HannaCode CSS Book

Style the Web – A Complete Beginner’ s Guide to CSS

Author: Oladapo A.O

Table of Contents

Chapter 1: Introduction to CSS

What is CSS

CSS, which stands for Cascading Style Sheets, is a stylesheet language used for describing the presentation of a document written in HTML or XML (including XML dialects such as SVG, MathML or XHTML). CSS describes how elements should be rendered on screen, on paper, in speech, or on other media.

How CSS works with HTML

HTML provides the structure of a web page, while CSS provides the styling. Think of HTML as the skeleton and CSS as the skin, clothes, and makeup. CSS rules are applied to HTML elements to control their appearance, such as colors, fonts, layout, and spacing.

Inline, internal, and external stylesheets

There are three main ways to include CSS in an HTML document:

Inline Styles

Inline styles are applied directly to individual HTML elements using the `style` attribute. This method is generally discouraged for larger projects as it mixes content with presentation and makes maintenance difficult.

```
<p style="color: blue; font-size: 16px;">This text is styled with inline CSS.
</p>
```

Internal Stylesheets

Internal stylesheets are defined within the `<style>` tags in the `<head>` section of an HTML document. This method is suitable for single pages or when you want to apply specific styles to a particular page.

```
<!DOCTYPE html>
<html>
<head>
<style>
p {
  color: red;
  text-align: center;
}
</style>
</head>
<body>

<p>This paragraph is styled with internal CSS.</p>

</body>
</html>
```

External Stylesheets

External stylesheets are separate `.css` files linked to the HTML document using the `<link>` tag in the `<head>` section. This is the most recommended method for larger projects as it promotes separation of concerns, improves maintainability, and allows styles to be reused across multiple pages.

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <p>This paragraph is styled with external CSS.</p>
</body>
</html>
```

styles.css:

```
p {
  color: green;
  font-family: Arial, sans-serif;
}
```

Chapter 2: CSS Syntax & Selectors

CSS rules consist of a selector and a declaration block.

Basic syntax and rules

A CSS rule-set consists of a selector and a declaration block:

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

- **Selector**: Points to the HTML element you want to style (e.g., `p` for paragraph).
- **Declaration block**: Contains one or more declarations separated by semicolons. Each declaration includes a CSS property name (e.g., `color`) and a value (e.g., `blue`), separated by a colon.

Selectors: element, class, ID

Selectors are used to "find" (or select) the HTML elements you want to style.

- **Element Selector**: Selects HTML elements based on their tag name.

```
h1 {
  color: navy;
}
```

- **Class Selector**: Selects HTML elements with a specific `class` attribute. Classes are denoted by a dot (`.`). Multiple elements can share the same class.

```
<p class="intro">This is an introduction.</p>
<p class="intro">This is also an introduction.</p>
```

```
.intro {
  font-weight: bold;
}
```

- **ID Selector**: Selects a unique HTML element with a specific `id` attribute. IDs are denoted by a hash (`#`). An ID must be unique within an HTML document.

```
<h2 id="main-title">Welcome to My Website</h2>
```

```
#main-title {
  text-align: center;
}
```

Grouping, combinators, pseudo-classes

- **Grouping Selectors**: If you have elements with the same style, you can group them to minimize code.

```
h1, h2, p {
  text-align: center;
  color: black;
}
```

- **Combinators**: Explain the relationship between selectors.
    - **Descendant Selector (space)**: Selects all `<p>` elements inside `<div>` elements. `css div p { background-color: yellow; }`
    - **Child Selector (>)**: Selects all `<p>` elements that are direct children of a `<div>` element. `css div > p { border: 1px solid red; }`
    - **Adjacent Sibling Selector (+)**: Selects an element that is immediately preceded by a specified element. `css div + p { margin-top: 20px; }`
    - **General Sibling Selector (~)**: Selects all elements that are siblings of a specified element. `css div ~ p { color: purple; }`
- **Pseudo-classes**: Used to define a special state of an element. For example, it can be used to style an element when a user mouses over it, or to style visited/unvisited links.

```
a:hover {
  color: hotpink;
}

p:first-child {
  font-style: italic;
}
```

Chapter 3: Colors and Backgrounds

Colors and backgrounds are fundamental to the visual appeal of a website. CSS provides various ways to define and apply them.

Color names, HEX, RGB, HSL

CSS allows you to specify colors using several methods:

- **Color Names**: Predefined color names like `red`, `blue`, `green`, `white`, `black`, etc.

