Pro CSS Techniques

Jeff Croft, Ian Lloyd, and Dan Rubin

Apress®

Pro CSS Techniques

Copyright © 2006 by Jeff Croft, Ian Lloyd, and Dan Rubin

ISBN-13 (pbk): 978-1-59059-732-3

ISBN-10 (pbk): 1-59059-732-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/ Download section.

CHAPTER 2

The Language of Style Sheets

OK, now we've had a look at modern web design, and how CSS fits into that, let's turn our attention to recapping the basics of CSS to make sure we are all on the same page before we start immersing ourselves in all the exciting techniques and productivity-improving tips that make up the rest of the book. Even if you think you have a good handle on the basics of CSS, at least skim through the chapter anyway. Specifically we cover the following:

- Adding style to your document
- Creating a style sheet and declarations
- Choosing selectors to apply styles to your markup
- Understanding XHTML and how it can be represented as an element node tree
- Daisy-chaining and grouping selectors

Adding Style to Your (X)HTML Document

In this section we'll explore the various ways in which you can add CSS styles to your (X)HTML document. We'll cover several approaches, and offer some best practices for making your markup and presentation information work well together.

The <link> Tag

There are a few ways to add CSS style information to an (X)HTML document, but it's almost always a best practice to use the `<link>` tag. The tag's purpose is simple: to associate one document with another. For our purposes, let's use it to associate a CSS document with an XHTML document, like so:

```
<link rel="stylesheet" type="text/css" href="styles.css" media="all" />
```

This simply tells the XHTML document to use the file `styles.css` as a source of CSS information. We call CSS documents referenced in this way *external style sheets*. Link tags that reference external style sheets must be placed inside the `head` element of your (X)HTML document (but not inside any other element).

The attributes of the `<link>` tag are fairly straightforward. The `rel`, or relation, attribute, describes the sort of relationship the linked file has to the (X)HTML file calling it. For CSS style sheets, its value will always be `stylesheet`. The `type` attribute defines the file type of the linked file, and its value should always be `text/css` for CSS style sheets. As you might expect, the `href` attribute declares the URL of the style sheet file you'd like to attach to your document. This URL may be absolute or relative. The final attribute, `media`, states for which presentation media the styles contained in the external style sheet should be applied. As Table 2-1 shows, several possible media types are defined in CSS—and they have varying degrees of browser support (more on this in Chapter 13).

Table 2-1. *Available CSS Media Types*

Column Heading 1	Column Heading 2
`all`	Applies styles for all media types
`aural`	Applies styles when listening to the document with a screen reader or similar audio-rendering device
`braille`	Applies styles when presenting the document with a Braille device
`embossed`	Applies styles when printing the document with a Braille device
`handheld`	Applies styles when viewing the document with a handheld personal device, such as a cell phone or PDA
`print`	Applies styles when printing the document (or when displaying a print preview)
`projection`	Applies styles when using a projection medium, such as a digital projector
`screen`	Applies styles when presenting the document on a screen, such as that on a typical desktop computer
`tty`	Applies styles when displaying the document in a fixed-character width setting, such as Teletype printers
`tv`	Applies styles when the document is being presented on a television-style screen

It's up to each viewing device that supports CSS—be it a desktop computer's web browser, a cell phone, a PDA, or a web TV device—to render the styles appropriate for the context in which it's being used. Desktop browsers, for example, know to use styles destined for the "screen" media type when they display web sites normally but to use those defined for the "print" type when you are printing; cell phones and PDAs know they should use styles for the "handheld" type; and so on.

You can use a single style sheet in multiple media by setting the value of the `media` attribute to a list, separated by commas. For example, to use the style sheet named `styles.css` in both screen and projection media, you'd use the following:

```
<link rel="stylesheet" type="text/css" href="styles.css" ➥
media="screen, projection" />
```

We'll cover using CSS media types in much greater detail in Chapter 13.

Using Multiple Style Sheets

You can also attach multiple style sheets to a single (X)HTML document with the `<link>` tag. The browser will use all linked style sheets and render the page accordingly.

