CSS Syntax Dictionary

This document provides a comprehensive reference for CSS syntax, including selectors, properties, values, and units, with examples and best practices.

CSS Selectors

Basic Selectors

Type Selector

Syntax:

```
element { property: value; }
```

Description: The type selector, also known as the element selector, selects all HTML elements of the specified type. It targets elements based on their node name (tag name). This is one of the most basic and commonly used selectors in CSS, allowing you to apply styles to all instances of a particular HTML element.

Example:

```
/* Selects all paragraph elements */
p {
  color: navy;
  line-height: 1.5;
  margin-bottom: 1em;
}

/* Selects all heading level 2 elements */
h2 {
  font-size: 1.5em;
  color: #333;
  margin-top: 1.5em;
}

/* Selects all button elements */
button {
  background-color: #4CAF50;
  color: white;
```

```
  padding: 10px 15px;
  border: none;
  border-radius: 4px;
}
```

Effect: The type selector applies the specified styles to all elements of the matching type throughout the document. This creates consistent styling for all instances of that element, establishing a base appearance that can be further customized with more specific selectors.

Best Practices: - Use type selectors for setting base styles that should apply to all instances of an element. - Avoid overly specific declarations with type selectors to maintain flexibility. - Consider the cascade and specificity when using type selectors with other more specific selectors. - Type selectors have low specificity, so their styles can be easily overridden when needed. - For more targeted styling, combine type selectors with class or attribute selectors.

Related Items: - Class selectors - ID selectors - Universal selector - Attribute selectors

Class Selector

Syntax:

```
.classname { property: value; }
```

Description: The class selector selects all elements with the specified class attribute. It targets elements based on the value of their `class` attribute, allowing you to apply the same styles to multiple elements regardless of their type. Class selectors are preceded by a period (.) character. Multiple classes can be applied to a single element, and a single class can be applied to multiple elements, making class selectors highly versatile for styling.

Example:

```
 /* Selects all elements with class="highlight" */
.highlight {
  background-color: yellow;
  font-weight: bold;
}

/* Selects all elements with class="btn" */
.btn {
  display: inline-block;
  padding: 8px 16px;
```

```
  background-color: #4285f4;
  color: white;
  border-radius: 4px;
  text-decoration: none;
}

/* Selects all elements with class="error" */
.error {
  color: red;
  border: 1px solid red;
  padding: 10px;
  background-color: #ffeeee;
}
```

HTML Usage:

```
 <!-- Element with a single class -->
<p class="highlight">This paragraph is highlighted.</p>

<!-- Element with multiple classes -->
<div class="card shadow rounded">
  This div has three classes: card, shadow, and rounded.
</div>

<!-- Different elements sharing the same class -->
<button class="btn">Click Me</button>
<a href="#" class="btn">Link Button</a>
```

Effect: The class selector applies the specified styles to all elements that have the matching class attribute, regardless of their element type. This allows for consistent styling across different elements and enables the reuse of style rules throughout a document.

Best Practices: - Use meaningful class names that describe the purpose or content rather than appearance. - Follow a consistent naming convention (e.g., BEM, SMACSS) for larger projects. - Use lowercase letters and hyphens for class names (e.g., "text-center" instead of "textCenter"). - Apply multiple classes to an element to layer on styles as needed. - Prefer classes over IDs for styling when the style will be applied to multiple elements. - Keep class names concise but descriptive.

Related Items: - Type selectors - ID selectors - Attribute selectors - Pseudo-classes

ID Selector

Syntax:

```
#idname { property: value; }
```

Description: The ID selector selects a single element with the specified ID attribute. It targets an element based on the value of its `id` attribute, which should be unique within the document. ID selectors are preceded by a hash (#) character. Because IDs must be unique within a document, ID selectors are used to style one specific element rather than a group of elements.

Example:

```
/* Selects the element with id="header" */
#header {
  background-color: #333;
  color: white;
  padding: 20px;
  position: sticky;
  top: 0;
}

/* Selects the element with id="main-content" */
#main-content {
  max-width: 1200px;
  margin: 0 auto;
  padding: 20px;
}

/* Selects the element with id="footer" */
#footer {
  background-color: #f8f8f8;
  padding: 20px;
  text-align: center;
  border-top: 1px solid #ddd;
}
```

HTML Usage:

```
<header id="header">
  <h1>Website Title</h1>
  <nav>Navigation links</nav>
</header>

<main id="main-content">
  <p>This is the main content area of the page.</p>
</main>

<footer id="footer">
```

```
  <p>&copy; 2025 My Website</p>
</footer>
```

Effect: The ID selector applies the specified styles to the single element with the matching ID attribute. Because IDs are unique within a document, ID selectors provide a way to target and style specific, one-of-a-kind elements.

Best Practices: - Ensure each ID is unique within the document. - Use meaningful ID names that describe the purpose or content. - Use lowercase letters and hyphens for ID names (e.g., "main-content" instead of "mainContent"). - Prefer classes over IDs for styling when the style will be applied to multiple elements. - Use IDs primarily for unique page sections or one-of-a-kind components. - Be aware that ID selectors have high specificity, which can make them difficult to override. - Consider using IDs more for JavaScript hooks and anchor links rather than for styling.

Related Items: - Class selectors - Type selectors - Attribute selectors - CSS specificity

Universal Selector

Syntax:

```
 * { property: value; }
```

Description: The universal selector selects all elements in the document. It is represented by an asterisk (*) and matches any element of any type. The universal selector can be used on its own to apply styles to every element, or it can be combined with other selectors to target specific elements.

Example:

```
 /* Applies to all elements */
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

/* Applies to all elements inside a div */
div * {
  color: blue;
}

/* Applies to all elements with a class attribute */
*[class] {
```

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

Effect: The universal selector applies the specified styles to all elements in the document or within a specific context when combined with other selectors. It's often used for CSS resets or to set global properties like `box-sizing`.

Best Practices: - Use the universal selector sparingly, as it can impact performance when applied to large documents. - It's particularly useful for CSS resets and setting global box-sizing. - Be cautious when setting properties like `margin` or `padding` with the universal selector, as it can have unintended consequences. - Consider using a more targeted approach for most styling needs. - When combined with child or descendant combinators, it can be useful for targeting all elements within a specific container.

Related Items: - Type selectors - Class selectors - ID selectors - CSS resets

Attribute Selector

Syntax:

```
 [attribute] { property: value; }
[attribute="value"] { property: value; }
[attribute~="value"] { property: value; }
[attribute|="value"] { property: value; }
[attribute^="value"] { property: value; }
[attribute$="value"] { property: value; }
[attribute*="value"] { property: value; }
```

Description: Attribute selectors select elements based on the presence or value of their attributes. They provide a powerful way to target elements without adding classes or IDs. There are several types of attribute selectors, each with different matching criteria:

- `[attribute]`: Selects elements with the specified attribute, regardless of its value.
- `[attribute="value"]`: Selects elements where the attribute exactly matches the specified value.
- `[attribute~="value"]`: Selects elements where the attribute value contains the specified word (space-separated).
- `[attribute|="value"]`: Selects elements where the attribute value is exactly "value" or begins with "value" followed by a hyphen.
- `[attribute^="value"]`: Selects elements where the attribute value begins with the specified value.
- `[attribute$="value"]`: Selects elements where the attribute value ends with the specified value.

- `[attribute*="value"]` : Selects elements where the attribute value contains the specified value anywhere.

Example:

```css
/* Selects all elements with a title attribute */
[title] {
  cursor: help;
}

/* Selects elements with href exactly matching "https://example.com" */
[href="https://example.com"] {
  color: purple;
}

/* Selects elements with class containing the word "button" */
[class~="button"] {
  padding: 5px 10px;
}

/* Selects elements with lang attribute starting with "en" */
[lang|="en"] {
  font-family: 'Arial', sans-serif;
}

/* Selects all links that start with "https" */
[href^="https"] {
  color: green;
}

/* Selects all links to PDF files */
[href$=".pdf"] {
  background-image: url('pdf-icon.png');
  background-repeat: no-repeat;
  padding-left: 20px;
}

/* Selects all elements with "user" anywhere in the data-id attribute */
[data-id*="user"] {
  border: 1px solid blue;
}
```

Effect: Attribute selectors apply the specified styles to elements that match the attribute criteria. They allow for precise targeting of elements based on their attributes without requiring additional class or ID attributes.

Best Practices: - Use attribute selectors when you want to target elements based on their existing attributes rather than adding classes. - They're particularly useful for

styling form elements, links to specific file types, or elements with data attributes. - Be aware that attribute selectors have the same specificity as class selectors. - Consider performance implications when using complex attribute selectors on large documents. - For case-insensitive matching, add `i` before the closing bracket (e.g., `[href$=".pdf" i]`).

Related Items: - Class selectors - Type selectors - Pseudo-classes - Data attributes

Combinators

Descendant Combinator

Syntax:

```
ancestor descendant { property: value; }
```

Description: The descendant combinator selects elements that are descendants of a specified element. It is represented by a space between two selectors. The descendant combinator matches all elements that are descendants of a specified element, not just direct children.