```
h1 {
  color: blue;
}
```

- **HEX (Hexadecimal) Values**: A 6-digit hexadecimal number preceded by a `#`. Each pair of digits represents the red, green, and blue components of the color (e.g., `#RRGGBB`).

```
p {
  color: #FF0000; /* Red */
}
```

- **RGB (Red, Green, Blue) Values**: Specifies color using the `rgb()` function with values from 0 to 255 for red, green, and blue. `rgba()` includes an alpha channel for opacity (0 to 1).

```css
body {
  color: rgb(0, 128, 0); /* Green */
}

div {
  background-color: rgba(255, 0, 0, 0.5); /* Semi-transparent red */
}
```

- **HSL (Hue, Saturation, Lightness) Values**: Specifies color using the `hsl()` function. Hue is a degree on the color wheel (0-360), saturation is a percentage (0-100%), and lightness is a percentage (0-100%). `hsla()` includes an alpha channel for opacity.

```css
span {
  color: hsl(120, 100%, 50%); /* Green */
}

article {
  background-color: hsla(240, 100%, 50%, 0.7); /* Semi-transparent blue */
}
```

`background-color`, `background-image`, gradients

- `background-color`: Sets the background color of an element.

```css
body {
  background-color: lightblue;
}
```

- `background-image`: Sets one or more background images for an element. By default, a background image is repeated both vertically and horizontally.

```css
div {
  background-image: url('paper.gif');
  background-repeat: no-repeat;
  background-position: center;
}
```

- **Gradients**: CSS gradients allow you to display smooth transitions between two or more specified colors. They can be linear or radial.
    - **Linear Gradients**: Transition colors linearly (e.g., top to bottom, left to right, or at an angle).

`css .linear-gradient { background-image: linear-gradient(red, yellow); }`

- **Radial Gradients**: Transition colors from a center point outwards.

`css .radial-gradient { background-image: radial-gradient(red, yellow, green); }`

Chapter 4: Text and Fonts

CSS offers extensive control over text appearance, including font families, sizes, weights, and various text decorations.

`font-family`, `font-size`, `line-height`

- `font-family`: Specifies the font for an element. You can list multiple font names as a "fallback" system. The browser will use the first font it supports.

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

- `font-size`: Sets the size of the font. Can be specified in pixels (`px`), ems (`em`), rems (`rem`), percentages (`%`), or keywords (`small` , `medium` , `large`).

```
h1 {
  font-size: 32px;
}

p {
  font-size: 1.2em; /* 1.2 times the parent element's font-size */
}

.responsive-text {
  font-size: 2vw; /* 2% of the viewport width */
}
```

- `line-height`: Sets the height of a line of text. Can be a number (multiplied by the font size), a length unit (px, em), or a percentage.

```css
p {
  line-height: 1.5; /* 1.5 times the font-size */
}
```

Text alignment, decoration, transformation

- `text-align`: Aligns the text within an element. Values include `left`, `right`, `center`, and `justify`.

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

- `text-decoration`: Specifies the line decoration for text. Common values are `none`, `underline`, `overline`, and `line-through`.

```css
a {
  text-decoration: none; /* Removes underline from links */
}

.strike {
  text-decoration: line-through;
}
```

- `text-transform`: Controls the capitalization of text. Values include `none`, `uppercase`, `lowercase`, and `capitalize`.

```css
h3 {
  text-transform: uppercase;
}
```

- `font-weight`: Sets how thick or thin characters in text should be displayed. Values can be keywords (`normal`, `bold`) or numbers (100-900).

```css
strong {
  font-weight: bold;
}

.light-text {
  font-weight: 300;
}
```

- `font-style`: Specifies the font style for text. Values include `normal`, `italic`, and `oblique`.

```
em {
  font-style: italic;
}
```

Chapter 5: Box Model

In CSS, every element is considered a rectangular box. This "box model" is fundamental to understanding how elements are displayed and how they interact with each other on a page.

Margin, border, padding, content

The CSS box model consists of four parts:

1. **Content**: The actual content of the element, such as text, images, or other media. The size of the content area can be controlled by `width` and `height` properties.
2. **Padding**: The space between the content and the border. Padding is transparent and takes on the background color of the element. You can control padding on all four sides (`padding-top`, `padding-right`, `padding-bottom`, `padding-left`) or use the shorthand `padding` property.
3. **Border**: A line that goes around the padding and content. You can control the `border-width`, `border-style`, and `border-color` for each side or use the shorthand `border` property.
4. **Margin**: The space outside the border, separating the element from other elements. Margins are transparent. You can control margins on all four sides (`margin-top`, `margin-right`, `margin-bottom`, `margin-left`) or use the shorthand `margin` property.