Also, there exists the concept of an *alternate style sheet*. In some browsers (notably those powered by the Gecko rendering engine, such as Mozilla, Firefox, and recent versions of Netscape), these alternate visual presentations of your document will appear in a menu for the visitor to select between. For example:

```
<link rel="stylesheet" type="text/css" href="styles.css" ➥
media="all" title="Default" />
<link rel="alternate stylesheet" type="text/css" ➥
href="low_vision_styles.css" media="all" title="Low vision" />
```

Here, we've defined references to two different style sheets. Because the second one has the value `alternate stylesheet` for the `rel` attribute, it will not be used when the page is first presented but will be offered to the user as a choice (only in those browsers that support it, though). Note the use of the `title` attribute to give each style sheet a name (which will appear in the visitor's menu of choices). Alternate style sheets are not widely used, probably because of their limited browser support.

It's important to note that any `<link>` tag with both a `title` attribute and a `rel` value of `stylesheet` will be considered a preferred style sheet. Only one preferred style sheet will be used when the page is initially loaded. Therefore, it's important that you don't assign multiple `<link>` tags both a `title` attribute and a `rel` attribute value of `stylesheet`. `<link>` tags *without* a `title` attribute are designated as persistent style sheets— which means they'll *always* be used in the display of the document. More often than not, this is the behavior you'll want. Because of this, you should only apply a `title` attribute to your `<link>` tags if you are referring to an alternate style sheet.

The style Element

The `style` element is an (X)HTML element that allows you to embed CSS style information directly in the page you're working on, rather than abstracting it out to an external style sheet. Although this may sound like a convenience at first, it's almost always a hindrance in the real world. One of the major benefits of CSS is the ability to have all of your style information in a single external style sheet that many pages refer to. That way, when you want to change something, you only have to change it one time. In the case of embedded style sheets created with the `style` element, the style information applies only to the (X)HTML document you're working with. It's possible this behavior is what you want, but more often than not it's beneficial to have the style information abstracted.

If you do want to embed a style sheet, use code like this in the `head` element of your (X)HTML document:

```
<style type="text/css" media="screen, print ">
...
</style>
```

The practical applications of embedded style sheets are few. If you are making a simple, one-page site, it may be more convenient to keep your style information embedded in your (X)HTML document. You may also have one page within a larger site that needs additional or overriding styles along with the styles applied to the rest of the site. In this case, you can get by with an embedded style sheet, but it still may be a best practice to abstract the styles into their own file. You never know when you may need to add another page utilizing those styles, or reuse them in another section of the site. Having them in their own file provides this additional flexibility.

Creating a Style Sheet

CSS syntax is quite simple: we're dealing with nothing more than a list of rules. A simple style sheet might look something like this:

```
h1 {
  color: blue;
}
h2 {
  color: green;
}
```

Save those two lines into a text file and give it a name ending in `.css`, and you've got yourself a perfectly valid (albeit simple) external style sheet. Put those two lines in a `<style>` element within the head of your (X)HTML document, and you've made an embedded style sheet.

Each style rule is made up of two parts: a *selector* and one or more *declarations*—each of which consists of a property and a value.

Declarations

Let's go about this in reverse order. Declarations are property/value pairs that define visual styles. *Properties* are things like background color, width, and font family. *Values* are their counterparts, such as white, 400 pixels, and Arial—or, in the proper syntax:

```
background-color: white;
width: 400px;
font-family: Arial;
```

Declarations are always formatted as the property name, followed by a colon, followed by a value, and then a semicolon. It is common convention to put a space after the colon, but this is not necessary. The semicolon is an indication that the declaration is concluded. Declarations are grouped within curly brackets, and the wrapped group is called a *declaration block*.

Selectors

Selectors define which part(s) of your (X)HTML document will be affected by the declarations you've specified. Several types of selectors are available in CSS. Note that some of them are not supported in all browsers, as you will learn in Chapter 4.

Element Selectors

The most basic of all selectors is the *element selector* (you may have heard them called *tag selectors*). It is simply the name of an (X)HTML element, and—not surprisingly—it selects all of those elements in the document. Let's look again at the previous example:

```
h1 {
  color: blue;
}
h2 {
  color: green;
}
```

We've used `h1` and `h2` as selectors. These are element selectors that select `h1` and `h2` elements within the (X)HTML document, respectively. Each rule indicates that the declarations in the declaration block should be applied to the selected element. So, in the previous example, all `h1` elements in the page would be blue and all `h2` elements would be green. Simple enough, right?

