# Web Design & Development Study Notes



Author: Son Nguyen

## 1. HTML Structure

**HTML (Hypertext Markup Language)** forms the backbone of all web pages. It provides the structure and content of a webpage. The basic structure of an HTML document looks like this:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, in
itial-scale=1.0">
        <title>Document Title</title>
        <!-- Other necessary header tags go here -->
    </head>
    <body>
        <!-- Content of the webpage goes here -->
    </body>
</html>
```

- `<!DOCTYPE html>`: Declares the document type and version (HTML5).
- `<html lang="en">`: Root element of the HTML document, with the `lang` attribute specifying the document language.

- `<head>`: Contains meta-information (metadata) about the document, like the character set, viewport settings, and links to stylesheets.
- `<body>`: Contains the actual content of the webpage (text, images, videos, etc.).

## 2. Necessary `<head>` Tags

Within the `<head>` section, various tags provide essential information and resources for the webpage:

- `<meta charset="UTF-8">`: Specifies the character encoding for the document, ensuring it displays correctly.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: Sets the viewport width to match the device's width, enabling responsive design.
- `<title>`: Sets the title of the webpage, displayed in the browser tab.
- `<link rel="stylesheet" href="styles.css">`: Links to an external CSS file for styling the webpage.
- `<script src="script.js"></script>`: Links to an external JavaScript file.
- `<meta name="description" content="Description of the webpage">`: Provides a brief description of the webpage, useful for SEO.
- `<meta name="keywords" content="HTML, CSS, Web Development">`: Specifies relevant keywords for SEO.

## 3. HTML5 Features

HTML5 introduced new features that enhance the functionality, accessibility, and performance of web pages:

- **New Semantic Elements**: `<header>`, `<footer>`, `<article>`, `<section>`, `<nav>`, `<aside>`, etc. to improve the semantic structure of documents.
- **Multimedia Elements**: `<audio>` and `<video>` for embedding media without plugins.
- **Form Enhancements**: New input types (`email`, `date`, `range`, etc.), form attributes (`required`, `placeholder`), and new elements (`<datalist>`, `<output>`).

- **Canvas and SVG**: `<canvas>` element for drawing graphics and SVG for scalable vector graphics.
- **Local Storage**: Provides APIs for storing data in the browser (**localStorage and sessionStorage**).
- **Geolocation API**: Allows web apps to access the user's geographical location.

## 4. Semantic HTML

Semantic HTML uses meaningful tags that convey the purpose of the content within them. This improves accessibility, SEO, and code readability.

- **Examples of Semantic Elements:**
  - `<header>`: Defines a header for a document or section.
  - `<footer>`: Defines a footer for a document or section.
  - `<article>`: Represents an independent piece of content (like a blog post or news article).
  - `<section>`: Defines a section in a document, often grouped by theme.
  - `<nav>`: Represents a section of navigation links.
  - `<aside>`: Contains content that is tangentially related to the main content.

## 5. Basic HTML Tags

HTML uses tags to structure and format content:

- **Headings**: `<h1>` to `<h6>` tags define headings.
- **Paragraphs**: `<p>` defines a paragraph.
- **Links**: `<a href="url">` creates a hyperlink.
- **Images**: `<img src="image.jpg" alt="description">` embeds an image.
- **Lists**: `<ul>` for unordered lists, `<ol>` for ordered lists, and `<li>` for list items within `<ul>` or `<ol>`.
- **Tables**: `<table>`, `<tr>`, `<td>`, `<th>`, etc., create tables.
- **Forms**: `<form>`, `<input>`, `<textarea>`, `<button>`, etc., create interactive forms.

## 6. HTML Deep Dive

- **6.1 Forms and Input Elements:**
  - HTML forms are fundamental for creating interactive user interfaces. They allow user data input and submission to servers.
  - **Form Elements and Their Usage:**
    - `<form>`: The container for all input elements.
      - **Attributes**:
        - `action`: Specifies where the form data is sent (a URL).
        - `method`: Specifies the HTTP method (`GET` or `POST`).
        - `enctype`: Used with `method="POST"` to specify the encoding type (e.g., `multipart/form-data` for file uploads).
        - `autocomplete`: Controls whether form fields should have autocomplete enabled (`on` or `off`).
    - **Input Elements**:
      - `<input type="text">`: Single-line text input.
      - `<input type="password">`: Text input that masks the characters.
      - `<input type="email">`: Validates email format.
      - `<input type="number">`: Accepts numeric input.
      - `<input type="radio">`: Radio button for selecting one option from a group.
      - `<input type="checkbox">`: Checkbox for selecting multiple options.
      - `<input type="date">`: Date picker input.
      - `<input type="file">`: File upload input.
      - `<input type="range">`: Slider control for selecting a value within a range.
    - **Additional Form Elements**:
      - `<textarea>`: Multi-line text input.

- `<select>` **and** `<option>`: Drop-down list for selecting an option.
- `<button>`: Button that can submit the form or trigger JavaScript.
- `<label>`: Ties a text label to a form control, improving accessibility.

- **Form Validation:**
  - HTML5 introduced built-in form validation, reducing the need for JavaScript.
  - **Validation Attributes**:
    - `required`: Makes an input field mandatory.
    - `min` **and** `max`: Specifies minimum and maximum values for numeric inputs.
    - `minlength` **and** `maxlength`: Specifies minimum and maximum character lengths for text inputs.
    - `pattern`: Provides a regular expression for validating input.
    - `step`: Specifies the increment for numeric or date inputs.
  - **Example of Form Validation:**

```
<form action="/submit" method="post">
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <label for="password">Password:</label>
  <input type="password" id="password" name="password"

  <label for="age">Age:</label>
  <input type="number" id="age" name="age" min="18" max

  <button type="submit">Submit</button>
</form>
```

- **6.2. Global Attributes and Their Use Cases**

- Global attributes are attributes that can be applied to any HTML element, providing additional functionality and flexibility.
  - `id`: Uniquely identifies an element, used for CSS styling or JavaScript manipulation.
  - `class`: Assigns one or more class names to an element, used for CSS styling or JavaScript selection.
  - `style`: Specifies inline CSS styles for an element.
  - `title`: Provides additional information about an element (often shown as a tooltip).
  - `hidden`: Hides an element from view, although it remains in the document.
  - `tabindex`: Controls the keyboard navigation order for an element.
  - `data-*`: Custom data attributes that store extra information on an element without affecting the layout or presentation.
- **Example of Global Attributes:**

```
<p id="intro" class="highlight" title="Introduction paragr
aph" data-info="important">Welcome to our website!</p>
```

- **6.3. Custom Data Attributes (`data-*` Attributes)**
  - Custom data attributes (using the `data-` prefix) allow embedding custom, non-visible data in HTML elements. This data can then be accessed via JavaScript.
  - **Use Cases of Custom Data Attributes:**
    - **Storing additional data**: For example, a user ID or product ID that can be referenced later.
      - **JavaScript integration**: Pass data from HTML to JavaScript for dynamic interactions.
  - **Example of Custom Data Attributes:**

```
<button data-product-id="12345" data-product-name="Lapto
p">Add to Cart</button>
```

- JavaScript can access these attributes using the `dataset` property:

```
const button = document.querySelector('button');
console.log(button.dataset.productId); // Output: "12345"
console.log(button.dataset.productName); // Output: "Laptop"
```

- **6.4. Advanced Media Handling**
  - HTML5 provides elements and APIs for handling multimedia content efficiently.
  - `<audio>`: Embeds audio content in a webpage.
    - Attributes: `controls`, `autoplay`, `loop`, `muted`, `src`.
  - `<video>`: Embeds video content in a webpage.
    - Attributes: `controls`, `autoplay`, `loop`, `muted`, `poster`, `src`, `width`, `height`.
  - **Examples:**

```
<audio controls>
  <source src="audiofile.mp3" type="audio/mpeg">
  Your browser does not support the audio element.
</audio>