Example:

```
/* Selects all p elements that are descendants of div elements */
div p {
  color: blue;
  line-height: 1.6;
}

/* Selects all li elements that are descendants of ul elements with class "menu" */
ul.menu li {
  list-style-type: none;
  padding: 5px 10px;
}

/* Selects all span elements that are descendants of article elements */
article span {
  font-style: italic;
}
```

HTML Context:

```
<div>
  <p>This paragraph is blue because it's a descendant of a div.</p>
  <section>
```

```
    <p>This paragraph is also blue because it's still a descendant of the div.</p>
  </section>
</div>
<p>This paragraph is not blue because it's not inside a div.</p>
```

Effect: The descendant combinator applies the specified styles to all elements that match the second selector and are contained within elements that match the first selector, at any level of nesting.

Best Practices: - Use descendant combinators when you want to style elements based on their context within the document. - Be aware that descendant combinators can lead to unintended styling if the document structure changes. - For large documents, descendant combinators can be less performant than more specific selectors. - Consider using child combinators (`>`) when you only want to target direct children. - Avoid overly complex descendant selector chains, as they can be difficult to maintain.

Related Items: - Child combinator - Adjacent sibling combinator - General sibling combinator - Specificity

Child Combinator

Syntax:

```
parent > child { property: value; }
```

Description: The child combinator selects elements that are direct children of a specified element. It is represented by a greater-than sign (`>`) between two selectors. Unlike the descendant combinator, the child combinator only matches elements that are immediate children of the parent element, not all descendants.

Example:

```
/* Selects all li elements that are direct children of ul elements */
ul > li {
  border-bottom: 1px solid #ddd;
}

/* Selects all p elements that are direct children of div elements with class "container" */
div.container > p {
  margin-left: 20px;
}

/* Selects all span elements that are direct children of h1 elements */
h1 > span {
  color: #666;
```

```
  font-size: 0.8em;
}
```

HTML Context:

```
<ul>
 <li>This item has a bottom border because it's a direct child of ul.</li>
 <li>This item also has a bottom border.
  <ul>
   <li>This nested item does NOT have a bottom border because it's not a direct
child of the first ul.</li>
  </ul>
 </li>
</ul>
```

Effect: The child combinator applies the specified styles only to elements that match the second selector and are direct children of elements that match the first selector.

Best Practices: - Use child combinators when you want to target only direct children, not all descendants. - Child combinators are useful for creating clear parent-child relationships in your CSS. - They help prevent styles from cascading too deeply into nested structures. - Child combinators are more specific than descendant combinators, which can help avoid unintended styling. - Consider the document structure carefully when using child combinators, as they are sensitive to changes in the HTML hierarchy.

Related Items: - Descendant combinator - Adjacent sibling combinator - General sibling combinator - Specificity

Adjacent Sibling Combinator

Syntax:

```
former + target { property: value; }
```

Description: The adjacent sibling combinator selects an element that immediately follows another specific element and shares the same parent. It is represented by a plus sign (`+`) between two selectors. The adjacent sibling combinator matches the second element only if it immediately follows the first element in the document tree.

Example:

```
/* Selects all h2 elements that immediately follow h1 elements */
h1 + h2 {
  margin-top: 0;
```

```
}

/* Selects all paragraphs that immediately follow images */
img + p {
  font-style: italic;
}

/* Selects all list items that immediately follow another list item */
li + li {
  border-top: 1px solid #ddd;
}
```

HTML Context:

```
 <h1>Main Heading</h1>
<h2>This subheading has no top margin because it immediately follows an h1.</
h2>
<p>Some paragraph text.</p>
<h2>This subheading has normal top margin because it doesn't immediately follow
an h1.</h2>

<ul>
  <li>First item (no border)</li>
  <li>Second item (has top border because it immediately follows another li)</li>
  <li>Third item (has top border because it immediately follows another li)</li>
</ul>
```

Effect: The adjacent sibling combinator applies the specified styles only to elements that match the second selector and immediately follow elements that match the first selector, when both share the same parent.

Best Practices: - Use adjacent sibling combinators when you want to style elements based on what immediately precedes them. - They're particularly useful for adding spacing or borders between consecutive elements. - Adjacent sibling combinators can help reduce the need for additional classes or markup. - Be aware that these selectors are sensitive to the exact order of elements in the HTML. - Consider using general sibling combinators (`~`) when you need to target all following siblings, not just the immediate one.

Related Items: - General sibling combinator - Child combinator - Descendant combinator - Specificity

General Sibling Combinator

Syntax:

```
former ~ target { property: value; }
```

Description: The general sibling combinator selects elements that follow another specific element and share the same parent. It is represented by a tilde (`~`) between two selectors. Unlike the adjacent sibling combinator, the general sibling combinator matches all elements that follow the first element, not just the immediate one.

Example:

```
/* Selects all p elements that follow an h2 and share the same parent */
h2 ~ p {
  margin-left: 20px;
}

/* Selects all list items that follow a list item with class "active" */
li.active ~ li {
  color: #999;
}

/* Selects all div elements that follow a section element */
section ~ div {
  border-top: 1px dashed #ccc;
}
```

HTML Context:

```
<div>
  <h2>Heading</h2>
  <p>This paragraph has a left margin because it follows an h2.</p>
  <div>This div is not affected because it's not a paragraph.</div>
  <p>This paragraph also has a left margin because it follows an h2 (even though there's a div in between).</p>
</div>

<ul>
  <li>Normal item</li>
  <li class="active">Active item</li>
  <li>This item is gray because it follows the active item.</li>
  <li>This item is also gray for the same reason.</li>
</ul>
```

Effect: The general sibling combinator applies the specified styles to all elements that match the second selector and follow elements that match the first selector, when both share the same parent.

Best Practices: - Use general sibling combinators when you want to style all following siblings, not just the immediate one. - They're useful for creating progressive disclosure interfaces or styling groups of elements after a trigger element. - Be aware that these selectors only target elements that come after the reference element in the document. - Consider using adjacent sibling combinators (`+`) when you only need to target the immediate next sibling. - General sibling combinators can help reduce the need for additional classes or JavaScript for state-based styling.

Related Items: - Adjacent sibling combinator - Child combinator - Descendant combinator - Specificity

Pseudo-Classes

:hover Pseudo-Class

Syntax:

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

Description: The `:hover` pseudo-class selects elements when the user hovers over them with a pointing device such as a mouse. It is commonly used to create interactive effects like changing colors, showing additional information, or indicating that an element is clickable.

Example:

```
/* Change link color on hover */
a:hover {
  color: #ff6600;
  text-decoration: underline;
}

/* Create a button hover effect */
.button:hover {
  background-color: #0056b3;
  transform: translateY(-2px);
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
}

/* Show additional content on hover */
.tooltip:hover .tooltip-text {
  visibility: visible;
  opacity: 1;
}
```

Effect: The `:hover` pseudo-class applies the specified styles only when the user hovers their cursor over the matching elements, creating interactive effects that respond to user input.

Best Practices: - Use `:hover` to provide visual feedback for interactive elements like links and buttons. - Ensure hover effects are subtle and enhance usability rather than distract. - Remember that hover effects are not available on touch devices, so don't rely on them for critical functionality. - Consider using transitions to create smooth hover effects. - For complex hover interactions, combine with other pseudo-classes like `:focus` to ensure keyboard accessibility. - Test hover effects to ensure they don't cause layout shifts or other disruptive behaviors.

Related Items: - :active pseudo-class - :focus pseudo-class - :visited pseudo-class - CSS transitions

:active Pseudo-Class

Syntax:

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

Description: The `:active` pseudo-class selects elements when they are being activated by the user. For example, when a button is being clicked or when a link is being clicked. It represents the "pressed" state of an interactive element and is commonly used to provide immediate visual feedback during interaction.

Example:

```
/* Change button appearance when clicked */
button:active {
  background-color: #003366;
  transform: translateY(1px);
  box-shadow: 0 1px 2px rgba(0, 0, 0, 0.2);
}

/* Style links when clicked */
a:active {
  color: #cc0000;
}

/* Create a "pressed" effect for any clickable element */
.clickable:active {
  opacity: 0.8;
```

```
  transform: scale(0.98);
}
```

Effect: The `:active` pseudo-class applies the specified styles only during the brief moment when the user is actively clicking or pressing on the element, creating a responsive feel to user interactions.

Best Practices: - Use `:active` to provide immediate visual feedback when users interact with clickable elements. - Keep active state changes subtle but noticeable to enhance the feeling of interactivity. - Consider the order of link pseudo-classes (`:link`, `:visited`, `:hover`, `:active`) to ensure proper cascade. - Combine with `:hover` and `:focus` states for a complete interactive experience. - Test active states on both mouse and touch devices to ensure they work as expected. - Use transitions sparingly with `:active` states, as the active state is typically very brief.

Related Items: - :hover pseudo-class - :focus pseudo-class - :visited pseudo-class - Interactive elements (buttons, links)

:focus Pseudo-Class

Syntax:

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

Description: The `:focus` pseudo-class selects elements that have received focus, either through keyboard navigation (pressing Tab) or by being clicked with a mouse. It is essential for accessibility, as it provides visual feedback to users navigating with a keyboard about which element is currently active.

