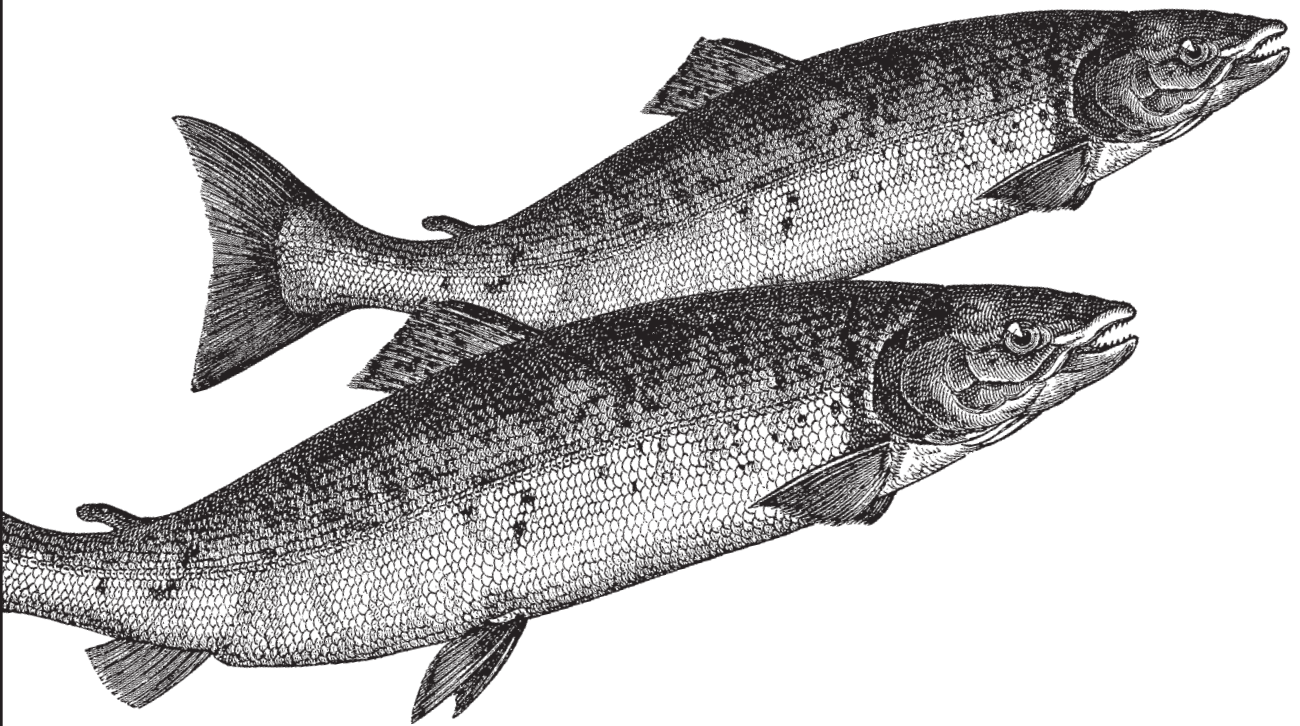


Visual Presentation for the Web

2nd Edition
Covers CSS2 & CSS2.1

Cascading Style Sheets

The Definitive Guide



O'REILLY®

Eric A. Meyer

Once of the primary advantages of CSS—particularly to designers—is its ability to easily apply a set of styles to all elements of the same type. Unimpressed? Consider this: by editing a single line of CSS, you can change the colors of all your headings. Don't like the blue you're using? Change that one line of code, and they can all be purple, yellow, maroon, or any other color you desire. That lets you, the designer, focus on design, rather than grunt work. The next time you're in a meeting and someone wants to see headings with a different shade of green, just edit your style and hit Reload. *Voilà!* The results are accomplished in seconds and are there for everyone to see.

Of course, CSS can't solve all your problems—you can't use it to change the color of your GIFs, for example—but it can make some global changes much easier. So let's begin with selectors and structure.

Basic Rules

As I've stated, a central feature of CSS is its ability to apply certain rules to an entire set of element types in a document. For example, let's say that you want to make the text of all h2 elements appear gray. Using old-school HTML, you'd have to do this by inserting `...` tags in all your h2 elements:

```
<h2><font color="gray">This is h2 text</font></h2>
```

Obviously, this is a tedious process if your document contains a lot of h2 elements. Worse, if you later decide that you want all those h2s to be green instead of gray, you'd have to start the manual tagging all over again.

CSS allows you to create rules that are simple to change, edit, and apply to all the text elements you define (the next section will explain how these rules work). For example, simply write this rule once to make all your h2 elements gray:

```
h2 {color: gray;}
```

If you want to change all h2 text to another color—for example, silver—simply alter the rule:

```
h2 {color: silver;}
```

Rule Structure

In order to understand the concept of rules in more detail, let's break down the structure.

Each rule has two fundamental parts, the *selector* and the *declaration block*. The declaration block is composed of one or more *declarations*, and each declaration is a pairing of a *property* and a *value*. Every style sheet is made up of a series of rules. Figure 2-1 shows the parts of a rule.

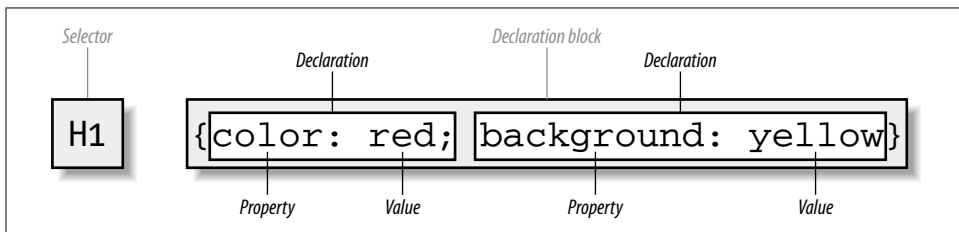


Figure 2-1. The structure of a rule

The selector, shown on the left side of the rule, defines which piece of the document will be affected. In Figure 2-1, h1 elements are selected. If the selector were p, then all p (paragraph) elements would be selected.

The right side of the rule contains the declaration block, which is made up of one or more declarations. Each declaration is a combination of a CSS property and a value of that property. In Figure 2-1, the declaration block contains two declarations. The first states that this rule will cause parts of the document to have a color of red, and the second states that part of the document will have a background of yellow. So, all of the h1 elements in the document (defined by the selector) will be styled in bold purple.

