

Lesson 12

Writing Your Best Code

In this Lesson

12

HTML

- [HTML Coding Practices](#)
- [Additional Resources](#)

CSS

- [CSS Coding Practices](#)

SHARE

There's a lot to learn—different elements, attributes, properties, values, and more—in order to write HTML and CSS. Every lesson until this point has had the primary objective of explaining these various components of HTML and CSS, in hopes of helping you to understand the core fundamentals of both languages. This lesson takes a step back and looks at a more abstract picture of HTML and CSS.

More specifically, this lesson focuses on the best coding practices for both HTML and CSS. These coding practices serve as an overarching framework for writing HTML and CSS. They apply to every lesson and should always be kept in mind when programming.

When you're reviewing these best practices think about how they may be used in other areas or programming languages, too. For example, the use of comments to organize code (as we cover in this lesson) is beneficial in all programming languages. Keep an open mindset and consider how you can fully utilize each practice.

HTML Coding Practices

A lot of coding best practices emphasize keeping code lean and well organized. The general practices within HTML are no different. The goal is to write well-structured and standards-compliant markup. The guidelines described here provide a brief introduction to HTML coding practices; this is by no means an exhaustive list.

Write Standards-Compliant Markup

HTML, by nature, is a forgiving language that allows poor code to execute and render to varying levels of accuracy. Successful rendering, however, does not mean that our code is semantically correct or guarantee that it will validate as standards compliant. In addition, poor code is unpredictable, and you can't be certain what you're going to get when it renders. We have to pay close attention when writing HTML and be sure to nest and close all elements correctly, to use IDs and classes appropriately, and to always validate our code.

The code that follows has multiple errors, including using the `intro` ID attribute value multiple times when it should be a unique value, closing the `<p>` and `` elements in the wrong order within the first paragraph, and not closing the `<p>` element at all in the second paragraph.

Bad Code

```
1      <p id="intro">New items on the menu today include <strong>caramel appl
2      <p id="intro">The caramel apple cider is delicious.
```



Good Code

```
1      <p class="intro">New items on the menu today include <strong>caramel a
2      <p class="intro">The caramel apple cider is delicious.</p>
```



Make Use of Semantic Elements

The library of elements in HTML is fairly large, with well over 100 elements available for use. Deciding which elements to use to describe different content may be difficult, but these elements are the backbone of semantics. We need to research and double-check our code to ensure we are using the proper semantic elements. Users will thank us in the long run for building a more accessible website, and your HTML will arguably be easier to style. If you are ever unsure of your code, find a friend to help out and perform routine code reviews.

Here the HTML doesn't use the proper heading and paragraph elements; instead, it uses meaningless elements to style and group content.

Bad Code

```
1      <span class="heading"><strong>Welcome Back</span></strong>
2      <br><br>
3
```

```
4      It has been a while. What have you been up to lately?  
      <br><br>
```

Good Code

```
1      <h1>Welcome Back</h1>  
2      <p>It has been a while. What have you been up to lately?</p>
```

Use the Proper Document Structure

As previously mentioned, HTML is a forgiving language and, therefore, pages will render without the use of the `<!DOCTYPE html>` doctype or `<html>`, `<head>`, and `<body>` elements. Without a doctype and these structural elements, pages will not render properly in every browser.

We must always be sure to use proper document structure, including the `<!DOCTYPE html>` doctype, and the `<html>`, `<head>`, and `<body>` elements. Doing so keeps our pages standards compliant and fully semantic, and helps guarantee they will be rendered as we wish.

Bad Code

```
1      <html>  
2      <h1>Hello World</h1>  
3      <p>This is a web page.</p>  
4      </html>
```

Good Code

```
1      <!DOCTYPE html>  
2      <html>  
3      <head>  
4          <title>Hello World</title>  
5      </head>  
6      <body>  
7          <h1>Hello World</h1>  
8          <p>This is a web page.</p>  
9
```

```
10      </body>
      </html>
```

Keep the Syntax Organized


As pages grow, managing HTML can become quite a task. Thankfully there are a few quick rules that can help us keep our syntax clean and organized. These include the following:

- Use lowercase letters within element names, attributes, and values
- Indent nested elements
- Strictly use double quotes, not single or completely omitted quotes
- Remove the forward slash at the end of self-closing elements
- Omit the values on Boolean attributes

Observing these rules will help keep our code neat and legible. Looking at the two sets of HTML here, the good code is easier to digest and understand.

