# 1 CSS introduction

This chapter covers

- A brief overview of CSS
- Basic CSS styling
- How to select HTML elements effectively

Cascading Style Sheets (CSS) is used to control the appearance of the elements of a web page. CSS uses style rules to instruct the browser to select certain elements and apply styles and effects to them.

Chapter 1 is a good place to start if you're new to CSS or in need of a refresher. We'll start with a brief history of CSS and swiftly move on to getting started with CSS, looking at ways to link CSS with HTML.

When we have our CSS up and running, we'll look at the structure of CSS by creating a static, single-column article page with basic media compo-

nents such as headings, content, and imagery to see how everything works together.

# .1 Overview of CSS

Håkon Wium Lie proposed the idea of CSS in 1994, a few years after Tim Berners-Lee created HTML in 1990. CSS was introduced to separate styling from the content of the web page through the options of colors, layout, and typography.

### 1.1.1 Separation of Concerns

This separation of content and presentation is based on the design principle Separation of Concerns (SoC). The idea behind this principle is that a computer program or application should be broken into individual, distinct sections segregated by purpose. The benefits of keeping good SoC include

- Decreased code duplication and, therefore, easier maintainability
- Extendibility, because it requires elements to focus on a single purpose

- Stability, because code is easier to maintain and test

With this principle in mind, HTML serves as the structure and content of a web page, CSS is the presentation, and JavaScript (JS) provides additional functionality. Together, they form the web pages. Figure 1.1 displays a diagram of this process.

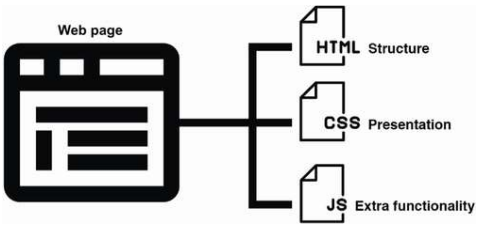


Figure 1.1 A breakdown of a web page

Since the introduction of smartphones in the mid-2000s, the web has expanded to mobile websites (often using m. subdomains, such as m.mywebsite.com), which tend to have fewer features than the desktop versions, and to responsive and adaptive designs. There are benefits and drawbacks to creating responsive/adaptive or mobile-specific websites.

The difference between responsive and adaptive designs

*Responsive design* uses a single fluid layout that can change based on factors such as screen size, orientation, and device preferences. *Adaptive design* can also change based on these factors. But instead of having a single fluid layout, we can create multiple fixed layouts, which gives us greater control of each one—at the cost of taking more time than a singular responsive layout. In practice, we can use both methods in conjunction with one another.

In general, responsive and adaptive designs are the way the industry is moving, especially as CSS expands, giving us more ability to apply CSS based on window sizes and media types (such as screen or print). Since the announcement of CSS in 1994, there have been three overall releases:

- 1996—First World Wide Web Consortium (W3C) recommendation of CSS
- 1997—First working draft of CSS2

- 1999—First three CSS3 drafts (color profiles, multi-column layouts, and paged media; https://www.w3.org/Style/CSS20)

After 1999, the release strategy was changed to allow for faster, more frequent releases of new features. Now CSS is divided into modules, with numbered levels starting at 1 and incrementing upward as features and functionality evolve and expand.

A CSS level-1 module is something that's brand new to CSS, such as a property that hasn't existed as an official standard before. Modules that have gone through a few versions—such as media queries, color, fonts, and cascading and inheritance modules—have higher-level numbers.

The benefit of breaking CSS into modules is that each part can move independently, without requiring large sweeping changes to the language as a whole. There have been some discussions about the need for someone to declare the current stage as CSS4, even if only to acknowledge that CSS has

changed a lot since 1999. This idea hasn't gained any traction so far, however.

### 1.1.2 What is CSS?

CSS is a declarative programming language: the code tells the browser what needs to be done rather than how to do it. Our code says we want a certain heading to be red, for example, and the browser determines how it's going to apply the style. This is useful because if we want to increase the line height of a paragraph to improve the reading experience, it's up to the browser to determine the layout, sizing, and formatting of that new line height, which reduces effort for the developer.

Domain-specific language

CSS is a *domain-specific language* (DSL)—a specialized language created to solve a specific problem. DSLs are generally less complex than general-purpose languages (GPLs) such as Java and C#. CSS's specific purpose is to style web content. Languages such as SQL, HTML, and XPath are also DSLs.

CSS has come a long way since 1994. Now we have ways to animate and transition elements, create motion paths to animate Scalable Vector Graphics (SVG) images, and conditionally apply styles based on viewport size. This type of functionality used to be possible only through JavaScript or Adobe Flash (now retired). We can look at CSS Zen Garden (**www.csszengarden.com**) for a glimpse of the possibilities; by looking at the first versus last designs, we can observe CSS's progression over time (**https://www.w3.org/Style/CSS20**).

In the past, design choices such as the use of transparency, rounded corners, masking, and blending were possible but required unconventional CSS techniques and hacks. As CSS evolved, properties were added to replace these hacks with standard, documented features.

CSS preprocessors

The evolution of CSS also led to the creation of CSS preprocessors and the introduction of Syntactically Awesome

Style Sheets (Sass), released in 2006. They were created to facilitate writing code that's easier to read and maintain, as well as to provide added functionality that's not available in CSS alone. We'll use a preprocessor to style a page in chapter 12.

It could be said that CSS is in a golden age. With the continual development of the language, opportunities for new and creative experiences are virtually endless.

## .2 Getting started with CSS by creating an article layout

In our first project, we'll explore a common use case on the web: creating a single-column article. This chapter focuses on how to link CSS to HTML and explores the selectors we can use to style our HTML.

The first thing we need to understand is how to tie our CSS to our HTML and how to select an element. Then we can worry about what properties and values we want to apply. Let's start by going over some basics.

If you're new to coding, you can often find free tools to use for these projects. You have the option of coding online, or you can do the work on your computer, using a code editor such as Sublime Text (**https://www.sublimetext.com**), Brackets (**https:// brackets.io**), or Visual Studio Code (**https://code.visualstudio.com**). Alternatively, you can use a basic text editor such as TextEdit for Mac (**http://mng.bz/rd9x**), Windows Notepad (**http://mng.bz/VpAN**), or gedit for Linux (**https://wiki.gnome.org/Apps/Gedit**).

The downside to using a basic text editor instead of a code editor or integrated development environment (IDE) is that it lacks syntax highlighting. This highlighting displays text in different colors and fonts according to its purpose in the code, which helps readability.

You can also use a free online development editor such as CodePen (**https://codepen.io**). Online development editors are great ways to test ideas; they provide quick, easy ac-

cess for frontend projects. CodePen provides a paid pro option that allows you to host assets such as images, which you'll need in later chapters. Another option is to link to the GitHub location where the images are stored, as all assets that are uploaded to GitHub are stored in the `raw.githubusercontent.com` domain.

When you have a code editor installed on your computer or have chosen an online editor and created an account, you'll need to get the starter code for the chapter. We created a code repository in GitHub (**https://github.com/michaelgearon/Tiny-CSS-Projects**) containing all the code you'll need to follow along with each chapter. Figure 1.2 shows a screenshot of the repository.