<video controls width="400">
  <source src="videofile.mp4" type="video/mp4">
  Your browser does not support the video element.
</video>
```

- **6.5. Accessibility Practices (a11y)**
  - Accessibility (often abbreviated as a11y) ensures that web content is usable by people of all abilities and disabilities.

- **Key Accessibility Practices in HTML:**
  - **Use Semantic Elements**: Use `<header>`, `<nav>`, `<main>`, `<footer>`, etc., to improve screen reader navigation.
  - **Provide Alternative Text**: Use `alt` attributes for images to describe their content.
  - **Use ARIA Roles and Properties**: Accessible Rich Internet Applications (ARIA) roles, states, and properties make dynamic content accessible.
  - **Label Form Elements Properly**: Use `<label>` tags and the `for` attribute to associate labels with form inputs.
  - **Keyboard Navigation**: Ensure that all interactive elements (links, buttons, forms) are keyboard accessible (use `tabindex` and `accesskey`).
  - **Contrast and Text Size**: Use sufficient contrast and readable text sizes for all users.

- **6.6. HTML5 APIs**
  - HTML5 introduced several powerful JavaScript APIs that enhance interactivity and functionality on the client side:
  - **Geolocation API:**
    - Allows web applications to access the user's geographical location with their consent.

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(function(positi
    console.log('Latitude: ' + position.coords.latitude);
    console.log('Longitude: ' + position.coords.longitude
  });
}
```

  - **Canvas API:**
    - Enables drawing graphics and animations directly in the browser.

```
<canvas id="myCanvas" width="200" height="100"></canvas
>
```

```
const canvas = document.getElementById('myCanvas');
const context = canvas.getContext('2d');
context.fillStyle = 'red';
context.fillRect(10, 10, 150, 75);
```

- **Local Storage and Session Storage APIs**
  - Provide methods for storing data on the client side, similar to cookies but with more capacity and flexibility.

```
// Store data
localStorage.setItem('username', 'JohnDoe');

// Retrieve data
const username = localStorage.getItem('username');
console.log(username); // Output: "JohnDoe"
```

## 7. CSS Deep Dive

- CSS (Cascading Style Sheets) is used to style and layout web pages. It controls the visual presentation and behavior of HTML elements.
- **Ways to Apply Styles to Elements**

1. **Inline Styles**: Directly within an HTML element's `style` attribute.

```
<p style="color: blue; font-size: 16px;">This is a para
graph.</p>
```

2. **Internal Stylesheet**: Within a `<style>` tag inside the `<head>` section.

```
<head>
    <style>
```

```
        p { color: blue; font-size: 16px; }
    </style>
</head>
```

3. **External Stylesheet**: Using a linked CSS file.

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

- **CSS Selectors**: Used to target HTML elements and apply styles.
  - **Type Selector**: Targets all instances of a particular element (`p { color: blue; }`).
  - **Class Selector**: Targets elements with a specific class (`.classname { color: blue; }`).
  - **ID Selector**: Targets an element with a specific ID (`#idname { color: blue; }`).
  - **Attribute Selector**: Targets elements with a specific attribute (`input[type="text"] { color: blue; }` ⇒ Targets all elements of type `input` with attribute `type="text"`).
- **Measurement Units:**
  - **Fixed Units:** Mostly are pixels `px`.
  - **Relative Units:**
    - `em`: Relative to the font size of the parent element.
      - E.g. `1em`: 1x the font size of the parent element.
    - `rem`: Relative to the font size of the `:root` element (which is typically `<html>` or the browser default, which is usually `16px`).
      - E.g. `1rem`: 1x the font size of the root element, which is 16px.
  - **Fluid Units:**
    - `%`: Percentage of the parent element's width/height.
      - For a font, it is the percentage of the parent element's assigned font size.

        - E.g. `1%`: usually, 1% of the element's parent element's width/height.
    - `vh`: Percentage of the viewport's height (browser window's width).
        - E.g. `1vh`: 1/100 of the height of the viewport (browser window)
    - `vw`: Percentage of the viewport's width (browser window's width).
        - `1vw`: 1/100 of the width of the viewport (browser window)

⇒ Think **fluid** for container and overall element widths, **relative** for fonts, **fixed** for small details like padding, margin, border, and anything that *cannot* change. But there's room for exceptions and preferences.

## 8. CSS Pseudo-Classes and Pseudo-Elements

- **8.1. Pseudo-Classes**
    - **Pseudo-classes** are used to define the special state of an element. They target elements based on information that is not present in the document tree, such as the element's state or its relationship with other elements.
    - **8.1.1. Dynamic Pseudo-Classes: Interaction States**
        - These pseudo-classes are used to style elements based on user interaction.
        - `:hover`: Applies styles when the user hovers over an element.
        - `:focus`: Applies styles when an element (like an input field) is focused, typically via mouse click or keyboard tab.
        - `:active`: Applies styles when an element is being activated, such as when a button is pressed.
        - `:visited`: Applies styles to links that have been visited.
        - `:link`: Applies styles to links that have not been visited.
        - **Example: Styling Interaction States**

```
a:link {
  color: blue; /* Unvisited link color */
}
```

```css
a:visited {
  color: purple; /* Visited link color */
}

button:hover {
  background-color: lightblue; /* Button color on hover */
}

input:focus {
  border-color: green; /* Input border color on focus */
}

button:active {
  background-color: darkblue; /* Button color when clicked
*/
}
```

- **8.1.2. Structural Pseudo-Classes**
  - Structural pseudo-classes target elements based on their position or characteristics within the document.
  - `:first-child`: Selects the first child of its parent.
  - `:last-child`: Selects the last child of its parent.
  - `:nth-child(n)`: Selects the nth child of its parent. Supports patterns like `nth-child(2n)` (even children) or `nth-child(2n+1)` (odd children).
  - `:nth-last-child(n)`: Selects the nth child from the end of its parent.
  - `:only-child`: Selects an element that is the only child of its parent.
  - `:first-of-type`: Selects the first element of its type within its parent.
  - `:last-of-type`: Selects the last element of its type within its parent.
  - `:nth-of-type(n)`: Selects the nth element of its type within its parent.

- `:nth-last-of-type(n)`: Selects the nth element from the end of its type within its parent.
- **Example: Using Structural Pseudo-Classes**

```
p:first-child {
  font-weight: bold; /* Makes the first paragraph bold */
}

li:nth-child(odd) {
  background-color: #f0f0f0; /* Styles odd list items */
}

tr:nth-of-type(even) {
  background-color: #e0e0e0; /* Styles even table rows */
}

p:last-of-type {
  color: red; /* Styles the last paragraph differently */
}
```

- **8.1.3. Form-Related Pseudo-Classes**
  - These pseudo-classes target form elements based on their state:
  - `:disabled`: Selects disabled elements.
  - `:enabled`: Selects enabled elements.
  - `:checked`: Selects checked radio buttons or checkboxes.
  - `:required`: Selects elements with the `required` attribute.
  - `:optional`: Selects elements without the `required` attribute.
  - `:valid`: Selects form elements with valid input.
  - `:invalid`: Selects form elements with invalid input.
  - **Example: Styling Form States**

```
input:required {
  border: 2px solid blue; /* Required fields have a blue b
order */
}

input:invalid {
  border-color: red; /* Invalid fields have a red border
*/
}

input:checked {
  background-color: green; /* Checked checkboxes or radio
buttons are green */
}

button:disabled {
  background-color: grey; /* Disabled buttons are grey */
}
```

- **8.2. Pseudo-Elements**
  - **Pseudo-elements** are used to style specific parts of an element's content. They allow you to apply styles to sub-parts of elements without having to add more HTML.
  - **8.2.1 `::before` and `::after`:**
    - These pseudo-elements are used to insert content before or after an element's content.
    - `::before`: Inserts content before the content of the selected element.
    - `::after`: Inserts content after the content of the selected element.
    - **Note:** When using `::before` and `::after`, the `content` property is mandatory. It defines what content to insert (can be text, image URLs, etc.).
    - **Example: Using `::before` and `::after`**

```
.quote::before {
  content: "“"; /* Adds a quote mark before the text */
  font-size: 2rem;
  color: grey;
}

.quote::after {
  content: "”"; /* Adds a quote mark after the text */
  font-size: 2rem;
  color: grey;
}
```