Element Selectors

A selector is most often an HTML element, but not always. For example, if a CSS file contains styles for an XML document, a selector might look something like this:

```
QUOTE {color: gray;}
BIB {color: red;}
BOOKTITLE {color: purple;}
MYElement {color: red;}
```

In other words, the elements of the document serve as the most basic selectors. In XML, a selector could be anything since XML allows for the creation of new markup languages that can have just about anything as an element name. If you're styling an

HTML document, on the other hand, the selector will generally be one of the many HTML elements such as p, h3, em, a, or even html itself. For example:

```
html {color: black;}
h1 {color: gray;}
h2 {color: silver;}
```

The results of this style sheet are shown in Figure 2-2.

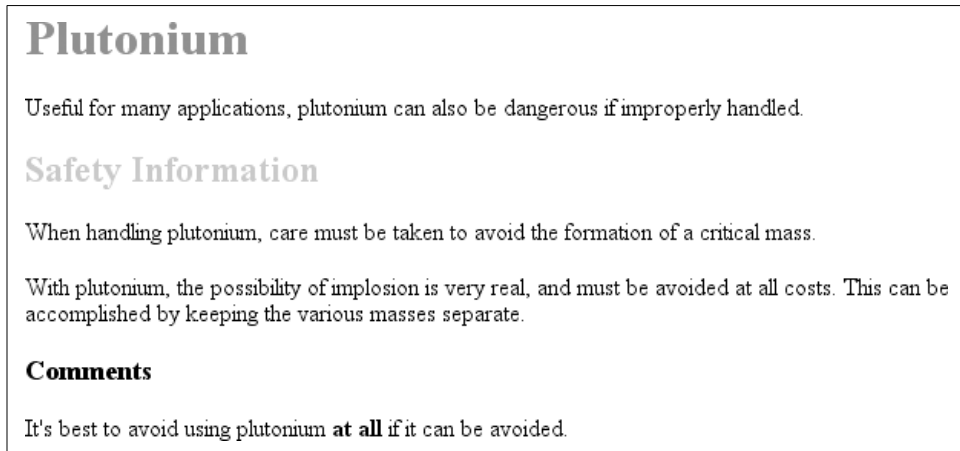


Figure 2-2. Simple styling of a simple document

Once you've globally applied styles directly to elements, you can shift those styles from one element to another. Let's say you decide that the paragraph text, not the h1 elements, in Figure 2-2 should be gray. No problem. Simply change the h1 selector to p:

```
html {color: black;}
p {color: gray;}
h2 {color: silver;}
```

The results are shown in Figure 2-3.

Declarations and Keywords

The declaration block contains one or more declarations. A declaration is always formatted as a property followed by a colon and then a *value* followed by a semicolon. The colon and semicolon can be followed by zero or more spaces. In nearly all cases, a value is either a single keyword or a space-separated list of one or more keywords that are permitted for that property. If you use either an incorrect property or value in a declaration, the whole thing will be ignored. Thus, the following two declarations would fail:

```
brain-size: 2cm; /* unknown property */
color: ultraviolet; /* unknown value */
```

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-3. Moving a style from one element to another

In an instance where you can use more than one keyword for a property's value, the keywords are usually separated by spaces. Not every property can accept multiple keywords, but many, such as the font property, can. Let's say you want to define medium-sized Helvetica for paragraph text, as illustrated in Figure 2-4.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-4. The results of a property value with multiple keywords

The rule would read as follows:

```
p {font: medium Helvetica;}
```

Note the space between `medium` and `Helvetica`, each of which is a keyword (the first is the font's size and the second is the actual font name). The space allows the user agent to distinguish between the two keywords and apply them correctly. The semi-colon indicates that the declaration has been concluded.

These space-separated words are referred to as keywords because, taken together, they form the value of the property in question. For instance, consider the following fictional rule:

```
rainbow: red orange yellow green blue indigo violet;
```

There is no such property as `rainbow`, of course, and two of the colors used aren't valid either, but the example is useful for illustrative purposes. The value of `rainbow` is `red orange yellow green blue indigo violet`, and the seven keywords add up to a single, unique value. We can redefine the value for `rainbow` as follows:

```
rainbow: infrared red orange yellow green blue indigo violet ultraviolet;
```

Now we have a new value for `rainbow` composed of nine keywords instead of seven. Although the name of the two values is the same, the two are as unique and different as zero and one.



As we've seen, CSS keywords are separated by spaces—except in one instance. In the CSS property `font`, there is exactly one place where a forward-slash (/) can be used to separate two specific keywords. Here's an example:

```
h2 {font: large/150% sans-serif;}
```

The slash separates the keywords that set the element's font size and line height. This is the only place the slash is allowed to appear in the font declaration. All of the other keywords allowed for `font` are separated by spaces.

Those are the basics of simple declarations, but they can get much more complex. The next section begins to show you just how powerful CSS can be.

Grouping

So far, we've learned fairly simple techniques for applying a single style to a single selector. But what if you want the same style to apply to multiple elements? If that's the case, you'll want to use more than one selector or apply more than one style to an element or group of elements.

Grouping Selectors

Let's say you want `h2` elements and paragraphs to have gray text. The easiest way to accomplish this is to use the following declaration:

```
h2, p {color: gray;}
```

By placing the `h2` and `p` selectors on the left side of the rule and separating them with a comma, you've defined a rule where the style on the right (`color: gray;`) applies to the elements referenced by both selectors. The comma tells the browser that there

are two different selectors involved in the rule. Leaving out the comma would give the rule a completely different meaning, which we'll explore later in the section "Descendant Selectors."

There are really no limits on how many selectors you can group together. For example, if you want to display a large number of elements in gray, you might use something like the following rule:

```
body, table, th, td, h1, h2, h3, h4, p, pre, strong, em, b, i {color: gray;}
```

Grouping allows an author to drastically compact certain types of style assignments, which makes for a shorter style sheet. The following alternatives produce exactly the same result, but it's pretty obvious which one is easier to type:

```
h1 {color: purple;}
h2 {color: purple;}
h3 {color: purple;}
h4 {color: purple;}
h5 {color: purple;}
h6 {color: purple;}
```

```
h1, h2, h3, h4, h5, h6 {color: purple;}
```

Grouping allows for some interesting choices. For example, all of the groups of rules in the following example are equivalent—each merely shows a different way of grouping both selectors and declarations:

```
/* group 1 */
h1 {color: silver; background: white;}
h2 {color: silver; background: gray;}
h3 {color: white; background: gray;}
h4 {color: silver; background: white;}
b {color: gray; background: white;}

/* group 2 */
h1, h2, h4 {color: silver;}
h2, h3 {background: gray;}
h1, h4, b {background: white;}
h3 {color: white;}
b {color: gray;}

/* group 3 */
h1, h4 {color: silver; background: white;}
h2 {color: silver;}
h3 {color: white;}
h2, h3 {background: gray;}
b {color: gray; background: white;}
```

Any of these will yield the result shown in Figure 2-5. (These styles use grouped declarations, which are explained in the upcoming section.)