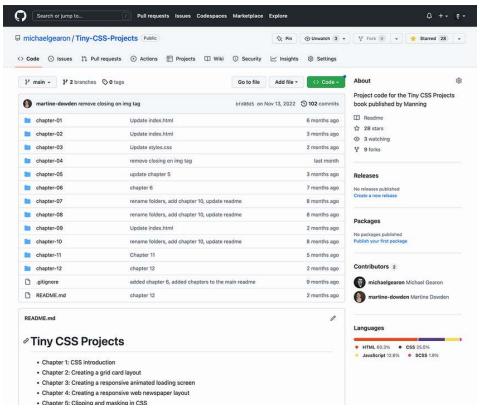


Figure 1.2 Tiny-CSS-Projects repository in GitHub

The code is organized in folders by chapter. Inside each chapter folder are two versions of the code:

- `before` —Contains the starter code for the project. You'll want this version if you're coding along with the chapter.
- `after` —Contains the completed project as it is at the end of the chapter with the presented CSS applied.

Download (or, if you're familiar with Git, clone) the project, using the Code drop-down menu at the top of the screen. If you're coding along with the chapter, grab the files from the `before` folder for chapter 1 and copy them to your project folder or pen. You should see an HTML file with some starter code and an empty CSS file. If you open the HTML file in a web browser or copy the contents of the `<body>` tag into CodePen, you'll see that the content is unstyled except for the defaults provided by your browser (figure 1.3). Now you're ready to start styling the content with CSS, as shown in listing 1.1.

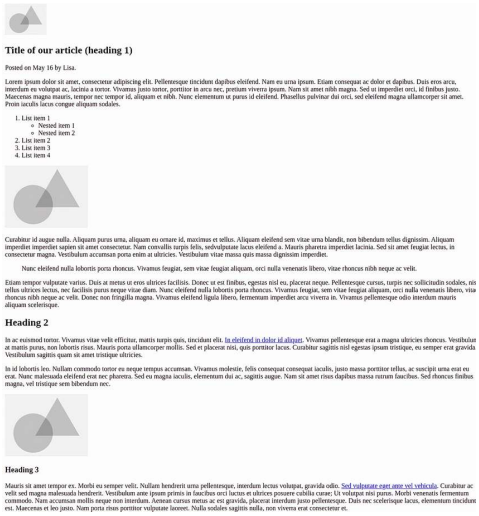


Figure 1.3 Starter HTML for our article

**NOTE** CodePen handles the information in the `<head>` tag for you automatically. Therefore, if you're following along in CodePen or a similar online editor, you need to copy only the code within the `<body>` tag.

Listing 1.1 Starting HTML

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Chapter 1 - CSS introduction</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <img src="sample-image.svg" width="100" height="75" alt="">
  <article>
    <header>
      <h1>Title of our article (heading 1)</h1>
      <p>
        Posted on
        <time datetime="2015-05-16 19:00">May 16</time>
```

```
    by Lisa.
  </p>
</header>
<p>Lorem ipsum dolor sit amet, ...</p>
<ol class="ordered-list">
  <li>List item 1
    <ul>
      <li>Nested item 1</li>
      <li>Nested item 2</li>
    </ul>
  </li>
  <li>List item 2</li>
  <li>List item 3</li>
  <li>List item 4</li>
</ol>
<img src="sample-image.svg" width="200" height="150" alt="">
<p>Curabitur id augue nulla ...</p>
<blockquote id="quote-by-author">
 Nunc eleifend nulla lobortis ...
</blockquote>
<p>Etiam tempor vulputate varius ...</p>
<h2>Heading 2</h2>
<p>
   In ac euismod tortor ...
   <a target="_blank" href="#">In eleifend in dolor id aliquet</a>
   ...
</p>
<p>In id lobortis leo ...</p>
<img src="sample-image.svg" width="200" height="150" alt="">
<h3>Heading 3</h3>
<p>
  Mauris sit amet tempor ex ...
  <a href="#">Sed vulputate eget ante vel vehicula</a>.
  Curabitur ac velit sed ...
</p>
<p>Quisque vel erat et ...</p>
<h4 class="small-heading">Heading 4</h4>
<p>Aliquam porttitor, ex ...
  <a href="#">Cras sed finibus libero</a>
  Duis lobortis, ipsum ut consectetur ...
</p>
<h2>Heading 2</h2>
<h3>Heading 3</h3>
```

```
      <svg xmlns="http:/ /www.w3.org/2000/svg" width="300" height="150">
        <circle cx="70" cy="70" r="50"></circle>
        <rect y="80" x="200" width="50" height="50" />
      </svg>
      <h4>Heading 4</h4>
      <h5 class="small-heading">Heading 5</h5>
      <p>In finibus ultrices nulla ut rhoncus ...</p>
      <h6 class="small-heading">Heading 6</h6>
      <p lang="it">Questo paragrafo è definito in italiano.</p>
      <ul class="list">
        <li>List item 1
          <ul>
            <li>Nested item 1</li>
            <li>Nested item 2</li>
          </ul>
        </li>
        <li>List item 2</li>
        <li>List item 3</li>
        <li>List item 4</li>
      </ul>
      <footer>
        <p>Footer text</p>
      </footer>
    </article>
    <p>Nam rutrum nunc at lectus ...</p>
  </body>
  </html>
```

## .3 Adding CSS to our HTML

When we're styling with CSS, we have three ways to apply CSS to our HTML:

- Inline
- Embedded
- External

### 1.3.1 Inline CSS

We can inline the CSS by adding a `style` attribute to an element. This method has us add the CSS to the element directly in the HTML.

Attributes are always specified in the opening tag and typically consist of the name of the attribute—in this case, `style`. The attribute is sometimes followed by an equal sign ( `=` ) and its value in quotes. All the CSS goes inside the opening and closing quotation marks.

As an example, let's set the color of our `heading` to `crimson`: `<h1 style="color: crimson"> Title of our article (heading 1)</h1>`. If we save our HTML and view it in a browser, we'll see that it's crimson. If we're using a code editor rather than a web client (CodePen), we need to refresh the browser page to view our changes. Figure 1.4 shows the output. Notice that the only element affected is the `<h1>` to which we applied the style.

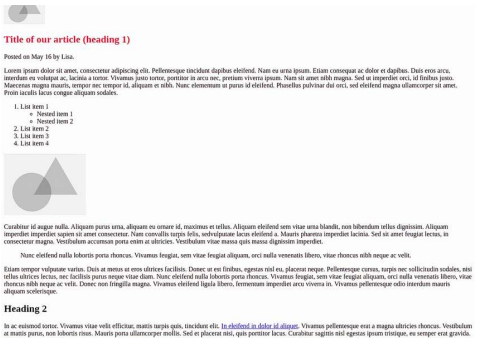


Figure 1.4 Crimson header

One downside of inline CSS is that it takes the highest specificity in CSS, which we'll look at in more detail soon.
Another major downside to inline CSS is that it can become unmanageable quickly.
Suppose that we have 20 paragraphs within an HTML document. We would need to apply the same style attributes with the same CSS properties 20 times to make sure that all our paragraphs look the same.
This case involves two problems:

- Our concerns are no longer separated. Our HTML, which is responsible for the content, and our CSS, which is responsible for styling, are now in the same place and tightly coupled.

- We're repeating the code in many places, which makes it extremely difficult to maintain and keep our styles consistent.

The benefit of inline CSS is page-load performance. The browser loads the HTML file first and then loads any other files it needs to render the page. When the CSS is already in the HTML file, the browser doesn't need to wait for it to load from a separate location. Let's undo the style we added to the `<h1>` and look at a different technique that has the same benefits as inline but fewer drawbacks.

### 1.3.2 Embedded CSS

To resolve the problem of repeating code, we can add our CSS within an embedded (sometimes referred to as internal) `<style>` element. The `<style>` element must be placed between the opening and closing `<head>` tags. To color all our heading elements crimson, we can use the snippet of code in the following listing.

Listing 1.2 Embedded CSS