- **8.2.2 `::first-line` and `::first-letter`:**
  - These pseudo-elements allow styling of the first line or the first letter of a block-level element.
  - `::first-line`: Applies styles to the first line of a block-level element.
  - `::first-letter`: Applies styles to the first letter of a block-level element.
  - **Example: Styling the First Line and First Letter**

```
p::first-line {
  font-weight: bold; /* Makes the first line bold */
  color: blue;
}

p::first-letter {
  font-size: 2em; /* Makes the first letter larger */
  color: red;
  float: left; /* Floats the first letter to the left */
  margin-right: 5px;
}
```

- **8.2.3. `::selection`:**

- The `::selection` pseudo-element styles the portion of an element that is selected by the user.
- **Example: Using `::selection`**

```
::selection {
  background-color: yellow; /* Changes the background colo
r of selected text */
  color: black; /* Changes text color of selected text */
}
```

- **8.2.4. `::placeholder`:**
  - The `::placeholder` pseudo-element allows you to style the placeholder text of an input or textarea.
  - **Example: Using `::placeholder`**

```
input::placeholder {
  color: grey; /* Changes color of the placeholder text */
  font-style: italic; /* Makes placeholder text italic */
}
```

## 9. Rendering Engines

- Rendering engines (browsers' core components) interpret HTML, CSS, and JavaScript to display web pages:
  - **Blink**: Used by Chrome, Opera, and Edge.
  - **WebKit**: Used by Safari.
  - **Gecko**: Used by Firefox.
- Each engine has slight differences in rendering, so cross-browser compatibility testing is important.

## 10. CSS Box Model: Deep Dive

- The **CSS Box Model** is the foundation of layout design in CSS. It represents the rectangular boxes that are generated for elements in the document tree and consists of the following four components:
  1. **Content**: The actual content of the element (text, image, etc.).
  2. **Padding**: Space between the content and the border.
  3. **Border**: A line that surrounds the padding and content.
  4. **Margin**: The outermost space that creates distance between the element and its neighboring elements.
- **10.1. Understanding the Box Model Components**
  - **Content**: The actual content inside the element (e.g., text or images).
  - **Padding**: The space inside the element, between the content and the border. Padding adds space inside the element's background but does not affect the element's position relative to other elements.
    - Controlled with: `padding`, `padding-top`, `padding-right`, `padding-bottom`, `padding-left`.
  - **Border**: The line surrounding the padding (if any) and the content. It adds thickness around the content and padding area.
    - Controlled with: `border`, `border-width`, `border-style`, `border-color`.
  - **Margin**: The outermost space between the element's border and other elements. Margins create space outside the border.
    - Controlled with: `margin`, `margin-top`, `margin-right`, `margin-bottom`, `margin-left`.
- **10.2. Visualizing the Box Model**

```
                              |-------- Margin ---------|
                              |-------------------------|
                              |   |----- Border ------|   |
                              |   |-------------------|   |
                              |   | |--- Padding ---| |   |
                              |   | |---------------| |   |
```

```
                              |   | |    Content    | |   |
                              |   | |---------------| |   |
                              |   | |--- Padding ---| |   |
                              |   |-------------------|   |
                              |   |----- Border ------|   |
                              |---------------------------|
                              |-------- Margin ---------|
```

- **10.3. CSS Properties Related to the Box Model**

1. **Content**: Defined by properties like `width`, `height`, `min-width`, `max-width`, `min-height`, and `max-height`.
2. **Padding**:
   - `padding`: Shorthand to set padding on all four sides.
   - `padding-top`, `padding-right`, `padding-bottom`, `padding-left`: Set padding on individual sides.
   - **Example:**

```css
.box {
  padding: 10px; /* Sets padding of 10px on all sides */
  padding-top: 20px; /* Sets padding of 20px on the top */
}
```

3. **Border**:
   - `border`: Shorthand to set the width, style, and color of all four sides.
   - `border-width`, `border-style`, `border-color`: Control the width, style (solid, dashed, etc.), and color of the border.
   - **Example:**

```css
.box {
  border: 2px solid black; /* 2px solid border around the
```

```
box */
}
```

4. **Margin**:
    - `margin`: Shorthand to set margin on all four sides.
    - `margin-top`, `margin-right`, `margin-bottom`, `margin-left`: Set margin on individual sides.
    - **Example:**

```
.box {
  margin: 20px; /* Sets margin of 20px on all sides */
  margin-bottom: 10px; /* Sets margin of 10px on the botto
m */
}
```

- **10.4. Box-Sizing Property:**
    - The `box-sizing` property controls how the total width and height of an element are calculated.
    - `content-box` (default): The `width` and `height` properties include only the content. Padding and border are added outside the width and height.
    - `border-box`: The `width` and `height` properties include content, padding, and border. This makes managing element size easier as it includes everything inside the specified width and height.
    - **Example: Using `box-sizing`**

```
.box {
  width: 200px;
  padding: 20px;
  border: 5px solid black;
  box-sizing: border-box; /* The total width is 200px, inc
luding padding and border */
}
```

- Using `box-sizing: border-box` simplifies calculations and avoids overflow issues when adding padding and borders.

- **10.5. Margin Collapsing**
  - When two vertical margins meet, they may **collapse** into a single margin. The resulting margin will be equal to the larger of the two adjacent margins.
  - **Example: Margin Collapsing**
    - If the `first` paragraph has a `margin-bottom` of 20px, and the `second` paragraph has a `margin-top` of 30px, the total margin between them is 30px (the larger of the two margins), not 50px.

```
<div class="container">
    <p class="first">First paragraph</p>
    <!-- Has margin-bottom: 20px -->
    <p class="second">Second paragraph</p>
    <!-- Has margin-top: 30px -->
</div>
```

## 11. Advanced Box Model Techniques

- **11.1. Centering Elements**
  - **Horizontally Centering with Margins:**
    - To center a block element (like a `<div>`) horizontally, you can use `margin: auto`:

```
.centered-box {
  width: 50%; /* Set a width */
  margin: 0 auto; /* Auto margins will center the block */
}
```

  - **Vertically Centering with Flexbox:**
    - To center content vertically and horizontally **within** a container, use `display: flex` and align properties:

```css
.center-container {
  display: flex;
  justify-content: center; /* Horizontal alignment */
  align-items: center; /* Vertical alignment */
  height: 100vh; /* Full viewport height */
}
```

- **11.2. Controlling Overflow**
  - `overflow`: Specifies what happens if content overflows an element's box.
    - `visible` (default): Overflowing content is not clipped and may be visible outside the element's box.
    - `hidden`: Overflowing content is clipped and not visible.
    - `scroll`: Adds scrollbars to view overflowing content.
    - `auto`: Adds scrollbars only when needed.
  - **Example: Using Overflow**

```css
.overflow-box {
  width: 200px;
  height: 100px;
  overflow: scroll; /* Adds scrollbars if content overflow
s */
}
```

- **11.3. Advanced Layout with Grid and Flexbox**
  - **CSS Flexbox**: A one-dimensional layout model that distributes space along a single axis (horizontal or vertical).
  - **CSS Grid**: A two-dimensional layout model that allows for more complex designs.
  - **Flexbox Example:**

```
.flex-container {
  display: flex;
  justify-content: space-between; /* Distributes space eve
nly between items */
  align-items: center; /* Aligns items vertically in the c
enter */
}
```

- **Grid Example:**

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr); /* Creates 3 equa
l-width columns */
  gap: 20px; /* Adds space between grid items */
}
```

## 12. CSS Cascade Hierarchy

The **CSS Cascade** determines which styles are applied to an element when multiple conflicting rules could apply. It considers three main factors:

1. **Source Order (Order of Appearance)**
2. **Specificity**
3. **Importance (Use of `!important`)**

- **12.1. Source Order:**
  - When two or more rules have the same specificity, the rule that appears last in the CSS file (or inline styles) takes precedence. This is known as the "source order."
  - **Example:**

```
p {
  color: red;
```

```
}

p {
  color: blue; /* This rule will be applied as it comes la
ter */
}
```

- **12.2. Specificity:**
  - **Specificity** is a weight system that determines which rule is more "specific" and thus should be applied. Specificity is calculated using four levels:
    - **Inline Styles**: `1000` points (if an element has a style defined directly in the HTML via the `style` attribute).
    - **ID Selectors**: `0100` points (each ID selector adds 100 points).
    - **Class, Attribute, and Pseudo-Class Selectors**: `0010` points (each class, attribute, or pseudo-class adds 10 points).
    - **Element and Pseudo-Element Selectors**: `0001` points (each element or pseudo-element adds 1 point).
  - The specificity score is calculated by adding these points. A rule with a higher specificity will always override a rule with lower specificity.
  - **Example:**
    - In this example, the `#header` selector will be applied because it has the highest specificity (0100). If an element matches multiple selectors, the one with the highest specificity takes precedence.