Example:

```
/* Style form inputs when they receive focus */
input:focus, textarea:focus, select:focus {
  border-color: #4d90fe;
  outline: none;
  box-shadow: 0 0 0 2px rgba(77, 144, 254, 0.3);
}

/* Style links when they receive focus */
a:focus {
  outline: 2px solid #0066cc;
  outline-offset: 2px;
}
```

```
/* Create a custom focus style for buttons */
button:focus {
  background-color: #f0f0f0;
  color: #333;
  border: 2px solid #333;
}
```

Effect: The `:focus` pseudo-class applies the specified styles when an element receives focus through keyboard navigation or mouse clicks, helping users identify which element is currently active.

Best Practices: - Never completely remove focus styles (e.g., `outline: none;`) without providing alternative visual focus indicators. - Ensure focus styles are clearly visible with good contrast for accessibility. - Consider using `outline-offset` to create space between the element and its outline. - Use consistent focus styles throughout your interface for a cohesive user experience. - Test focus styles with keyboard navigation to ensure they're effective. - Consider using `:focus-visible` (with appropriate fallbacks) to show focus styles only when navigating with a keyboard.

Related Items: - :focus-visible pseudo-class - :focus-within pseudo-class - :hover pseudo-class - :active pseudo-class - Accessibility

:nth-child() Pseudo-Class

Syntax:

```
selector:nth-child(pattern) { property: value; }
```

Description: The `:nth-child()` pseudo-class selects elements based on their position among a group of siblings. It accepts a formula as an argument, which can be a number, a keyword (`odd` or `even`), or a formula pattern like `an+b` where: - `a` is a step size - `n` is a counter (0, 1, 2, ...) - `b` is an offset

This powerful selector allows for targeting elements based on their position in a sequence.

Example:

```
/* Select every odd row in a table */
tr:nth-child(odd) {
  background-color: #f2f2f2;
}

/* Select every even list item */
```

```css
li:nth-child(even) {
  background-color: #e6f7ff;
}

/* Select every third element */
div:nth-child(3n) {
  border-right: 2px solid #ccc;
}

/* Select the first 5 elements */
p:nth-child(-n+5) {
  font-weight: bold;
}

/* Select every 4th element starting from the 3rd */
.item:nth-child(4n+3) {
  color: red;
}
```

Effect: The `:nth-child()` pseudo-class applies the specified styles to elements that match the position pattern within their parent container, allowing for alternating styles, highlighting specific positions, or creating grid-like patterns.

Best Practices: - Use `:nth-child(odd)` and `:nth-child(even)` for simple alternating patterns like zebra striping tables. - For more complex patterns, use the formula `an+b` where `a` is the frequency and `b` is the offset. - Remember that `:nth-child()` counts all siblings, not just those of the same type. - For type-specific selection, consider using `:nth-of-type()` instead. - Test your patterns with different numbers of elements to ensure they behave as expected. - Use comments to explain complex patterns for better maintainability.

Related Items: - :nth-of-type() pseudo-class - :first-child pseudo-class - :last-child pseudo-class - :only-child pseudo-class

:not() Pseudo-Class

Syntax:

```css
selector:not(negation-selector) { property: value; }
```

Description: The `:not()` pseudo-class, also known as the negation pseudo-class, selects elements that do not match the selector provided as an argument. It allows you to apply styles to all elements except those that match a specific condition, making it a powerful tool for exception-based styling.

Example:

```
 /* Style all buttons except those with class "primary" */
button:not(.primary) {
  background-color: #f0f0f0;
  color: #333;
}

/* Add margin to all paragraphs except the first one */
p:not(:first-child) {
  margin-top: 1em;
}

/* Style all inputs except checkboxes and radio buttons */
input:not([type="checkbox"]):not([type="radio"]) {
  width: 100%;
  padding: 8px;
  border: 1px solid #ccc;
}

/* Style all list items except those with a special class */
li:not(.special) {
  list-style-type: square;
}
```

Effect: The `:not()` pseudo-class applies the specified styles to elements that do not match the negation selector, allowing for exception-based styling without requiring additional classes or overrides.

Best Practices: - Use `:not()` to reduce the need for overrides and exceptions in your CSS. - Chain multiple `:not()` pseudo-classes for multiple exclusions. - Be aware that `:not()` doesn't increase specificity much (only by the specificity of its argument). - Consider readability when using complex `:not()` selectors; sometimes separate rules might be clearer. - In modern browsers, `:not()` can accept complex selectors and even multiple selectors separated by commas. - Use `:not()` to create more maintainable CSS by focusing on exceptions rather than creating numerous specific rules.

Related Items: - :is() pseudo-class - :where() pseudo-class - Specificity - Combinators

CSS Properties

Color and Background Properties

color

Syntax:

```
selector { color: value; }
```

Description: The `color` property sets the foreground color value of an element's text content and text decorations. It also sets the `currentcolor` value, which can be used as an indirect value on other properties.

Values: - `<color>` : Specifies a color value (e.g., named colors like `red` , `blue` ; hexadecimal values like `#ff0000` , `#00f` ; RGB values like `rgb(255, 0, 0)` ; HSL values like `hsl(0, 100%, 50%)`). - `inherit` : Inherits the color value from the parent element. - `initial` : Sets the property to its default value (browser-dependent, usually black). - `revert` : Reverts the property to the user-agent stylesheet value. - `unset` : Resets the property to its inherited value if it inherits from its parent or to its initial value if not.

Example:

```
/* Set paragraph text color to dark gray */
p {
  color: #333333;
}

/* Set heading color to navy blue */
h1 {
  color: navy;
}

/* Set link color to a specific shade of green */
a {
  color: rgb(0, 128, 0);
}

/* Use currentcolor for border */
.box {
  color: blue;
  border: 2px solid currentcolor; /* Border will be blue */
}
```

Effect: Changes the color of the text content within the selected elements. This is one of the most fundamental properties for controlling the visual appearance of text.

Best Practices: - Ensure sufficient contrast between text color and background color for readability and accessibility (use WCAG guidelines). - Use consistent color palettes throughout your website for a cohesive design. - Consider using CSS custom properties (variables) to manage colors centrally. - Use named colors for simplicity or hexadecimal/RGB/HSL for precise control. - Remember that `color` sets the `currentcolor`, which can be useful for related properties like borders or SVG fills.

Related Items: - background-color property - currentcolor keyword - CSS Color Values - Accessibility and Color Contrast

background-color

Syntax:

```
selector { background-color: value; }
```

Description: The `background-color` property sets the background color of an element. The background extends underneath the content and padding box of the element.

Values: - `<color>`: Specifies a color value (e.g., `red`, `#ff0000`, `rgba(255, 0, 0, 0.5)`). - `transparent`: Specifies a transparent background (default value). - `inherit`: Inherits the background-color value from the parent element. - `initial`: Sets the property to its default value (`transparent`). - `revert`: Reverts the property to the user-agent stylesheet value. - `unset`: Resets the property to its inherited value if it inherits from its parent or to its initial value if not.

Example:

```
/* Set the background color of the body */
body {
  background-color: #f0f0f0; /* Light gray */
}

/* Set the background color for a specific div */
.highlight-box {
  background-color: yellow;
  padding: 10px;
}

/* Set a semi-transparent background color */
.overlay {
  background-color: rgba(0, 0, 0, 0.7); /* Dark semi-transparent */
```

```
  color: white;
}

/* Set header background */
header {
  background-color: #4a90e2; /* Blue */
}
```

Effect: Changes the background color of the selected elements, filling the area behind their content and padding.

Best Practices: - Ensure sufficient contrast between background color and foreground text color for readability. - Use `background-color` in conjunction with `color` to define the overall look. - Consider using `rgba()` or `hsla()` to create semi-transparent backgrounds. - Use `transparent` when you want the background of the parent element to show through. - Combine with other background properties (like `background-image`) for more complex effects.

Related Items: - color property - background property (shorthand) - background-image property - background-clip property - background-origin property

Font and Text Properties

font-family

Syntax:

```
selector { font-family: family-name | generic-family [, family-name | generic-family]*; }
```

Description: The `font-family` property specifies a prioritized list of one or more font family names and/or generic family names for the selected element. The browser will use the first font in the list that is installed on the user's system or can be downloaded using `@font-face`.

Values: - `family-name`: The name of a specific font family (e.g., "Times New Roman", "Arial", "Helvetica"). Font names containing spaces should be quoted. - `generic-family`: A generic font family name (e.g., `serif`, `sans-serif`, `monospace`, `cursive`, `fantasy`). These provide fallback options if specific fonts are unavailable. - `inherit`, `initial`, `revert`, `unset`.