```
<!DOCTYPE html>
<html lang="en">
 <head>
   ...
  <style>
   h1, h2, h3, h4, h5, h6 {
    color: crimson;
   }
  </style>
 </head>
 <body>
  ...
 </body>
</html>
```

The benefit of this approach is that now we're grouping all our CSS together, and the CSS will be applied to the whole HTML document. In our example, all headings ( `<h1>` , `<h2>` , `<h3>` , `<h4>` , `<h5>` , and `<h6>` ) within that web page will be crimson, as we can observe in figure 1.5.

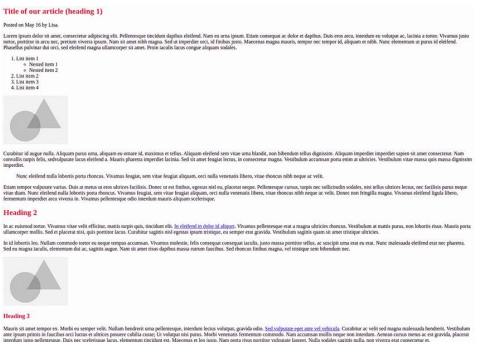


Figure 1.5 Styles applied to all headings

We also see a difference in how the embedded CSS is written compared with inline CSS.

When we're writing embedded CSS, we create what are known as *rulesets*, which are composed of the parts shown in figure 1.6.

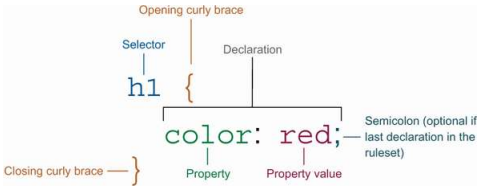


Figure 1.6 An example of a CSS rule

The part of the rule that defines which elements to apply the styles to is called the *selector*. The rule in figure 1.6 will be applied to all `<h1>` elements; its selector is `h1`.

To apply multiple selectors, we write them as a comma-delimited list before the opening curly brace. To select all `<h1>` and `<h2>` elements, for example, we would write `h1,h2 { ... }`.

The declaration is made up of the property—in this case, `color`—followed by a colon and then the property value (`red`). The declaration defines how the element selected will be styled. Both properties and values must be written in American English. Spelling variations such as *colour* and

*capitalise* aren't supported and won't be recognized by the browser. When a browser comes across invalid CSS, it ignores it. If a rule has an invalid declaration inside it, valid declarations will still be applied; only those that are invalid will be ignored.

Embedded CSS works well for one-off web pages in which the styles are specific to that page. It groups CSS nicely, allowing us to write rules that are applied across elements, preventing us from having to copy and paste the same styles in multiple places. It also has the same performance benefits as inline styles, in that the browser has immediate access to the CSS; it doesn't have to wait for the CSS to be fetched from a different location.

The downside of having our CSS within our HTML document is that the CSS will work for only that document. So if our website has multiple pages, which is often the case, we'd need to copy that CSS into each HTML document. Unless these styles are being generated by a template of backend language (such as

PHP), this task will become unmaintainable quickly, especially for large applications such as blogs and e-commerce websites. Next, let's undo the changes to our project one last time and look at a third technique.

### 1.3.3 External CSS

Like embedded CSS, the external CSS approach keeps our styles grouped together, but it places the CSS in a separate `.css` file. By separating our HTML and CSS, we can effectively separate our concerns: content and style.

We link the stylesheet to the HTML by using the `<link>` HTML tag. The link element needs two attributes for stylesheets: the `rel` attribute, which describes the relationship between the HTML document and the thing being linked to, and the `href` attribute, which stands for *hypertext reference* and indicates where to find the document that we want to include. The following listing shows how we link our stylesheet to our HTML for our project.

Listing 1.3 Applying external CSS to HTML

```
<!DOCTYPE html>
<html>
<head>
   <link rel="stylesheet" href="styles.css">
 </head>
 <body>
   <h1>Inline CSS</h1>
 </body>
</html>
```

Most of the time, this approach is the one we see across the web, so it's the approach we'll use throughout this book. The benefit of external stylesheets is that our CSS is in one single document that can be modified once to apply the changes across all of our HTML pages. The downside to this approach is that it takes an extra request from the browser to retrieve that document, losing the performance benefit provided by putting the CSS directly inside the HTML.

## .4 The cascade of CSS

One fundamental feature of CSS that we need to understand is the cascade. When CSS was created, it was developed around the concept of

*cascading,* which allows styles to overwrite or inherit from one another. This concept paved the way for multiple stylesheets that compete over the presentation of the web page.

For this reason, while inspecting an element with the browser's developer tools, we sometimes see multiple CSS values fighting to be the one rendered by the browser. The browser decides which CSS property values to apply to an element through specificity. Specificity allows the browser (or the user agent) to determine which declarations are relevant to the HTML and apply the styling to that element.

One aspect in which specificity is calculated is the order in which stylesheets are applied. When multiple stylesheets are applied, the styles in a later stylesheet will override styles provided by the preceding stylesheet. In other words, assuming that the same selector is used, the last one declared wins. CSS has three different stylesheet origins:

- User-agent stylesheets

- Author stylesheets
- User stylesheets

### 1.4.1 User-agent stylesheets

The first origin is the browser's default styles. When we opened the project, before we added any styles to it, our elements didn't all look the same. Our headers are bigger and bolder than our text, for example. This formatting is defined by *user-agent* (UA) *stylesheets*. These stylesheets have the lowest priority of the three types, and we find that different browsers present HTML properties slightly differently.

Most of the time, UA stylesheets set the font size, border styles, and some basic layout for form elements such as the text input and progress bar, which can be useful if the user stylesheet can't be found or a file-loading error occurs. The UA stylesheet provides some fallback styling, which makes the page more readable and maintains visual differentiation between element types.

### 1.4.2 Author stylesheets

The stylesheets that we developers write are known as *author stylesheets*, which typically have the second-highest priority in terms of the styles that the browser displays. When we create a web page, the CSS we write (embedded, external, or inline) and apply to our web pages consists of author stylesheets.

### 1.4.3 User stylesheets

A user who is accessing our web page can use their own stylesheet to override both author and UA styles. This option can improve their experience, especially for disabled users.

Users may use their own stylesheets for a variety of reasons, such as to set a minimum font size, choose a custom font, improve contrast, or increase the spacing between elements. Any user can apply a user stylesheet to a web page. How these stylesheets are applied to the web page depends on the browser, usually through browser settings or a plugin.

The user stylesheet is applied only for the user who added it, and only in the browser in which they applied it. Whether the change is carried over from one device to another depends on the browser itself and its ability to sync user settings and installed plugins across multiple devices.

### 1.4.4 CSS reset

Default styles provided by the browser aren't consistent. Each browser has its own stylesheet. Default styles are different in Google Chrome from the way they are in Apple's Safari, for example. This difference can create some challenges if we want our applications to look the same across all browsers.

Luckily, two options are available: CSS resets and CSS normalizers (such as Normalize.css; **https://github.com/necolas/normalize.css**). Although both can be used to solve cross-browser styling problems, they work in radically different ways.

By using a CSS reset, we undo the browser's default styles; we're telling the browser we don't want any defaults at all. Without any author styles applied, all elements, regardless of what they are, look like plain text (figure 1.7).

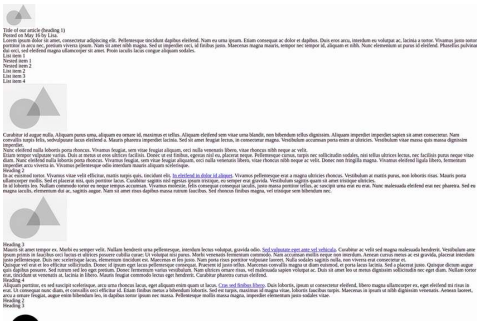

Figure 1.7 CSS reset applied

To apply a CSS reset to our project, first we create a reset stylesheet to add to our project. In our project folder, we create a file called `reset.css`. Then we copy the reset CSS into the file. Many reset options exist; one commonly used option is available at **https://meyerweb.com/eric/tools/css/reset**.

Finally, we need to link our stylesheet to our HTML. Because order matters, we want to make sure to include the reset CSS *before* our author styles in our `<head>`. Our

HTML, therefore, will look like listing 1.4.

Page-load performance

For readability, having the reset and our styles in separate files is a lot nicer than having everything in one file. This approach isn't ideal for page-load performance, however.

In a production environment, we'd want to do one of the following things:

- Place the reset CSS at the beginning of the same file we have our own styles in so that we load only one stylesheet. We could do this manually or as part of a build process.
- Load the reset code from a content delivery network (CDN) before our own styles. By loading it from a CDN, we increase the likelihood that our users will have the code already cached on their machines.

Listing 1.4 Adding a CSS reset