```css
div {
  width: 200px;
  height: 100px;
  padding: 25px; /* 25px on all sides */
  border: 5px solid blue;
  margin: 50px; /* 50px on all sides */
  background-color: lightgray;
}
```

`box-sizing` property

The `box-sizing` property allows you to include the padding and border in an element's total width and height.

- **`content-box` (default)**: The `width` and `height` properties only apply to the content area. Padding and border are added to this, increasing the total size of the element.
- **`border-box`**: The `width` and `height` properties include the content, padding, and border. This makes it much easier to lay out elements, as the total size of the box remains consistent regardless of padding and border values.

It is a common practice to set `box-sizing: border-box;` globally for easier layout management.

```css
/* Global box-sizing reset */
html {
  box-sizing: border-box;
}
*,
*::before,
*::after {
  box-sizing: inherit;
}

/* Example with border-box */
.box {
  width: 200px;
  height: 100px;
  padding: 20px;
  border: 10px solid green;
  background-color: lightyellow;
  box-sizing: border-box; /* Now width and height include padding and border */
}
```

Chapter 6: Layout Techniques

CSS provides several properties to control the layout of elements on a web page. Understanding these is crucial for building well-structured and visually appealing designs.

Display types: `block`, `inline`, `inline-block`, `none`

- **`block`**: Elements with `display: block;` start on a new line and take up the full width available. Examples: `<div>`, `<p>`, `<h1>`.

  ```
  css .block-element { display: block; background-color: lightcoral; margin: 10px; padding: 10px; }
  ```

- **`inline`**: Elements with `display: inline;` do not start on a new line and only take up as much width as necessary. You cannot set `width`, `height`, `margin-top`, or `margin-bottom` on inline elements. Examples: `<span>`, `<a>`, `<strong>`.

  ```
  css .inline-element { display: inline; background-color: lightgreen; padding: 5px; }
  ```

- **`inline-block`**: Elements with `display: inline-block;` behave like inline elements but allow you to set `width`, `height`, and vertical margins/padding. They do not start on a new line.

  ```
  css .inline-block-element { display: inline-block; width: 150px; height: 50px; background-color: lightblue; margin: 10px; padding: 10px; }
  ```

- **`none`**: Hides an element completely. The element will not take up any space on the page.

  ```
  css .hidden-element { display: none; }
  ```

Position: `static`, `relative`, `absolute`, `fixed`

The `position` property controls how an element is positioned on the page.

- **`static` (default)**: Elements are positioned according to the normal flow of the document. `top`, `right`, `bottom`, `left` properties have no effect.

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

- **`relative`**: Elements are positioned relative to their normal position. `top`, `right`, `bottom`, `left` properties will move the element from its normal position, but still occupy its original space in the document flow.

  ```
  css .relative-box { position: relative; top: 20px; left: 30px; }
  ```

- **`absolute`**: Elements are positioned relative to their closest positioned ancestor (an ancestor with `position: relative`, `absolute`, or `fixed`). If no positioned ancestor exists, it's positioned relative to the initial containing block (usually the `<html>` element). Absolute positioned elements are removed from the normal document flow.

  ```
  css .parent { position: relative; } .absolute-box { position: absolute; top: 0; right: 0; }
  ```

- **`fixed`**: Elements are positioned relative to the viewport, meaning they stay in the same position even if the page is scrolled. Fixed elements are removed from the normal document flow.

  ```
  css .fixed-header { position: fixed; top: 0; width: 100%; background-color: #333; color: white; }
  ```

Float and clear

- **`float`**: Used to wrap text around images or to create multi-column layouts. Elements can float `left`, `right`, or `none`.