Bad Code

```
1      <Aside>
2      <h3>Chicago</h3>
3      <H5 HIDDEN='HIDDEN'>City in Illinois</H5>
4      <img src=chicago.jpg alt="Chicago, the third most populous city in th
5      <ul>
6      <li>234 square miles</li>
7      <li>2.715 million residents</li>
8      </ul>
9      </ASIDE>
```



Good Code

```
1      <aside>
2      <h3>Chicago</h3>
3      <h5 hidden>City in Illinois</h5>
4      
6      <li>234 square miles</li>
7      <li>2.715 million residents</li>
```

```
8         </ul>
9     </aside>
```

Use Practical ID & Class Values

Creating ID and class values can be one of the more difficult parts of writing HTML. These values need to be practical, relating to the content itself, not the style of the content. Using a value of `red` to describe red text isn't ideal, as it describes the presentation of the content. Should the style of the text ever need to be changed to blue, not only does the CSS have to be changed, but so does the HTML in every instance where the class `red` exists.

The HTML here assumes that the alert message will be red. However, should the style of the alert change to orange the class name of `red` will no longer make sense and will likely cause confusion.

Bad Code

```
1     <p class="red">Error! Please try again.</p>
```

Good Code

```
1     <p class="alert">Error! Please try again.</p>
```

Use the Alternative Text Attribute on Images

Images should always include the `alt` attribute. Screen readers and other accessibility software rely on the `alt` attribute to provide context for images.

The `alt` attribute value should be very descriptive of what the image contains. If the image doesn't contain anything of relevance, the `alt` attribute should still be included; however, the value should be left blank so that screen readers will ignore it rather than read the name of the image file.

Additionally, if an image doesn't have a meaningful value—perhaps it is part of the user interface, for example—it should be included as a CSS background image if at all possible, not as an `` element.

Bad Code

```
1 
```

Good Code

```
1 
```



Separate Content from Style

Never, ever, use inline styles within HTML. Doing so creates pages that take longer to load, are difficult to maintain, and cause headaches for designers and developers. Instead, use external style sheets, using classes to target elements and apply styles as necessary.

Here, any desired changes to styles within the bad code must be made in the HTML. Consequently, these styles cannot be reused, and the consistency of the styles will likely suffer.

Bad Code

```
1 <p style="color: #393; font-size: 24px;">Thank you!</p>
```

Good Code

```
1 <p class="alert-success">Thank you!</p>
```

Avoid a Case of “Divitis”

When writing HTML, it is easy to get carried away adding a `<div>` element here and a `<div>` element there to build out any necessary styles. While this works, it can add quite a bit of bloat to a page, and before too long we’re not sure what each `<div>` element does.

We need to do our best to keep our code lean and to reduce markup, tying multiple styles to a single element where possible. Additionally, we should use the HTML5 structural elements where suitable.

Bad Code

```
1      <div class="container">
2          <div class="article">
3              <div class="headline">Headlines Across the World</div>
4          </div>
5      </div>
```

Good Code

```
1      <div class="container">
2          <article>
3              <h1>Headlines Across the World</h1>
4          </article>
5      </div>
```

Continually Refactor Code

Over time websites and code bases continue to evolve and grow, leaving behind quite a bit of cruft. Remember to remove old code and styles as necessary when editing a page. Let's also take the time to evaluate and refactor our code after we write it, looking for ways to condense it and make it more manageable.

CSS Coding Practices

Similar to those for HTML, the coding practices for CSS focus on keeping code lean and well organized. CSS also has some additional principles regarding how to work with some of the intricacies of the language.