```
<head>
  ...
  <link rel="stylesheet" href="reset.css">      ①
```

```
    <link rel="stylesheet" href="styles.css">     ②
  </head>
```

① Resets stylesheet

② Author stylesheet

The benefit of the CSS reset is that we have a blank slate to start from. As shown in figure 1.7, all our elements look like plain text now. The downside is that we need to define basic styles for all elements, including adding bullets to lists and differentiating header levels. Furthermore, each version of CSS reset will be slightly different, based on the version and the developer who authored it.

Our other option is using a normalizer. Instead of resetting the styles, a normalizer specifically targets elements that have differences across browsers and applies rules to standardize them.

### 1.4.5 Normalizer

Like a CSS reset, a normalizer styles things slightly differently depending on the version and author. One commonly used CSS normalizer is available at

https://necolas.github.io/normalize.css.

We can apply it to our project in much the same way that we did the CSS reset code: create a file, copy the code into the file, and link it to our HTML. Note that the same performance consideration holds true here.

When the normalizer is applied (figure 1.8), our HTML looks the same as it did originally, as most of the discrepancies it handles are on elements that aren't being used in this particular project. Depending on the browser we're using, we may notice a difference in the size of the `<h1>` s.

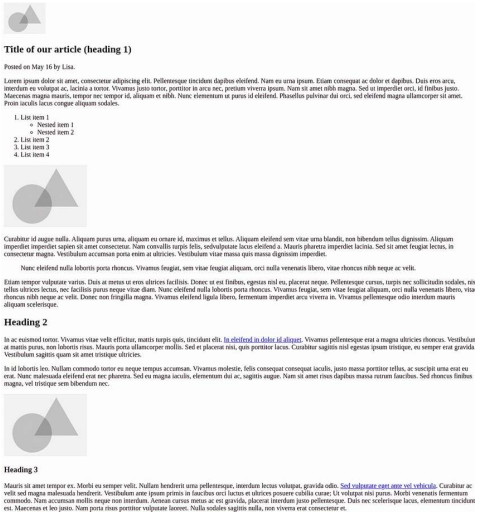


Figure 1.8 A normalizer applied to our project

The good news is that UA stylesheet differences are far less problematic than they were more than 10 years ago.

Today, browsers are more consistent in styling, so using a CSS reset or a normalizer is more a personal choice than a necessity.

Some differences still exist, however. Whether or not we use a CSS reset or a normalizer, we should be testing our code across a variety of devices and browsers.

### 1.4.6 The !important annotation

The `!important` annotation is one you may have seen in some stylesheets. Often used as a last resort when all else fails, it's a way to override the specificity and declare that a particular value is the most important thing. With great power, however, comes great responsibility. The `!important` annotation was originally created as an accessibility feature.

Remember that we talked about users being able to apply their own styles to have a better user experience? This annotation was created to help users define their own styles without having to worry

about specificity. Because it overrides any other styles, it ensures that a user's styles always have the highest importance and therefore are the ones applied.

Using `!important` is considered to be bad practice, so we should generally avoid using it in our author stylesheets. Also, this annotation breaks the natural cascade of the CSS and can make it harder to manage the stylesheet going forward.

## .5 Specificity in CSS

When multiple property values are being applied to an element, one will win over the others. We determine the winner through a multistep process. We'll ignore `!important` (section 1.4.6) for the time being, as it breaks the normal flow; we'll come back to it later.

First, we look at where the value comes from. Anything explicitly defined in a rule will override inherited values. In listings 1.5 and 1.6, for example, if we set the font color to `red` on the `<body>` element,

the elements inside `<body>` will have red text.

The font color is inherited by child elements. If we specifically set a different color on a paragraph inside the body, the inherited `red` value would be overridden by the more specific `blue` value set on the paragraph. Therefore, that paragraph's text color would be blue.

Listing 1.5 Example of inheritance (HTML)

```
<body>
  <h1>Example</h1>          ①
  <p>My paragraph</p>       ②
</body>
```

① Our header would inherit the red color.

② The paragraph's color would be blue, as set by the paragraph rule.

Listing 1.6 Example of inheritance (CSS)

