media.
- **Padding**: Transparent space between content and border. Increases the “clickable” or “readable” area without changing the border.
- **Border**: The line surrounding padding and content. Styleable with `border-width`, `border-style`, and `border-color`.
- **Margin**: Transparent space outside the border, pushing neighboring elements away.

1.1 How It Affects Layout

- **Total element size** = content size + padding + border + margin.
- Default `box-sizing: content-box` means width/height only include the content area.
- Using `box-sizing: border-box` makes width/height include padding and border, simplifying responsive layouts.

```
/* Default behavior */
.box {
  width: 200px;           /* content width only */
  padding: 20px;          /* +40px total horizontal */
  border: 5px solid #000; /* +10px total horizontal */
  margin: 10px;           /* external spacing */
}

/* Border-box makes sizing easier */
.box-border {
  box-sizing: border-box;
  width: 200px;
  padding: 20px;
  border: 5px solid #000;
  /* Total width remains exactly 200px */
}
```

---

2. Display Values

Controls how an element generates boxes and interacts with its siblings:

Value	Description
`block`	Starts on a new line, takes full available width. Can set width/height, margin, padding.
`inline`	Flows within text, width/height ignored, only horizontal padding/margins apply, no line break.
`inline-block`	Like inline, but you can set width/height and vertical padding/margin; sits inline with text.
`none`	Element is not rendered (no box, no space).
`flex`	Creates a flex container, enabling the Flexbox layout model for its children.

```
<div class="block">Block</div>
<span class="inline">Inline</span>
<div class="inline-block">Inline-Block</div>
<div class="hidden">Hidden</div>
<div class="flex">Flex Container</div>
```

---

3. Positioning

Defines how an element is placed in the document flow:

Position	Behavior
`static`	Default. Flows normally in the document. Offsets (`top`, `left`) don't apply.
`relative`	Remains in flow, but you can offset it relative to its normal position (`top`, `left`, etc.).
`absolute`	Removed from flow, positioned relative to the nearest ancestor with non-`static` position, or viewport.
`fixed`	Removed from flow, fixed relative to the viewport, stays in place on scroll.
`sticky`	Acts like `relative` until scroll position crosses a threshold, then acts like `fixed`.

```
/* Relative example */
.relative-box {
  position: relative;
  top: 10px; left: 20px;
}

/* Absolute example */
.container {
  position: relative;
}
.absolute-box {
  position: absolute;
```

```
  bottom: 0; right: 0;
}

/* Fixed example */
.fixed-box {
  position: fixed;
  top: 0; left: 0;
}

/* Sticky example */
.sticky-box {
  position: sticky;
  top: 10px;
}
```

---

4. z-Index & Stacking Context

- **z-index** controls the stacking order of positioned elements (`position` ≠ `static`).
- Higher `z-index` appears on top.
- A new **stacking context** is created by elements that are positioned (`relative`, `absolute`, `fixed`, `sticky`) with a non-`auto` `z-index`, or by certain CSS properties (e.g., `opacity` < 1, `transform`, `filter`).
- Within each context, child elements stack independently of outside elements.

```
.parent {
  position: relative;
  z-index: 10;          /* establishes stacking context */
  background: white;
}
.child {
  position: absolute;
  z-index: 5;           /* stacks within .parent only */
  background: lightblue;
}
```

---

5. Flexbox Layout

Flexbox provides an efficient way to layout, align, and distribute space among items in a container.

5.1 Container Properties

```
.flex-container {
  display: flex;               /* establishes flex formatting context */
  flex-direction: row;         /* main axis: row | row-reverse | column |
column-reverse */
  justify-content:            /* distribution along main axis */
    flex-start | flex-end | center | space-between | space-around | space-
evenly;
  align-items:               /* alignment along cross axis */
    stretch | flex-start | flex-end | center | baseline;
  flex-wrap: nowrap | wrap | wrap-reverse; /* controls wrapping of flex
```

```
items */
}
```

- **`flex-direction`**:
    - `row` (default): items laid out left to right.
    - `column`: top to bottom.
- **`justify-content`**: distributes any extra free space along the main axis.
- **`align-items`**: aligns items within the container's cross axis.
- **`flex-wrap`**: allows items to wrap to the next line when they overflow.

