HubSpot

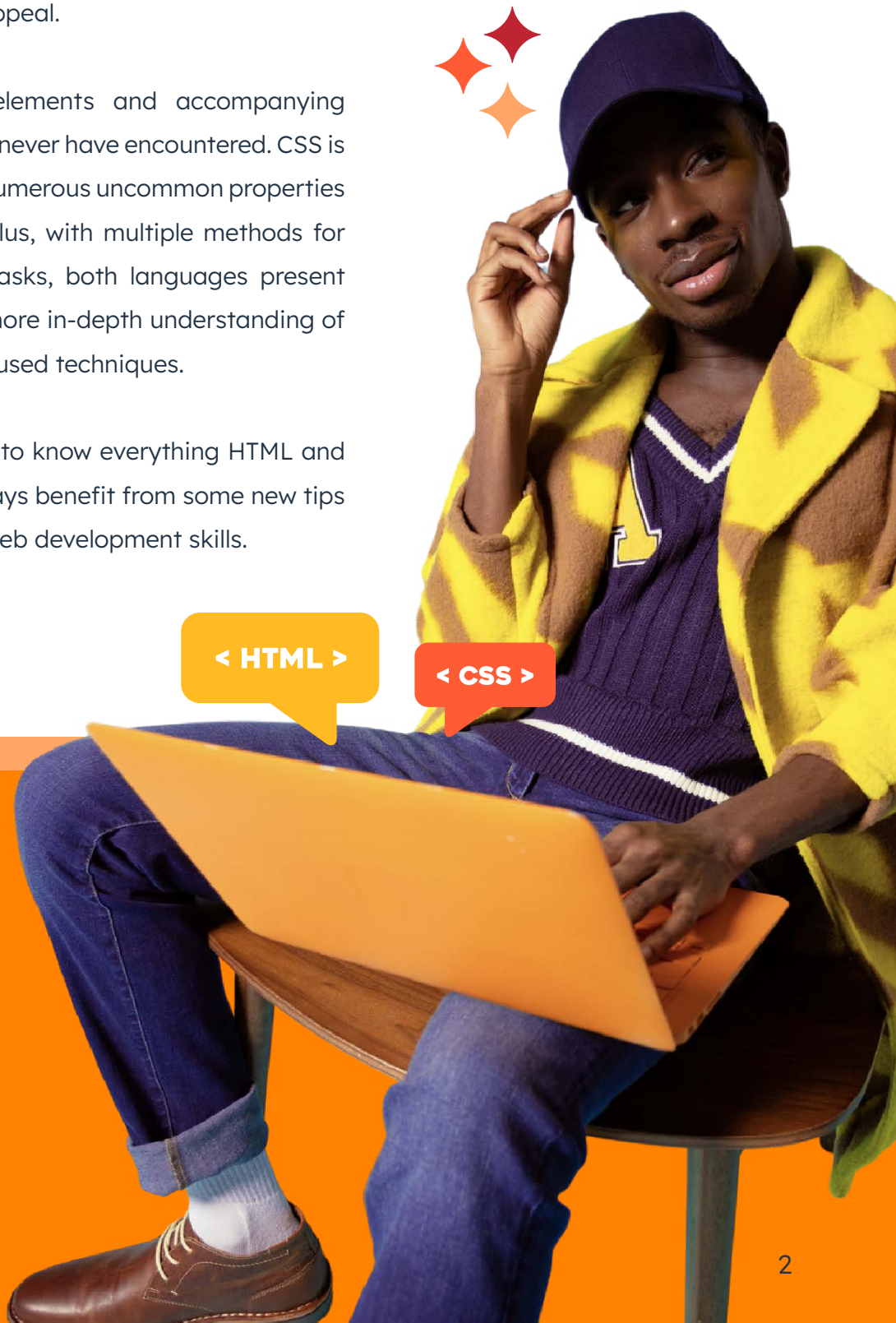
# 25 Tips and Tricks for Using HTML and CSS Like a Pro

# Introduction

HTML and CSS are powerful tools for creating web pages. HTML declares the markup — adding buttons, images, headings, and paragraphs — while CSS styles the elements for visual appeal.

HTML features many elements and accompanying attributes that you might never have encountered. CSS is no different, containing numerous uncommon properties and style declarations. Plus, with multiple methods for accomplishing specific tasks, both languages present opportunities to gain a more in-depth understanding of best practices and underused techniques.

So, while you don't need to know everything HTML and CSS can do, you can always benefit from some new tips and tricks to hone your web development skills.