  ```
  css .image-float { float: left; margin-right: 15px; }
  ```

- **`clear`**: Used to control the behavior of elements next to floated elements. An element with `clear: both;` (or `left`, `right`) will move below any floated elements on its specified side.

  `css .clearfix::after { content: ""; display: table; clear: both; }`

  While `float` was historically used for layout, modern CSS layout methods like Flexbox and Grid are generally preferred for their flexibility and predictability.

Chapter 7: Flexbox

Flexbox (Flexible Box Layout Module) is a one-dimensional layout method for arranging items in a container. It makes it easier to design flexible and responsive layout structures without using floats or positioning.

Container and item properties

To use Flexbox, you define a flex container and flex items within it.

Flex Container Properties (applied to the parent element)

- **`display: flex;`**: Initializes a flex container. All direct children become flex items.

  `css .container { display: flex; }`

- **`flex-direction`**: Defines the direction of the main axis (the direction flex items are placed in).
  - `row` (default): left to right
  - `row-reverse`: right to left
  - `column`: top to bottom
  - `column-reverse`: bottom to top

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

- **`flex-wrap`**: Controls whether flex items are forced onto one line or can wrap onto multiple lines.
    - `nowrap` (default): all flex items will be on one line
    - `wrap`: flex items will wrap onto multiple lines
    - `wrap-reverse`: flex items will wrap onto multiple lines in reverse order

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

- **`justify-content`**: Aligns flex items along the main axis.
    - `flex-start` (default): items packed to the start of the flex-direction
    - `flex-end`: items packed to the end of the flex-direction
    - `center`: items centered along the line
    - `space-between`: items evenly distributed; first item is at the start, last item at the end
    - `space-around`: items evenly distributed with equal space around them
    - `space-evenly`: items evenly distributed with equal space between them and around the edges

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

- **`align-items`**: Aligns flex items along the cross axis (perpendicular to the main axis).
    - `stretch` (default): items stretch to fill the container
    - `flex-start`: items aligned to the start of the cross axis
    - `flex-end`: items aligned to the end of the cross axis
    - `center`: items centered on the cross axis
    - `baseline`: items aligned such that their baselines align

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

- **`align-content`**: Aligns flex lines when there is extra space in the cross-axis and `flex-wrap` is set to `wrap` or `wrap-reverse`. (Similar to `justify-content` but for multiple lines).

```
css .container { align-content: space-around; }
```

Flex Item Properties (applied to the child elements)

- **`order`**: Specifies the order of a flex item relative to the rest of the flex items inside the same container. Default is 0.

  ```
  css .item-1 { order: 2; } .item-2 { order: 1; }
  ```

- **`flex-grow`**: Specifies the ability of a flex item to grow if necessary. Accepts a unitless value that serves as a proportion.

  ```
  css .item { flex-grow: 1; }
  ```

- **`flex-shrink`**: Specifies the ability of a flex item to shrink if necessary. Accepts a unitless value.

  ```
  css .item { flex-shrink: 0; }
  ```

- **`flex-basis`**: Specifies the initial size of a flex item before any available space is distributed. Can be a length (e.g., `200px`) or a keyword (`auto`).

  ```
  css .item { flex-basis: 150px; }
  ```

- **`flex` (shorthand)**: A shorthand for `flex-grow`, `flex-shrink`, and `flex-basis`.

  ```
  css .item { flex: 1 1 auto; /* flex-grow: 1, flex-shrink: 1, flex-basis: auto */ }
  ```

- **`align-self`**: Overrides the `align-items` value for a single flex item.

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

Alignments and spacing

Flexbox makes aligning and distributing space among items very straightforward using `justify-content` and `align-items` (and `align-content` for multi-line flex containers).

Example: Centering an item

To perfectly center an item horizontally and vertically within its container:

```
.container {
  display: flex;
  justify-content: center; /* Centers horizontally */
  align-items: center;    /* Centers vertically */
  height: 300px; /* Container needs a defined height */
}
```

Flex direction and wrap

These properties (`flex-direction` and `flex-wrap`) are crucial for defining the primary flow and responsiveness of your flex container.

Example: Responsive Navigation Bar

```
<nav class="navbar">
  <a href="#">Home</a>
  <a href="#">About</a>
  <a href="#">Services</a>
  <a href="#">Contact</a>
</nav>
```

```
.navbar {
  display: flex;
  flex-direction: row; /* Items arranged in a row */
  flex-wrap: wrap;     /* Allow items to wrap to the next line on smaller
screens */
  justify-content: space-around; /* Distribute space around items */
  background-color: #f8f8f8;
  padding: 10px;
}

.navbar a {
  padding: 10px 15px;
  text-decoration: none;
  color: #333;
  border: 1px solid #ddd;
  margin: 5px;
}
```

Chapter 8: Grid Layout

CSS Grid Layout is a two-dimensional layout system for the web. It allows you to arrange content in rows and columns, making it ideal for complex page layouts.

Basic grid syntax

To use CSS Grid, you define a grid container and grid items within it.

Grid Container Properties (applied to the parent element)

- **`display: grid;`**: Initializes a grid container. All direct children become grid items.

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

- **`grid-template-columns`**: Defines the number and width of columns in the grid.