```
body { color: red }
p { color: blue }
```

Not all property values will be inherited. Theme-related styles such as color and font size will generally be inherited; layout considerations generally are not. This guideline is a loose one, with definite exceptions, but it's a good place to start. We'll cover exceptions on a case-by-case basis throughout the projects.

If the property value isn't being inherited, the browser looks at the type of selector that was used and mathematically calculates the specificity value. We'll get into more detail about what each selector type is in section 1.6, but first let's look at how the math is applied.

The browser looks at the selector, categorizes the types of selectors being used by the rule, and applies the type value. Then it adds all the values and gets a final specificity value. Figure 1.9 diagrams the process. The biggest number wins, so rule 1 in the diagram would win over rule 2.

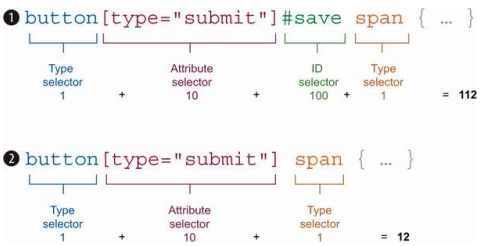


Figure 1.9 Calculating specificity

Specificity values by selector type are as follows:

- `100` —ID selectors
- `10` —Class selectors, attribute selectors, and pseudo-classes
- `1` —Type selectors and pseudo-elements
- `0` —Universal selectors

If we still have a tie, the browser looks at which stylesheet the style originated from. If both values come from the same stylesheet, the one later in the document wins. If the values come from different stylesheets, the order is as follows:

1. User stylesheet
2. Author stylesheets (in the order in which they're being imported; the last one wins)
3. UA stylesheet

We set `!important` to the side earlier. Now that we under-

stand the normal flow, let's add it back into the mix. When a value has the `!important` annotation, the process is short-circuited, and the value with the annotation automatically wins.

If both values have the `!important` annotation, the browser follows the normal flow. Figure 1.10 shows the flow through the stylesheets, including `!important` declarations.

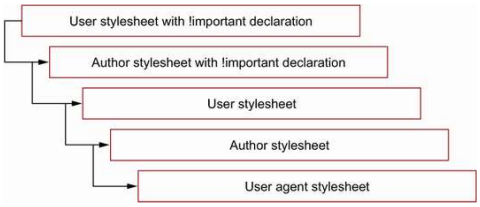


Figure 1.10 CSS order of precedence

We've established that the type of selector will affect specificity. Let's take a closer look at the selectors and use them in our project.

## .6 CSS selectors

The selector sets what HTML elements we want to target. In CSS, we have seven ways to target the HTML elements we want to style, as discussed in the following sections.

### 1.6.1 Basic selectors

The most common method of applying styles to HTML elements is selecting them based on name, ID, or class name. These are used most often because of their one-to-one mapping to the HTML element itself or attributes set on the element.

#### Type selector

The *type selector* targets the HTML element by name. The benefit of using the type selector is that when we read through our CSS, we can quickly work out which HTML elements would be affected if we made changes in the rule. This selector doesn't require us to add any particular markup to the HTML to target the element.

Let's use a type selector to target all our headings ( `<h1>` through `<h6>` ) and change their color to crimson. Our CSS would be `h1, h2, h3, h4, h5, h6 { color: crimson; }` . Figure 1.11 shows that our headers have changed colors.

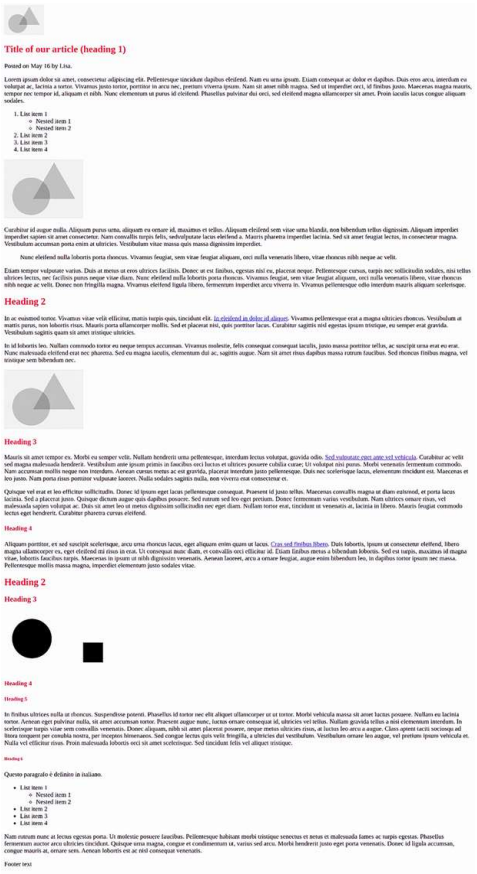


Figure 1.11 Header color change

## Class selectors

We can use class selectors on as many different elements as we want. By applying a class name to elements, we're grouping multiple HTML elements so that when we apply styles, they'll roll out to any element with that class name.

To add classes to HTML, we use the `class` attribute. Within the `class` attribute, we can add as many values (or classes) as we want, separating each with a space.

We have many ways and methods to write our class names, such as Block, Element, Modifier (BEM) methodology (**https://en.bem.info**) and Scalable and Modular Architecture for CSS (SMACSS; **http://smacss.com**), which are style guides for writing consistent stylesheets.

The main point is to write class names that make sense to everyone. Adding the class name `text` to paragraph elements, for example, would be highly confusing. Other elements, such as our headings, can also be thought of as text, so it may not be clear which specific element we're referring to.

Applying class names based on a specific style, such as a `color`, can also be dangerous. Giving an element the class name `blue` might work immediately, but if the design changes and the color applied is now red, our class name will no longer make sense.

In our HTML, we find that some of our headings have a class of `small-heading`. We're going to create a rule that se-

lects `small-heading` and changes the text of the elements to uppercase.

To select the `small-heading` class name, in the CSS we first type dot ( `.` ) followed by the class name `small-heading` . Then our styles go into curly braces as follows: `.small-heading { text-transform: uppercase }` . Figure 1.12 shows our uppercased headings. Notice that the other headings aren't affected—only those to which the class was applied.

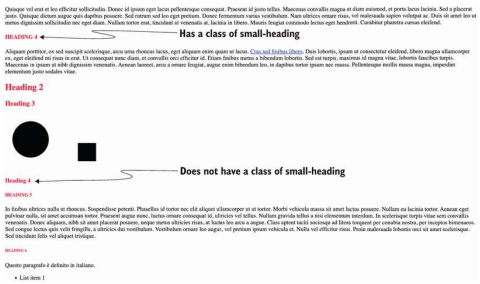


Figure 1.12 Class selector applied to elements that have the class name `small-heading`

### ID SELECTOR

In HTML, IDs are unique. Any given ID should be used only once on a web page. If an ID is repeated, our code is considered to be invalid HTML.

Generally we should avoid using ID selectors; because they

need to be unique in the HTML, rules constructed against the ID aren't easy to reuse. Furthermore, an ID selector is one of the most specific selectors available, making the styles applied with an ID selector difficult to override. Unless the uniqueness of the element is key, avoid using ID attributes.

Our example article contains a `blockquote` with an `ID` attribute containing the value `quote-by-author`. In our CSS, to select the `blockquote` we use a hash ( `#` ), followed immediately by the `ID` we want to target. Then we have curly braces, inside which we place our declarations, as shown in the following listing.

Listing 1.7 ID selector