# Table of Contents

## Expert Insights:

# Becoming an HTML/CSS Power User

This section offers a mixture of hands-on demonstrations and high-level advice about best practices. Where possible, it uses code snippets to demonstrate concepts and features, such as HTML tags and CSS properties.

# HTML: Code to Convention

Here are a few HTML best practices you can use to become an even better developer.

## 01. Write HTML Before CSS

When creating pages or their components, you may be tempted to begin styling before finishing the markup. But, while it can be satisfying to add some aesthetic appeal, you'll likely take longer to finish. Adding style while simultaneously finishing markup means many more modifications, deletions, and additions in your stylesheet.

Instead, first ensure that you complete your HTML, using properly sequenced semantic elements to understand how each element fits on the page.Then, you won't have to restyle your CSS based on HTML-induced layout changes.

## 02. Rely on the Proper Semantic HTML Elements

HTML has numerous element tags, each for a particular purpose. These include **<a>** tags for anchor links, **header** and **footer** tags that determine where your content displays, **<ul>** for unordered (bulleted) list items, and more.

Theoretically, you could use CSS to make a standard paragraph look like a section header or style a ** ** element like an unordered list. However, it's far better to implement appropriate HTML elements where they can work as intended.

There are countless benefits to using proper semantic elements, including:

- Keeping your code simple and readable
- Improving accessibility
- Enabling search engines to better understand your page, which allows for accurate indexing and improved SEO

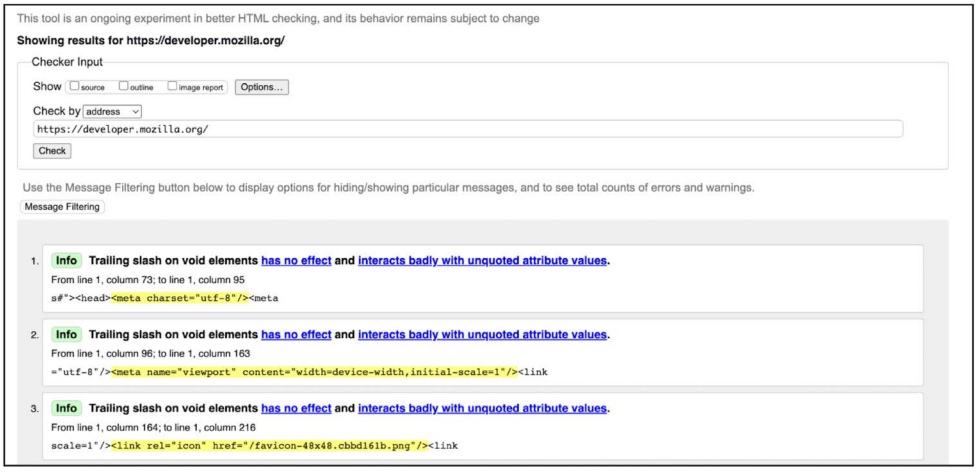
This practice also ensures that you use CSS to its greatest capacity, enabling you to reference specific semantic elements rather than generic ** ** classes.

## 03. Always Validate HTML Code

Web apps must follow certain standards to ensure security, accessibility, and a positive user experience. Ensuring that your web application follows these standards means that it will remain usable, efficient, and optimally secure.

To verify that your web application follows these standards, you can manually compare your site with the W3C documentation. Alternatively, you can use validator tools like the W3C Markup Validator. This tool allows you to provide a URL, upload an HTML file, or paste HTML code directly. Then, it provides validation information, warnings, and errors.

Here's the results page for MDN:



# CSS: Consistency is Key

## 04. Use CSS Resets

Many elements have default CSS styles. Some of these styles come from a browser's default style settings. As a result, even a simple paragraph styled with the CSS below may appear differently across multiple browsers.

```
p {
  border: 2px solid blue;
}
```

To maintain consistency, you should reset all default browser styles using CSS resets. Fortunately, most popular CSS frameworks employ this approach automatically, but you can also reset all styles manually without a framework.

However, instead of scripting a full reset stylesheet, you can use Eric Meyer's Reset, a ubiquitous stylesheet designed for this purpose. The stylesheet code is publicly available and ready to paste into your project.

## 05. Use a CSS Preprocessor to Improve Development Experience

Developers typically want to do more work with less code, and often prefer tools that extend a language's native features. For CSS, you can use a preprocessor that provides features outside its native capabilities. Two popular CSS preprocessors are Sass and Less.

These preprocessors are scripting languages that compile into browser-friendly CSS. You set them up in a development environment, using Sass files for development and the generated CSS files for production.