```
#header { color: red; }   /* Specificity: 0100 (ID) */
.header { color: blue; }  /* Specificity: 0010 (class) */
header { color: green; }  /* Specificity: 0001 (element)
*/
```

  - **Another Example:**

- `#example` has the highest specificity (0100) and will make the text color red.

```
<div id="example" class="highlight">
    Example Text
</div>
```

```
/* CSS Rules */
div { color: black; }           /* Specificity: 0001 */
#example { color: red; }        /* Specificity: 0100 */
.highlight { color: blue; }     /* Specificity: 0010 */
```

- **12.3. Importance (`!important`):**
  - The `!important` rule is a powerful way to override other styles, regardless of their specificity or source order. When a rule is marked as `!important`, it overrides all other conflicting rules unless another `!important` rule with higher specificity is present.
  - **Example:**

```
p {
    color: blue !important; /* This will take precedence o
ver any other rule */
}

p {
    color: red; /* This rule is ignored */
}
```

  - **Important Note:** Overusing `!important` can make CSS hard to maintain and debug. It's best used sparingly and only when necessary.

## 13. Block Elements vs. Inline Elements in HTML

- **13.1 Block Elements: Block elements** (or **block-level elements**) are elements that:

1. Always start on a new line, occupying the full width available (their parent container's width) by default.
2. Take up the full width of the container, unless a specific width is set.
3. Can contain other block elements or inline elements.
4. Allow you to set properties like `width`, `height`, `margin`, and `padding`, which affect the layout on a larger scale.

- **Common Block Elements in HTML:**
  - `<div>`: A generic container used to group content for styling or layout purposes.
  - `<p>`: Represents a paragraph of text.
  - `<h1>` **to** `<h6>`: Headings, where `<h1>` is the highest (largest) level and `<h6>` is the lowest (smallest).
  - `<header>`: Represents the introductory content or a set of navigational links for its nearest ancestor sectioning content or the `body` element.
  - `<footer>`: Represents the footer for its nearest ancestor sectioning content or the `body` element.
  - `<section>`: Represents a standalone section of content that is thematically related.
  - `<article>`: Represents a self-contained piece of content, like a blog post or news article.
  - `<nav>`: Represents a section with navigation links.
  - `<aside>`: Represents content tangentially related to the content around it (like a sidebar).
  - `<main>`: Represents the main content of a document.
  - `<form>`: Represents a form that collects user input.
  - `<ul>`, `<ol>`, **and** `<li>`: Lists, both unordered (`<ul>`) and ordered (`<ol>`), with list items (`<li>`).
  - `<table>`: Represents a table, which includes other block-level elements like `<tr>`, `<th>`, and `<td>`.

        - `<figure>`: Represents self-contained content, like illustrations, diagrams, photos, or code snippets.

- **13.2. Inline Elements: Inline elements** are elements that:
    1. Do not start on a new line; they appear "in line" with the content around them.
    2. Only take up as much width as necessary (width of their content).
    3. Cannot contain block-level elements (but they can contain other inline elements).
    4. Generally, do not respect the `width` and `height` properties (these properties have no effect on inline elements).
    5. Only allow properties like `padding`, `margin`, and `border` on the left and right sides, not on the top and bottom.

    - **Common Inline Elements in HTML**
        - `<span>`: A generic container used to group inline elements or text for styling.
        - `<a>`: Represents a hyperlink.
        - `<img>`: Embeds an image.
        - `<strong>`: Represents strong importance, typically displayed as bold text.
        - `<em>`: Represents emphasized text, typically displayed as italic.
        - `<br>`: Inserts a line break.
        - `<i>`: Represents text in an alternative voice or mood (italic text).
        - `<b>`: Represents text stylistically different from normal text without conveying additional importance (bold text).
        - `<u>`: Represents text with an underline.
        - `<small>`: Represents side comments or small print.
        - `<mark>`: Represents text that has been highlighted or marked for reference.

- `<code>`: Represents a fragment of computer code.
- `<input>`: Represents an input control in a form.

- **13.3. Difference Between Block and Inline Elements**

| Property | **Block Elements** | **Inline Elements** |
|---|---|---|
| **Display Behavior** | Starts on a new line | Does not start on a new line |
| **Width** | Takes up the full width of its container | Takes only as much width as necessary |
| **Containment** | Can contain both block and inline elements | Can contain only other inline elements |
| **CSS Properties** | `width`, `height`, `margin`, `padding` fully applicable | Only `padding` and `margin` on the left and right sides are fully applicable |
| **Example Elements** | `<div>`, `<p>`, `<h1>`, `<ul>`, `<table>`, etc. | `<span>`, `<a>`, `<img>`, `<strong>`, `<em>`, etc. |

- **13.4. Converting Between Block and Inline Elements:** You can change an element's display behavior using the `display` property in CSS:
  - **Converting Block to Inline:**

```
.block-to-inline {
  display: inline;
}
```

  - **Converting Inline to Block:**

```
.inline-to-block {
  display: block;
}
```

  - **Other Display Values:**
    - `display: inline-block;`: Behaves like an inline element but allows setting width, height, margins, and padding.

- `display: flex;`: Applies a flex container layout to the element.
- `display: grid;`: Applies a grid container layout to the element.
- `display: none;`: Hides the element from the page entirely.

## 14. Text Formatting in CSS

Text formatting in CSS includes controlling aspects like font family, size, color, weight, line height, letter spacing, text decoration, and more. These properties help make text more readable and visually appealing.

- **14.1 Font Properties**

1. `font-family`: Specifies the font or fonts used for the text. You can specify multiple fonts as a fallback mechanism.

   **Example:**

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

2. `font-size`: Specifies the size of the font. It can be defined using absolute units (`px`, `pt`) or relative units (`em`, `rem`, `%`).

   **Example:**

   ```
   h1 {
     font-size: 2rem; /* Relative to the root element */
   }

   p {
     font-size: 16px; /* Absolute size */
   }
   ```

3. `font-weight`: Specifies the thickness of the font. Values range from `100` (thin) to `900` (bold). You can also use predefined keywords like `normal`, `bold`, `bolder`, `lighter`.

   **Example:**

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

h2 {
  font-weight: 300; /* Light font weight */
}
```

4. `font-style`: Specifies the style of the font, such as `normal`, `italic`, or `oblique`.

   **Example:**

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

5. `font-variant`: Specifies whether or not a text should be displayed in a small-caps font. It can take values like `normal` or `small-caps`.

   **Example:**

   ```
   p {
     font-variant: small-caps;
   }
   ```

6. `font` **(Shorthand Property)**: A shorthand for setting `font-style`, `font-variant`, `font-weight`, `font-size`, `line-height`, and `font-family` all in one declaration.

   **Example:**

   ```
   p {
     font: italic small-caps bold 16px/1.5 "Arial", sans-serif;
   }
   ```

- **14.2 Color and Background Properties**

  1. `color`: Specifies the color of the text.

     **Example:**

     ```
     p {
       color: #333333; /* Hexadecimal color */
     }
     ```

  2. `background-color`: Specifies the background color behind the text.

     **Example:**

     ```
     p {
       background-color: yellow;
     }
     ```

  3. `background-image`: Adds a background image behind the text.