```
#quote-by-author {
  background: lightgrey;
  padding: 10px;
  line-height: 1.75;
}
```

Figure 1.13 shows the code applied to our project.

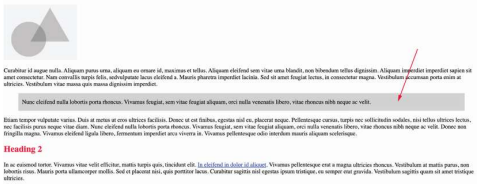

Figure 1.13 Styles applied by `#quote-by-author`

## 1.6.2 Combinators

Another way to write CSS is through *combinators*, which allow for more complex CSS without overusing `class` or `ID` names. There are four combinators:

- Descendant combinator ( `space` )
- Child combinator ( > )
- Adjacent sibling combinator ( + )
- General sibling combinator ( ~ )

One important concept to understand is the relationships between elements. In the next couple of examples, we'll look at how we can use the relationships between elements to target different HTML elements to style our article. Figure 1.14 introduces the types of relationships we're going to examine.

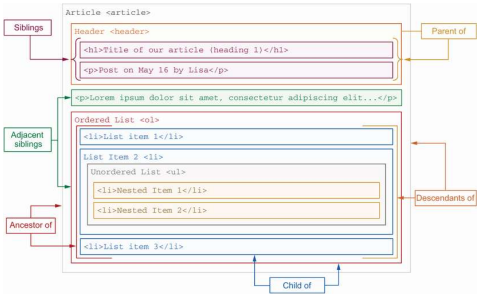


Figure 1.14 The relationships between elements in HTML

### Descendant combinator (space)

Selectors that use descendant combinators select all the HTML elements within a parent. A selector that uses a descendant combinator is made up of three parts. The first part is the parent, which in this case is the article element. The parent is followed by a space and then by any element we want to select. Figure 1.15 diagrams the syntax.

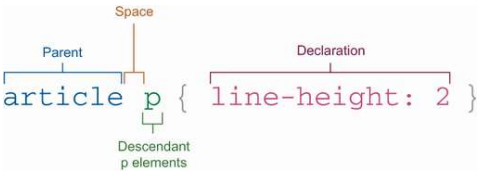


Figure 1.15 An example of a selector using a descendant combinator

In this example, the browser would find any `<article>` element, target all descendant paragraphs ( `<p>` ) in the parent `<article>` element, and make the text double-spaced. When this selector is applied,

our article looks like figure 1.16.

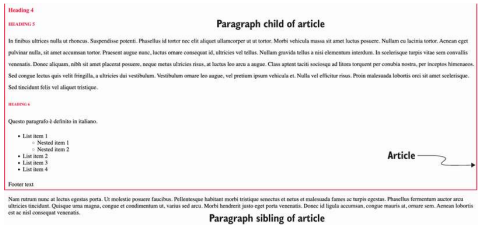


Figure 1.16 Child paragraphs are double-spaced

### Child combinator (>)

The child combinator allows us to target the immediate child elements of a particular selector. This combinator is different from a selector that uses a descendant combinator because in the case of a child combinator, the targeted element must be an immediate child. A selector that uses a descendant combinator can select any descendent (child, grandchild, great-grandchild, and so on).

In our project, we'll style the list items in the article. As listing 1.8 shows, we have an unordered list ( `<ul>` ) with list items ( `<li>` ). That first child element has its own nested items, which would be grandchildren and great-grandchildren.

Listing 1.8 HTML list items

```
<ul class="list">                    ①
  <li>List item 1                    ②
    <ul>                             ③
      <li>Nested item 1</li>         ④
      <li>Nested item 2</li>         ④
    </ul>
  </li>
  <li>List item 2</li>               ②
  <li>List item 3</li>               ②
  <li>List item 4</li>               ②
 </ul>                               ①
```

① Parent item (.list)

② Children of .list

③ Grandchild of .list

④ Great-grandchildren of .list

We're going to style only the first-level list items—or immediate children of the `<ul>` with a `class` attribute value containing `list`—in a crimson color, without affecting the nested list items (the great-grandchildren). So the browser will find elements containing the `list` class, target only their immediate children that are list items ( `<li>` ), and change `color` to `crimson` . We'll use the following CSS:

```
.list > li { color: crimson; }
```

With this CSS, the entire list becomes crimson, not just the top-level list items. The color is applied to the `<li>` element and all of its descendants. Even though we select the immediate child, because color is inherited, the children also turn crimson.

To select only the top-level elements, we therefore need to add a second rule

```
.list > li ul { color: initial }
```

which returns the nested list items to their initial color, as shown in figure 1.17.


Figure 1.17 Child combinator applied to `list` items

We can perform this operation in reverse and select the parent of the child element, right? The short answer is no, as the following example wouldn't work: `article < p { color: blue; }`. If we want to select the parent or ances-

tor of an element, we need to use the `has()` pseudo-class—`article:has(p) { color: blue; }`—covered in section 1.6.3.

### Adjacent sibling combinator (+)

When we need to style an element that's at the same level as another, the way your brother or sister is on the same level of the family tree as you, we can use the adjacent sibling combinator. If we want to target the element that's directly after another, we can use a selector that uses an adjacent sibling combinator.

In listing 1.9, the browser will find any uses of the `<header>` element, target the first paragraph ( `<p>` ) immediately after (or adjacent to) the `<header>` element, and change the `font-size` to `1.5rem` and the `font-weight` to `bold`. Figure 1.18 shows the code applied to our article.

Listing 1.9 Adjacent sibling combinator