Organize Code with Comments

CSS files can become quite extensive, spanning hundreds of lines. These large files can make finding and editing our styles nearly impossible. Let's keep our styles organized in logical groups. Then, before each group, let's provide a comment noting what the following styles pertain to.

Should we wish, we can also use comments to build a table of contents at the top of our file. Doing so reminds us—and others—exactly what is contained within the file and where the styles are located.

Bad Code

```
1      header { ... }
2      article { ... }
```

```
3      .btn { ... }
```

Good Code

```
1      /* Primary header */
2      header { ... }
3
4      /* Featured article */
5      article { ... }
6
7      /* Buttons */
8      .btn { ... }
```

Write CSS Using Multiple Lines & Spaces

When writing CSS, it is important to place each selector and declaration on a new line. Then, within each selector we'll want to indent our declarations.

After a selector and before the first declaration comes the opening curly bracket, {, which should have a space before it. Within a declaration, we need to put a space after the colon, :, that follows a property and end each declaration with a semicolon, ;.

Doing so makes the code easy to read as well as edit. When all of the code is piled into a single line without spaces, it's hard to scan and to make changes.

Bad Code

```
1      a,.btn{background:#aaa;color:#f60;font-size:18px;padding:6px;}
```

GOOD CODE

```
1      a,
2      .btn {
3          background: #aaa;
4          color: #f60;
5          font-size: 18px;
6          padding: 6px;
7      }
```


Comments & Spacing

These two recommendations, organizing code with comments and using multiple lines and spaces, are not only applicable to CSS, but also to HTML or any other language. Overall we need to keep our code organized and well documented. If a specific part of our code is more complex, let's explain how it works and what it applies to within comments. Doing so helps others working on the same code base, as well as ourselves when we revisit our own code down the road.

Use Proper Class Names

Class names (or values) should be modular and should pertain to content within an element, not appearance, as much as possible. These values should be written in such a way that they resemble the syntax of the CSS language. Accordingly, class names should be all lowercase and should use hyphen delimiters.

Bad Code

```
1      .Red_Box { ... }
```

Good Code

```
1      .alert-message { ... }
```

Build Proficient Selectors

CSS selectors can get out of control if they are not carefully maintained. They can easily become too long and too location specific.

The longer a selector is and the more prequalifiers it includes, the higher specificity it will contain. And the higher the specificity the more likely a selector is to break the CSS cascade and cause undesirable issues.

Also in line with keeping the specificity of our selectors as low as possible, let's not use IDs within our selectors. IDs are overly specific, quickly raise the specificity of a selector, and quite often break the cascade within our CSS files. The cons far outweigh the pros with IDs, and we are wise to avoid them.

Let's use shorter and primarily direct selectors. Nest them only two to three levels deep, and remove as many location-based qualifying selectors as possible.

Bad Code

```
1      #aside #featured ul.news li a { ... }
2      #aside #featured ul.news li a em.special { ... }
```

Good Code

```
1      .news a { ... }
2      .news .special { ... }
```

Use Specific Classes When Necessary

There are times when a CSS selector is so long and specific that it no longer makes sense. It creates a performance lag and is strenuous to manage. In this case, using a class alone is advised. While applying a class to the targeted element may create more code within HTML, it will allow the code to render faster and will remove any managing obstacles.

For example, if an `` element is nested within an `<h1>` element inside of an `<aside>` element, and all of that is nested within a `<section>` element, the selector might look something like `aside h1 em`. Should the `` element ever be moved out of the `<h1>` element the styles will no longer apply. A better, more flexible selector would use a class, such as `text-offset`, to target the `` element.

Bad Code

```
1      section aside h1 em { ... }
```

Good Code

```
1      .text-offset { ... }
```

Use Shorthand Properties & Values

One feature of CSS is the ability to use shorthand properties and values. Most properties and values have acceptable shorthand alternatives. As an example, rather than using four different `margin`-based property and value declarations to set the margins around all four sides of an element, use one single `margin` property and value declaration that sets the values for all four sides at once. Using the shorthand alternative allows us to quickly set and identify styles.