  ```
  css .container { grid-template-columns: 100px 1fr 2fr; /* 100px, then 1 part, then 2 parts of remaining space */ }
  ```

- **`grid-template-rows`**: Defines the number and height of rows in the grid.

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

- **`grid-gap`** **(shorthand for `grid-row-gap` and `grid-column-gap`)**: Sets the size of the gap between rows and columns.

  ```
  css .container { grid-gap: 10px 20px; /* row-gap 10px, column-gap 20px */ }
  ```

Grid Item Properties (applied to the child elements)

- **`grid-column`**: Specifies the starting and ending column lines for a grid item.

  ```
  css .item-a { grid-column: 1 / 3; /* Starts at column line 1, ends at column line 3 (spans 2 columns) */ }
  ```

- **`grid-row`**: Specifies the starting and ending row lines for a grid item.

  ```
  css .item-b { grid-row: 2 / 4; /* Starts at row line 2, ends at row line 4 (spans 2 rows) */ }
  ```

Template columns and rows

`grid-template-columns` and `grid-template-rows` are powerful for defining the structure of your grid.

- **Fixed values**: `px`, `em`, `rem`.
- **Flexible values**: `fr` (fractional unit) distributes available space proportionally.
- **`repeat()` function**: A shorthand to repeat columns or rows.

  ```
  css .container { grid-template-columns: repeat(3, 1fr); /* Three equal columns */ grid-template-rows: repeat(2, 100px); /* Two rows, each 100px tall */ }
  ```

- **`minmax()` function**: Defines a size range greater than or equal to `min` and less than or equal to `max`.

  ```
  css .container { grid-template-columns: repeat(auto-fit, minmax(200px, 1fr)); /* Responsive columns */ }
  ```

Grid gap and alignment

- **`grid-gap`**: As mentioned, this property sets the spacing between grid cells.

  ```
  css .container { grid-gap: 20px; /* 20px gap between all rows and columns */ }
  ```

- **`justify-items`**: Aligns grid items along the row (inline) axis within their grid area.
  - `start`, `end`, `center`, `stretch` (default)

  ```
  css .container { justify-items: center; }
  ```

- **`align-items`**: Aligns grid items along the column (block) axis within their grid area.
    - `start`, `end`, `center`, `stretch` (default)

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

- **`justify-content`**: Aligns the grid itself along the row axis within the grid container.

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

- **`align-content`**: Aligns the grid itself along the column axis within the grid container.

    `css .container { align-content: space-between; }`

Example Grid Layout:

```
<div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
</div>
```

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr); /* 3 equal columns */
  grid-gap: 15px;
  background-color: #f3f3f3;
  padding: 20px;
}

.grid-item {
  background-color: #e0e0e0;
  border: 1px solid #ccc;
  padding: 20px;
  text-align: center;
}
```

Chapter 9: Responsive Design

Responsive web design is an approach to web design that makes web pages render well on a variety of devices and window or screen sizes from minimum to maximum display size. It is about creating flexible layouts that adapt to the user's screen.

Media queries

Media queries are a CSS3 module that allows content to adapt to different conditions, such as screen resolution (e.g., smartphone vs. desktop screen). They consist of a media type and zero or more expressions that check for the conditions of particular media features.

```
/* Styles for screens smaller than 600px */
@media screen and (max-width: 600px) {
  body {
    background-color: lightblue;
  }
  .container {
    flex-direction: column;
  }
}

/* Styles for screens larger than 900px */
@media screen and (min-width: 900px) {
  body {
    background-color: lightgreen;
  }
  .sidebar {
    display: block;
  }
}
```

Common media features used in queries: - `width` / `height`: Width/height of the viewport. - `min-width` / `max-width`: Minimum/maximum width of the viewport. - `orientation`: Orientation of the viewport (`portrait` or `landscape`). - `resolution`: Resolution of the output device.

Mobile-first design

Mobile-first design is a strategy that prioritizes designing for mobile devices first, then progressively enhancing the design for larger screens. This approach ensures that the

core content and functionality are accessible on smaller devices, and then more complex layouts and features are added for desktop users.

Why Mobile-First? - **Performance**: Mobile devices often have slower internet connections and less processing power. Designing mobile-first encourages optimizing for performance from the start. - **User Experience**: Forces designers to focus on essential content and features, leading to a cleaner and more intuitive user experience. - **SEO**: Search engines often prioritize mobile-friendly websites.