```
header + p {
  font-size: 1.25rem;
```

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

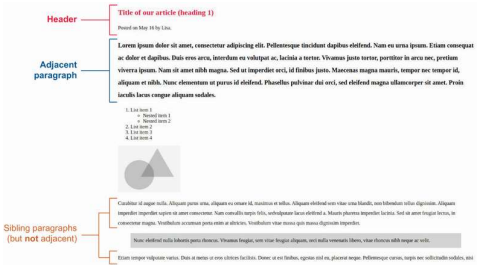


Figure 1.18 Styling the paragraph immediately after the header

This approach could be useful if we're trying to style the first element differently from the others to make it stand out. We might see this effect in a newspaper. The first paragraph of an article might be made to look more prominent than the rest to catch our attention.

Another use case is for error handling in forms. Adjacent sibling combinators allow us to display an error message to the user immediately following an invalid value in a form control.

### General sibling combinator (~)

The general sibling combinator is more open-ended than the other methods, as it allows us to target all elements that

are siblings after the element targeted by the selector.

In our example, we'll style all images that come after the element `<header>`. Notice that we have three placeholder images. The first image is small (it could be a logo or an author photo) and resides above the `<header>`. We don't want to style it. The other two images are farther down in the article. We want to apply a border around them to keep the color theme consistent with the rest of the article.

Our rule will be as follows: `header ~ img { border: 4px solid crimson; }`. The browser will find the `<header>` element; target all the sibling images (`<img>`) after that element; and add a `border` that's `4px` in thickness, that's a `solid` line (as opposed to a `dotted`, `dashed`, or `double` line), and that's colored `crimson`. We can see the code applied to our article in figure 1.19.

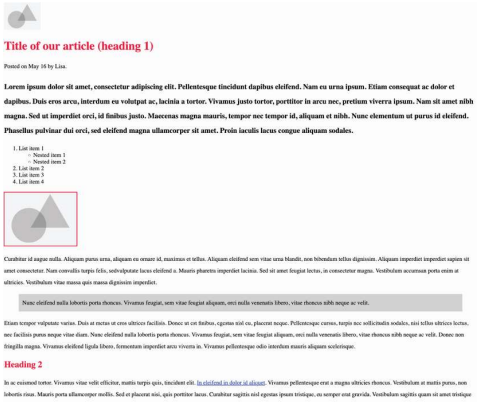


Figure 1.19 General sibling combinator targeting sibling images of header

### 1.6.3 Pseudo-class and pseudo-element selectors

CSS has selectors called *pseudo-classes* and *pseudo-elements*. You may wonder where the names come from. *Pseudo* means “not genuine, false, or pretend.” This definition makes sense because, technically, we’re targeting a state or parts of an element that may not exist yet. We’re simply pretending.

Not all pseudo-elements and pseudo-classes work on all HTML elements. Throughout this book, we’ll look at where we can use pseudo-classes and with which HTML elements.

#### Pseudo-class

A *pseudo-class* is added to a selector to target a specific state of the element. Pseudo-classes

are especially useful for elements that the user will interact with, such as links, buttons, and form fields. Pseudo-classes use a single colon (`:`) followed by the state of the element.

Our article contains a few links. We haven't styled the links in any way; therefore, their styles will come from the UA stylesheet. Most browsers underline links and display them in a color based on whether the link was previously visited—that is, whether the URL appears in the browser's history.

With links, we have a few states to consider. The most common are

- `link` —An anchor tag (`<a>`) contains an `href` attribute and a URL that doesn't appear in the user's browser history.
- `visited` —An anchor (`<a>`) element contains an `href` attribute and a URL that does appear in the user's browser history.
- `hover` —The user has the cursor over the element but hasn't clicked it.

- `active` —The user is clicking and holding the element.
- `focus` —A *focused element* is an element that receives keyboard events by default. When a user clicks a focusable element, it automatically gains focus (unless some JavaScript alters this behavior). Using the keyboard to navigate among form fields, links, and buttons also changes the element that is in focus.
- `focus-within` —When `focus-within` is applied to a parent element and the child of the parent has focus, `focus-within` styles will be applied.
- `focus-visible` —When elements are selected using `focus-visible`, styles are applied only when focus has been gained via keyboard navigation or the user is interacting with the element via the keyboard.

We mentioned `:has()` earlier. Also a pseudo-class but not specific to links, `:has()` applies when the element has at least one descendent that meets the selector specified inside the parentheses. When

we wrote this book, `:has()` had not yet been implemented in all major browsers.

In our current article project, we'll create an `a:link` rule to change the color of anchor tags that contain an `href` attribute and haven't been visited to light blue, using the hex color code `#1D70B8`. The `:visited` state should be different from the `:link` state because it should indicate to the user that they haven't visited that page before (that is, the URL isn't present in their browser history). Often, websites don't differentiate between the two states, but discerning them can provide a better user experience. In our example, we'll change the `:visited` state to a purple `color`, using the hex code value `#4C2C92`.

Then we'll handle the `:hover` state. This state doesn't apply to mobile users, as there's no way to recognize a user hovering over a link on a mobile device. In our article, we'll change the `:hover` state text color to a dark blue, using the hex code value `#003078`.

Finally, we'll handle the `:focus` state. We can use this state on any focusable elements. Links, buttons, and form fields are focusable by default (unless disabled), but we can make any element focusable by using a positive-numbered `tabindex`, in which case focus-based styles could be applied. The `:focus` state is shown when the user clicks or taps an element. When the element is focused, we add a 1-pixel crimson outline to the element. All put together, our link rules appear as shown in the following listing.

Listing 1.10 Styling links using pseudo-elements

```
a:link {
  color: #1D70B8;
}
a:visited {
  color: #4C2C92;
}
a:hover {
  color: #003078;
}
a:focus {
 outline: solid 1px crimson;
}
```

Note that the order in which these rulesets are written mat-

ters, as they have the same level of specificity. The condition that's farthest down in the stylesheet will win if multiple conditions apply. In our example, if a link has been visited but is being hovered over, the link will take the color assigned to it by the `a:hover {}` rule because it comes after the `a:visited {}` rule in our stylesheet.

Although developer tools vary in features and how those features are accessed, in most browsers, we can view the different element states by going into our browser, right-clicking, and choosing Inspect from the contextual menu. Typically, we get a view of the HTML with the CSS on the side. By clicking the `:hov` button in the Styles section, we see a panel that may say something like *force element state*, and then we can toggle different pseudo-classes on and off. Figure 1.20 shows the Chrome developer tools with the `:hov` panel open.

Developer tools in browsers

All major browsers have developer tools that allow devel-

opers to modify, debug, and optimize websites. For this book, we will use developer tools to examine our code. We will also examine the compiled code in the browser tools to understand how the browser is processing our CSS. For more information about developer tools and how to access them, see the appendix.

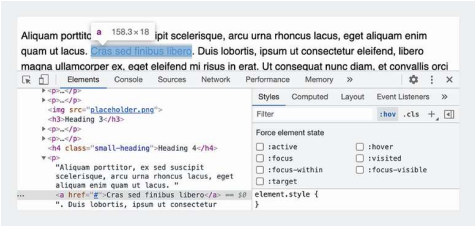


Figure 1.20 Viewing different element states by using the browser's developer tools

### Pseudo-element

*Pseudo-elements* use a double colon ( `::` ). The purpose of pseudo-elements is to allow us to style a specific part of an element. Sometimes, pseudo-elements are written with a single colon, although using two is strongly recommended. The ability to ignore the second colon is for backward compatibility; the two-colon syntax was introduced as part of CSS3 to better differentiate between pseudo-classes and pseudo-elements.

Using the `::first-letter` pseudo-element, we can target the first letter of a paragraph rather than wrap the letter in something like a `span` element, which would break the word apart and clutter our HTML. This approach allows us to create complex CSS without complicating the HTML.

In our article, we used the adjacent sibling combinator to make our first paragraph bold and in a larger font size than the rest. Now we're going to change the color of the first letter of that first paragraph and change the font style to `italic`.

First, we target the `header` element; then, we target the first letter ( `::first-letter` ) of the paragraph ( `<p>` ). With our selector created, we add our declarations. Our CSS will look like the following listing.

Listing 1.11 Selecting the first letter

```
header + p::first-letter {
  color: crimson;
  font-style: italic;
}
```

When this code is applied, the first letter is red and italicized (figure 1.21).

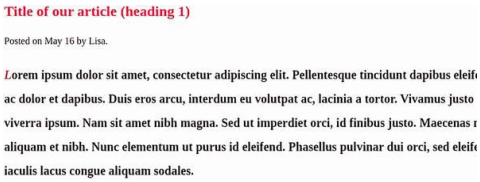


Figure 1.21 Pseudo-element targeting the first letter of the first paragraph immediately after the header

### 1.6.4 Attribute value selectors

Commonly used for styling links and form elements, the attribute selector styles HTML elements that include a specified attribute. The attribute value selector looks for a specific attribute with the same value.

In our article, we have some content in Italian. The language of the paragraph is specified by the `lang` attribute, as shown in the following listing.

Listing 1.12 Specifying Italian content