When we're only setting one value, though, shorthand alternatives should not be used. If a box only needs a bottom `margin`, use the `margin-bottom` property alone. Doing so ensures that other margin values will not be overwritten, and we can easily identify which side the `margin` is being applied to without much cognitive effort.

Bad Code

```
1      img {
2          margin-top: 5px;
3          margin-right: 10px;
4          margin-bottom: 5px;
5          margin-left: 10px;
6      }
7      button {
8          padding: 0 0 0 20px;
9      }
```

Good Code

```
1      img {
2          margin: 5px 10px;
3      }
4      button {
5          padding-left: 20px;
6      }
```

Use Shorthand Hexadecimal Color Values

When available, use the three-character shorthand hexadecimal color value, and always use lowercase characters within any hexadecimal color value. The idea, again, is to remain consistent, prevent confusion, and embrace the syntax of the language the code is being written in.

Bad Code

```
1      .module {
2          background: #DDDDDD;
3          color: #FF6600;
4      }
```

Good Code

```
1      .module {  
2          background: #ddd;  
3          color: #f60;  
4      }
```

Drop Units from Zero Values

One way to easily cut down on the amount of CSS we write is to remove the unit from any zero value. No matter which length unit is being used—pixels, percentages, em, and so forth—zero is always zero. Adding the unit is unnecessary and provides no additional value.

Bad Code

```
1      div {  
2          margin: 20px 0px;  
3          letter-spacing: 0%;  
4          padding: 0px 5px;  
5      }
```

Good Code

```
1      div {  
2          margin: 20px 0;  
3          letter-spacing: 0;  
4          padding: 0 5px;  
5      }
```

Group & Align Vendor Prefixes

With CSS3, vendor prefixes gained some popularity, adding quite a bit of code to CSS files. The added work of using vendor prefixes is often worth the generated styles; however, they have to be kept organized. In keeping with the goal of writing code that is easy to read and modify, it's best to group and indent individual vendor prefixes so that the property names stack vertically, as do their values.

Depending on where the vendor prefix is placed, on the property or the value, the alignment may vary. For example, the following good code keeps the `background` property aligned to the left, while the prefixed `linear-gradient()` functions are indented to keep their values vertically stacked. Then, the prefixed `box-sizing`

property is indented as necessary to keep the box-sizing properties and values vertically stacked.

As always, the objective is to make the styles easier to read and to edit.

Bad Code

```
1      div {
2          background: -webkit-linear-gradient(#a1d3b0, #f6f1d3);
3          background: -moz-linear-gradient(#a1d3b0, #f6f1d3);
4          background: linear-gradient(#a1d3b0, #f6f1d3);
5          -webkit-box-sizing: border-box;
6          -moz-box-sizing: border-box;
7          box-sizing: border-box;
8      }
```

Good Code

```
1      div {
2          background: -webkit-linear-gradient(#a1d3b0, #f6f1d3);
3          background:      -moz-linear-gradient(#a1d3b0, #f6f1d3);
4          background:          linear-gradient(#a1d3b0, #f6f1d3);
5          -webkit-box-sizing: border-box;
6              -moz-box-sizing: border-box;
7                  box-sizing: border-box;
8      }
```

Vendor Prefixes

When using vendor prefixes we need to make sure to place an unprefixed version of our property and value last, after any prefixed versions. Doing so ensures that browsers that support the unprefixed version will render that style according to its placement within the cascade, reading styles from the top of the file to the bottom.

The good news is that browsers are largely moving away from using vendor prefixes. Over time this will become less of a concern; however, for now we're well advised to double-check which styles require a vendor prefix and to keep those prefixes organized.

Modularize Styles for Reuse

CSS is built to allow styles to be reused, specifically with the use of classes. For this reason, styles assigned to a class should be modular and available to share across elements as necessary.

If a section of news is presented within a box that includes a border, background color, and other styles, the class of `news` might seem like a good option. However, those same styles may also need to be applied to a section of upcoming events. The class of `news` doesn't fit in this case. A class of `feat-box` would make more sense and may be widely used across the entire website.