Example Mobile-First CSS Structure:

```css
/* Base styles (for mobile devices) */
body {
  font-size: 16px;
  padding: 10px;
}

.header {
  text-align: center;
}

/* Styles for tablets and larger screens */
@media screen and (min-width: 768px) {
  body {
    font-size: 18px;
    padding: 20px;
  }
  .header {
    text-align: left;
  }
}

/* Styles for desktops and larger screens */
@media screen and (min-width: 1024px) {
  body {
    font-size: 20px;
    max-width: 1200px;
    margin: 0 auto;
  }
  .header {
    display: flex;
    justify-content: space-between;
    align-items: center;
  }
}
```

Relative units (%, em, rem, vw, vh)

Using relative units instead of absolute units (like `px`) is crucial for responsive design, as they scale relative to other elements or the viewport size.

- `%` **(Percentage)**: Relative to the parent element. For example, `width: 50%;` means 50% of the parent's width.

  ```
  css .parent { width: 500px; } .child { width: 50%; /* 250px */ }
  ```

- `em`: Relative to the font-size of the element itself. If not set on the element, it inherits from its parent.

  ```
  css .container { font-size: 16px; } .text { font-size: 1.5em; /* 24px */ padding: 1em; /* 16px */ }
  ```

- `rem` **(root em)**: Relative to the font-size of the root HTML element (`<html>`). This makes scaling more predictable as it doesn't depend on nested element font sizes.

  ```
  css html { font-size: 16px; } .title { font-size: 2rem; /* 32px */ } .paragraph { font-size: 1rem; /* 16px */ }
  ```

- `vw` **(viewport width)**: Relative to 1% of the viewport's width. `100vw` is the full width of the viewport.

  ```
  css h1 { font-size: 5vw; /* 5% of viewport width */ }
  ```

- `vh` **(viewport height)**: Relative to 1% of the viewport's height. `100vh` is the full height of the viewport.

  ```
  css .full-screen-div { height: 100vh; }
  ```

Chapter 10: CSS Transitions and Animations

CSS transitions and animations allow you to create dynamic and engaging user interfaces without relying on JavaScript. They provide smooth changes to CSS properties over a specified duration.

`transition` properties

Transitions provide a way to animate changes in CSS properties. You define which properties to animate, how long the animation should take, and how it should progress.

- **`transition-property`**: Specifies the CSS property to which the transition should be applied (e.g., `width`, `color`, `opacity`). Use `all` to transition all properties.
- **`transition-duration`**: Specifies how long the transition will take to complete (e.g., `0.5s`, `500ms`).
- **`transition-timing-function`**: Specifies the speed curve of the transition (e.g., `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`).
- **`transition-delay`**: Specifies a delay before the transition starts.

Shorthand `transition` property:

```
.box {
  width: 100px;
  height: 100px;
  background-color: blue;
  transition: width 2s ease-in-out 1s, background-color 1s linear; /* Animate
width over 2s after 1s delay, and background-color over 1s */
}

.box:hover {
  width: 200px;
  background-color: red;
}
```

`@keyframes` and `animation`

CSS animations allow for more complex, multi-step animations. They are defined using `@keyframes` rules and then applied to elements using the `animation` properties.

`@keyframes` rule

Defines the animation sequence. You specify styles at different stages (keyframes) of the animation using percentages or `from` / `to` (0% and 100%).

```css
@keyframes slidein {
  from {
    margin-left: 100%;
    width: 300%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}

@keyframes fade-and-grow {
  0% {
    opacity: 0;
    transform: scale(0.5);
  }
  50% {
    opacity: 1;
    transform: scale(1.2);
  }
  100% {
    opacity: 0.8;
    transform: scale(1);
  }
}
```

`animation` properties

Applied to the element you want to animate.

- **`animation-name`**: Specifies the name of the `@keyframes` rule to bind to the element.
- **`animation-duration`**: Specifies how long the animation takes to complete one cycle.
- **`animation-timing-function`**: Specifies the speed curve of the animation.
- **`animation-delay`**: Specifies a delay before the animation starts.
- **`animation-iteration-count`**: Specifies how many times the animation should play (`infinite` for continuous).
- **`animation-direction`**: Specifies whether the animation should play forwards, backwards, or alternate between the two (`normal`, `reverse`, `alternate`, `alternate-reverse`).
- **`animation-fill-mode`**: Specifies a style for the element when the animation is not playing (before it starts, after it ends, or both).

- `animation-play-state`: Specifies whether the animation is running or paused.