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium at all if it can be avoided.

Figure 2-5. The result of equivalent style sheets

The universal selector

CSS2 introduced a new simple selector called the *universal selector*, displayed as an asterisk (*). This selector matches any element at all, much like a wildcard. For example, to make every single element in a document red, you would write:

```
* {color: red;}
```

This declaration is equivalent to a grouped selector that lists every single element contained within the document. The universal selector lets you assign the color value red to every element in the document in one efficient stroke. Beware, however: although the universal selector is convenient, it can have unintended consequences, which I'll discuss in the next chapter.

Grouping Declarations

Since you can group selectors together into a single rule, it follows that you can also group declarations. Assume that you want all h1 elements to appear in purple, 18-pixel-high Helvetica text on an aqua background (and you don't mind blinding your readers). You could write your styles like this:

```
h1 {font: 18px Helvetica;}  
h1 {color: purple;}  
h1 {background: aqua;}
```

But this method is inefficient; imagine creating such a list for an element that will carry 10 or 15 styles! Instead, you can group your declarations together:

```
h1 {font: 18px Helvetica; color: purple; background: aqua;}
```

This will have exactly the same effect as the three-line style sheet just shown.

Note that using semicolons at the end of each declaration is crucial when you're grouping them. Browsers ignore whitespace in style sheets, and the user agent must rely on correct syntax to parse the style sheet. You can fearlessly format styles like the following:

```
h1 {
  font: 18px Helvetica;
  color: purple;
  background: aqua;
}
```

If the second semicolon is omitted, however, the user agent will interpret the style sheet as follows:

```
h1 {
  font: 18px Helvetica;
  color: purple background: aqua;
}
```

Since `background:` is not a valid value for `color`, and also since `color` can be given only one keyword, a user agent will ignore the `color` declaration (including the `background: aqua` part) entirely. It might render `h1`s as purple text without an aqua background, but more likely, you won't even get purple `h1`s. Instead, they'll be the default color (usually black) with no background at all. (The declaration `font: 18px Helvetica` will still take effect since it was correctly terminated with a semicolon.)



While it is not technically necessary to follow the last declaration of a rule with a semicolon, it is generally good practice to do so. First, it will keep you in the habit of terminating your declarations with semicolons, the lack of which is one of the most common causes of rendering errors. Second, if you decide to add another declaration to a rule, you won't have to worry about forgetting to insert an extra semicolon. Finally, some older browsers such as Internet Explorer 3.x have a greater tendency to become confused if the semicolon is left off the final declaration in a rule. Avoid all these problems: always follow a declaration with a semicolon, wherever the rule appears.

As with selector grouping, declaration grouping is a convenient way to keep your style sheets short, expressive, and easy to maintain.

Grouping Everything

You now know that you can group selectors, and you can group declarations. By combining both kinds of grouping in single rules, you can define very complex styles using only a few statements. Now, what if you want to assign some complex styles to all the headings in a document, and you want the same styles to be applied to all of them? Here is how you do this:

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white; padding: 0.5em;
  border: 1px solid black; font-family: Charcoal, sans-serif;}
```

You've grouped the selectors, so the styles on the right side of the rule will be applied to all the headings listed, and grouping the declarations means that all of the listed styles will be applied to the selectors on the left side of the rule. The result of this rule is shown in Figure 2-6.

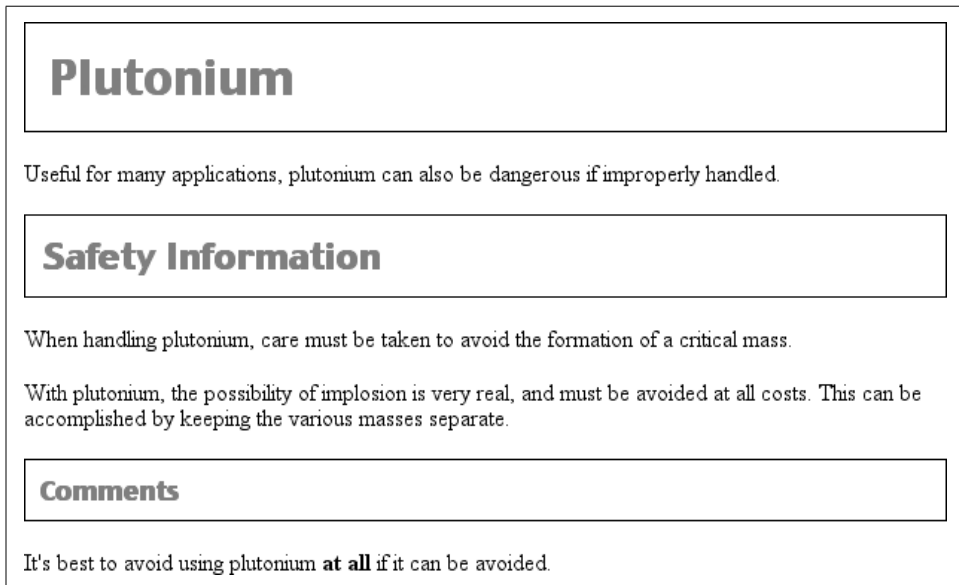


Figure 2-6. Grouping both selectors and rules

This approach is preferable to the drawn-out alternative, which would begin with something like this:

```
h1 {color: gray;}
h2 {color: gray;}
h3 {color: gray;}
h4 {color: gray;}
h5 {color: gray;}
h6 {color: gray;}
h1 {background: white;}
h2 {background: white;}
h3 {background: white;}
```

and continue for many lines. You certainly can write out your styles the long way, but I wouldn't recommend it. Editing your styles would be almost as tedious as using font tags everywhere!

It's possible to add even more expression to selectors and to apply styles in a way that cuts across elements in favor of types of information. Of course, to get something so powerful, you'll have to do a little work in return, but it's well worth it.

Class and ID Selectors

So far, we've been grouping selectors and declarations together in a variety of ways, but the selectors we've been using are still simple ones. The selectors we've used refer only to document elements; they're fine up to a point, but there are times when you need something a little more specialized.

In addition to raw document elements, there are two other types of selectors: *class selectors* and *ID selectors*, which let you assign styles in a way that is independent of document elements. These selectors can be used on their own or in conjunction with element selectors. However, they work only if you've marked up your document appropriately, so using them generally involves a little forethought and planning.

For example, say you're drafting a document that discusses ways of handling plutonium. The document contains various warnings about safely dealing with such a dangerous substance. You want each warning to appear in boldface text so that it will stand out. However, you don't know which elements these warnings will be. Some warnings could be entire paragraphs, while others could be a single item within a lengthy list or a small section of text. So, you can't define a rule using simple selectors of any kind. Suppose you tried this route:

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

All paragraphs would be bold, not just those that contain warnings. You need a way to select only the text that contains warnings, or more precisely, a way to select only those elements that are warnings. How do you do it? You apply styles to parts of the document that have been marked in a certain way, independent of the elements involved, by using class selectors.

Class Selectors

The most common way to apply styles without worrying about the elements involved is to use class selectors. Before you can use them, however, you need to modify your actual document markup so that the class selectors will work. Enter the class attribute:

```
<p class="warning">When handling plutonium, care must be taken to avoid  
the formation of a critical mass.</p>  
<p>With plutonium, <span class="warning">the possibility of implosion is  
very real, and must be avoided at all costs</span>. This can be accomplished  
by keeping the various masses separate.</p>
```

In order to associate the styles of a class selector with an element, you must assign a class attribute to the appropriate value. In the previous code, a class value of `warning` was assigned to two elements: the first paragraph and the `span` element in the second paragraph.

All you need now is a way to apply styles to these classed elements. In HTML documents, you can use a very compact notation where the name of a class is preceded by a period (.) and can be joined with a simple selector:

```
*.warning {font-weight: bold;}
```

When combined with the example markup shown earlier, this simple rule has the effect shown in Figure 2-7. That is, the style of `font-weight: bold` will be applied to every element (thanks to the presence of the universal selector) that carries a class attribute with a value of `warning`.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, **the possibility of implosion is very real, and must be avoided at all costs.** This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-7. Using a class selector

As you can see, the class selector works by directly referencing a value that will be found in the `class` attribute of an element. This reference is *always* preceded by a period (.), which marks it as a class selector. The period helps keep the class selector separate from anything with which it might be combined—like an element selector. For example, you may want boldface text only when an entire paragraph is a warning:

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

The selector now matches any `p` elements that have a `class` attribute containing the word `warning`, but no other elements of any kind, classed or otherwise. The selector `p.warning` translates to: “Any paragraph whose `class` attribute contains the word `warning`.” Since the `span` element is not a paragraph, the rule’s selector doesn’t match it, and it won’t be converted to bold text.

If you did want to assign different styles to the `span` element, you could have used the selector `span.warning`:

```
p.warning {font-weight: bold;}
span.warning {font-style: italic;}
```

In this case, the warning paragraph is boldfaced, while the warning span is italicized. Each rule applies only to a specific type of element/class combination and so does not leak over to other elements.

Another option is to use a combination of a general class selector and an element-specific class selector to make the styles even more useful, as in the following markup:

```
.warning {font-style: italic;}  
span.warning {font-weight: bold;}
```

The results are shown in Figure 2-8.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, ***the possibility of implosion is very real, and must be avoided at all costs.*** This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-8. Using generic and specific selectors to combine styles

In this situation, any warning text will be italicized, but only the text within a span element and text with a class of warning will be boldfaced and italicized.

Notice the format of the general class selector in the previous example: it's simply a class name preceded by a period without any element name. In cases where you only want to select all elements that share a class name, you can omit the universal selector from a class selector without any ill effects.

Multiple Classes

In the previous section, we dealt with class values that contained a single word. In HTML, it's possible to have a space-separated list of words in a single class value. For example, if you want to mark a particular element as being both urgent and a warning, you could write:

```
<p class="urgent warning">When handling plutonium, care must be taken to  
avoid the formation of a critical mass.</p>
```

```
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

The order of the words doesn't actually matter; `warning urgent` would also suffice.

Now let's say you want all elements with a class of `warning` to be boldface, those with a class of `urgent` to be italic, and those elements with both values to have a silver background. This would be written as follows:

```
.warning {font-weight: bold;}
.urgent {font-style: italic;}
.warning.urgent {background: silver;}
```

By chaining two class selectors together, you can select only those elements that have both class names, in any order. As you can see, the HTML source contains `class="urgent warning"` but the CSS selector is written `.warning.urgent`. Regardless, the rule will still cause the "When handling plutonium..." paragraph to have a silver background, as illustrated in Figure 2-9.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, **the possibility of implosion is very real, and must be avoided at all costs.** This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

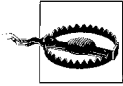
Figure 2-9. Selecting elements with multiple class names

If a multiple class selector contains a name that is not in the space-separated list, then the match will fail. Consider the following rule:

```
p.warning.help {background: red;}
```

As you would expect, the selector will match only those `p` elements with a class containing the words `warning` and `help`. Therefore, it will not match a `p` element with just the words `warning` and `urgent` in its class attribute. It would, however, match the following:

```
<p class="urgent warning help">Help me!</p>
```



Internet Explorer for both platforms has problems correctly handling multiple class selectors. While you can select a single class name out of a list, selecting based on multiple names in a list will not work properly. Thus, `p.warning` would work as expected, but `p.warning.help` would match any `p` elements that have a `class` attribute with the word `help` because it came last in the selector. If you wrote `p.help.warning`, then Explorer would match any `p` elements that have `warning` in their `class` value.

ID Selectors

In some ways, ID selectors are similar to class selectors, but there are a few crucial differences. First, ID selectors are preceded by an octothorpe (`#`)—also known as a pound sign, hash mark, or tic-tac-toe board—instead of a period. Thus, you might see a rule like this one:

```
*#first-para {font-weight: bold;}
```

This rule applies boldface text to any element whose `id` attribute has a value of `first-para`.

The second difference is that instead of referencing values of the `class` attribute, ID selectors refer, unsurprisingly, to values found in `id` attributes. Here's an example of an ID selector in action:

```
*#lead-para {font-weight: bold;}
```

```
<p id="lead-para">This paragraph will be boldfaced.</p>  
<p>This paragraph will NOT be bold.</p>
```

Note that the value `lead-para` could have been assigned to any element within the document. In this particular case, it is applied to the first paragraph, but you could have applied it just as easily to the second or third paragraph.

As with class selectors, it is possible to omit the universal selector from an ID selector. In the previous example, you could also have written:

```
#lead-para {font-weight: bold;}
```

The effect of this selector would be the same.

Deciding Between Class and ID

You may assign classes to any number of elements, as demonstrated earlier; the class name `warning` was applied to both a `p` and a `span` element, and it could have been applied to many more elements. IDs, on the other hand, are used once, and only once, within an HTML document. Therefore, if you have an element with an `id` value of `lead-para`, no other element in that document can have an `id` value of `lead-para`.