     **Example:**

     ```
     h1 {
       background-image: url('background.png');
     }
     ```

- **14.3 Spacing Properties**

  1. `line-height`: Sets the height of each line of text, improving readability. A value of `1.5` is generally considered optimal for most text.

     **Example:**

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

  2. `letter-spacing`: Controls the spacing between letters (also called tracking). Positive values increase spacing, while negative values decrease it.

**Example:**

```
h2 {
  letter-spacing: 2px; /* Adds 2 pixels of space betwee
n each letter */
}
```

3. `word-spacing`: Controls the spacing between words.

   **Example:**

   ```
   p {
     word-spacing: 5px; /* Adds 5 pixels of space between
   each word */
   }
   ```

- **14.4 Text Alignment and Decoration**

1. `text-align`: Specifies the horizontal alignment of text within its container. Common values are `left`, `right`, `center`, and `justify`.

   **Example:**

   ```
   p {
     text-align: justify; /* Stretches the text to align b
   oth left and right */
   }
   ```

2. `text-decoration`: Adds decorations like `underline`, `overline`, `line-through`, or `none` to the text.

   **Example:**

   ```
   a {
     text-decoration: underline;
   }
   ```

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

   **Example:**

```
h1 {
  text-transform: uppercase; /* Converts all text to up
percase */
}
```

4. `text-indent`: Indents the first line of text.

   **Example:**

```
p {
  text-indent: 50px; /* Indents the first line by 50 pi
xels */
}
```

5. `text-shadow`: Adds a shadow effect to the text. The values define the horizontal offset, vertical offset, blur radius, and color.

   **Example:**

```
h2 {
  text-shadow: 2px 2px 5px grey;
}
```

- **14.5 More Advanced Text Formatting**

1. `white-space`: Controls how whitespace inside an element is handled. Common values are `normal`, `nowrap`, `pre`, `pre-wrap`, and `pre-line`.

   **Example:**

```
p {
  white-space: pre; /* Preserves whitespace and line br
```

```
eaks */
}
```

2. `direction`: Specifies the text direction (`ltr` for left-to-right, `rtl` for right-to-left).

   **Example:**

```
p {
  direction: rtl; /* Sets text direction to right-to-le
ft */
}
```

## 15. Text Layout in CSS

Text layout involves how text is positioned and arranged within its container, which includes properties for aligning, spacing, and controlling overflow.

- **15.1 Aligning Text and Content:**

  1. **Horizontal Alignment with `text-align`:**
     - `left`: Aligns text to the left.
     - `right`: Aligns text to the right.
     - `center`: Centers the text.
     - `justify`: Stretches the text to fill the line width.

     **Example:**

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

  2. **Vertical Alignment with `vertical-align`:**
     - Used to vertically align inline or inline-block elements. Common values are `baseline`, `top`, `middle`, `bottom`, and `sub`.

     **Example:**

```css
.text-vertical {
  vertical-align: middle;
}
```

- **15.2 Text Overflow and Clipping:** When the text exceeds its container, you can control how it is displayed using:
  1. `overflow`: Controls how content is handled when it overflows its container.
     - `visible`: Default; overflow is visible.
     - `hidden`: Overflow is clipped, and the rest is invisible.
     - `scroll`: Adds scroll bars to see overflow content.
     - `auto`: Adds scroll bars only when necessary.

     **Example:**

     ```css
     .overflow-hidden {
       overflow: hidden;
     }
     ```

  2. `text-overflow`: Specifies how to signal overflowed content. Values are `clip` (default) and `ellipsis`.

     **Example:**

     ```css
     .ellipsis {
         white-space: nowrap;
         overflow: hidden;
         text-overflow: ellipsis;
         /* Displays "..." for overflowed text */
     }
     ```

  3. `white-space`: Controls the handling of white space and text wrapping.

     **Example:**

```css
.no-wrap {
  white-space: nowrap; /* Prevents line breaks */
}
```

- **15.3 Creating Text Columns:** You can create multi-column text layouts with the following properties:
  1. `column-count`: Specifies the number of columns.

     **Example:**

     ```css
     .columns {
       column-count: 3; /* Creates three columns */
     }
     ```

  2. `column-gap`: Specifies the gap between columns.

     **Example:**

     ```css
     .columns {
       column-gap: 20px; /* Sets 20px gap between columns */
     }
     ```

  3. `column-rule`: Specifies a line (rule) between columns.

     **Example:**

     ```css
     .columns {
       column-rule: 1px solid #ccc;
       /* Adds a 1px solid line between columns */
     }
     ```

## 16. CSS Flexbox

- **16.1. Flexbox Overview**
  - Flexbox (Flexible Box Layout) is a CSS module that provides a more efficient way to lay out, align, and distribute space among items in a

container, even when their sizes are unknown or dynamic. It works well for both small-scale layouts (e.g., navigation bars) and larger applications (e.g., entire web pages).

- **16.2. Key Concepts of Flexbox**
  - **Flex Container**: The parent element that holds the flex items. It is defined using `display: flex;` or `display: inline-flex;`.
  - **Flex Items**: The direct children of the flex container. They are the elements that will be laid out using Flexbox.
- **16.3. Flex Container Properties**
  - **16.3.1.** `display` **Property:** Sets the flex container to either block-level or inline-level.
    - `display: flex;`: Creates a block-level flex container.
    - `display: inline-flex;`: Creates an inline-level flex container.
  - **16.3.2.** `flex-direction` **Property:** Defines the direction in which the flex items are placed in the flex container.
    - `row` (default): Items are placed horizontally from left to right.
    - `row-reverse`: Items are placed horizontally from right to left.
    - `column`: Items are placed vertically from top to bottom.
    - `column-reverse`: Items are placed vertically from bottom to top.

    **Example:**

    ```
    .flex-container {
      display: flex;
      flex-direction: row;
      /* Items are placed horizontally (default) */
    }
    ```

  - **16.3.3.** `flex-wrap` **Property:** Specifies whether the flex items should wrap onto multiple lines.
    - `nowrap` (default): All flex items will be on one line.

- `wrap`: Flex items will wrap onto multiple lines from top to bottom.
- `wrap-reverse`: Flex items will wrap onto multiple lines from bottom to top.

**Example:**

```
.flex-container {
  display: flex;
  flex-wrap: wrap;
  /* Items will wrap onto multiple lines */
}
```

- **16.3.4.** `justify-content` **Property:** Defines how the flex items are aligned along the main axis (e.g. horizontal axis):
  - `flex-start` (default): Items are packed toward the start of the main axis.
  - `flex-end`: Items are packed toward the end of the main axis.
  - `center`: Items are centered along the main axis.
  - `space-between`: Items are evenly distributed with equal space between them.
  - `space-around`: Items are evenly distributed with equal space around them.
  - `space-evenly`: Items are evenly distributed with equal space between and around them.

  **Example:**

```
.flex-container {
  display: flex;
  justify-content: center;
  /* Centers items horizontally */
}
```

- **16.3.5.** `align-items` **Property:** Defines how flex items are aligned along the cross axis (e.g. vertical axis):

- `stretch` (default): Items are stretched to fill the container (if no height is set).
- `flex-start`: Items are aligned to the start of the cross axis.
- `flex-end`: Items are aligned to the end of the cross axis.
- `center`: Items are centered along the cross axis.
- `baseline`: Items are aligned such that their baselines align.

**Example:**

```
.flex-container {
  display: flex;
  align-items: center;
  /* Aligns items vertically to the center */
}
```

- **16.3.6.** `align-content` **Property:** Defines how multiple lines of flex items are aligned along the cross axis (e.g. vertical y axis) when there is extra space in the flex container. It only applies when `flex-wrap` is set to `wrap` or `wrap-reverse`.
  - `flex-start`: Lines are packed toward the start of the cross axis.
  - `flex-end`: Lines are packed toward the end of the cross axis.
  - `center`: Lines are packed toward the center of the cross axis.
  - `space-between`: Lines are evenly distributed with equal space between them.
  - `space-around`: Lines are evenly distributed with equal space around them.
  - `space-evenly`: Lines are evenly distributed with equal space between and around them.
  - `stretch` (default): Lines stretch to fill the container.