Example:

```css
 /* Use a common sans-serif stack */
body {
  font-family: Arial, Helvetica, sans-serif;
}

/* Use a serif font for headings */
h1, h2, h3 {
  font-family: "Georgia", Times, serif;
}

/* Use a monospace font for code blocks */
pre, code {
  font-family: "Courier New", Courier, monospace;
}

/* Using a custom web font */
@font-face {
  font-family: "MyCustomFont";
  src: url("mycustomfont.woff2") format("woff2");
}
.custom-text {
  font-family: "MyCustomFont", sans-serif;
}
```

Effect: Sets the typeface used for rendering text within the selected elements. The browser attempts to use fonts from the list in order, falling back to the next option if a font is not available.

Best Practices: - Always provide fallback generic font families (like `serif` or `sans-serif`) at the end of the list. - Use quotes around font names that contain spaces or special characters. - Prioritize web-safe fonts or use `@font-face` to embed custom fonts for consistent appearance. - Consider performance implications when using multiple custom web fonts. - Choose fonts that are readable and appropriate for the content's tone.

Related Items: - font-size property - font-weight property - font-style property - line-height property - @font-face rule

font-size

Syntax:

```css
selector { font-size: value; }
```

Description: The `font-size` property specifies the size of the font. Setting the font size is crucial for readability and visual hierarchy.

Values: - `<length>`: An absolute length value (e.g., `16px`, `12pt`). - `<percentage>`: A percentage value relative to the parent element's font size (e.g., `120%`). - Relative length units: `em` (relative to parent's font size), `rem` (relative to root element's font size), `vw` (relative to viewport width), `vh` (relative to viewport height). - Keywords: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large` (relative to user's default font size). - Math functions: `calc()`, `min()`, `max()`, `clamp()`. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
 /* Set a base font size on the body */
body {
  font-size: 16px;
}

/* Use relative units for headings */
h1 {
  font-size: 2.5rem; /* 2.5 times the root font size */
}
h2 {
  font-size: 1.8em; /* 1.8 times the parent font size */
}

/* Set a specific pixel size for captions */
figcaption {
  font-size: 14px;
}

/* Use viewport width for responsive headings */
.responsive-title {
  font-size: clamp(1.5rem, 5vw, 3rem); /* Size between 1.5rem and 3rem based on
viewport width */
}
```

Effect: Controls the size of the text within the selected elements. Different units offer different levels of control and responsiveness.

Best Practices: - Set a base font size on the `html` or `body` element (often `16px` or `100%`). - Use relative units like `rem` or `em` for most elements to allow users to scale text easily and maintain consistency. - Use `px` for elements that should not scale with user preferences or parent sizes, but use sparingly. - Use `vw` or `vh` cautiously for responsive typography, often combined with `clamp()` for control. - Ensure sufficient font size for readability, especially for body text (typically 16px or larger). - Test font sizes across different devices and screen resolutions.

Related Items: - font-family property - line-height property - font-weight property - CSS Length Units

font-weight

Syntax:

```
selector { font-weight: value; }
```

Description: The `font-weight` property specifies the weight (or boldness) of the font. The available weights depend on the font family being used.

Values: - `normal`: Normal font weight. Equivalent to `400`. - `bold`: Bold font weight. Equivalent to `700`. - `lighter`: One step lighter than the parent element's font weight. - `bolder`: One step bolder than the parent element's font weight. - Numeric values: `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, `900`. Not all fonts support all numeric weights. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
/* Make headings bold */
h1, h2, h3 {
  font-weight: bold; /* or 700 */
}

/* Use a lighter weight for a subtitle */
.subtitle {
  font-weight: 300;
}

/* Emphasize text with a bolder weight */
strong, b {
  font-weight: bolder; /* Makes it bolder relative to parent */
}

/* Set normal weight for paragraphs */
p {
  font-weight: normal; /* or 400 */
}
```

Effect: Changes the thickness or boldness of the text characters. The visual effect depends on the specific font family and the weights it provides.

Best Practices: - Use `bold` (or `700`) for standard bold text and `normal` (or `400`) for regular text. - Use numeric values when the font family supports specific weights (e.g.,

light, semi-bold) and you need fine control. - Ensure the font family you are using supports the specified weights; otherwise, the browser will approximate. - Use `font-weight` semantically (e.g., for headings, `<strong>` elements) rather than purely for visual styling. - Avoid using too many different font weights on a single page, as it can look cluttered.

Related Items: - font-family property - font-style property - strong element - b element

text-align

Syntax:

```
selector { text-align: value; }
```

Description: The `text-align` property describes how inline content like text is aligned within its parent block element. It affects text, inline elements, and inline-block elements.

Values: - `left`: Aligns content to the left edge (default for left-to-right languages). - `right`: Aligns content to the right edge (default for right-to-left languages). - `center`: Centers the content horizontally. - `justify`: Stretches the lines so that each line has equal width (except the last line), creating straight left and right edges. - `start`: Aligns content to the start edge (left in LTR, right in RTL). - `end`: Aligns content to the end edge (right in LTR, left in RTL). - `match-parent`: Similar to `inherit`, but `start` and `end` values are calculated based on the parent's direction. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
/* Center headings */
h1, h2 {
  text-align: center;
}

/* Justify paragraph text */
p.justified {
  text-align: justify;
}

/* Align table cell content to the right */
td.numeric {
  text-align: right;
}

/* Align text based on language direction */
.content {
```

```
  text-align: start; /* Left for English, Right for Arabic */
}
```

Effect: Controls the horizontal alignment of text and other inline content within block-level containers.

Best Practices: - Use `left` (or `start`) for body text in most Western languages for optimal readability. - Use `center` sparingly, typically for headings or short lines of text. - Use `justify` with caution, as it can create uneven spacing between words (rivers) which can hinder readability. Consider using hyphenation with justified text. - Use `right` (or `end`) for languages written right-to-left or for aligning specific content like numbers in tables. - Use `start` and `end` for better internationalization support, as they adapt to the text direction.

Related Items: - direction property - writing-mode property - text-justify property - text-indent property

Box Model Properties

width

Syntax:

```
selector { width: value; }
```

Description: The `width` property specifies the width of an element's content area. By default, block-level elements take up the full available width, while inline elements take up only the width of their content.

Values: - `<length>`: An absolute length (e.g., `300px`, `10cm`). - `<percentage>`: A percentage relative to the containing block's width (e.g., `50%`). - `auto`: The browser calculates the width (default). - `max-content`: The intrinsic preferred width. - `min-content`: The intrinsic minimum width. - `fit-content(<length-percentage>)`: Uses the available space, but not more than the specified argument. - Math functions: `calc()`, `min()`, `max()`, `clamp()`. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
/* Set a fixed width for a container */
.container {
  width: 960px;
  margin: 0 auto; /* Center the container */
```

```
}

/* Set a percentage width for a responsive column */
.column {
  width: 75%;
  float: left;
}

/* Set width based on content */
.button {
  width: max-content;
  padding: 10px 20px;
}

/* Use calc() for complex width calculation */
.sidebar {
  width: calc(100% - 200px);
}
```

Effect: Controls the width of the element's content box. The total width occupied by the element also includes padding, border, and margin (unless `box-sizing: border-box` is used).

Best Practices: - Use `auto` (the default) or percentage-based widths for responsive design. - Use fixed pixel widths (`px`) sparingly, typically for elements that should not resize. - Use `max-width` in conjunction with `width: auto` or `width: 100%` to create flexible layouts that don't exceed a certain size. - Use `max-content` or `min-content` when the width should depend strictly on the content. - Remember that `width` applies to the content box by default; use `box-sizing: border-box` to make it apply to the border box instead.

Related Items: - height property - max-width property - min-width property - box-sizing property - padding, border, margin properties

height

Syntax:

```
selector { height: value; }
```

Description: The `height` property specifies the height of an element's content area. By default, the height of an element is determined by its content.

Values: - `<length>` : An absolute length (e.g., `200px` , `5cm`). - `<percentage>` : A percentage relative to the containing block's height. Note: This often requires the

containing block to have an explicitly set height. - `auto` : The browser calculates the height based on content (default). - `max-content` : The intrinsic preferred height. - `min-content` : The intrinsic minimum height. - `fit-content(<length-percentage>)` : Uses the available space, but not more than the specified argument. - Math functions: `calc()` , `min()` , `max()` , `clamp()` . - `inherit` , `initial` , `revert` , `unset` .

Example:

```
 /* Set a fixed height for an image container */
.image-container {
  height: 400px;
  overflow: hidden;
}

/* Set a percentage height (requires parent height) */
html, body {
  height: 100%;
}
.full-height-section {
  height: 100%;
}

/* Set height based on content */
.card {
  height: auto; /* Default behavior */
}

/* Use viewport height */
.hero-banner {
  height: 80vh; /* 80% of the viewport height */
}
```

Effect: Controls the height of the element's content box. The total height occupied by the element also includes padding, border, and margin (unless `box-sizing: border-box` is used).

Best Practices: - Allow height to be determined by content (`height: auto`) whenever possible for flexibility. - Use fixed heights (`px`) when necessary for specific layout constraints (e.g., fixed-size components, aspect ratios). - Be cautious with percentage heights, as they depend on the parent element having an explicit height. - Use `min-height` to ensure an element is at least a certain height while still allowing it to grow with content. - Use viewport units (`vh`) for sections that should relate to the screen height. - Remember that `height` applies to the content box by default; use `box-sizing: border-box` to make it apply to the border box instead.