5.2 Item Properties

```
.flex-item {
  flex-grow: 0;    /* ability to grow relative to siblings (default 0) */
  flex-shrink: 1;  /* ability to shrink if needed (default 1) */
  flex-basis: auto;/* default size before growing/shrinking (can be length
or % ) */
  align-self:      /* override container's align-items for this item */
    auto | stretch | flex-start | flex-end | center | baseline;
  order: 0;        /* controls visual order (default 0) */
}
```

- **`flex` shorthand**:

```
.flex-item {
  flex: [flex-grow] [flex-shrink] [flex-basis];
  /* e.g. flex: 1 0 200px; */
}
```

- **`align-self`** lets individual items override the container's cross-axis alignment.

---

6. Putting It All Together

```
<div class="card-container">
  <div class="card">Card A</div>
  <div class="card">Card B</div>
  <div class="card special">Card C</div>
</div>
```

```
.card-container {
  display: flex;
  flex-direction: row;
  justify-content: space-around;
  align-items: center;
  padding: 20px;
  border: 2px solid #ccc;
  box-sizing: border-box;
}
```

```
.card {
  flex: 1 1 150px;          /* grow, shrink, base size */
  margin: 10px;             /* outer spacing */
  padding: 15px;            /* inner spacing */
  border: 1px solid #999;
  box-sizing: border-box;
  position: relative;
}

.special {
  z-index: 2;               /* brought to front within stacking context */
  top: -10px;               /* slight overlap */
  flex-grow: 2;             /* takes up more space */
  background: #f9f9f9;
}
```

---

1. CSS Grid Layout

CSS Grid is a two-dimensional layout system that lets you define rows and columns, and place items precisely within that grid.

1.1 Creating a Grid Container

```
.grid-container {
  display: grid;                          /* establishes grid formatting
context */
  grid-template-columns:                 /* defines column tracks */
    200px 1fr 2fr;                        /* fixed, flexible, flexible */
  grid-template-rows:                    /* defines row tracks (optional) */
    auto 300px;
  grid-gap: 20px;                         /* shorthand for row-gap and column-
gap */
  /* or use */
  column-gap: 20px;
  row-gap: 10px;
}
```

- **`grid-template-columns`**
  - You can specify fixed units (`px`, `em`), flexible fractions (`fr`), percentages, or the `repeat()` function:

    ```
    grid-template-columns: repeat(3, 1fr);       /* three equal
    columns */
    grid-template-columns: 100px auto 2fr 1fr;    /* mix of fixed and
    flexible */
    ```

- **`grid-gap`** (now `gap`)
  - Controls spacing between both rows and columns.
  - You can also specify them separately:

    ```
    gap: 10px 20px;    /* row-gap column-gap */
    ```

1.2 Positioning Grid Items

By default, items are placed in the next available grid cell, but you can explicitly position them:

```
/* Place item to span from column 2 to 4, and row 1 to 3 */
.grid-item {
  grid-column: 2 / 4;   /* start at grid line 2, end before grid line 4 */
  grid-row: 1 / 3;      /* start at line 1, end before line 3 */
}

/* Shorthand */
.grid-item {
  grid-area: 1 / 2 / 3 / 4; /* row-start / col-start / row-end / col-end */
}
```

- **Auto-placement** If you omit positioning, the grid auto-places items in source order.

---

2. Pseudo-Classes

Pseudo-classes target elements based on user interaction or document structure.

Pseudo-class	Description
`:hover`	When the user hovers over the element with a pointer.
`:active`	While the element is being activated (e.g., clicked).
`:nth-child(n)`	Selects elements based on their position in a parent.

```
button:hover {
  background: #555;
  color: white;
}

a:active {
  color: red;
}

li:nth-child(odd) {
  background: #f0f0f0;
}

li:nth-child(3n) {
  font-weight: bold;
}
```

- **Formula syntax** for `:nth-child()`
    - `an + b` e.g. `2n+1` selects odd, `3n` every third.

---

3. Pseudo-Elements

Pseudo-elements let you style parts of an element's content.