In the real world, browsers don't usually check for the uniqueness of IDs in HTML, which means that if you sprinkle an HTML document with several elements, all of which have the same value for their ID attributes, you'll probably get the same styles applied to each. This is incorrect behavior, but it happens anyway. Having more than one of the same ID value in a document also makes DOM scripting more difficult, since functions like `getElementById()` depend on there being one, and only one, element with a given ID value.

Unlike class selectors, ID selectors can't be combined, since ID attributes do not permit a space-separated list of words.

On a purely syntactical level, the dot-class notation (e.g., `.warning`) is not guaranteed to work for XML documents. As of this writing, the dot-class notation works in HTML, SVG, and MathML, and it may well be permitted in future languages, but it's up to each language's specification to decide that. The hash-ID notation (e.g., `#lead`) will work in any document language that has an attribute that enforces uniqueness within a document. Uniqueness can be enforced with an attribute called `id`, or indeed anything else, as long as the attribute's contents are defined to be unique within the document.

Another difference between `class` and `id` names is that IDs carry a heavier weight when trying to determine which styles should be applied to a given element. I'll explain this in greater detail in the next chapter.

Like classes, IDs can also be selected independently of an element. There may be circumstances in which you know that a certain ID value will appear in a document, but you don't know the element on which it will appear (as in the plutonium-handling warnings), so you'll want to declare standalone ID selectors. For example, you may know that in any given document, there will be an element with an ID value of `mostImportant`. You don't know whether that most important thing will be a paragraph, a short phrase, a list item, or a section heading. You know only that it will exist in each document, occur in an arbitrary element, and appear no more than once. In that case, you would write a rule like this:

```
#mostImportant {color: red; background: yellow;}
```

This rule would match any of the following elements (which, as I noted before, should *not* appear together in the same document because they all have the same ID value):

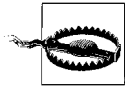
```
<h1 id="mostImportant">This is important!</h1>  
<em id="mostImportant">This is important!</em>  
<ul id="mostImportant">This is important!</ul>
```

Also note that class and ID selectors may be case-sensitive, depending on the document language. HTML and XHTML define class and ID values to be case-sensitive,

so the capitalization of your class and ID values must match that found in your documents. Thus, in the following pairing of CSS and HTML, the element will not be boldfaced:

```
p.criticalInfo {font-weight: bold;}  
  
<p class="criticalinfo">Don't look down.</p>
```

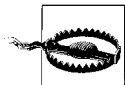
Because of the change in case for the letter “I”, the selector will not match the element shown.



Some older browsers did not treat class and ID names as case-sensitive, but all current browsers as of this writing enforce case sensitivity.

Attribute Selectors

When it comes to both class and ID selectors, what you’re really doing is selecting values of attributes. The syntax used in the previous two sections is particular to HTML, SVG, and MathML documents (as of this writing). In other markup languages, these class and ID selectors may not be available. To address this situation, CSS2 introduced *attribute selectors*, which can be used to select elements based on their attributes and the values of those attributes. There are four types of attribute selectors.



Attribute selectors are supported by the Opera and Gecko browsers but not by Internet Explorer through IE5/Mac and IE6/Win.

Simple Attribute Selection

If you want to select elements that have a certain attribute, regardless of the attribute’s value, you can use a simple attribute selector. For example, to select all h1 elements that have a class attribute with any value and make their text silver, write:

```
h1[class] {color: silver;}
```

So given the following markup:

```
<h1 class="hoopla">Hello</h1>  
<h1 class="severe">Serenity</h1>  
<h1 class="fancy">Fooling</h1>
```

you get the result shown in Figure 2-10.

This strategy is very useful in XML documents, as XML languages tend to have element and attribute names that are very specific to their purpose. Consider an XML language that is used to describe planets of the solar system (we’ll call it PlanetML).



Figure 2-10. Selecting elements based on their attributes

If you want to select all `planet` elements with a `moons` attribute and make them boldface, thus calling attention to any planet that has moons, you would write:

```
planet[moons] {font-weight: bold;}
```

This would cause the text of the second and third elements in the following markup fragment to be boldfaced, but not the first:

```
<planet>Venus</planet>  
<planet moons="1">Earth</planet>  
<planet moons="2">Mars</planet>
```

In HTML documents, you can use this feature in a number of creative ways. For example, you could style all images that have an `alt` attribute, thus highlighting those images that are correctly formed:

```
img[alt] {border: 3px solid red;}
```

(This particular example is useful more for diagnostic purposes—that is, determining whether images are indeed correctly formed—than for design purposes.)

If you wanted to boldface any element that includes `title` information, which most browsers display as a “tooltip” when a cursor hovers over the element, you could write:

```
*[title] {font-weight: bold;}
```

Similarly, you could style only those anchors (`a` elements) that have an `href` attribute.

It is also possible to select based on the presence of more than one attribute. This is done simply by chaining the attribute selectors together. For example, to boldface the text of any HTML hyperlink that has both an `href` and a `title` attribute, you would write:

```
a[href][title] {font-weight: bold;}
```

This would boldface the first link in the following markup, but not the second or third:

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />  
<a href="http://www.webstandards.org">Standards Info</a><br />  
<a title="Not a link">dead.letter</a>
```

Selection Based on Exact Attribute Value

In addition to selecting elements with attributes, you can further narrow the selection process to encompass only those elements whose attributes are a certain value. For example, let's say you want to boldface any hyperlink that points to a certain document on the web server. This would look something like:

```
a[href="http://www.css-discuss.org/about.html"] {font-weight: bold;}
```

Any attribute and value combination can be specified for any element. However, if that exact combination does not appear in the document, then the selector won't match anything. Again, XML languages can benefit from this approach to styling. Let's return to our PlanetML example. Suppose you want to select only those planet elements that have a value of 1 for the attribute moons:

```
planet[moons="1"] {font-weight: bold;}
```

This would boldface the text of the first and second elements in the following markup fragment, but not the first or third:

```
<planet>Venus</planet>  
<planet moons="1">Earth</planet>  
<planet moons="2">Mars</planet>
```

As with attribute selection, you can chain together multiple attribute-value selectors to select a single document. For example, to double the size of the text of any HTML hyperlink that has both an href with a value of `http://www.w3.org/` and a title attribute with a value of `W3C Home`, you would write:

```
a[href="http://www.w3.org/"][title="W3C Home"] {font-size: 200%;}
```

This would double the text size of the first link in the following markup, but not the second or third:

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />  
<a href="http://www.webstandards.org"  
  title="Web Standards Organization">Standards Info</a><br />  
<a href="http://www.example.org/" title="W3C Home">dead.link</a>
```

The results are shown in Figure 2-11.