**Example:**

```
.flex-container {
  display: flex;
  flex-wrap: wrap; /* Allows wrapping */
  align-content: space-between; /* Distributes wrapped lin
es evenly */
}
```

- **16.4. Flex Item Properties:**
  - **16.4.1.** `order` **Property:** Defines the order in which flex items are laid out in the flex container. The default value is `0`. Items with a lower value appear first.

    **Example:**

    ```
    .flex-item {
      order: 2; /* This item will appear second */
    }

    .flex-item-first {
      order: 1; /* This item will appear first */
    }
    ```

  - **16.4.2.** `flex-grow` **Property:** Defines how much a flex item should grow relative to the rest of the flex items when there is extra space in the container. The default value is `0` (no growth).

    **Example:**

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

  - **16.4.3.** `flex-shrink` **Property:** Defines how much a flex item should shrink relative to the rest of the flex items when there is not enough space in the container. The default value is `1` (shrink).

**Example:**

```
.flex-item {
  flex-shrink: 0;
  /* Item will not shrink even if there is not enough s
pace */
}
```

- **16.4.4.** `flex-basis` **Property:** Defines the initial size of a flex item before the remaining space is distributed. It can be set using units like `px`, `em`, `rem`, or percentages.

  **Example:**

  ```
  .flex-item {
    flex-basis: 200px;
    /* Item will start with a width of 200px */
  }
  ```

- **16.4.5.** `align-self` **Property:** Allows the default alignment (`align-items`) to be overridden for individual flex items.
  - Values: `auto` (default), `flex-start`, `flex-end`, `center`, `baseline`, `stretch`.

  **Example:**

  ```
  .flex-item {
    align-self: center;
    /* This item will be centered along the cross axis */
  }
  ```

- **16.5. Practical Examples of Flexbox:**
  - **16.5.1. Centering Elements with Flexbox:** Flexbox makes it easy to center elements both horizontally and vertically.

```
<div class="flex-container">
    <div class="flex-item">Centered Item</div>
```

```
</div>
```

```
.flex-container {
  display: flex;
  justify-content: center; /* Center horizontally */
  align-items: center; /* Center vertically */
  height: 100vh; /* Full viewport height */
}
```

- **16.5.2. Creating a Simple Navigation Bar:** Flexbox can also be used to create a responsive navigation bar.

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

```
.nav-bar {
  display: flex;
  /* Evenly distributes space between items */
  justify-content: space-between;
  background-color: #333;
  padding: 10px;
}

.nav-bar a {
  color: white;
  text-decoration: none;
  padding: 10px;
}
```

## 17. CSS Grid: Deep Dive

- **17.1. Grid Overview**
  - **CSS Grid Layout** is a two-dimensional layout system for the web. It allows you to create complex, responsive layouts that are optimized for various devices and screen sizes.
- **17.2. Key Concepts of Grid**
  - **Grid Container**: The parent element that contains grid items. Defined by applying `display: grid;` or `display: inline-grid;`.
  - **Grid Items**: The direct children of the grid container. These elements are laid out within the grid.
- **17.3. Grid Container Properties**
  - **17.3.1.** `display` **Property:** Sets the element as a grid container.
    - `display: grid;`: Creates a block-level grid container.
    - `display: inline-grid;`: Creates an inline-level grid container.
  - **17.3.2.** `grid-template-columns` **and** `grid-template-rows` **Properties:** Defines the number and size of columns and rows in the grid.
    - `grid-template-columns`: Specifies the number and size of columns.
    - `grid-template-rows`: Specifies the number and size of rows.

    **Example: Defining Columns and Rows**

```
.grid-container {
  display: grid;
  grid-template-columns: 200px 1fr 100px;
  /* Defines 3 columns: 200px, 1 fraction unit, 100px */
  grid-template-rows: auto 1fr auto;
  /* Defines 3 rows: auto height, 1 fraction unit, auto he
ight */
}
```

  - **17.3.3.** `grid-column-gap`, `grid-row-gap`, **and** `grid-gap` **Properties:** Defines the gap between columns, rows, or both.

- `grid-column-gap`: Sets the gap between columns.
- `grid-row-gap`: Sets the gap between rows.
- `grid-gap`: Shorthand for setting both `grid-column-gap` and `grid-row-gap`.

**Example: Using Grid Gaps**

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-gap: 20px;
  /* Adds a 20px gap between columns and rows */
}
```

- **17.3.4. `justify-items` and `align-items` Properties:** Aligns grid items along the column axis (`justify-items`) or row axis (`align-items`).
  - `justify-items`: Aligns items horizontally within their grid area (`start`, `end`, `center`, `stretch`).
  - `align-items`: Aligns items vertically within their grid area (`start`, `end`, `center`, `stretch`).

**Example: Using Justify and Align Items**

```
.grid-container {
  display: grid;
  justify-items: center; /* Center-aligns items horizontal
ly */
  align-items: center; /* Center-aligns items vertically
*/
}
```

- **17.3.5. `justify-content` and `align-content` Properties:** Controls the alignment of the entire grid within the grid container.
  - `justify-content`: Aligns the grid horizontally within the container (`start`, `end`, `center`, `space-between`, `space-around`, `space-evenly`).

- `align-content`: Aligns the grid vertically within the container (`start`, `end`, `center`, `space-between`, `space-around`, `space-evenly`).

**Example: Using Justify and Align Content**

```
.grid-container {
  display: grid;
  justify-content: center;
  /* Center-aligns the entire grid horizontally */
  align-content: center;
  /* Center-aligns the entire grid vertically */
}
```

- **17.4. Grid Item Properties**
  - **17.4.1 `grid-column` and `grid-row` Properties:** Defines how many columns or rows an item will span.
    - `grid-column`: Specifies the starting and ending columns for an item.
    - `grid-row`: Specifies the starting and ending rows for an item.

**Example: Spanning Grid Items**

```
.grid-item {
  grid-column: 1 / 3; /* Spans from column line 1 to 3 */
  grid-row: 2 / 4; /* Spans from row line 2 to 4 */
}
```

  - **17.4.2 `grid-area` Property:** Assigns an item to a named grid area defined by `grid-template-areas`.

**Example: Using Grid Areas**

```
.grid-item {
  grid-area: header;
  /* Places the item in the 'header' grid area */
}
```

- **17.5. Practical Examples of Grid: Creating a Basic Grid Layout**

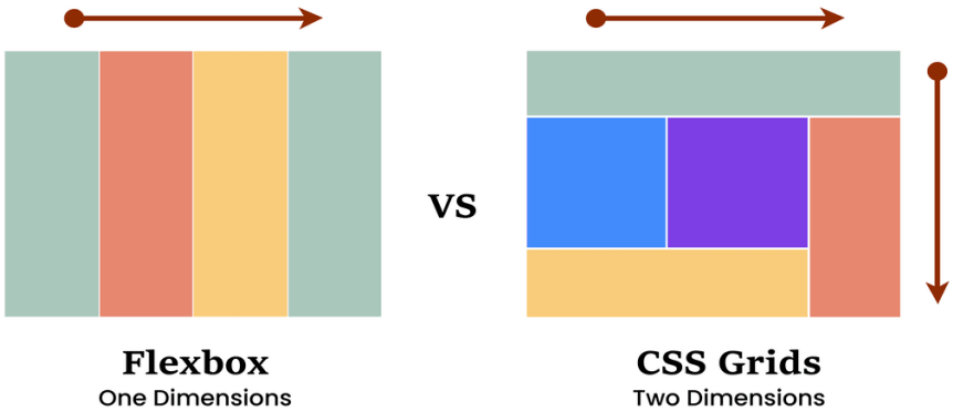
```
<div class="grid-container">
    <div class="header">Header</div>
    <div class="nav">Navigation</div>
    <div class="content">Content</div>
    <div class="aside">Aside</div>
    <div class="footer">Footer</div>
</div>
```

```
.grid-container {
    display: grid;
    grid-template-areas:
        "header header header"
        "nav content aside"
        "footer footer footer";
    grid-gap: 10px;
}