```
<p lang="it">Questo paragrafo è definito in italiano.</p>
```

To hint to users that this content is in Italian, we'll use CSS to add the Italian-flag emoji.

The browser will find the language ( `lang` ) attribute with the value of Italian ( `it` ) and then add an Italian-flag emoji `before` it. Listing 1.13 uses a `::before` pseudo-element as well. We can use multiple types of selectors to target the exact part of the HTML we want to style.

Listing 1.13 Using multiple types of selectors to add a flag before Italian content

```
[lang="it"]::before {
  content: "🇮🇹"
}
```

When this code is applied, our Italian content has an emoji flag before it (figure 1.22).

Emoji differences across devices and applications

If you're coding along with this chapter, your output may differ from figure 1.22. Emojis present differently depending on the device, operating system, and application being used. Sites such as Emojipedia (**https://emojipedia.org**) show how a particular emoji would look across applications and devices. You can find details

on the Italian flag at **https://emojipedia.org/flag-italy**.

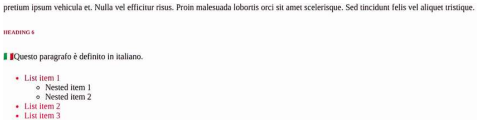


Figure 1.22 Italian flag applied by the attribute selector and a pseudo-element

### 1.6.5 Universal selector

The broadest type of selector is the universal selector, which uses the asterisk symbol ( `*` ). Any declarations made with the universal selector will be applied to all the HTML elements.

Sometimes, this selector can be used to reset CSS, but in terms of specificity, it has a specificity value of `0`, which means that it can be overridden easily if necessary. This is important because it targets every element. The universal selector can also be used to target any and all descendants of a particular selector, as in `.foo * { background: yellow; }`, in which any and all descendants of an element with the class `foo` would be given a yellow background.

In our example project, we'll use a universal selector ( `*` ) to set the `font-family` to `sans-serif` so that the font will be sans-serif consistently throughout the article, as shown in the following listing.

Listing 1.14 Making our `font-family` consistent

```
* { font-family: sans-serif; }
```

When this code is applied, all the text in our document uses a sans-serif font regardless of element type (figure 1.23).

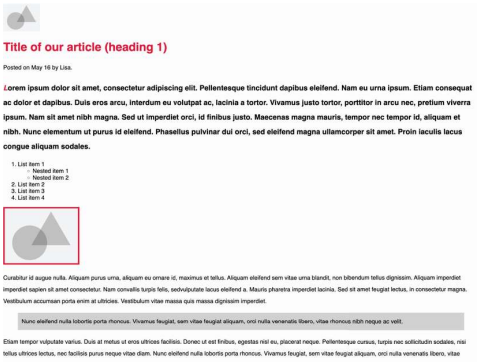


Figure 1.23 Using the universal selector to change the font type on all elements

## .7 Different ways to write CSS

CSS allows flexibility in the way we write our rules and formatting. In this section, we'll look at shorthand properties (which we will keep

coming back to throughout the book) and ways to format CSS.

### 1.7.1 Shorthand

*Shorthand* replaces writing multiple CSS properties with merging all the values into one property. We can do this with a few properties such as padding, margin, and animation, all of which are covered at various points throughout this book. The benefit of writing shorthand is that it reduces the size of our stylesheet, which improves readability, performance, and memory use.

Each shorthand property takes different values. Let's explore the one we used in our project. We have a `blockquote` in our article. When we styled it, we used the `padding` property and declared our padding as follows: `padding: 10px`. In doing so, we used shorthand. Instead, we could have written the code as shown in the following listing.

Listing 1.15 Padding expanded

```
padding-top: 10px;
padding-right: 10px;
```

```
    padding-bottom: 10px;
    padding-left: 10px;
```

It's completely fine to write each declaration separately, but doing that is expensive in terms of computing performance, especially because all the property values are the same. Instead, we can use the `padding` property and put all four values on the same line. The order is `top`, `right`, `bottom`, and `left`. We can also combine `right` / `left` and `top` / `bottom` values if they're identical, as depicted in figure 1.24.

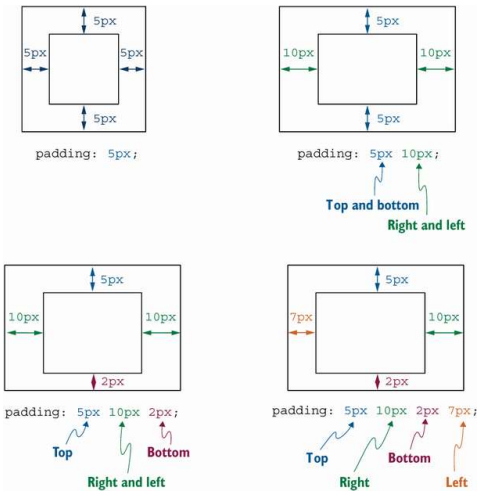


Figure 1.24 Padding shorthand property explained

As shown in the figure, we can declare all four values to define the `top`, `right`, `bottom`, and `left` values. But if we say that `right` and `left` are the same and `top` and `bottom`

are different, we can specify three values, in the order `top`, `right & left`, `bottom`.

If two values are declared, we're saying the first value is what the `top` and `bottom` should be; then the second value sets the `right` and `left`. Finally, if only one value is declared, the value sets all four sides.

### 1.7.2 Formatting

We can write CSS in a few ways, and often when we view other people's code, we see different formats. This section shows a few examples.

The multiline format shown in listing 1.16 is likely the most popular choice for formatting. Each declaration is on its own line and indented by means of tabs or spaces.

Listing 1.16 Multiline format

```
h1 {
  color: red;
  font-size: 16px;
  font-family: sans-serif
}
```

A variation on the multiline format, shown in listing 1.17,

places the opening curly brace on its own line. This example is something we might see in the PHP language. It could be considered unnecessary to place the opening brace on its own line.

Listing 1.17 A variation on multiline format

```
h1
{
  color: red;
  font-size: 16px;
  font-family: sans-serif
}
```

The single-line format shown in listing 1.18 makes a lot of sense; it's compact, and we can scan a file knowing that the first part is the selector. The downside is that it can be difficult to read if a rule contains many declarations.

Listing 1.18 Single-line format

```
h1 { color: red; font-size: 16px; font-family: sans-serif }
```

All these options have positives and negatives, but the projects in this book use a combination of options one and three. The main thing to know is that there's no right or

wrong method; the choice generally comes down to what works best for you and/or your team. As long as the code is easy to understand, that's all that matters.

Those with an eagle eye will notice that in listings 1.16, 1.17, and 1.18, there's no semicolon ( `;` ) at the end of the last declaration of the rules. This semicolon is optional. One of the best aspects of CSS is that we can write it in the way that's most comfortable for us.

## ummary

- CSS is a well-established coding language, and each part of CSS is made up of modules.
- Modules replaced large releases such as CSS3.
- Inline CSS can take the highest priority and has good performance, but it's repetitive and hard to maintain.
- External CSS keeps our CSS separate from our HTML, maintaining SoC.
- Along with our own CSS, the browser applies default styling.

- The user may also apply their own CSS, which can override the author and UA stylesheets.
- Using `!important` is considered to be bad practice.
- A CSS rule consists of a selector and one or more declarations.
- We can create rules for many types of selectors, and each rule can have its own level of specificity.