Figure 2-11. Selecting elements based on attributes and their values

Note that this format requires an exact match for the attribute's value. Matching becomes an issue when the form encounters values that can in turn contain a space-separated list of values (e.g., the HTML attribute `class`). For example, consider the following markup fragment:

```
<planet type="barren rocky">Mercury</planet>
```

The only way to match this element based on its exact attribute value is to write:

```
planet[type="barren rocky"] {font-weight: bold;}
```

If you had written `planet[type="barren"]`, the rule would not have matched the example markup and thus would have failed. This is true even for the `class` attribute in HTML. Consider:

```
<p class="urgent warning">When handling plutonium, care must be taken to  
avoid the formation of a critical mass.</p>
```

To select this element based on its exact attribute value, you would have to write:

```
p[class="urgent warning"] {font-weight: bold;}
```

This is not the dot-class notation we covered earlier.

Also, be aware that ID selectors and attribute selectors that target the `id` attribute are not precisely the same. In other words, there is a subtle but crucial difference between `h1#page-title` and `h1[id="page-title"]`. This difference is explained in the next chapter.

Selection Based on Partial Attribute Values

For any attribute that accepts a space-separated list of words, it is possible to select based on the presence of any one of those words. The classic example in HTML is the `class` attribute, which can accept one or more words as its value. Consider our usual example text:

```
<p class="urgent warning">When handling plutonium, care must be taken to  
avoid the formation of a critical mass.</p>
```

Let's say you want to select elements whose `class` attribute contains the word `warning`. You can do this with an attribute selector:

```
p[class~="warning"] {font-weight: bold;}
```

Note the presence of the tilde (`~`) in the selector. It is the key to selection based on the presence of a space-separated word within the attribute's value. If you omit the tilde, you would have an exact-value matching requirement, as discussed in the previous section.

This selector construct is equivalent to the dot-class notation discussed earlier. Thus, `p.warning` and `p[class~="warning"]` are equivalent when applied to HTML documents. Let's return to our previous XML example:

```
<planet type="barren rocky">Mercury</planet>  
<planet type="barren cloudy">Venus</planet>  
<planet type="life-bearing cloudy">Earth</planet>
```

In order to make all elements with the word `barren` in their `type` attribute italicized, you write:

```
planet[type~="barren"] {font-style: italic;}
```

This rule’s selector will match the first two elements in the example XML and thus italicize their text, as shown in Figure 2-12.

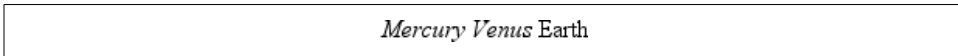


Figure 2-12. Selecting elements based on portions of attribute values

Even in HTML, this form of attribute selector can be useful. For example, you might have a document that contains a number of images, only some of which are figures. You can use a partial-value attribute selector aimed at the title text to select only those figures:

```
img[title~="Figure"] {border: 1px solid gray;}
```

This rule will select any image whose title text contains the word Figure. Therefore, as long as all your figures have title text that looks something like “Figure 4. A bald-headed elder statesman,” the rule above will match those images. For that matter, the selector `img[title~="Figure"]` will also match a title attribute with the value “How To Figure Out Who’s In Charge.” Any image that does not have a title attribute, or whose title value doesn’t contain the word “Figure,” won’t be matched.

A Particular Attribute Selection Type

A fourth type of attribute selector, the particular attribute selector, is easier to show than it is to describe. Consider the following rule:

```
*[lang|=“en”] {color: white;}
```

This rule will select any element whose lang attribute is equal to en or begins with en-. Therefore, the first three elements in the following example markup would be selected, but the last two would not:

```
<h1 lang="en">Hello!</h1>
<p lang="en-us">Greetings!</p>
<div lang="en-au">G'day!</div>
<p lang="fr">Bonjour!</p>
<h4 lang="cy-en">Jrooana!</h4>
```

In general, the form `[att|=“val”]` can be used for any attribute and its values. Let’s say you have a series of figures in an HTML document, each of which has a filename like `figure-1.gif` and `figure-3.jpg`. You can match all of these images using the following selector:

```
img[src|=“figure”] {border: 1px solid gray;}
```

The most common use for this type of attribute selector is to match language values, as demonstrated later in this chapter.

Using Document Structure

As I've mentioned before, CSS is powerful because it uses the structure of HTML documents to determine appropriate styles and how to apply them. That's only part of the story since it implies that such determinations are the only way CSS uses document structure. Structure plays a much larger role in the way styles are applied to a document. Let's take a moment to discuss structure before moving on to more powerful forms of selection.

Understanding the Parent–Child Relationship

In order to understand the relationship between selectors and documents, you need to once again examine how documents are structured. Consider this very simple HTML document:

```
<html>
<head>
  <base href="http://www.meerkat.web/">
  <title>Meerkat Central</title>
</head>
<body>
  <h1>Meerkat <em>Central</em></h1>
  <p>
    Welcome to Meerkat <em>Central</em>, the <strong>best meerkat web site
    on <a href="inet.html">the <em>entire</em> Internet</a></strong>!</p>
  <ul>
    <li>We offer:
      <ul>
        <li><strong>Detailed information</strong> on how to adopt a meerkat</li>
        <li>Tips for living with a meerkat</li>
        <li><em>Fun</em> things to do with a meerkat, including:
          <ol>
            <li>Playing fetch</li>
            <li>Digging for food</li>
            <li>Hide and seek</li>
          </ol>
        </li>
      </ul>
    </li>
    <li>...and so much more!</li>
  </ul>
  <p>
    Questions? <a href="mailto:suricate@meerkat.web">Contact us!</a>
  </p>
</body>
</html>
```

Much of the power of CSS is based on the *parent–child relationship* of elements. HTML documents (actually, most structured documents of any kind) are based on a hierarchy of elements, which is visible in the “tree” view of the document

(Figure 2-13). In this hierarchy, each element fits somewhere into the overall structure of the document. Every element in the document is either the *parent* or the *child* of another element, and it's often both.