Shorthand `animation` property:

```
.animated-box {
  width: 100px;
  height: 100px;
  background-color: purple;
  animation: fade-and-grow 4s ease-in-out infinite alternate;
}
```

Chapter 11: Best Practices

Writing clean, organized, and efficient CSS is crucial for maintainable and scalable web projects. Following best practices helps in collaboration, debugging, and long-term project health.

Organizing your CSS

- **Modular CSS**: Break down your CSS into smaller, reusable modules. Instead of one large stylesheet, use multiple files for different components (e.g., `header.css`, `footer.css`, `buttons.css`).
- **Logical Grouping**: Group related CSS properties together. For example, all typography-related properties (`font-size`, `font-family`, `line-height`) should be together.
- **Comments**: Use comments to explain complex sections of code, clarify intentions, or provide context. This is especially helpful when working in teams.

  ```css / *Global styles* / body { font-family: Arial, sans-serif; line-height: 1.6; color: #333; }

  / --- *Components* --- /

  / *Button styles* / .btn { display: inline-block; padding: 10px 20px; background-color: #007bff; color: white; text-decoration: none; border-radius: 5px; } ```
  ```

Naming conventions (BEM basics)

Consistent naming conventions make your CSS classes and IDs predictable and easier to understand. BEM (Block, Element, Modifier) is a popular methodology for naming CSS classes.

- **Block**: Standalone entity that is meaningful on its own (e.g., `header`, `menu`, `button`).
- **Element**: A part of a block that has no standalone meaning and is semantically tied to its block (e.g., `menu__item`, `button__icon`). Elements are separated from blocks by two underscores (`__`).
- **Modifier**: A flag on a block or element. It is used to change the appearance or behavior of a block or element (e.g., `button--primary`, `menu__item--disabled`). Modifiers are separated from blocks or elements by two hyphens (`--`).

```
<div class="card">
  <h2 class="card__title">Card Title</h2>
  <p class="card__text card__text--highlighted">Some text.</p>
  <button class="card__button card__button--disabled">Buy Now</button>
</div>
```

```css
.card {
  /* styles for the card block */
}

.card__image {
  /* styles for the image element within the card */
}

.card__title {
  /* styles for the title element within the card */
}

.card__text {
  /* styles for the text element within the card */
}

.card__text--highlighted {
  /* modifier for highlighted text */
  color: orange;
}

.card__button {
  /* styles for the button element within the card */
}

.card__button--disabled {
  /* modifier for disabled button */
  opacity: 0.5;
  cursor: not-allowed;
}
```

Avoiding redundancy

- **DRY (Don't Repeat Yourself)**: Avoid writing the same CSS properties multiple times. Use grouping selectors, shorthand properties, and variables (CSS custom properties) to reduce repetition.
- **Shorthand Properties**: Use shorthand properties (e.g., `margin`, `padding`, `background`, `font`, `border`) to combine multiple related properties into one.

  ```css / *Instead of this:* / .box { margin-top: 10px; margin-right: 20px; margin-bottom: 10px; margin-left: 20px; }

  / *Use this:* / .box { margin: 10px 20px; / *top/bottom left/right* / } ```
- **CSS Custom Properties (Variables)**: Define reusable values (like colors, font sizes) as custom properties. This makes it easy to update values across your entire stylesheet.
  ```

```css :root { --primary-color: #007bff; --font-size-base: 16px; }

body { color: var(--primary-color); font-size: var(--font-size-base); }

.button { background-color: var(--primary-color); } ```

# Chapter 12: Mini Projects

These mini-projects will help you practice and apply the CSS concepts you've learned, building practical styling skills.

## Responsive card layout

Create a responsive card layout that displays information in a grid-like structure. The cards should adapt to different screen sizes, stacking vertically on small screens and arranging in columns on larger screens.

**HTML Structure:**

```

    <h3>Card Title 1</h3>
    Short description for card 1.

    <h3>Card Title 2</h3>
    Short description for card 2.

    <h3>Card Title 3</h3>
    Short description for card 3.

```

**CSS (using Flexbox and Media Queries):**
```

```css
.card-container {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-around;
  padding: 20px;
}

.card {
  background-color: #f9f9f9;
  border: 1px solid #ddd;
  border-radius: 8px;
  margin: 10px;
  padding: 20px;
  width: 100%; /* Default for small screens */
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

@media (min-width: 600px) {
  .card {
    width: calc(50% - 40px); /* Two columns on medium screens */
  }
}

@media (min-width: 900px) {
  .card {
    width: calc(33.333% - 40px); /* Three columns on large screens */
  }
}
```

Styled form

Take a basic HTML form (like the one from the HTML book) and apply appealing CSS styles to it. Focus on:

- Consistent input field styling.
- Attractive button design.
- Clear labeling and spacing.
- Responsive behavior.