■**Note** Although this book is about using CSS to style (X)HTML documents, CSS can be used for other types of documents as well (notably XML). Therefore, it's entirely possible that you will run across element selectors that are not valid (X)HTML elements.

Class Selectors

So far we've been assigning styles exclusively to (X)HTML elements, using element selectors. But there are several other types of selectors, and the *class* and *ID selectors* may be next in line as far as usefulness. Modern markup (as discussed in Chapter 1) often involves the assigning of classes and IDs to elements. Consider the following:

```
<h1 class="warning">Be careful!</h1>
<p class="warning">Every 108 minutes, the button ➥
must be pushed. Do not attempt to use the computer ➥
for anything other than pushing the button.</p>
```

Here, we've specified a `class` of `warning` to both the `h1` element and the `p` (paragraph) element. This gives us a hook on which we can hang styles that is independent of the element type. In CSS, class selectors are indicated by a class name preceded by a period (`.`); for example:

```
.warning {
  color: red;
  font-weight: bold;
}
```

This CSS will apply the styles noted in the declaration (a red, bold font) to all elements that have the class name `warning`. In our markup, both the `h1` and the `p` elements would become red and bold. We can join an element selector with a class selector like this:

```
p.warning {
  color: red;
  font-weight: bold;
}
```

This rule will assign the red color and bold weight *only to paragraph elements* that have been assigned the class `warning`. It will not apply to other type elements, even if they have the `warning` class assigned. So, the `h1` in our previous markup would be ignored by this style rule, and it would not become red and bold. You can use these rules in combination to save yourself some typing. Take a look at this block of CSS code. We've got two style rules, and each has several of the same declarations:

```
p.warning {
  color: red;
  font-weight: bold
  font-size: 11px;
  font-family: Arial;
}
h1.warning {
  color: red;
  font-weight: bold
  font-size: 24px;
  font-family: Arial;
}
```

A more efficient way to write this is

```
.warning {
  color: red;
  font-weight: bold
  font-size: 11px;
  font-family: Arial;
}
h1.warning {
  font-size: 24px;
}
```

Class selectors can also be chained together to target elements that have multiple class names. For example:

```
<h1 class="warning">Be careful!</h1>
<p class="warning help">Every 108 minutes, the button ➥
must be pushed. Do not attempt to use the computer ➥
for anything other than pushing the button.</p>
<p class="help">The code is 4-8-15-16-23-42.</p
```

A `.warning` selector will target both the `h1` and first `p` elements, since both have the `class` value `warning`. A `.help` selector will target both `p` elements (both have a `class` value of `help`). A chained selector such as `.warning.help` will select only the first paragraph, since it is the only element that has both classes (`warning` and `help`) assigned to it.

ID Selectors

ID selectors are similar to class selectors, but they are prefaced by a pound sign (#) instead of a period. So, to select this `div` element:

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

we would need a selector like this:

```
#main-content {
  width: 400px;
}
```

or this:

```
div#main-content {
  width: 400px;
}
```

■**Note** You may ask yourself why you'd ever need to join an element selector with an ID selector, since IDs are valid only once in each (X)HTML document. The answer lies in the fact that a single style sheet can be used over many documents. So, while one document may have a `div` element with the ID of content, the next might have a paragraph with the same ID. By leaving off the element selector, you can select both of these elements. Alternatively, you can ensure that only one of them is selected by using the element selector in conjunction with your ID selector.

ID selectors cannot be chained together, since it is invalid to have more than one ID on a given element in (X)HTML. However, it is possible to chain class selectors and ID selectors, such as `div#main-content.error`.

(X)HTML's Family Tree

(X)HTML documents are hierarchical in nature. Nearly every element in your (X)HTML document is the child of another element. For example, `head` and `body` are children of the `html` element. A `ul` element will likely have children `li` elements.

Another way to describe the parent and child relationship is as *ancestor* and *descendant*. Although they may seem alike, there is a vital difference: an item's parent element is exactly one level up the family tree from it. While the parent is an ancestor, there are likely additional ancestors two or more levels above it. Consider the following (X)HTML code:

```
<body>
  <h1>This is a <em>really</em> important header</h1>
  <p>This is a <strong>basic</strong> paragraph</p>
</body>
```

The top-level element in this example is `body`. Below it, there is an `h1` element, which has a child `em` element. The `em` is a descendant of `body`, but `body` is not its parent (`h1` is its parent). The `h1` element is both a child and a descendant of `body`, and it is the parent of `em`. Similarly, there is a paragraph element that has `body` as a parent and `strong` as a descendent. Therefore, the XHTML code can be represented using the element node tree seen in Figure 2-1.