Among many other features, preprocessors like Sass support nesting style declarations:

```
div {
  color: yellow;

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

The p above matches paragraph tags that are descendants of the **`div`** tag.

## 06. Use a Naming Convention Such as the BEM Convention

When building applications, several collaborators may contribute to a codebase. Every developer has unique preferences, which might result in inconsistencies within the code. Fortunately, maintaining defined structures solves this problem.

One of the most common methodologies is the Block, Element, Modifier (BEM) convention. This convention creates a structure that guides you and your collaborators in writing understandable, maintainable CSS.

It recommends adding classes to HTML elements as follows:

- Use block to specify an independent component.
- Use an element only within the block.
- Use a modifier to specify the appearance or behavior of a block or an element.

# Accommodate Diverse Displays

## 07. Follow a Mobile-First Approach for Writing CSS

Building responsive applications can be a headache, so it's best to start building to accommodate the mobile view. This is because it's easier to add elements to fit a desktop view than to remove elements so a desktop site works on a mobile device.

When writing CSS, first write for narrow, small-screened mobile devices, then make additions for larger screens.

## 08. Use Media Queries for Device Responsiveness

There are different ways to achieve device responsiveness when building applications. One way is efficiently using the **`flex`** or **`grid`** properties or values. However, these values do not apply to every situation. This is where using media queries can be a great solution.

By using media queries, you can target specific screen size ranges and modify your styles to fit the desired range. Suppose you have style modifications for a mobile device, tablet, desktop computer, and an ultra-wide desktop monitor.

```
div {
  max-width: 200px;
  width: 100%;
}

@media screen and (min-width: 400px) {
  div {
    max-width: 400px;
  }
}

@media screen and (min-width: 800px) {
  div {
    max-width: 600px;
  }
}
```

In the style declaration above, the **div** has a **max-width** of **200px**. By using media queries, you can modify this property to have a value of **400px** on a screen less than **400px** wide. For a screen less than **800px** in width, you can modify the property to have a value of **600px**.

## 09. Use Lazy Loading to Improve Application Performance

Lazy loading involves delaying the retrieval of certain resources until needed. This concept improves page load time and system performance.

A typical web page has a header, landing background, several sections, media, and a footer. When you try to load such a page, the browser requests all resources from the server. The time it takes to complete the request depends on the page's resources.

By implementing lazy loading (especially for images), a browser only requests the necessary resources, avoiding the wait for other resources until the user reaches them.

To enable lazy loading of images, you can use the **lazy** value for the **loading** attribute:

```
<img src="image url" alt="image alt" loading="lazy" />
```

# Be Browser-Friendly

## 10. Avoid Using the !important Flag

CSS can prove immensely frustrating when you combine stylesheets. For example, let's say you use a templated stylesheet and your own stylesheet. You may find an element doesn't display as intended because the template style is overwriting your style. The fastest and most straightforward solution is to use the **`!important`** flag.

However, this quick fix can create additional problems. It might conflict with another stylesheet that uses the same flag or reduce your stylesheet's readability. More broadly, it invites poor coding practices that prioritize a less robust solution over high-quality code.

For example, consider the following **`index.html`** file:

```
<link rel="stylesheet" href="./global.css" />
<link rel="stylesheet" href="./index.css" />

<div class="box1">
  <p>Paragraph 1</p>
</div>

<div class="box2">
  <p>Paragraph 2</p>
</div>
```

The HTML above references `global.css` and **`index.css`**. Let's say **`global.css`** has this code: Here, the child **`p`** elements of **`div`** have the **`color`** value blue.

```
div p {
  color: blue;
}
```

Additionally, you want the p elements of **`index.html`** to appear in red. You code this as follows:

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

This style declaration won't work; the **`p`** element would remain blue because the style in the **`global.css`** file has greater specificity than the one we declared in **`index.css`**. Theoretically, we could simply achieve our desired outcome by using the **`!important flag:`**

```
p {
  color: red !important;
}
```

This would indeed change the color of the p element to red. However, what if we want to make the p element in the **`div`** block labeled box2 appear in green? We could add another bit of code:

```
.box2 p {
color: green;
}
```

However, importance takes precedence over specificity in applying styles, so the p element in all **`div`** blocks would remain red. To recolor the p element in the **`div`** block box2 to green, we would need to add another **`!important`** flag:

```
.box2 p {
color: green !important;
}
```

However, adding more **!important** flags quickly disrupts the CSS cascading order in our document — actually reversing it for elements with the flag applied. Browsers use this ordering to decide which styles to display, and it can quickly become difficult to debug a cascade with **!important** flags.

For example, if we wanted to change the color of our p elements to match a user's theme selection, we would need to apply **!important** flags to the declaration for each element in which we've already added an **!important** flag.

Instead, it's better practice to change the specificity of these elements.

One way to remove the need for an **!important** flag in our **index.css** file is by increasing the declaration's specificity to be equal to the one in **global.css**, in which case the last declared rule takes precedence. For example, we can add the **div** type to the declaration in **index.css**:

```
div p {
color: red;
}
```

When importing styles from external stylesheets, we treat their ordering as if the declarations they contain were all concatenated and declared in the referencing document. In our case, **index.css** is declared last, so the p element will have a color value of red.

Then, to make sure the color of the p element in the **div** block labeled box2 is green, we would increase the specificity of its declaration as well by targeting it with a class selector, which takes precedence over the **div** type:

```
.box2 p {
color: green;
}
```

Because this declaration has higher specificity than the other two, it doesn't matter if it's declared before or after them — it will take precedence.

In general, the `!important` flag should only be used to ensure individual rules cannot be overridden later, and avoided at all other times. To override stylesheets you can't remove, it's best to use cascade layers. So, the optimal way to display all of our chosen colors is to reorder our list of styles or change the specificity of the elements they reference. This approach makes it easier to debug our CSS when our content doesn't display as expected.

## 11. Use Fallback Fonts

You must create some web applications following a style guide. This guide prescribes the colors you should use, element sizes, media, and typography. Regarding typography, the style guide will include the font face, type, size, character spacing, and even line spacing.

These font types could be:

- Hosted on the same server as the web application
- Hosted on an external server (like Google fonts)
- Installed on the user's device

So when do you use fallback fonts?

In certain scenarios, a poor internet connection or broken/nonexistent font link can prevent your CSS-defined fonts from loading.

That's why it's best practice to "prepare" by providing fallbacks. Here's what a fallback font declaration looks like:

```
@import url('https://fonts.googleapis.com/
css?family=Random');