Related Items: - width property - max-height property - min-height property - box-sizing property - padding, border, margin properties

margin

Syntax:

```
 selector { margin: value; } /* All four sides */
selector { margin: top/bottom left/right; } /* Vertical, Horizontal */
selector { margin: top right/left bottom; } /* Top, Horizontal, Bottom */
selector { margin: top right bottom left; } /* Top, Right, Bottom, Left */

/* Individual properties */
selector { margin-top: value; }
selector { margin-right: value; }
selector { margin-bottom: value; }
selector { margin-left: value; }
```

Description: The `margin` property sets the margin area on all four sides of an element. It is shorthand for `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. Margins create space around elements, outside of any defined borders. Vertical margins between adjacent block elements often collapse.

Values: - `<length>`: An absolute length (e.g., `10px`, `1em`). Negative values are allowed. - `<percentage>`: A percentage relative to the containing block's width. - `auto`: The browser calculates the margin. Often used to center block elements horizontally (`margin: 0 auto`). - `inherit`, `initial`, `revert`, `unset`.

Example:

```
 /* Set 10px margin on all sides */
.box {
  margin: 10px;
}

/* Set 5px top/bottom margin, 15px left/right margin */
.button {
  margin: 5px 15px;
}

/* Set specific margins */
.content {
  margin-top: 20px;
  margin-bottom: 30px;
  margin-left: 10px;
  margin-right: 10px;
```

```
}

/* Center a block element horizontally */
.container {
  width: 80%;
  margin: 0 auto;
}

/* Use negative margin to overlap elements */
.overlap {
  margin-top: -20px;
}
```

Effect: Creates empty space around the element, outside its border. Margins push other elements away and are transparent.

Best Practices: - Use margins to create space between elements. - Understand margin collapsing: adjacent vertical margins of block elements often combine into a single margin. - Use `margin: 0 auto` on elements with a defined width to center them horizontally within their container. - Use consistent margin values for a balanced layout. - Avoid using excessive margins; consider padding or layout techniques like Flexbox/Grid gaps instead. - Use negative margins cautiously, as they can complicate layouts.

Related Items: - padding property - border property - Margin collapsing - box-sizing property

padding

Syntax:

```
 selector { padding: value; } /* All four sides */
selector { padding: top/bottom left/right; } /* Vertical, Horizontal */
selector { padding: top right/left bottom; } /* Top, Horizontal, Bottom */
selector { padding: top right bottom left; } /* Top, Right, Bottom, Left */

/* Individual properties */
selector { padding-top: value; }
selector { padding-right: value; }
selector { padding-bottom: value; }
selector { padding-left: value; }
```

Description: The `padding` property sets the padding area on all four sides of an element. It is shorthand for `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`. Padding creates space within an element, between its content and its border.

Values: - `<length>` : An absolute length (e.g., `10px` , `0.5em`). Negative values are not allowed. - `<percentage>` : A percentage relative to the containing block's width. - `inherit` , `initial` , `revert` , `unset` .

Example:

```
 /* Set 15px padding on all sides */
.card {
  padding: 15px;
  border: 1px solid #ccc;
}

/* Set 10px top/bottom padding, 20px left/right padding */
.button {
  padding: 10px 20px;
}

/* Set specific padding values */
.article {
  padding-top: 10px;
  padding-bottom: 20px;
  padding-left: 5px;
  padding-right: 5px;
}

/* Use percentage padding */
.responsive-box {
  padding: 5%; /* 5% of the container width on all sides */
}
```

Effect: Creates empty space inside the element, between the content and the border. The background of the element extends into the padding area.

Best Practices: - Use padding to create space between an element's content and its border. - Use consistent padding values for visual harmony. - Remember that padding increases the total size of the element (unless `box-sizing: border-box` is used). - Use padding to increase the clickable area of small elements like icons or buttons. - Avoid using padding solely to create space between elements; use margins for that purpose.

Related Items: - margin property - border property - box-sizing property - width property - height property

border

Syntax:

```css
/* Shorthand for width, style, and color */
selector { border: width style color; }

/* Individual properties */
selector { border-width: value; }
selector { border-style: value; }
selector { border-color: value; }

/* Per-side shorthands */
selector { border-top: width style color; }
selector { border-right: width style color; }
selector { border-bottom: width style color; }
selector { border-left: width style color; }

/* Per-side individual properties */
selector { border-top-width: value; }
selector { border-top-style: value; }
selector { border-top-color: value; }
/* ...and so on for right, bottom, left */
```

Description: The `border` property is a shorthand for setting the width, style, and color of an element's border on all four sides. Borders are drawn between the padding and margin of an element.

Values: - `border-width`: `<length>` (e.g., `1px`, `thick`), `medium` (default), `thin`. - `border-style`: `none` (default), `hidden`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`. - `border-color`: `<color>` (e.g., `black`, `#ccc`, `rgba(0,0,0,0.5)`).

Example:

```css
/* Solid black 1px border on all sides */
.box {
  border: 1px solid black;
  padding: 10px;
}

/* Dashed gray border on the bottom only */
hr {
  border: none;
  border-bottom: 1px dashed #ccc;
}

/* Different borders on different sides */
.panel {
  border-top: 3px solid blue;
  border-bottom: 1px solid #ddd;
  border-left: 1px solid #ddd;
  border-right: 1px solid #ddd;
```

```
  padding: 15px;
}

/* Rounded borders */
.rounded-button {
  border: 2px solid green;
  border-radius: 5px; /* Use border-radius for rounded corners */
  padding: 8px 16px;
}
```

Effect: Draws a line around the element, between its padding and margin. The style, width, and color of the border can be controlled.

Best Practices: - Use the `border` shorthand property for simplicity when setting all three values. - Set `border-style` to something other than `none` for the border to be visible. - Use `border: none;` or `border: 0;` to remove borders. - Use `border-radius` to create rounded corners. - Remember that borders add to the total size of the element (unless `box-sizing: border-box` is used). - Use borders for visual separation, emphasis, or decoration.

Related Items: - border-radius property - box-shadow property - outline property - padding property - margin property - box-sizing property

Layout Properties

display

Syntax:

```
selector { display: value; }
```

Description: The `display` property specifies the display behavior (the type of rendering box) of an element. It determines how an element is rendered in the document flow, whether it behaves as a block or inline element, and how its children are laid out (e.g., using Flexbox or Grid).

Values: - **Outer Display Types:** - `block` : Element generates a block box, starting on a new line and taking up available width. - `inline` : Element generates an inline box, flowing with text, width/height have no effect. - `inline-block` : Element generates a block box flowed with surrounding content as if it were a single inline box. - **Inner Display Types (for children layout):** - `flow` (default): Children laid out using normal flow (block and inline). - `flex` : Children laid out using the flexible box model. - `grid` : Children laid

out using the grid model. - `table`, `table-row`, `table-cell`, etc.: Element behaves like corresponding HTML table elements. - **Other Values:** - `none`: Element is not displayed and removed from the layout entirely. - `contents`: Element's children act as direct children of the element's parent. - `list-item`: Element behaves like a `<li>` element. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
 /* Make list items appear inline */
nav ul li {
  display: inline-block;
  margin-right: 10px;
}

/* Hide an element */
.hidden {
  display: none;
}

/* Use Flexbox for layout */
.container {
  display: flex;
  gap: 10px;
}

/* Use Grid for layout */
.grid-container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 15px;
}

/* Make a span behave like a block */
span.block-span {
  display: block;
  margin-bottom: 10px;
}
```

Effect: Fundamentally changes how an element is rendered and interacts with other elements in the layout. It controls the element's box type and its participation in the document flow.

Best Practices: - Understand the difference between `block`, `inline`, and `inline-block` for basic layout control. - Use `display: flex` or `display: grid` for modern, powerful layout techniques. - Use `display: none` to completely remove an element from the page (contrast with `visibility: hidden`, which hides but preserves space). - Avoid using `display:`

`table` properties for general layout; prefer Flexbox or Grid. - Use `display: contents` carefully, as it can affect accessibility if not used correctly.

Related Items: - position property - float property - visibility property - Flexbox - Grid Layout

position

Syntax:

```
selector { position: value; }
```

Description: The `position` property specifies the type of positioning method used for an element. It works in conjunction with the `top`, `right`, `bottom`, and `left` properties to determine the final location of positioned elements.

Values: - `static`: Default value. Element is positioned according to the normal flow of the document. - `relative`: Element is positioned according to the normal flow, and then offset relative to itself based on `top`, `right`, `bottom`, `left`. The space the element would have occupied in normal flow is preserved. - `absolute`: Element is removed from the normal flow, and positioned relative to its nearest positioned ancestor (or the initial containing block). Other elements behave as if the absolutely positioned element does not exist. - `fixed`: Element is removed from the normal flow, and positioned relative to the viewport. It stays in the same place even when the page is scrolled. - `sticky`: Element is treated as `relative` until its containing block crosses a specified threshold (e.g., scrolling past a certain point), at which point it is treated as `fixed` until it meets the opposite edge of its containing block. - `inherit`, `initial`, `revert`, `unset`.

Example:

```
 /* Default positioning */
.normal-element {
  position: static;
}

/* Offset an element slightly */
.offset-element {
  position: relative;
  top: -5px;
  left: 10px;
}

/* Position an overlay */
.overlay {
```

```
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0,0,0,0.5);
}

/* Create a fixed header */
.fixed-header {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  background-color: white;
  z-index: 1000;
}

/* Create a sticky sidebar section */
.sticky-section {
  position: sticky;
  top: 20px;
}
```

Effect: Determines how an element is positioned within the document or viewport, allowing elements to be taken out of the normal flow, layered, or fixed in place.

Best Practices: - Use `static` (the default) unless you specifically need to change an element's position. - Use `relative` primarily to establish a positioning context for `absolute` children or for minor offsets. - Use `absolute` for elements that need to be positioned precisely relative to a container, often for overlays, tooltips, or custom layouts. - Use `fixed` for elements that should always stay visible relative to the viewport, like headers, footers, or modals. - Use `sticky` for elements that should scroll normally but then stick to an edge (like section headers or sidebars). - Remember that `absolute`, `fixed`, and `sticky` positioning often require setting `top`, `right`, `bottom`, or `left` properties. - Use the `z-index` property to control the stacking order of positioned elements.

Related Items: - top, right, bottom, left properties - z-index property - display property - float property

float

Syntax:

```
selector { float: value; }
```

Description: The `float` property places an element on the left or right side of its container, allowing text and inline elements to wrap around it. The element is removed from the normal flow, though it still remains part of the flow (unlike absolute positioning).

Values: - `left` : Element floats to the left of its container. - `right` : Element floats to the right of its container. - `none` : Default value. Element does not float. - `inline-start` : Element floats to the start side (left in LTR, right in RTL). - `inline-end` : Element floats to the end side (right in LTR, left in RTL). - `inherit` , `initial` , `revert` , `unset` .

Example:

```
/* Float an image to the left, text wraps around */
img.float-left {
  float: left;
  margin-right: 15px;
  margin-bottom: 10px;
}

/* Float a sidebar to the right */
.sidebar {
  float: right;
  width: 200px;
  margin-left: 20px;
}

/* Clear floats to prevent container collapse */
.clearfix::after {
  content: "";
  display: block;
  clear: both;
}
```

Effect: Moves the element to the left or right, allowing other content to flow around it. Floated elements are taken out of the normal block flow but still affect the layout.

Best Practices: - Use `float` primarily for its original purpose: allowing text to wrap around images. - For general page layout (e.g., creating columns), prefer modern techniques like Flexbox or Grid over floats. - Remember to clear floats using techniques like the `clearfix` hack or `overflow: hidden` on the container to prevent layout issues (e.g., container collapse). - Use the `clear` property on subsequent elements to control

whether they flow around the floated element or appear below it. - Test floated layouts carefully, as they can be fragile and difficult to manage.

Related Items: - clear property - display property (Flexbox, Grid) - overflow property - position property

CSS Values and Units

Length Units

Absolute Length Units

Syntax:

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

Description: Absolute length units represent a physical measurement and are fixed in size, regardless of the device or viewing environment. These units are generally used when the physical properties of the output medium are known, such as for print layouts.

Values: - `px` (pixels): One pixel. Traditionally tied to physical display pixels, but now often represents a device-independent pixel. - `pt` (points): 1/72 of an inch. - `pc` (picas): 1 pica = 12 points. - `in` (inches): 1 inch = 96px = 2.54cm. - `cm` (centimeters): 1cm = 96px/2.54. - `mm` (millimeters): 1mm = 1/10th of a centimeter. - `Q` (quarter-millimeters): 1Q = 1/40th of a centimeter.

Example:

```
/* Using pixels for screen display */
.container {
  width: 300px;
  padding: 20px;
  border: 1px solid black;
}

/* Using print units */
@media print {
  body {
    font-size: 12pt;
    margin: 0.5in;
  }
```

```
  .page-break {
    page-break-after: always;
  }
}

/* Using physical measurements */
.card {
  width: 8.5cm;
  height: 5.4cm;
  border: 0.5mm solid #ccc;
}
```

Effect: Absolute length units define sizes that remain consistent regardless of the context. They provide precise control over dimensions but may not adapt well to different screen sizes or user preferences.

Best Practices: - Use `px` for screen designs when you need consistent sizing across devices. - Use `pt`, `pc`, `in`, `cm`, and `mm` primarily for print stylesheets. - Avoid using absolute units for font sizes in responsive web design, as they don't scale with user preferences. - Remember that `px` is not actually tied to physical pixels on high-density displays. - For web layouts, consider using relative units instead of absolute units for better responsiveness.

Related Items: - Relative length units - Viewport units - Media queries

Relative Length Units

Syntax:

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

Description: Relative length units specify a length relative to another length property. They are more flexible than absolute units and are essential for creating responsive designs that adapt to different screen sizes and user preferences.

Values: - `em`: Relative to the font-size of the element (e.g., 2em means 2 times the font size). - `rem`: Relative to the font-size of the root element (html). - `ex`: Relative to the x-height of the current font (rarely used). - `ch`: Relative to the width of the "0" (zero) character. - `lh`: Relative to the line-height of the element. - `rlh`: Relative to the line-height of the root element. - `vw`: 1% of the viewport's width. - `vh`: 1% of the viewport's height. - `vmin`: 1% of the viewport's smaller dimension. - `vmax`: 1% of the viewport's larger dimension. - `%`: Relative to the parent element.

Example:

```css
 /* Set base font size on root element */
html {
  font-size: 16px;
}

/* Using em units (relative to parent's font size) */
.nested-element {
  font-size: 0.875em; /* 14px if parent is 16px */
  margin-bottom: 1.5em; /* 21px if element's font size is 14px */
}

/* Using rem units (relative to root font size) */
h1 {
  font-size: 2rem; /* 32px if root is 16px */
  margin-bottom: 1rem; /* 16px regardless of element's font size */
}

/* Using viewport units */
.hero {
  height: 80vh; /* 80% of viewport height */
  width: 100vw; /* 100% of viewport width */
}

/* Using percentage */
.column {
  width: 50%; /* Half the width of parent */
  padding: 5%; /* 5% of parent width */
}

/* Using ch for readable line lengths */
p {
  max-width: 60ch; /* Approximately 60 characters wide */
}
```

Effect: Relative length units create flexible and adaptable layouts that respond to their context, such as the parent element's size, the viewport dimensions, or the font size.

Best Practices: - Use `rem` for font sizes to ensure they scale with user preferences while maintaining consistent proportions. - Use `em` for spacing related to the element's font size (e.g., padding, margins within components). - Use `%` for creating responsive layouts that adapt to their container. - Use viewport units (`vw`, `vh`, `vmin`, `vmax`) for designs that need to respond directly to the viewport size. - Use `ch` for controlling line length in paragraphs for optimal readability. - Combine relative units with `calc()` for complex responsive calculations. - Remember that `em` compounds in nested elements, while `rem` always refers to the root font size.

Related Items: - Absolute length units - calc() function - Media queries - Responsive design

Color Values

Named Colors

Syntax:

```
selector { color-property: color-name; }
```

Description: CSS provides a set of predefined color names that can be used as color values. These range from basic colors to more specific shades and are supported across all browsers.

Values: - Basic colors: `black`, `white`, `red`, `green`, `blue`, `yellow`, `purple`, `gray`, etc. - Extended colors: `aqua`, `coral`, `crimson`, `fuchsia`, `indigo`, `lime`, `olive`, `teal`, etc. - Shades of gray: `darkgray`, `gray`, `lightgray`, `dimgray`, etc. - Various other named colors (140+ in total).

Example:

```
/* Using basic named colors */
h1 {
  color: navy;
}

p {
  color: black;
  background-color: white;
}

/* Using extended named colors */
.warning {
  color: red;
}

.success {
  color: green;
}

.info {
  color: teal;
}
```

```
/* Creating borders with named colors */
.box {
  border: 2px solid silver;
}
```

Effect: Named colors apply the specified predefined color to the targeted property. They provide a simple way to add color without needing to know specific color codes.

Best Practices: - Use named colors for quick prototyping or when the exact shade isn't critical. - Prefer more specific color formats (HEX, RGB, HSL) for precise brand colors or when consistency is important. - Be aware that some named colors might render slightly differently across browsers. - Use common named colors for better code readability when appropriate. - Consider using CSS custom properties (variables) for consistent color application throughout your site.

Related Items: - Hexadecimal colors - RGB and RGBA colors - HSL and HSLA colors - currentColor keyword - CSS custom properties (variables)

Hexadecimal Colors

Syntax:

```
 selector { color-property: #RRGGBB; } /* 6-digit format */
selector { color-property: #RGB; } /* 3-digit shorthand */
```

Description: Hexadecimal color notation represents colors using a pound/hash symbol (#) followed by either three or six hexadecimal digits. Each pair of digits in the 6-digit format (or each digit in the 3-digit format) represents the intensity of red, green, and blue components of the color, ranging from 00 to FF (0 to 255 in decimal).

Values: - `#RRGGBB` : 6-digit format where RR, GG, and BB are hexadecimal values (00-FF) for red, green, and blue. - `#RGB` : 3-digit shorthand where each digit is duplicated (e.g., #F00 is equivalent to #FF0000).

Example:

```
 /* Using 6-digit hex codes */
body {
  background-color: #FFFFFF; /* White */
  color: #000000; /* Black */
}

/* Using 3-digit hex codes */
.primary-button {
```

```
  background-color: #00F; /* Blue (#0000FF) */
  color: #FFF; /* White (#FFFFFF) */
}

/* Various hex colors */
.palette {
  --brand-red: #FF3366;
  --brand-blue: #3399CC;
  --brand-green: #66CC99;
  --light-gray: #EEEEEE;
  --dark-gray: #333333;
}
```

Effect: Hexadecimal colors apply the specified color to the targeted property, allowing for precise control over the exact shade.

Best Practices: - Use the 3-digit shorthand when possible for cleaner code (when each component pair has repeated digits). - Use lowercase or uppercase consistently for better readability. - Consider using CSS custom properties to define a color palette for your site. - Use meaningful variable names when storing hex colors in custom properties. - For colors with transparency, use RGBA or HSLA instead. - Use tools like color pickers to find the exact hex code for your desired color.

Related Items: - RGB and RGBA colors - HSL and HSLA colors - Named colors - CSS custom properties (variables)

RGB and RGBA Colors

Syntax:

```
 selector { color-property: rgb(r, g, b); } /* RGB format */
selector { color-property: rgba(r, g, b, a); } /* RGBA format */
```

Description: RGB color notation represents colors using the red, green, and blue color model. RGBA extends this by adding an alpha channel for transparency. Each color component (R, G, B) ranges from 0 to 255 (or 0% to 100%), while the alpha component ranges from 0 (completely transparent) to 1 (completely opaque).

Values: - `rgb(r, g, b)` : r, g, and b are integers from 0-255 or percentages from 0%-100%. - `rgba(r, g, b, a)` : r, g, and b as above, with a being a number from 0-1. - Modern CSS also supports the space-separated format: `rgb(r g b)` and `rgb(r g b / a)` .

Example:

```css
/* Using RGB with integer values */
.box {
  background-color: rgb(255, 0, 0); /* Red */
  color: rgb(255, 255, 255); /* White */
}

/* Using RGB with percentage values */
.container {
  background-color: rgb(50%, 75%, 25%);
}

/* Using RGBA for transparency */
.overlay {
  background-color: rgba(0, 0, 0, 0.5); /* Semi-transparent black */
}

.button:hover {
  background-color: rgba(0, 123, 255, 0.8); /* Semi-transparent blue */
}

/* Using modern space-separated syntax */
.modern {
  color: rgb(100 150 200 / 0.6); /* Semi-transparent blue-gray */
}
```

Effect: RGB and RGBA colors apply the specified color to the targeted property, with RGBA allowing for transparency effects that let underlying content show through.

Best Practices: - Use RGBA when you need transparency effects, such as overlays or hover states. - Consider using RGB when you're working with colors programmatically or when the color values come from a design tool. - Use consistent notation (either integers or percentages) throughout your codebase. - For simple transparency effects on existing colors, consider using the `opacity` property instead. - Remember that the alpha channel only affects the element it's applied to, not child elements. - Use the modern space-separated syntax for better readability when supported browsers are targeted.

Related Items: - Hexadecimal colors - HSL and HSLA colors - opacity property - CSS custom properties (variables)

HSL and HSLA Colors

Syntax:

```css
selector { color-property: hsl(h, s, l); } /* HSL format */
selector { color-property: hsla(h, s, l, a); } /* HSLA format */
```

Description: HSL color notation represents colors using the hue, saturation, and lightness color model. HSLA extends this by adding an alpha channel for transparency. HSL is often considered more intuitive for adjusting colors than RGB because it separates the color (hue) from its intensity (saturation) and brightness (lightness).

Values: - `hsl(h, s, l)`: - h (hue): An angle from 0 to 360 degrees representing the color wheel (0/360 is red, 120 is green, 240 is blue). - s (saturation): A percentage from 0% (gray) to 100% (full color). - l (lightness): A percentage from 0% (black) to 100% (white), with 50% being the normal color. - `hsla(h, s, l, a)`: h, s, and l as above, with a being a number from 0-1 for transparency. - Modern CSS also supports the space-separated format: `hsl(h s l)` and `hsl(h s l / a)`.

Example:

```
 /* Basic HSL colors */
.primary {
  color: hsl(0, 100%, 50%); /* Red */
}

.secondary {
  color: hsl(120, 100%, 50%); /* Green */
}

.tertiary {
  color: hsl(240, 100%, 50%); /* Blue */
}

/* Adjusting saturation and lightness */
.muted {
  color: hsl(0, 60%, 50%); /* Less saturated red */
}

.light {
  color: hsl(0, 100%, 70%); /* Lighter red */
}

.dark {
  color: hsl(0, 100%, 30%); /* Darker red */
}

/* Using HSLA for transparency */
.overlay {
  background-color: hsla(0, 0%, 0%, 0.5); /* Semi-transparent black */
}

/* Using modern space-separated syntax */
.modern {
```

```
  color: hsl(210 50% 50% / 0.8); /* Semi-transparent blue */
}
```

Effect: HSL and HSLA colors apply the specified color to the targeted property, with HSLA allowing for transparency effects. HSL makes it easy to create color variations by adjusting saturation and lightness while keeping the same hue.

Best Practices: - Use HSL when you need to create variations of a color (lighter, darker, more or less saturated). - Use HSL for creating color schemes based on a single hue. - Use HSLA when you need both HSL color control and transparency. - Consider using HSL for theming systems where colors need to be systematically adjusted. - Remember that the hue value is an angle (0-360), so 0 and 360 represent the same hue (red). - Use the modern space-separated syntax for better readability when supported browsers are targeted.

Related Items: - RGB and RGBA colors - Hexadecimal colors - opacity property - CSS custom properties (variables)

Functional Notation

calc() Function

Syntax:

```
selector { property: calc(expression); }
```

Description: The `calc()` function allows mathematical expressions with addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) to be used as property values. It can combine different units and perform calculations at runtime, making it powerful for responsive design.

Values: - `expression`: A mathematical expression that can include: - Numbers - Length units (px, em, rem, %, vw, etc.) - Arithmetic operators (+, -, *, /) - Parentheses for grouping

Example:

```
/* Basic calculations */
.container {
  width: calc(100% - 40px); /* Full width minus 40px margin */
  padding: calc(1rem + 5px); /* Combines relative and absolute units */
}
```

```css
/* Complex calculations */
.sidebar {
  width: calc(100% / 3); /* One-third of the container */
  margin-left: calc(100% / 6); /* Half of the sidebar width */
}

/* Responsive font sizing */
h1 {
  font-size: calc(1.5rem + 2vw); /* Base size plus viewport-relative increment */
}

/* Nested calculations */
.complex {
  height: calc(100vh - (header-height + footer-height)); /* Using CSS variables */
  width: calc((100% - (2 * 1rem)) / 3); /* Width for 3 columns with 1rem gutters */
}
```

Effect: The `calc()` function computes the expression at runtime, allowing for dynamic values that can combine different units or respond to changing conditions like viewport size.

Best Practices: - Use `calc()` to mix different units that otherwise couldn't be combined. - Always include spaces around operators (`+` and `-`) to avoid parsing errors. - Use `calc()` for responsive layouts that need to combine percentages with fixed values. - Consider using CSS custom properties (variables) within `calc()` for more maintainable code. - Use `calc()` to create flexible layouts that maintain specific proportions or constraints. - Remember that `calc()` expressions are evaluated at runtime, so they can adapt to changing conditions.

Related Items: - CSS custom properties (variables) - min(), max(), and clamp() functions - Length units - Viewport units

min(), max(), and clamp() Functions

Syntax:

```css
selector { property: min(value1, value2, ...); }
selector { property: max(value1, value2, ...); }
selector { property: clamp(min, preferred, max); }
```

Description: These CSS math functions provide powerful ways to set responsive values with built-in constraints: - `min()`: Returns the smallest value from a list of comma-separated expressions. - `max()`: Returns the largest value from a list of comma-

separated expressions. - `clamp()`: Returns a value constrained between a minimum and maximum, with a preferred value in between.

Values: - For `min()` and `max()`: A comma-separated list of values or expressions. - For `clamp()`: Three values representing minimum, preferred, and maximum values. - All can include numbers, length units, and calculations.

Example:

```
 /* Using min() */
.container {
  width: min(1200px, 90%); /* Use 90% width, but never exceed 1200px */
  padding: min(5%, 30px); /* Use 5% padding, but never exceed 30px */
}

/* Using max() */
.text {
  font-size: max(16px, 1.2vw); /* At least 16px, but grow with viewport */
  line-height: max(1.5, 1em + 0.5rem); /* Ensure minimum line height */
}