.header {
    grid-area: header;
}

.nav {
    grid-area: nav;
}

.content {
    grid-area: content;
}

.aside {
    grid-area: aside;
}
```

```
.footer {
    grid-area: footer;
}
```

## 18. Choosing Between Flexbox and Grid

- **Use Flexbox** for one-dimensional layouts where you need to distribute space along a single axis (row or column). Flexbox is ideal for navigation bars, aligning items, or creating components that need flexibility.
- **Use Grid** for two-dimensional layouts where you need control over both rows and columns. Grid is perfect for creating complete page layouts, complex components, or any design that requires precise control over the placement of elements.




## 19. Responsive Web Design

- **Responsive Web Design (RWD)** is a web development approach that ensures a website's layout and content adapt dynamically to different screen sizes, orientations, and devices. This approach enhances usability and provides an optimal viewing experience for users, whether they're on a desktop, tablet, or smartphone.

- Responsive web design relies on **flexible grids and layouts**, **media queries**, and **responsive images** to adjust content based on the device's characteristics. Let’s dive deeper into these concepts and learn how to create a responsive web design.
- **19.1. Core Principles of Responsive Web Design:**
  1. **Fluid Grids**: Use relative units like percentages, `em`, or `rem` instead of fixed units like pixels (`px`) to create layouts that resize proportionally.
  2. **Flexible Images**: Ensure images and other media scale appropriately within their containing elements using CSS properties like `max-width: 100%`.
  3. **Media Queries**: Use CSS media queries to apply different styles for different screen sizes and device capabilities.
- **19.2. Setting Up a Responsive Layout**
  - **19.2.1. Fluid Grids:** Fluid grids use relative units like percentages (`%`), `em`, or `rem` to define widths and spacing, ensuring that the layout scales fluidly across different devices.

**Example: Creating a Fluid Grid Layout**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initi
al-scale=1.0"> <!-- Ensures proper scaling on mobile devic
es -->
  <title>Responsive Web Design Example</title>
  <style>
    /* Basic reset for box sizing */
    * {
      box-sizing: border-box;
      margin: 0;
      padding: 0;
    }
```

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

    /* Fluid Grid Container */
    .container {
      width: 90%;
      /* Use a percentage for fluid width */
      max-width: 1200px; /* Maximum width constraint */
      margin: 0 auto; /* Centers the container */
      padding: 10px;
    }

    .row {
      display: flex;
      flex-wrap: wrap; /* Allows the row to wrap onto mult
iple lines */
      margin: -10px; /* Negative margin to compensate for
padding */
    }

    .column {
      flex: 1; /* Each column will take equal space */
      padding: 10px;
    }

    /* Media query for smaller screens */
    @media (max-width: 768px) {
      .column {
          flex: 100%; /* Each column takes full width on s
maller screens */
      }
    }
  </style>
</head>
<body>
```

```
  <div class="container">
    <div class="row">
        <div class="column" style="background-color: light
blue;">Column 1</div>
        <div class="column" style="background-color: light
coral;">Column 2</div>
        <div class="column" style="background-color: light
green;">Column 3</div>
    </div>
  </div>
</body>
</html>
```

- **19.2.2. Flexible Images:** Flexible images adjust to the width of their containing elements, preventing them from overflowing and ensuring they look good on all screen sizes.

**Example: Making Images Responsive**

```
<img src="example.jpg" alt="Example Image" style="max-widt
h: 100%; height: auto;">
```

- **19.3. Media Queries**
  - **Media queries** are a key part of responsive design. They allow you to apply different styles depending on the user's device characteristics, such as the width, height, orientation, and resolution of the screen.
  - **Basic Syntax of Media Queries:** The basic syntax for a media query is:

    ```
    @media (condition) {
      /* CSS rules to apply when the condition is met */
    }
    ```

  - **Default Styles**: The default styles apply to all devices unless overridden by a media query.

- **Tablet Media Query**: The media query for screens `768px` wide or smaller adjusts the font size and padding.
- **Mobile Media Query**: The media query for screens `480px` wide or smaller adjusts the font size, changes the navbar layout to a vertical stack, and modifies the spacing between links.

- **19.4. Responsive Design with CSS Flexbox and Grid:** Both **Flexbox** and **Grid** are powerful tools for creating responsive layouts.
  - **19.4.1. Flexbox for Responsive Design:** Flexbox is great for one-dimensional layouts, like creating a responsive navbar or aligning items in a single row or column.

**Example: Responsive Navbar with Flexbox**

```
<nav class="navbar">
  <div class="navbar-logo">MyWebsite</div>
  <ul class="navbar-links">
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Services</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
  <div class="navbar-button">
    <a href="#" class="btn">Sign Up</a>
  </div>
</nav>
```

```
.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  background-color: #333;
  padding: 10px 20px;
}
```

```css
.navbar-links {
  display: flex;
  list-style: none;
}

.navbar-links li {
  margin-left: 20px;
}

.navbar-links a {
  color: white;
  text-decoration: none;
}

/* Responsive Media Query */
@media (max-width: 768px) {
  .navbar {
    flex-direction: column; /* Stacks navbar items vertica
lly */
    align-items: flex-start;
  }

  .navbar-links {
    flex-direction: column; /* Stacks links vertically */
    width: 100%; /* Full width for links */
  }

  .navbar-links li {
    margin: 10px 0; /* Adds space between links */
  }
}
```

- **19.5. Responsive Typography:**
  - Responsive typography adjusts the font size, line height, and other text properties based on the device or screen size.

- **19.5.1. Using Relative Units for Font Size:** Use relative units like `em` or `rem` to ensure that text scales appropriately.

```
body {
  font-size: 100%; /* Base font size (16px by default) */
}

h1 {
  font-size: 2em; /* 2 times the base size (32px) */
}

p {
  font-size: 1rem; /* Same as the base font size (16px) */
}
```

- **19.5.2 Fluid Typography with Viewport Units:** Use viewport units (`vw` and `vh`) for fluid typography that scales with the viewport size.