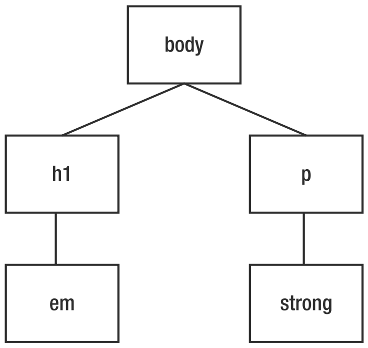

Figure 2-1. *An XHTML element node tree*

As you might have guessed, the reason for the quick family tree lesson is CSS's ability to piggyback on this inherent structure of (X)HTML.

Descendant Selectors

Descendant selectors, sometimes called *contextual selectors*, allow you to create style rules that are effective only when an element is the descendant of another one. Descendant selectors are indicated by a space between two elements. As an example, you may want to style only `li` elements that are descendants of `ul` lists (as opposed, say, to those who are part of `ol` lists). You'd do so like this:

```
ul li {
  color: blue;
}
```

This rule will make `li` text blue—but only when the `li` is contained within a `ul` element. So, in the following code, all `li` elements would be blue:

```
<ul>
  <li>Item one</li>
  <li>Item two</li>
  <li>Item three</li>
  <li>Item four has a nested list
    <ol>
      <li>Sub-item one</li>
      <li>Sub-item two</li>
    </ol>
  </li>
</ul>
```

Even though the nested list in item four is an `ol` element, the blue color will still be applied to its list items because they are the descendants of a `ul`.

Descendant selectors can be useful in targeting items deep in your (X)HTML structure, even when they don't have an ID or class assigned to them. By stringing together many elements, you can target `strong` elements inside `cite` elements inside `blockquote` elements inside `div` elements:

```
div blockquote cite strong {
  color: orange;
}
```

You can combine these with your class and ID selectors to get even more specific. Perhaps we want only `li` elements in `ul` elements with a `class` of `ingredients` inside our `div` with the `id` value `recipes`:

```
div#recipes ul.ingredients li {
  font-size 10px;
}
```

As you can imagine, descendant selectors are powerful, and it's no coincidence that descendant selectors are among the most-used types of CSS selectors.

Child Selectors

Child selectors are similar to descendant selectors, but they select only children rather than all ancestors. Child selectors are indicated by a greater-than sign (`>`).

■ **Note** Microsoft Internet Explorer 6 and below does not support child selectors.

Consider our example markup from earlier:

```
<ul>
  <li>Item one</li>
  <li>Item two</li>
  <li>Item three</li>
  <li>Item four has a nested list
    <ol>
      <li>Sub-item one</li>
      <li>Sub-item two</li>
    </ol>
  </li>
</ul>
```

Whereas our descendant selector, `ul li { color: blue; }`, targeted all `li` elements in this example, a similar child selector would only select the first four `li` elements, as they are direct children of a `ul` element:

```
ul > li {
  color: blue
}
```

It would *not* target those `li` elements in item four's nested `ol` list.

Adjacent Sibling Selectors

Adjacent sibling selectors allow you to target an element that immediately follows—and that has the same parent as—another element.

■ **Note** Microsoft Internet Explorer 6 and below does not support adjacent sibling selectors.

The concept of adjacent sibling selectors may sound a bit convoluted at first, but consider this example:

```
<body>
  <h1>This is a header</h1>
  <p>This is a paragraph.</p>
  <p>This is another paragraph.</p>
</body>
```

Paragraphs, by default, have a margin of 1 em above and below them. A common style effect is to remove the top margin of a paragraph when it is immediately after a header. The adjacent sibling selector, which is indicated by a plus sign (+), solves this problem for you:

```
h1 + p {
  margin-top: 0;
}
```

Although this is incredibly useful in theory, it's not so useful in the real world. Internet Explorer version 6 and older doesn't support this selector (Internet Explorer 7 does offer support for it, though). Including it in your styles doesn't hurt anything—IE will simply ignore it. But those visitors using IE won't see your adjacent sibling style rules, either.