/* Using clamp() */
h1 {
  font-size: clamp(1.5rem, 5vw, 3rem); /* Between 1.5rem and 3rem, preferably 5vw */
}

.flexible-width {
  width: clamp(300px, 50%, 800px); /* Between 300px and 800px, preferably 50% */
}
```

Effect: These functions allow for responsive values that automatically adjust within defined constraints, eliminating the need for many media queries.

Best Practices: - Use `min()` when you want to set a maximum constraint (the smaller value wins). - Use `max()` when you want to set a minimum constraint (the larger value wins). - Use `clamp()` to set both minimum and maximum constraints with a preferred value in between. - These functions are particularly useful for fluid typography and responsive layouts. - Combine with viewport units for truly responsive designs that adapt to screen size. - Remember that these functions can contain multiple values and even nested calculations. - Use these functions to reduce the need for media queries in responsive designs.

Related Items: - calc() function - CSS custom properties (variables) - Viewport units - Media queries - Responsive design

var() Function and Custom Properties

Syntax:

```
/* Defining custom properties */
selector {
  --property-name: value;
}

/* Using custom properties */
selector {
  property: var(--property-name, fallback-value);
}
```

Description: CSS custom properties (also known as CSS variables) allow you to store values that can be reused throughout a document. The `var()` function retrieves the value of a custom property. Custom properties are defined with a double hyphen prefix (`--`) and are accessed using the `var()` function.

Values: - `--property-name` : The name of the custom property, must start with double hyphens. - `fallback-value` : Optional. A value to use if the custom property is not defined or invalid.

Example:

```
/* Defining custom properties on the root element (global scope) */
:root {
  --primary-color: #3498db;
  --secondary-color: #2ecc71;
  --text-color: #333333;
  --spacing-unit: 1rem;
  --border-radius: 4px;
  --max-width: 1200px;
}

/* Using custom properties */
body {
  color: var(--text-color);
  max-width: var(--max-width);
  margin: 0 auto;
}

.button {
  background-color: var(--primary-color);
  padding: var(--spacing-unit) calc(var(--spacing-unit) * 2);
  border-radius: var(--border-radius);
}
```

```css
/* Local scope custom properties */
.card {
  --card-padding: 1.5rem;
  padding: var(--card-padding);
  border-radius: var(--border-radius, 8px); /* Using global with local fallback */
}

/* Using fallback values */
.legacy-support {
  color: var(--undefined-color, #666666); /* Uses fallback if variable not defined */
}
```

Effect: Custom properties create reusable values that can be changed in one place, affecting all instances where they're used. They also respect the cascade and can be dynamically updated with JavaScript.

Best Practices: - Define global custom properties on the `:root` selector for site-wide access. - Use meaningful, descriptive names for custom properties. - Group related custom properties together (e.g., colors, spacing, typography). - Provide fallback values for critical properties to ensure graceful degradation. - Use custom properties for values that are repeated or might change (e.g., theme colors, spacing units). - Take advantage of scoping to create component-specific variables. - Use custom properties with media queries to create responsive designs with fewer code repetitions. - Combine custom properties with `calc()` for dynamic calculations.

Related Items: - calc() function - Color values - Length units - Media queries - JavaScript DOM API for manipulating custom properties

Special Values

Global Keywords

Syntax:

```
selector { property: keyword; }
```

Description: CSS global keywords are special values that can be applied to any CSS property. They provide ways to control inheritance, reset properties to their initial values, or use special behaviors.

Values: - `inherit` : Takes the computed value of the property from the parent element. - `initial` : Sets the property to its initial value (as defined in the CSS specification). - `unset` :

Acts like `inherit` if the property is inherited, otherwise acts like `initial`. - `revert`: Reverts the property to the value it would have had if no styles were applied by the current style origin. - `revert-layer`: Reverts the property to the value from a previous cascade layer.

Example:

```
 /* Using inherit */
.child-element {
  color: inherit; /* Use the same color as the parent */
  font-family: inherit; /* Use the same font family as the parent */
}

/* Using initial */
.reset-element {
  margin: initial; /* Reset to the default margin (usually 0) */
  font-weight: initial; /* Reset to the default font weight (usually 400) */
}

/* Using unset */
.flexible-element {
  color: unset; /* Will inherit if color is inheritable, otherwise use initial */
  display: unset; /* Will use initial since display is not inheritable */
}

/* Using revert */
.browser-default {
  all: revert; /* Revert all properties to browser defaults */
}

/* Using multiple global keywords */
button.custom {
  border: 1px solid black;
}
button.reset {
  border: initial; /* Reset to no border */
  padding: inherit; /* Use parent's padding */
  background-color: unset; /* Inherit or initial depending on property */
}
```

Effect: Global keywords provide ways to control how properties behave in relation to inheritance, default values, and the cascade, offering powerful tools for managing styles across complex projects.

Best Practices: - Use `inherit` when you want to explicitly enforce inheritance for properties that don't inherit by default. - Use `initial` to reset specific properties to their default values without affecting others. - Use `unset` for a more adaptive reset that respects the natural behavior of properties. - Use `revert` when you want to go back to

browser defaults rather than CSS specification defaults. - Consider using the `all` property with these keywords to affect all properties at once. - Be aware that support for `revert` and `revert-layer` may vary in older browsers.

Related Items: - all property - CSS inheritance - CSS cascade - CSS specificity

Special Color Keywords

Syntax:

```
selector { color-property: keyword; }
```

Description: CSS provides several special color keywords that have unique behaviors beyond the standard named colors. These include transparent colors, system colors, and dynamic color values.

Values: - `transparent` : Fully transparent color (equivalent to `rgba(0,0,0,0)`). - `currentColor` : The computed value of the element's `color` property. - System colors (deprecated): `ActiveText` , `ButtonFace` , etc.

Example:

```
/* Using transparent */
.overlay {
  background-color: transparent; /* Fully transparent background */
}

.button {
  border: 1px solid black;
  background-color: transparent; /* Shows through to parent */
}

/* Using currentColor */
.icon {
  fill: currentColor; /* SVG will use the text color */
  stroke: currentColor;
}

.box {
  color: blue;
  border: 1px solid currentColor; /* Border will be blue */
}

/* Dynamic color usage */
a {
  color: blue;
  text-decoration-color: currentColor; /* Underline matches text color */
```

```
}

/* Combining with other properties */
.panel {
  color: #333;
  border-bottom: 3px solid currentColor;
  box-shadow: 0 2px 5px currentColor;
}
```

Effect: Special color keywords provide ways to create transparent elements or dynamically link colors to other properties, enhancing consistency and reducing code repetition.

Best Practices: - Use `transparent` when you need an element to be fully transparent or to show through to underlying content. - Use `currentColor` to maintain color consistency between related properties without repeating color values. - `currentColor` is particularly useful for SVG elements within HTML to inherit colors from their context. - Avoid using deprecated system colors as they may be removed in future browser versions. - Consider using `currentColor` with opacity variations (via `rgba()`) for related but distinct colors. - Remember that `currentColor` refers to the computed value of the `color` property, which may be inherited.

Related Items: - Color values - opacity property - CSS custom properties (variables) - SVG styling

none and auto Keywords

Syntax:

```
selector { property: none|auto; }
```

Description: `none` and `auto` are special keywords used across many CSS properties with context-dependent meanings. Generally, `none` removes or disables a feature, while `auto` lets the browser determine the appropriate value based on context.

Common Uses: - `display: none;` : Removes the element from the document flow and visual rendering. - `display: auto;` : Uses the default display value for that element type. - `border: none;` : Removes all borders. - `outline: none;` : Removes the outline (often used for focus states). - `width/height: auto;` : Sizes based on content or constraints. - `margin/padding: auto;` : Often used for horizontal centering. - `background: none;` : Removes all background properties. - `list-style: none;` : Removes list markers.

Example:

```css
/* Using none */
.hidden {
  display: none; /* Element is not rendered and takes no space */
}

.borderless {
  border: none; /* No border */
}

ul.clean {
  list-style: none; /* No bullets */
}

button.minimal {
  background: none;
  border: none;
  outline: none; /* Note: generally avoid removing focus outlines */
}

/* Using auto */
.responsive-image {
  width: 100%;
  height: auto; /* Maintain aspect ratio */
}

.centered {
  width: 80%;
  margin-left: auto;
  margin-right: auto; /* Horizontal centering */
}

.flexible-container {
  height: auto; /* Size based on content */
}
```

Effect: These keywords provide ways to either remove/disable features (`none`) or let the browser automatically determine appropriate values (`auto`) based on context.

Best Practices: - Use `display: none;` when you want to completely hide elements (but consider accessibility implications). - Avoid `outline: none;` without providing alternative focus indicators, as it harms accessibility. - Use `margin: 0 auto;` for horizontal centering of block elements with defined widths. - Use `height: auto;` to allow elements to size based on their content. - Use `list-style: none;` when creating custom-styled lists or navigation menus. - Remember that `auto` behavior varies significantly depending on the property and context. - For hiding elements while maintaining accessibility, consider alternatives like `visibility: hidden;` or positioning techniques.

Related Items: - display property - visibility property - width and height properties - margin property - border property - outline property