Pseudo-element	Description
`::before`	Insert content **before** an element's content.
`::after`	Insert content **after** an element's content.

```
h2::before {
  content: "§ ";
  color: #999;
}

p::after {
  content: " — Read more";
  font-style: italic;
}
```

- Always include a `content` property.
- Can be used for decorative icons, quotes, clearfix hacks, etc.

---

4. Responsive Design & Media Queries

Media queries allow you to apply styles based on viewport characteristics.

```
/* Mobile-first: base styles for small screens */
.container {
  padding: 1rem;
  font-size: 1rem;
}

/* Tablet and up */
@media (min-width: 600px) {
  .container {
    padding: 2rem;
    font-size: 1.125rem;
  }
}

/* Desktop and up */
@media (min-width: 1024px) {
  .container {
    padding: 3rem;
    font-size: 1.25rem;
  }
}
```

4.1 Mobile-First vs Desktop-First

- **Mobile-First**: Write base styles targeting small screens; add `min-width` breakpoints for larger devices.
- **Desktop-First**: Write base styles for large screens; use `max-width` queries to adjust for smaller devices.

Recommendation: Mobile-first is more future-proof as it naturally degrades upward.

---

5. Relative Units for Responsiveness

Unit	Relative to…	Use Cases
`%`	Parent element's size	Fluid widths/heights within containers
`em`	Computed font-size of the element itself	Sizing elements relative to text size
`rem`	Computed font-size of the root (`<html>`)	Consistent, scalable spacing and typography
`vw`	1% of the viewport's width	Full-width layouts, responsive typography
`vh`	1% of the viewport's height	Full-height sections, hero images

```
body {
  font-size: 16px;     /* base for rem calculations */
}

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

.container {
  width: 80%;          /* 80% of parent */
  margin: 2rem auto;   /* 32px top/bottom, centered */
}

.hero {
  height: 100vh;       /* full viewport height */
  padding: 5vh;        /* responsive vertical padding */
}
```

- **Why use relative units?**
    - They scale with user settings (accessibility).
    - Simplify proportional layouts.
    - Help maintain consistency across breakpoints.

Putting It Into Practice

```
<div class="grid-container">
  <header class="header">Header</header>
  <nav class="nav">Navigation</nav>
  <main class="main">Main Content</main>
  <aside class="aside">Sidebar</aside>
  <footer class="footer">Footer</footer>
</div>
```

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

.grid-container {
  display: grid;
  grid-template-columns: 1fr 3fr;
  grid-template-rows: auto 1fr auto;
  gap: 1rem;
  min-height: 100vh;
}

.header   { grid-area: 1 / 1 / 2 / 3; background: #eee; }
.nav      { grid-area: 2 / 1 / 3 / 2; background: #ddd; }
.main     { grid-area: 2 / 2 / 3 / 3; background: #fff; }
.aside    { grid-area: 2 / 3 / 3 / 4; background: #f9f9f9; }
.footer   { grid-area: 3 / 1 / 4 / 3; background: #ccc; }

.grid-container > * {
  padding: 1rem;
  box-sizing: border-box;
}

@media (max-width: 768px) {
  .grid-container {
    grid-template-columns: 1fr;
    grid-template-rows: auto repeat(3, 1fr) auto;
  }
}
```

- **Grid** defines a two-column layout on desktops and collapses to a single column on tablets/smaller devices via a media query.
- Use **relative units** (`rem`, `%`, `vh`) for scalable spacing.
- Add **pseudo-classes** for interactivity and **pseudo-elements** for decorative accents as needed.

---

1. CSS Transitions

Transitions allow you to animate the change of CSS properties over time when they switch from one state to another (e.g., on `:hover`).