This selector doesn't do a lot for you in the real world at the time of this writing, but it's still smart to be aware of it, as someday it will come in handy.

Attribute Selectors

While we're covering things that Internet Explorer doesn't support, let's talk about the humble *attribute selector*. This selector, which is indicated by square brackets ([ ]), allows you to target elements based on their attributes. You can select elements based on the presence of

- An attribute in an element
- An exact attribute value within an element

- A partial attribute value within an element (for example, as a part of a URL)
- A particular attribute name and value combination (or part thereof) in an element

In this section we will go into greater detail about these possibilities.

Note Microsoft Internet Explorer 6 and below does not support attribute selectors.

Presence of an Attribute

Consider anchor tags as an example of how simple attribute selection works. They are used both for links and for anchors within the page. Perhaps you want to style the two differently? You could select based on the presence of the `href` attribute, right? For example:

```
a[href] {
  color: red;
}
```

will select all anchor (`<a>`) elements that have an `href` attribute. The code

```
a[href][title] {
  color: red;
}
```

will select all anchor elements that have both `href` and `title` attributes. You can also use the universal selector, *, to target all elements that have a particular attribute. The code

```
*[src]
```

will select any element with an `src` attribute, which may include `img`, `embed`, and others.

Exact Attribute Value

You can also select based on the value of an attribute. For example, you may want to have links to your homepage appear differently than the rest of the links on your site:

```
<a href="http://ourcompany.com/" title="Our homepage">Home</a>
```

You could accomplish this in either of two ways:

```
a[title="Our homepage"] {
  color: red;
}
a[href="http://ourcompany.com"] {
  color: red;
}
```

Using this format matches attribute values exactly. In order for selection to occur, the match must be exact, even including case.

Partial Attribute Values

For attributes that accept a space-separated list of words (notably, the `class` attribute), it is possible to select based on the presence of a word within the list, rather than the exact match earlier. Remember our multiple-class example:

```
<p class="warning help">Every 108 minutes, the button ➥
must be pushed. Do not attempt to use the computer ➥
for anything other than pushing the button.</p>
```

To select this paragraph with the exact match selector earlier, you'd need to write: `p[ class="warning help"]`. Neither `p[class="warning"]` nor `p[class="help"]` would match. However, by using the tilde-equal (`~=`) indicator, you can select based on the presence of a word within the space-separated list, like so:

```
p[class~="help"] {
  color: red;
}
```

Note that this is functionally equivalent to the class selector (`p.warning` or `p.help`) we introduced earlier, and the class selector is far more widely supported. However, the partial attribute selector also works for attributes other than `class`.

Particular Attribute Selector

Perhaps better named the "equal to or starts with" attribute selector, the *partial attribute selector*—with its pipe-equal (`|=`) syntax—matches attribute values that either match exactly or begin with the given text. For example:

```
img[src|="vacation"] {
  float: left;
}
```

would target any image whose `src` value begins with `vacation`. It would match `vacation/photo1.jpg` and `vacation1.jpg`, but not `/vacation/photo1.jpg`.

Attribute selectors, like adjacent sibling selectors, would be more valuable if Internet Explorer 6 and lower supported them (again, they are supported in IE 7). Since it doesn't, many web developers are forced to admire them from afar.

Pseudo-Classes and Pseudo-Elements

And now for something completely different. OK, not completely—but close. Pseudo-class and pseudo-element selectors allow you to apply styles to elements that don't actually exist in your (X)HTML document. These are structures that you may have explicitly added to your (X)HTML document if you'd been able to predict them—but you can't. For example, it's often helpful to style the first line of a paragraph differently than the rest. But, given the vast array of devices, browsers, and screen sizes your site will be rendered on, there's simply no way to predict and define ahead of time what text encapsulates the first line.

Pseudo-Classes

Pseudo-classes are best understood by way of the anchor element, typically used for links between (X)HTML documents. It is commonplace for links to documents a user has visited in the past to be displayed differently than ones they haven't visited. But there's simply no way for you, as the web developer, to predefine this, because you haven't a clue what documents your visitor may have already hit.

To compensate for this, CSS 2.1 defines two pseudo-classes specifically for hyperlinks. Pseudo-classes are indicated by use of a colon (`:`). The two link-specific (with *link* defined as an anchor element with an `href` attribute) pseudo-classes are

- `:link`, which refers to links that point to documents that have not been visited. Some browsers interpret this incorrectly and apply it to all links, visited or not.
- `:visited`, which refers to hyperlinks to an address that has already been visited.