Bad Code

```
1      .news {
2          background: #eee;
3          border: 1px solid #ccc;
4          border-radius: 6px;
5      }
6      .events {
7          background: #eee;
8          border: 1px solid #ccc;
9          border-radius: 6px;
10     }
```

Good Code

```
1      .feat-box {
2          background: #eee;
3          border: 1px solid #ccc;
4          border-radius: 6px;
5      }
```

Additional Resources & Links

Every lesson has come with a few resources for additional learning and discovery. Outlined below is a longer list of resources, as well as beneficial links.

HTML & CSS

- [Mozilla Developer Network](#) via Mozilla
- [Opera.Dev](#) via Opera
- [HTML and CSS Tutorials](#) via HTML Dog
- [DevDocs](#) — Instant documentation search

Design Inspiration

- [Dribbble](#)
- [Premium Pixels](#)

Frameworks & Style Guides

- [Web Style Guide](#)
- [Bootstrap](#)
- [Foundation](#)
- [Skeleton Framework](#)
- [Google HTML/CSS Style Guide](#)
- [GitHub Styleguide](#)

Icons

- [Helveticons](#) via Goodbye Horses
- [Ion Icons](#) via Ben Sperry
- [Fugue Icons](#) via Yusuke Kamiyamane
- [famfamfam Silk Icons](#) via Mark James
- [Pictos](#) via Drew Wilson
- [Picons](#)
- [The Noun Project](#)

Miscellaneous

- [COLOURlovers](#) — Color trends and palettes
- [ColorHexa](#) — Color encyclopedia
- [Modernizr](#) — JavaScript feature detection library
- [jQuery](#) — Feature-rich JavaScript library
- [Google Hosted Libraries](#) — Content distribution network for JavaScript libraries
- [Copy Paste Character](#) — Copying the “hidden” characters

Summary

Hopefully the principles of writing beautiful HTML and CSS are starting to become clear here. While each language does have its own intricacies, the majority of these practices can be shared across the two languages—and many other computer languages.

Individually we need to do our best to uphold these practices, and when working on a team we need to do our best to help educate the team on these practices, too. Likewise, our teams may have valuable suggestions and practices that we should work together to follow.

To highlight some of the overarching themes of this lesson, our HTML and CSS should always

- Be well organized, so that it is easy to read, edit, and maintain
- Be modular and flexible, allowing us to reuse code and patterns as necessary
- Look as if one person wrote it, even if several people contributed

These practices are only the beginning, and as the languages evolve and we write more and more HTML and CSS, we'll develop new ones. It's all part of the beauty of knowing HTML and CSS.

You're now equipped with some very powerful knowledge about how to build websites with HTML and CSS, and I'm excited to see what you do with it. Keep me posted on how it goes, and happy building!

Lesson 11

[Organizing Data with Tables](#)

Learn More HTML & CSS or Study Other Topics

Learning how to code HTML & CSS and building successful websites can be challenging, and at times additional help and explanation can go a long way. Fortunately there are plenty of online schools, boot camps, workshops, and the alike, that can help.

Select your topic of interest below and I will recommend a course I believe will provide the best learning opportunity for you.

Select Your Topic of Interest:

Design & Product
Front-end Development
Web Development
Mobile Development
Data & Machine Learning
Info & Cyber Security

Your Course Recommendations:

Select a topic above to view your course recommendations.

Learn to Code HTML & CSS the Book



Learn to Code HTML & CSS is an interactive beginner's guide with one express goal: **teach you how to develop and style websites** with HTML and CSS. Outlining the fundamentals, this book covers all of the common elements of front-end design and development.

Buy Learn to Code HTML & CSS

Also available at [Amazon](#) and [Barnes & Noble](#)

Looking for Advanced HTML & CSS Lessons?

Checkout these [advanced HTML and CSS lessons](#) to take a deeper look at front-end design and development, perfect for any designer or front-end developer looking to round out their skills.

View Advanced HTML & CSS Lessons

Join the Newsletter

To stay up to date and learn when new courses and lessons are posted, please sign up for the newsletter—spam free.

Get Updates

© Shay Howe

Enjoy these lessons?

Follow @shayhowe