body {
  font-family: Random, Consolas, Helvetica, Arial
}
```

For the **`font-family`** property, there are four font names. The first font, Random, is the most-preferred and imported from the CSS file. If that font does not load, the second font, Consolas, replaces it, and so on.

With fallbacks, you can maintain consistent typography when a font does not load. For example, if Consolas is similar to Random, using Consolas will not significantly affect your site's appearance. To take advantage of fallbacks for typography, you can provide alternative fonts in order of their similarity to the original.

# Minimize Bugs, Optimize Performance

## 12. Use Linting Tools to Ensure Cleaner and Syntactically Correct CSS

Linting tools perform source code analysis to determine potential errors, especially regarding syntax errors. Additionally, after analysis, some linting tools can suggest code that matches a specific, predefined, or manually defined standard or best practice.

For example, Stylelint is an efficient CSS linter that has built-in rules for modern CSS. It allows you to create custom rules to accommodate your coding style. It also fixes discovered issues automatically where possible, which is exceptionally convenient for larger codebases.

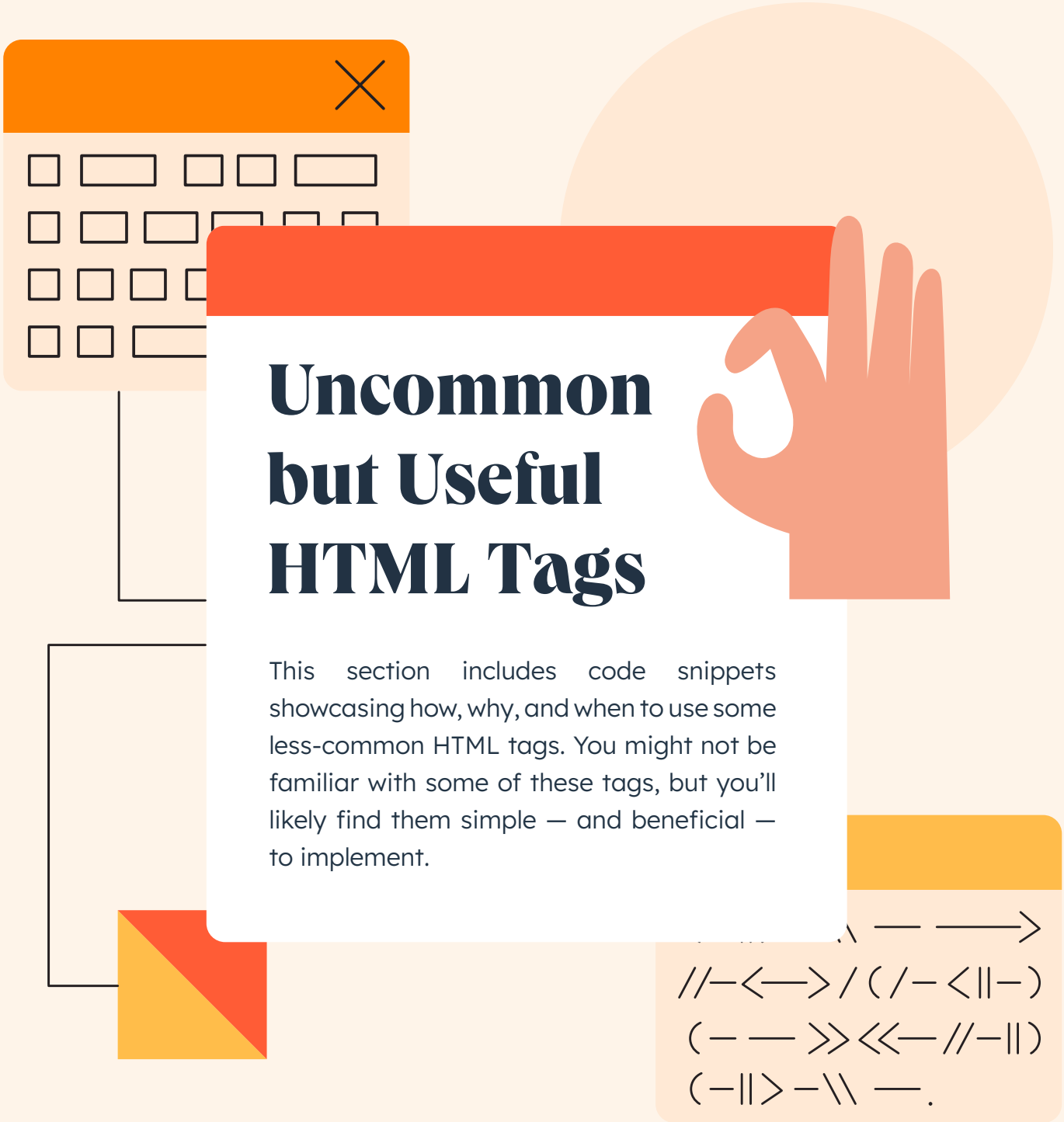
## 13. Use CSS Compression Tools to Optimize Website Performance

Whenever you visit a website, your browser requests resources from the host server. Larger files take longer to fetch and use more network bandwidth.

Therefore, you should consider using minifying or compression methods whenever possible, especially when working on large CSS codebases.

For small or simple projects, compression may make little difference. For such cases, you can use a minifying tool like CSS Minifier.

When you minify your CSS file, ensure you remove unnecessary content like whitespace and nonessential comments. While its results vary by case, this practice may significantly reduce file size and enhance site performance.



# Uncommon but Useful HTML Tags

This section includes code snippets showcasing how, why, and when to use some less-common HTML tags. You might not be familiar with some of these tags, but you'll likely find them simple — and beneficial — to implement.

# Interactive Elements

## 14. Autocomplete <datalist> tag

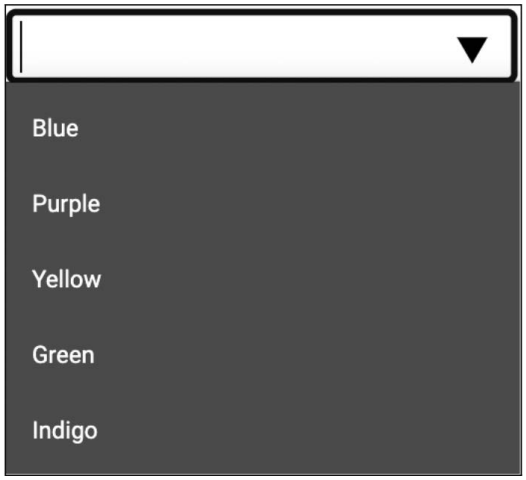
The **datalist** HTML tag allows you to provide a list of optional elements that users can choose from when providing inputs.

**Here's an example:**