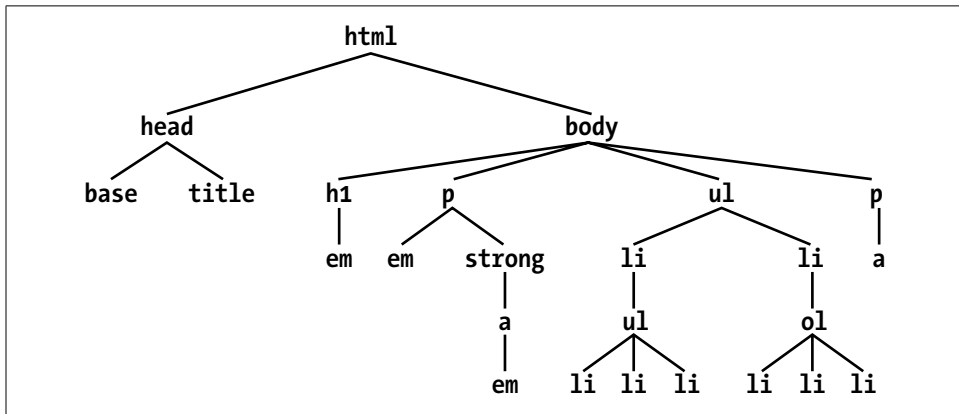


Figure 2-13. A document tree structure

An element is said to be the parent of another element if it appears directly above that element in the document hierarchy. For example, in Figure 2-13, the first `p` element is parent to an `em` and a `strong` element, while the `strong` is parent to an anchor element, which is, itself, parent to another `em` element. Conversely, an element is the child of another element if it is directly beneath the other element. Thus, the anchor element in Figure 2-13 is a child of the `strong` element, which is in turn child to the paragraph, and so on.

The terms parent and child are specific applications of the terms *ancestor* and *descendant*. There is a difference between them: in the tree view, if an element is exactly one level above another, then they have a parent–child relationship. If the path from one element to another continues through two or more levels, the elements have an ancestor–descendant relationship, but not a parent–child relationship. (Of course, a child is also a descendant, and a parent is an ancestor.) In Figure 2-13, the first `ul` element is parent to two `li` elements, but the first `ul` is also the ancestor of every element descended from its `li` element, all the way down to the most deeply nested `li` elements.

Also, in Figure 2-13, there is an anchor that is a child of `strong`, but also a descendant of paragraph, `body`, and `html` elements. The `body` element is an ancestor of everything that the browser will display by default, and the `html` element is ancestor to the entire document. For this reason, the `html` element is also called the *root element*.

Descendant Selectors

The first benefit derived from understanding this model is the ability to define *descendant selectors* (also known as contextual selectors). Defining descendant selectors is the act of creating rules that operate in certain structural circumstances but not others. As an example, let's say you want to style only those `em` elements that are descended from `h1` elements. You could put a `class` attribute on every `em` element found within an `h1`, but that's almost as time-consuming as using the `font` tag. It's obviously far more efficient to declare rules that match only `em` elements that are found inside `h1` elements.

To do so, write the following:

```
h1 em {color: gray;}
```

This rule will make gray any text in an `em` element—that is, the descendant of an `h1` element. Other `em` text, such as that found in a paragraph or a block quote, will not be selected by this rule. Figure 2-14 makes this clear.



Figure 2-14. Selecting an element based on its context

In a descendant selector, the selector side of a rule is composed of two or more space-separated selectors. The space between the selectors is an example of a *combinator*. Each space combinator can be translated as “found within,” “which is part of,” or “that is a descendant of,” but only if you read the selector right to left. Thus, `h1 em` can be translated as, “Any `em` element that is a descendant of an `h1` element.” (To read the selector left to right, you might phrase it something like, “Any `h1` that contains an `em` will have the following styles applied to the `em`.”)

You aren't limited to two selectors, of course. For example:

```
ul ol ul em {color: gray;}
```

In this case, as Figure 2-15 shows, any emphasized text that is part of an unordered list that is part of an ordered list that is itself part of an unordered list (yes, this is correct) will be gray. This is obviously a very specific selection criterion.

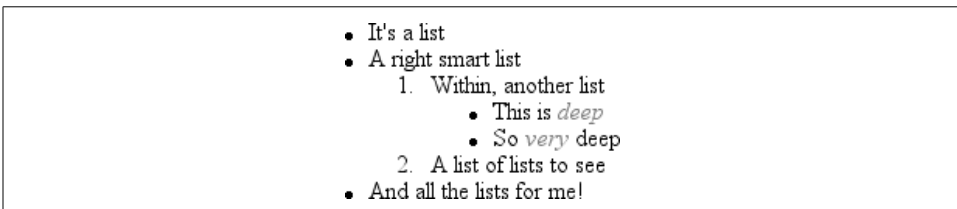


Figure 2-15. A very specific descendant selector

Descendant selectors can be extremely powerful. They make possible what could never be done in HTML—at least not without oodles of font tags. Let's consider a common example. Assume you have a document with a sidebar and a main area. The sidebar has a blue background, the main area has a white background, and both areas include lists of links. You can't set all links to be blue because they'd be impossible to read in the sidebar.

The solution: descendant selectors. In this case, you give the table cell that contains your sidebar a class of `sidebar`, and assign the main area a class of `main`. Then you write styles like this:

```
td.sidebar {background: blue;}
td.main {background: white;}
td.sidebar a:link {color: white;}
td.main a:link {color: blue;}
```

Figure 2-16 shows the result.



Figure 2-16. Using descendant selectors to apply different styles to the same type of element



`:link` refers to links to resources that haven't been visited. We'll talk about this in detail later in this chapter.

Here's another example: let's say that you want gray to be the text color of any `b` (boldface) element that is part of a `blockquote`, and also for any bold text that is found in a normal paragraph:

```
blockquote b, p b {color: gray;}
```

The result is that the text within `b` elements that are descended from paragraphs or block quotes will be gray.

One overlooked aspect of descendant selectors is that the degree of separation between two elements can be practically infinite. For example, if you write `ul em`, that syntax will select any `em` element descended from a `ul` element, no matter how deeply nested the `em` may be. Thus, `ul em` would select the `em` element in the following markup:

```

<ul>
<li>List item 1
<ol>
<li>List item 1-1</li>
<li>List item 1-2</li>
<li>List item 1-3
<ol>
<li>List item 1-3-1</li>
<li>List item <em>1-3-2</em></li>
<li>List item 1-3-3</li>
</ol></li>
<li>List item 1-4</li>
</ol></li>
</ul>

```

Selecting Children

In some cases, you don't want to select an arbitrarily descended element; rather, you want to narrow your range to select an element that is a child of another element. You might, for example, want to select a `strong` element only if it is a child (as opposed to a descendant) of an `h1` element. To do this, you use the child combinator, which is the greater-than symbol (`>`):

```
h1 > strong {color: red;}
```

This rule will make red the `strong` element shown in the first `h1` below, but not the second:

```

<h1>This is <strong>very</strong> important.</h1>
<h1>This is <em>really <strong>very</strong></em> important.</h1>

```

Read right to left, the selector `h1 > strong` translates as “selects any `strong` element that is a child of an `h1` element.” The child combinator is optionally surrounded by whitespace. Thus, `h1 > strong`, `h1> strong`, and `h1>strong` are all equivalent. You can use or omit whitespace as you like.