Using these pseudo-classes, you can make unvisited links blue and visited ones purple like so:

```
a:link {
  color: blue;
}
a:visited {
  color purple;
}
```

A couple of other common pseudo-classes are `:hover` and `:focus`. These are activated based on the current state an element is in with regard to the user's interaction with it. The hover state is activated when a user hovers on an element. Most typically, the hovering behavior is rolling over the element with a mouse. However, it's important to note that users on alternate devices may hover in a different manner. The focus state is activated when a user gives a particular element (especially a form field) focus by selecting it. In the typical desktop browser environment, this is done by tabbing to the element, or by clicking in a form field. Using these two pseudo-classes, you can easily change the display of an element only when these states are activated. For example:

```
a:hover {
  color: red;
}
tr:hover {
  background-color: #dfdfdf ;
}
input:focus {
  background-color: #dfdfdf ;
}
```

■ **Note** Microsoft Internet Explorer 6 and below supports pseudo-classes only on links (anchor elements with an `href` attribute). It does not allow for `:hover`, `:focus`, and so forth on arbitrary elements.

There are a handful of other pseudo-classes, all of which are covered in detail in Appendix A of this book.

Pseudo-Elements

As mentioned earlier, it is sometimes useful to style the first line of a paragraph or first letter of a header. These are examples of pseudo-elements. These work in a fashion similar to pseudo-classes:

```
p:first-line {
  font-weight: bold;
}
h1:first-letter {
  font-size: 2em;
}
```

In addition to `first-line` and `first-letter`, CSS offers `:before` and `:after` pseudo-elements, which let you generate content to be displayed just before or just after a particular element. For example, you may want to insert a comma `(,)` after every `<li>` element. Pseudo-elements are a topic of their own (and aren't very well supported across browsers); we cover them in detail in Appendix A.

Daisy-Chaining Selectors

It's important to note that all types of selectors can be combined and chained together. For example, take this style rule:

```
#primary-content div {
  color: orange
}
```

This code would make for orange-colored text in any `div` elements that are inside the element with an `id` value of `primary-content`. This next rule is a bit more complex:

```
#primary-content div.story h1 {
  font-style: italic
}
```

This code would italicize the contents of any `h1` elements within `div`s with the `class` value `story` inside any elements with an `id` value of `primary-content`. Finally, let's look at an over-the-top example, to show you just how complicated selectors can get:

```
#primary-content div.story h1 + ul > li a[href|="http://ourcompany.com"] em {
  font-weight: bold;
}
```

This code would boldface all `em` elements contained in anchors whose `href` attribute begins with `http://ourcompany.com` and are descendants of an `li` element that is a child of a `ul` element that is an adjacent sibling of an `h1` element that is a descendant of a `div` with the `class` named `story` assigned to it inside any element with an `id` value of `primary-content`. Seriously. Read it again, and follow along, right to left.

Grouping Selectors

You can also group selectors together to avoid writing the same declaration block over and over again. For example, if all your headers are going to be bold and orange, you could do this:

```
h1 {
  color: orange; font-weight: bold;
}
h2 {
  color: orange; font-weight: bold;
}
h3 {
  color: orange; font-weight: bold;
}
h4 {
  color: orange; font-weight: bold;
}
h5 {
  color: orange; font-weight: bold;
}
h6 {
  color: orange; font-weight: bold;
}
```

Or, for more efficiency, you could comma-separate your selectors and attach them all to a single declaration block, like this:

```
h1, h2, h3, h4, h5, h6 {
  color: orange; font-weight: bold;
}
```

Obviously this is much more efficient to write, and easier to manage later, if you decide you want all your headers green instead of orange.

Summary

CSS selectors range from simple to complex, and can be incredibly powerful when you begin to understand them fully. The key to writing efficient CSS is taking advantage of the hierarchical structure of (X)HTML documents. This involves getting especially friendly with descendant selectors. If you never become comfortable with the more advanced selectors, you'll find you write the same style rules over and over again, and that you add way more classes and IDs to your markup than is really necessary.

Another key concept of CSS is that of specificity and the cascade. We'll cover that topic in our next chapter.