```
<input list="colors-list" />

<datalist id="colors-list">
  <option value="Blue" />
  <option value="Purple" />
  <option value="Yellow" />
  <option value="Green" />
  <option value="Indigo" />
</datalist>
```

**Result:**




You connect the input and **datalist** by passing the **datalist** ID to the input's **list** attribute.

The input also uses the **datalist** for auto-completion. For example, if the user types "B," the data list shows Blue:

## 15. <optgroup>

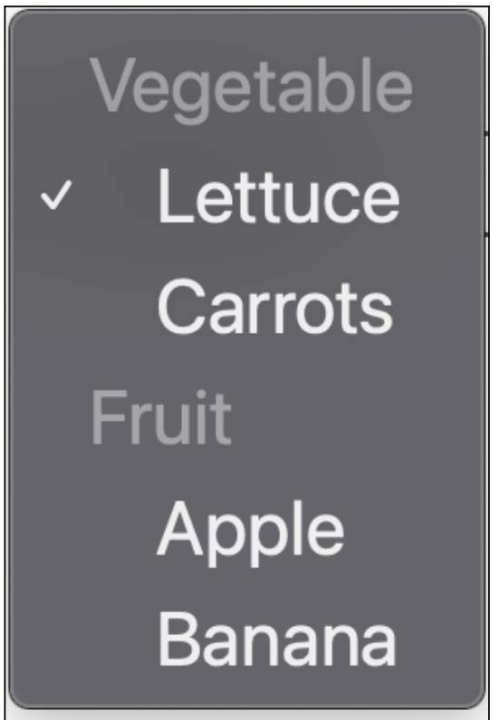
You can group the options in a select element using the **optgroup** tag. This tag is useful when you have a list of options with different categories and helps the user understand what option to choose.

**Here's an example:**

```
<select>
  <optgroup label="Vegetable">
    <option>Lettuce</option>
    <option>Carrots</option>
  </optgroup>

  <optgroup label="Fruit">
    <option>Apple</option>
    <option>Banana</option>
  </optgroup>
</select>
```

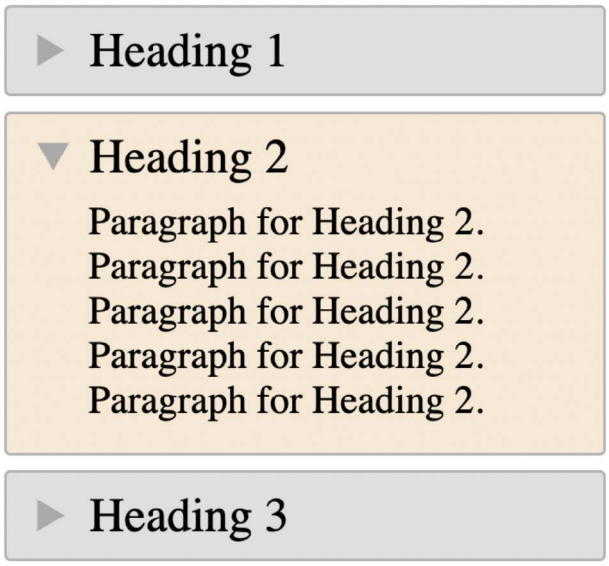
Depending on the default styles your browser provides for the **select** element, you should have a result similar to this:



## 16. <details> tag

Suppose you build a stacked list of elements known as an accordion. Each item can be expanded or collapsed.

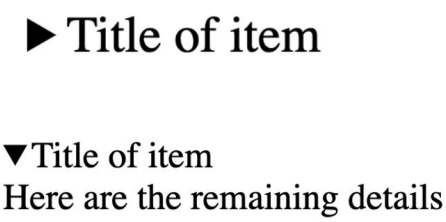
You can implement this functionality with JavaScript, but you can also do it with the **`details`** tag in HTML. Using this tag automatically makes the functionality accessible to every user. JavaScript requires more work to ensure it is accessible.




**Here's an example:**

```
<details>
  <summary>Title of item</
summary>
  Here are the remaining
details
</details>
```

**The result:**




By using some CSS, you can achieve the complete look you want. Also, the **`details`** element has an open attribute that can be **`true`** or **`false`**. With some JavaScript, you can programmatically expand or collapse the element and achieve the functionality you want.

# Displaying Content

## 17. <meter> tag

The **meter** tag is very useful for displaying values that fall into a particular range. It reduces the need for HTML, CSS, and even JavaScript.

Assume you want to display someone's score where:

- The minimum is 0.
- The maximum is 70.
- The failing score is anything less than 30.
- The passing score is anything more than 55.

You can do this with the **meter** tag with a value of 20 as follows:

```
<meter
  min="0"
  max="70"
  low="30"
  high="50"
  optimum="70"
  value="20"
>
  20
</meter>
```

You specify the minimum with the **min** attribute, the maximum with **max**, failure with **low**, passing with **high**, the best with **optimum**, and the value itself with **value**.

The result:

With a value of 20, the meter is red. Here's the same meter with values of 40 and 60, respectively:

## 18. <template> tag

As the name implies, the **`template`** tag allows you to create a template. This works because the browser doesn't render the template, but some JavaScript code can make the template appear during runtime after the page loads.

In essence, it is HTML code that you store somewhere to use later:

```
<h1>Template</h1>

<p>I am not a template</p>

<template id="para-
template">
  <p>I am a template</p>
</template>
```

**Result:**

**Template**

I am not a template

The browser doesn't render the text under the **`para-template`** template during page load. However, you can use JavaScript to display the template's contents:

```
const template = document.
getElementById("para-
template")
document.
querySelector("body").
appendChild(template.
content)
```

**Result:**

**Template**

I am not a template

I am a template

First, you get the template with the **`id`** of **`para-template`**. Then you append the content of the template to the body element. The result is that instead of creating elements in JavaScript, you can simply create a template in HTML and display it as needed.


# Lesser-Known CSS Properties to Elevate Your Skills

You can sharpen your code by using some of these CSS properties.

# Style to a T

## 19. all property

The all property allows you to reset all style properties of an element (except **`unicode-bidi`**, **`direction`**, and CSS Custom Properties). You can reset them to their initial or inherited values.

**For example:**

**HTML**

```
<div>
  <p>Hello</p>
</div>
```

**CSS**

```
p {
  color: green;
  border: 2px solid blue;
  font-weight: bold;
}
```

**The result:**


The **`p`** tag has **`color`**, **`border`**, and **`font-weight`** style declarations according to the code above. By adding the **`all`** property with an **`initial`** value, these properties return to their defaults:

```
p {
  color: green;
  border: 2px solid blue;
  font-weight: bold;
  all: initial;
}
```

**The result:**

Hello

If you use the **`all`** property with an **`inherit`** value, it inherits properties from the element's parents or ancestors:

```
div {
  color: blue;
  border: 2px solid blue;
}

p {
  color: green;
  font-weight: bold;
  all: inherit;
}
```

**The result:**


As you can see here, the p element inherits the **`color`** and **`border`** properties from the div, and the p loses its own styles.

## 20. currentColor

The **`currentColor`** keyword specifies the **`color`** property within its designated scope.

Sometimes, when writing CSS, you may style a child element with the color specified for the parent element. This could look like this:

**HTML**

```
<div>
  <p>I am a paragraph</p>
</div>
```

**CSS**

```
div {
  color: green;
}
p {
  border: 1px solid green;
}
```

This achieves the desired result, but if you want to change the color of the **div**, you also have to change it in the p style's declaration. A simple way to go about this is using **currentColor**:

```
div {
  color: green;
}

p {
  border: 1px solid
currentColor;
}
```

**The result:**

I am a paragraph

The current value of **currentColor** is green, as that's what the parent (**div**) or the target element (**p**) has.

## 21. CSS Counters

CSS counters are variables that change dynamically depending on the number of times you use them. With such variables, you can add numbering to elements without using the **ol** tag.

Suppose you have the following elements:

```
<section>
  <h2>First heading</h2>
  <h2>Second heading</h2>
  <h2>Third heading</h2>
</section>
```

If you wanted to number these headings (**1, 2, 3**), you would typically have to do that manually, as using **ol** and **h2** would be semantically incorrect. But with CSS counters, you can automatically enable the numbering:

```
section {
  counter-reset: heading;
}

section h2 {
  counter-increment:
heading;
}

section h2::before {
  content:
counter(heading) " - ";
}
```

**The result:**

**1 - First heading**

**2 - Second heading**

**3 - Third heading**

First, declare a counter variable (using the **counter-reset** property and the variable's name: **heading**). This variable has a default value of 0. The number of times **counter-increment** is called as a variable. It increments by a default value of 1. For the first h2, **counter-increment** is triggered on the **heading** variable, which increments it to 1. By using the **counter** function and passing the variable, you can display that value using the **::before** pseudo-element and the **content** property.

For the second and third h2 elements, the **heading** variable triggers the **counter-increment**, which increments it to 2 and 3, respectively.

If you add another h2 automatically, the variable becomes 4 and is displayed. Thanks to the CSS counters feature, you don't need to do anything.

# Fine-Grained Control

## 22. :is and :where

You can improve the way you target elements to style in CSS using the :is and :where pseudo-class. They allow you to write readable and shorter CSS.

The **`:is and :where`** pseudo-classes allow you to select elements that match a list of arguments you provide.

Instead of doing this:

```
h1:hover,
h2:hover,
h3:hover,
h4:hover,
h5:hover,
h6:hover {
  text-decoration:
underline;
  color: green;
}
```

You can use the **`:is (or :where)`** class like this:

```
:is(h1, h2, h3, h4, h5,
h6):hover {
  text-decoration: underline;
  color: green;
}