```
h1 {
  font-size: 4vw; /* 4% of the viewport width */
}
```

## 20. CSS Display Properties

- **The `display: block` Property:**
  - **Description**: An element with `display: block` is a block-level element. It takes up the full width available (by default) and starts on a new line.
  - **Examples**: `<div>`, `<p>`, `<h1>` to `<h6>`, `<section>`, `<article>`.
  - **Properties**: You can control width, height, margin, padding, etc.
- **The `display: inline` Property:**
  - **Description**: An element with `display: inline` is an inline-level element. It does not start on a new line and only takes up as much width as needed.
  - **Examples**: `<span>`, `<a>`, `<strong>`, `<em>`.

    - **Properties**: You cannot set width or height. Margins and padding will only affect spacing horizontally, not vertically.

- **The `display: inline-block` Property:**
    - **Description**: An inline-level element that allows setting width and height.
    - **Usage**: Useful for elements that need to flow inline but have some block-level properties (like width and height).
- **The `display: none` Property:**
    - **Description**: The element is completely removed from the document flow and does not take up any space. It's as if the element does not exist.
    - **Usage**: Used to hide elements without deleting them from the HTML.
- **The `display: flex` Property:**
    - **Description**: An element with `display: flex` becomes a flex container, and its children become flex items. Flexbox is a layout model that provides a flexible way to align and distribute space among items within a container.
    - **Usage**: Commonly used for responsive layouts, navigation menus, grids, etc.
    - **Properties**: Includes `justify-content`, `align-items`, `flex-direction`, `flex-wrap`, etc.
- **The `display: grid` Property:**
    - **Description**: An element with `display: grid` becomes a grid container, and its children become grid items. CSS Grid is a layout system for creating two-dimensional layouts (both rows and columns).
    - **Usage**: Useful for complex page layouts, such as creating a gallery or a webpage layout.
    - **Properties**: Includes `grid-template-columns`, `grid-template-rows`, `grid-gap`, etc.
- **The `display: table` Property:**
    - **Description**: Makes the element behave like a `<table>` element.
    - **Usage**: Used to create table-like layouts.

- **Related Properties**: `display: table-row`, `display: table-cell`, etc.

## 21. CSS Float & Clear Properties

**Float:** The `float` property in CSS is used for positioning and formatting content, e.g., floating an image to the left of a paragraph. It pushes the element to the left or right and allows text or inline elements to wrap around it.

- **Common Values for `float`:**
  - `float: left`: The element floats to the left side of its container.
  - `float: right`: The element floats to the right side of its container.
  - `float: none`: The element does not float (default).
- **How `float` Works:**
  - When you apply `float` to an element, it is removed from the normal document flow, meaning other elements will ignore its presence in terms of layout but will wrap around it.
  - For example, if you `float` an image to the left (`float: left`), the following text will wrap around it on the right side.
- **Problems with `float`:**
  - Floated elements are often removed from the normal flow of the document, which can cause container elements to collapse in height if they only contain floated elements.

**Clear**: The `clear` property is used to control the behavior of elements that follow floated elements. It prevents elements from wrapping around floated elements.

- **Common Values for `clear`:**
  - `clear: left`: The element will not appear next to floated elements on the left side.
  - `clear: right`: The element will not appear next to floated elements on the right side.
  - `clear: both`: The element will not appear next to floated elements on either side.

- `clear: none`: The element will allow floats on both sides (default).

- **How `clear` Works:**
  - If you have an element that follows a floated element and you do not want it to wrap around the floated element, you can use `clear` to push it down below the floated content.

## 22. CSS Position Properties

- **22.1. CSS `position` Property Overview:**
  - The `position` property specifies how an element is positioned in the document. The position can be controlled relative to its containing block or the viewport, depending on the value assigned to the property.
  - Common Values of `position`:
    1. `static` (default)
    2. `relative`
    3. `absolute`
    4. `fixed`
    5. `sticky`

- **22.2. CSS Property `position: static`:**
  - **Description**: This is the default value for all elements. Elements with `position: static` are placed according to the normal document flow.
  - **Characteristics**:
    - Elements do not move from their default position.
    - The `top`, `right`, `bottom`, and `left` properties have no effect on static elements.
  - **Use Case**: When no positioning is required. For example, regular paragraphs, headings, etc., usually remain `static`.

- **22.3. CSS Property `position: relative`:**

- **Description**: Elements with `position: relative` are positioned relative to their normal position in the document flow.
- **Characteristics**:
  - The element remains in the document flow, but you can move it using the `top`, `right`, `bottom`, and `left` properties.
  - The space for the element is still reserved in its original position, even if it's visually moved.
- **Use Case**: Used when you want to move an element slightly from its original place without affecting the layout around it.
- **Example**:

```
<div style="position: relative; top: 10px; left: 20px;">
  I am positioned 10px down and 20px to the right from my
original position.
</div>
```

- **22.4. CSS Property `position: absolute`:**
  - **Description**: An element with `position: absolute` is positioned relative to its nearest positioned ancestor (an ancestor with a position other than `static`), or the initial containing block (viewport) if no such ancestor exists.
  - **Characteristics**:
    - The element is removed from the normal document flow, meaning it does not affect the positioning of other elements.
      - It can be precisely positioned using the `top`, `right`, `bottom`, and `left` properties.
  - **Use Case**: Useful for positioning elements that need to be in a specific location regardless of the normal flow, such as a popup or tooltip.
  - **Example**:

```
<div style="position: relative;">
  <div style="position: absolute; top: 0; right: 0;">
```

```
    I am positioned absolutely within my relative parent.
  </div>
</div>
```

- In this example, the absolutely positioned element is aligned to the top-right corner of its relative parent.

- **22.5. CSS Property `position: fixed`:**
  - **Description**: An element with `position: fixed` is positioned relative to the viewport (the browser window).
  - **Characteristics**:
    - The element is removed from the normal document flow.
    - It stays fixed in the same place even when the page is scrolled.
    - The `top`, `right`, `bottom`, and `left` properties are used to control its position.
  - **Use Case**: Commonly used for sticky headers, footers, or elements that need to remain visible at all times (e.g., a back-to-top button).
  - **Example**:

```
<div style="position: fixed; bottom: 10px; right: 10px;">
  I am fixed to the bottom right corner of the viewport.
</div>
```

  - In this example, the element will always stay in the bottom-right corner of the viewport, regardless of scrolling.

- **22.6. CSS Property `position: sticky`:**
  - **Description**: An element with `position: sticky` toggles between `relative` and `fixed`, depending on the user's scroll position.
  - **Characteristics**:
    - It is positioned relative to its normal position until a certain scroll point is reached, after which it becomes fixed.

    - The `top`, `right`, `bottom`, or `left` properties define the point where the element will "stick".
  - **Use Case**: Useful for creating elements that should stick at the top of the page when scrolling, like a sticky header.
  - **Example**:

```
<div style="position: sticky; top: 0;">
  I become sticky when you scroll past me.
</div>
```

  - In this example, the element will stick to the top of the page once the page is scrolled to its position.

- **22.7. Using `top`, `right`, `bottom`, and `left` Properties:**
  - These properties are used to specify the exact position of an element when its `position` property is set to `relative`, `absolute`, `fixed`, or `sticky`.
    - `top`: Moves the element down from the top.
    - `right`: Moves the element left from the right.
    - `bottom`: Moves the element up from the bottom.
    - `left`: Moves the element right from the left.

## 23. CSS Animations

- **Key Concepts of CSS Animations:**
  - **@keyframes Rule:**
    - The @keyframes rule defines the animation's intermediate steps. You can set different styles at specific percentages (0% to 100%) of the animation's timeline.
  - **Animation Properties:** CSS properties applied to elements control how animations behave. Key ones include:
    - `animation-name`: The name of the `@keyframes` rule.
    - `animation-duration`: Duration of the animation (e.g., `2s`, `500ms`).

- `animation-timing-function`: Easing function (e.g., `ease`, `linear`, `ease-in-out`).
- `animation-delay`: Delay before the animation starts.
- `animation-iteration-count:` Number of times the animation repeats (e.g., `1`, `infinite`).
- `animation-direction`: Direction of animation (`normal`, `reverse`, `alternate`).
- `animation-fill-mode`: Specifies whether styles are applied before/after the animation.

- **Combining Animation Properties:** You can combine multiple animation properties in a shorthand:

```
animation: animation-name duration timing-function delay ite
```

- **Example 1: Basic Animation -** Animate a box moving from left to right:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <style>
    /* Keyframes for the animation */
    @keyframes moveRight {
      0% { transform: translateX(0); }
      100% { transform: translateX(200px); }
    }

    /* Apply the animation */
    .box {
      width: 50px;
      height: 50px;
      background-color: skyblue;
      animation: moveRight 2s ease-in-out infinite;
    }
  </style>
```

```
</head>
<body>
  <div class="box"></div>
</body>
</html>
```

- **Example 2: Fade In Animation -** Make an element fade in:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <style>
    @keyframes fadeIn {
      0% { opacity: 0; }
      100% { opacity: 1; }
    }

    .fade {
      width: 100px;
      height: 100px;
      background-color: coral;
      animation: fadeIn 3s ease-in-out forwards;
    }
  </style>
</head>
<body>
  <div class="fade"></div>
</body>
</html>
```

- **Example 3: Hover Triggered Animation:**

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <style>
    @keyframes grow {
      0% { transform: scale(1); }
      100% { transform: scale(1.5); }
    }

    .hover-box {
      width: 100px;
      height: 100px;
      background-color: lightgreen;
      transition: transform 0.3s;
    }

    .hover-box:hover {
      animation: grow 0.5s ease-in-out forwards;
    }
  </style>
</head>
<body>
  <div class="hover-box"></div>
</body>
</html>
```

- **Animation Timing Functions:** The animation-timing-function controls the speed of the animation over time. Common options:
  - `ease` (default): Starts slow, speeds up, then slows down.
  - `linear`: Moves at a constant speed.
  - `ease-in`: Starts slow.
  - `ease-out`: Ends slow.
  - `cubic-bezier`: Custom easing (e.g., cubic-bezier(0.68, -0.55, 0.27, 1.55)).