HTML (Example from HTML Book):

```
<form class="styled-form">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
  </div>
  <div class="form-group">
    <label for="message">Message:</label>
    <textarea id="message" name="message" rows="5" required></textarea>
  </div>
  <button type="submit" class="submit-btn">Send Message</button>
</form>
```

CSS:

```css
.styled-form {
  max-width: 500px;
  margin: 40px auto;
  padding: 30px;
  border: 1px solid #e0e0e0;
  border-radius: 10px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.05);
  background-color: #fff;
}

.form-group {
  margin-bottom: 20px;
}

.form-group label {
  display: block;
  margin-bottom: 8px;
  font-weight: bold;
  color: #333;
}

.form-group input[type="text"],
.form-group input[type="email"],
.form-group textarea {
  width: 100%;
  padding: 12px;
  border: 1px solid #ccc;
  border-radius: 5px;
  font-size: 16px;
  box-sizing: border-box; /* Include padding and border in element's total
width */
}

.form-group input[type="text"]:focus,
.form-group input[type="email"]:focus,
.form-group textarea:focus {
  border-color: #007bff;
  outline: none;
  box-shadow: 0 0 0 3px rgba(0, 123, 255, 0.25);
}

.submit-btn {
  display: block;
  width: 100%;
  padding: 15px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 5px;
  font-size: 18px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

.submit-btn:hover {
  background-color: #0056b3;
}
```

Navigation menu

Design a responsive navigation menu. It should be a horizontal menu on desktop screens and transform into a hamburger menu or a vertical stack on mobile devices.

HTML Structure:

```
<nav class="main-nav">
  <ul class="nav-list">
    <li class="nav-item"><a href="#">Home</a></li>
    <li class="nav-item"><a href="#">About</a></li>
    <li class="nav-item"><a href="#">Services</a></li>
    <li class="nav-item"><a href="#">Contact</a></li>
  </ul>
</nav>
```

CSS (using Flexbox and Media Queries):

```css
.main-nav {
  background-color: #333;
  padding: 10px 0;
}

.nav-list {
  list-style: none;
  margin: 0;
  padding: 0;
  display: flex; /* Horizontal layout for desktop */
  justify-content: center;
}

.nav-item a {
  display: block;
  color: white;
  text-align: center;
  padding: 14px 20px;
  text-decoration: none;
}

.nav-item a:hover {
  background-color: #575757;
}

/* Mobile styles */
@media (max-width: 768px) {
  .nav-list {
    flex-direction: column; /* Stack vertically on small screens */
    align-items: center;
  }

  .nav-item {
    width: 100%;
    text-align: center;
  }
}
```

End of Book

Final tips

Congratulations on completing the HannaCode CSS Book! You've gained a solid foundation in CSS, from understanding its syntax and selectors to mastering layout techniques like Flexbox and Grid, and building responsive designs. Remember these final tips:

- **Practice Regularly**: The best way to learn CSS is by building things. Experiment with different properties and layouts.
- **Inspect Elements**: Use your browser's developer tools (right-click -> Inspect) to examine how CSS is applied to elements on any website. This is an invaluable learning tool.
- **Stay Updated**: The web development landscape evolves quickly. Keep an eye on new CSS features and best practices.
- **Don't Be Afraid to Break Things**: Experimentation is key. If something doesn't work, debug it, learn from it, and try again.

Learning roadmap

To continue your journey in web development, consider exploring:

- **Advanced CSS**: Dive deeper into topics like CSS custom properties, preprocessors (Sass/Less), CSS-in-JS, and more advanced animations.
- **JavaScript**: Learn JavaScript to add interactivity and dynamic behavior to your web pages. This is the next logical step after mastering HTML and CSS.
- **Frontend Frameworks**: Explore popular JavaScript frameworks like React, Vue, or Angular to build complex single-page applications.
- **Backend Development**: Understand how servers and databases work to build full-stack applications.

HannaCode community

Stay connected with the HannaCode community for ongoing support, learning resources, and to share your progress:

- **Twitter**: @HannaCodeAcademy
- **Facebook**: HannaCode Academy
- **Discord**: HannaCode Discord Server

Happy styling!