/* or */

:where(h1, h2, h3, h4, h5,
h6):hover {
  text-decoration: underline;
  color: green;
}
```

The difference between **`:is`** and **`:where`** is the CSS specificity. The specificity of **`:is`** is the selector (in the arguments) with the highest specificity, so the specificity of **`:where`** is 0. Furthermore, the specificity of a selector (or selectors) determines its strength.

Additionally, the types of selectors you use in a style determine their strength. When you try to overwrite a style declaration, if the selectors in the new declaration are not as strong as those in the declaration you want to overwrite, such overwriting would not be successful. You can learn more about CSS specificity on MDN.

## 23. Isolation

The **`isolation`** property specifies a new stacking context or not. A stacking context controls how elements appear on a webpage over or under other elements.

Look at this example:

**HTML**

```
<body>
  <div class="card">
    <button>Click Me!</
button>
  </div>
</body>
```

**CSS**

```
div {
  background-color: blue;
  width: max-content;
  padding: 30px;
}

div button {
  position: relative;
}
```

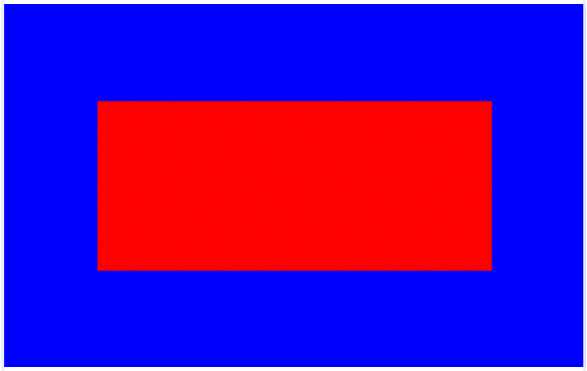
**Here is the result:**


The **body** element creates the stacking context, so every stack ordering for elements is determined by the body.

Suppose you have an **::after** pseudo-element on the button which provides a background for it:

```
div button::after {
  content: “”;
  position: absolute;
  left: -10px;
  right: -10px;
  top: -10px;
  bottom: -10px;
     background-color:
red;
}
```

**Here is the result:**



But now, to make the pseudo-element appear at the back of the button, use **z-index:**

```
div button::after {
  /* other code */
  z-index: -1;
}
```

**Here is the result:**

Because the current stacking context has the **div**, then the button on the V, using -1 for the **z-index** pushes the red background behind the **div**. There are different ways to make the red background appear between the **div** and the button. The easiest way is using **isolation**:

```
div {
     background-color:
blue;
  width: max-content;
  padding: 30px;
  isolation: isolate;
}
```

**The result:**


By using **isolation: isolate** on the **div**, the **div** creates its own stacking context for its children, so every stack ordering will now be relative to it. So using **z-index: -1** on the red background would extend beyond the **div**.

## 24. Contain

When the browser renders elements, it calculates their layouts, including how much space they occupy and how one element's layout affects another in the DOM. When an element's size changes, other elements will be affected — repainted — to fit the new layout.

Repainting can lead to performance problems, especially in large applications. This is where the **contain CSS property** comes in. This property allows you to control an element's "containment" to avoid outside side effects when the element's layout changes. It allows you to paint an element (and its descendants) independently, thereby improving performance as the browser does not need to recalculate layouts for the contents of the whole page.

It accepts a couple of values. Here are the layout and paint values:

```
element1 {
  contain: layout;
}

element2 {
  contain: paint;
}
```

The **`layout`** value indicates to the browser that the element's contents do not affect other elements' layouts, nor do other elements affect the contained element's internal layout. This way, the **`contained`** element renders independently.

The **`paint`** value indicates to the browser that it should not paint anything outside the **`contained`** element's border. So, any of the element's descendants styled to display outside the border are clipped.

## 25. Use Shorthand Properties for More Concise and Readable CSS

There are some properties in CSS that allow you to provide values for multiple properties at once. This pattern allows you to write clearer, more readable, and more concise code.

For styling element borders, you often use the following properties:

- `border-width`
- `border-color`
- `border-style`

Instead of individually declaring these properties, like this:

```
element {
  border-width: 2px;
  border-color: green;
  border-style: solid;
}
```

You can use the border shorthand property to declare the properties simultaneously:

```
element {

  border: 2px solid green;
}
```

Some shorthand properties, however, have a specific order, so providing the values in a different order may not result in a style modification.

# Conclusion

Implementing best practices for HTML and CSS in your web applications ensures that you can build accessible, user-friendly, and SEO-optimized applications. And while the plethora of available languages can help you create a visually rich and functional web page, you can benefit substantially from learning to optimize your use of the HTML and CSS, which are foundational to superior web design.