```
.selector {
  /* Define which property(ies) to animate, how long, and easing */
  transition-property: background-color, transform;
  transition-duration: 0.3s, 0.5s;
  transition-timing-function: ease-in, cubic-bezier(0.25, 0.1, 0.25, 1);
  transition-delay: 0s, 0.2s;  /* optional: staggered start */
}

/* Shorthand */
.selector {
  transition: background-color 0.3s ease-in-out 0s,
              transform        0.5s cubic-bezier(0.25,0.1,0.25,1) 0.2s;
}

.selector:hover {
  background-color: #3498db;
```

```
  transform: scale(1.05);
}
```

- **`transition-property`**: Which CSS property(ies) to animate.
- **`transition-duration`**: How long the transition takes (e.g., `0.3s`, `200ms`).
- **`transition-timing-function`**: Easing curve (`ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, or custom `cubic-bezier`).
- **`transition-delay`**: Delay before the animation starts.

Tips

- Only transition animatable properties (e.g., `opacity`, `transform`, `color`, `height`).
- Use `will-change` to hint the browser for better performance:

```
.selector {
  will-change: transform, opacity;
}
```

---

2. CSS Animations

Animations provide keyframe-driven control over what styles apply at various points in the animation sequence.

2.1 Defining Keyframes

```
@keyframes slide-in {
  0% {
    transform: translateX(-100%);
    opacity: 0;
  }
  50% {
    transform: translateX(10%);
    opacity: 0.5;
  }
  100% {
    transform: translateX(0);
    opacity: 1;
  }
}
```

2.2 Applying Animations

```
.animated-box {
  /* Name the keyframes, define duration, easing, delay, iteration count,
direction */
  animation-name: slide-in;
  animation-duration: 1s;
  animation-timing-function: ease-out;
  animation-delay: 0.2s;
  animation-iteration-count: 1;       /* or infinite */
  animation-direction: normal;        /* normal | reverse | alternate */
  animation-fill-mode: forwards;      /* none | forwards | backwards | both
```

```
*/
}
```

- **`animation-name`**: Must match an `@keyframes` identifier.
- **`animation-duration`**: Total time for one cycle.
- **`animation-timing-function`**: Easing of the animation.
- **`animation-delay`**: Time before starting.
- **`animation-iteration-count`**: How many times to repeat.
- **`animation-direction`**: Play direction each cycle.
- **`animation-fill-mode`**: Determines how styles are applied before/after animation runs.

2.3 Shorthand

```
.animated-box {
  animation: slide-in 1s ease-out 0.2s 1 normal forwards;
}
```

Order: `animation: [name] [duration] [timing-function] [delay] [iteration-count] [direction] [fill-mode];`

---

3. Keeping CSS Modular & Reusable

3.1 External Stylesheets

- Store all CSS in `.css` files, and link via `<link rel="stylesheet" href="styles.css">`.
- Break large stylesheets into multiple files by feature or component (e.g., `buttons.css`, `grid.css`, `theme.css`) and import with `@import` or build tooling (Webpack, PostCSS).

3.2 Naming Conventions: BEM

BEM (Block Element Modifier) is a methodology for clear, scalable, and flat class names:

```
<div class="card">
  <h2 class="card__title">Title</h2>
  <p class="card__text">Lorem ipsum…</p>
  <button class="card__btn card__btn--primary">Click</button>
</div>
```

- **Block**: standalone component (e.g., `card`).
- **Element**: child inside block, separated by `__` (e.g., `card__title`).
- **Modifier**: flag for variations, separated by `--` (e.g., `card__btn--primary`).

Benefits:

- Predictable, no cascading overrides.
- Easy to search and refactor.
- Promotes component isolation.

3.3 Commenting & Organizing Code

- Use clear, consistent comments to delineate sections:

```
/*
================================================================================
====
   Base Styles

================================================================================
==== */
/* Reset margins, headings, base typography */

/*
================================================================================
====
   Layout (Grid, Flex, Containers)

================================================================================
==== */

/*
================================================================================
====
   Components (Cards, Buttons, Forms)

================================================================================
==== */
```

- **Logical order** within each stylesheet:

    1. Variables / Custom Properties
    2. Reset / Normalize
    3. Base (typography, body, headings)
    4. Layout (grid containers, wrappers)
    5. Components (buttons, cards, navbars)
    6. Utilities (helpers like `.u-mt-1`, `.text-center`)
    7. States & Overrides (hover, focus, media queries)

- Consider using a **CSS preprocessor** (Sass, Less) or **PostCSS** to organize with partials, nesting, and imports, while still compiling to clean CSS.