When viewing the document as a tree structure, it's easy to see that a child selector restricts its matches to elements that are directly connected in the tree. Figure 2-17 shows part of a document tree.

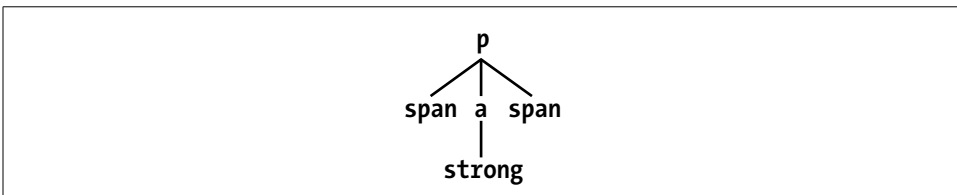


Figure 2-17. A document tree fragment

In this tree fragment, you can easily pick out parent–child relationships. For example, the `a` element is parent to the `strong`, but it is child to the `p` element. You could match elements in this fragment with the selectors `p > a` and `a > strong`, but not `p > strong`, since the `strong` is a descendant of the `p` but not its child.

You can also combine descendant and child combinations in the same selector. Thus, `table.summary td > p` will select any `p` element that is a child of a `td` element that is itself descended from a `table` element that has a `class` attribute containing the word `summary`.

Selecting Adjacent Sibling Elements

Let’s say you want to style the paragraph immediately after a heading or give a special margin to a list that immediately follows a paragraph. To select an element that immediately follows another element with the same parent, you use the *adjacent-sibling combinator*, represented as a plus symbol (+). Like the child combinator, the symbol can be surrounded by whitespace at the author’s discretion.

To remove the top margin from a paragraph immediately following an `h1` element, write:

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

The selector is read as, “selects any paragraph that immediately follows an `h1` element that shares a parent with the `p` element.”

In order to visualize how this selector works, it is easiest to once again consider a fragment of a document tree, shown in Figure 2-18.

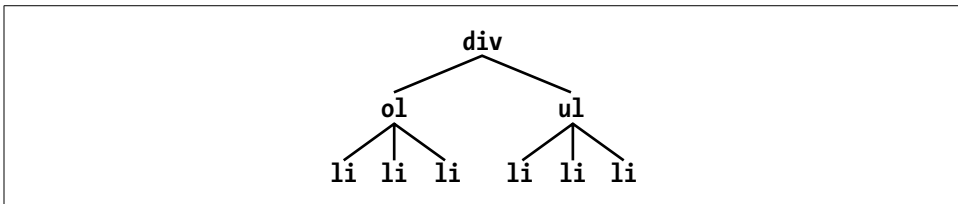


Figure 2-18. Another document tree fragment

In this fragment, a pair of lists descends from a `div` element, one ordered and the other not, each containing three list items. Each list is an adjacent sibling, and the list items themselves are also adjacent siblings. However, the list items from the first list are *not* siblings of the second, since the two sets of list items do not share the same parent element. (At best, they’re cousins.)

Remember that you can select the second of two adjacent siblings only with a single combinator. Thus, if you write `li + li {font-weight: bold;}`, only the second and third items in each list will be boldfaced. The first list items will be unaffected, as illustrated in Figure 2-19.

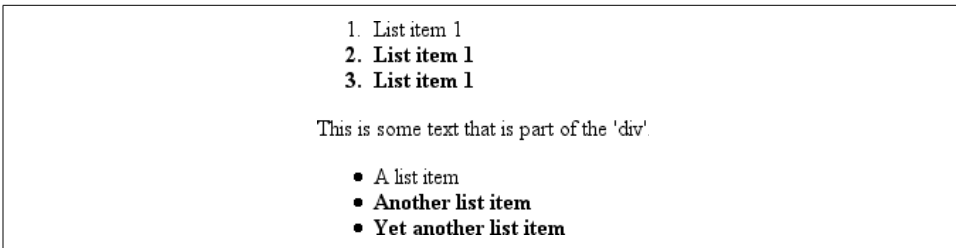


Figure 2-19. Selecting adjacent siblings

In order to work properly, CSS requires that the two elements appear in “source order.” In our example, an `ol` element is followed by a `ul` element. This would allow you to select the second element with `ol + ul`, but you cannot select the first using the same syntax. For `ul + ol` to match, an ordered list would have to immediately follow an unordered list.

In addition, text content between two elements does not prevent the adjacent-sibling combinator from working. Consider this markup fragment, whose tree view would be the same as that shown in Figure 2-18:

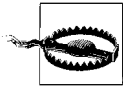
```
<div>
  <ol>
    <li>List item 1</li>
    <li>List item 1</li>
    <li>List item 1</li>
  </ol>
  This is some text that is part of the 'div'.
  <ul>
    <li>A list item</li>
    <li>Another list item</li>
    <li>Yet another list item</li>
  </ul>
</div>
```

Even though there is text between the two lists, you can still match the second list with the selector `ol + ul`. That’s because the intervening text is not contained with a sibling element, but is instead part of the parent `div`. If you wrapped that text in a paragraph element, it would then prevent `ol + ul` from matching the second list. Instead, you might have to write something like `ol + p + ul`.

As the following example illustrates, the adjacent-sibling combinator can be used in conjunction with other combinators:

```
html > body table + ul{margin-top: 1.5em;}
```

The selector translates as “selects any `ul` element that immediately follows a sibling `table` element that is descended from a `body` element that is itself a child of an `html` element.”



Internet Explorer for Windows through IE6, the most recent version as of this writing, does not support child and adjacent-sibling selectors.

Pseudo-Classes and Pseudo-Elements

Things get really interesting with *pseudo-class selectors* and *pseudo-element selectors*. These selectors let you assign styles to structures that don't necessarily exist in the document, or to phantom classes that are inferred by the state of certain elements, or even by the state of the document itself. In other words, the styles are applied to pieces of a document based on something other than the structure of the document, and in a way that cannot be precisely deduced simply by studying the document's markup.

It may sound like I'm applying styles at random, but I'm not. Instead, I'm applying styles based on somewhat ephemeral conditions that can't be predicted in advance. However, the circumstances under which the styles will appear are, in fact, well-defined. Think of it this way: during a sporting event, whenever the home team scores, the crowd will cheer. You don't know exactly when during a game the scoring will happen, but when it does, the crowd will cheer, just as predicted. The fact that you can't predict the moment of the cause doesn't make the effect any less expected.

Pseudo-Class Selectors

Let's begin by examining pseudo-class selectors since they're better supported by browsers and are therefore more widely used.

Consider the anchor element (`a`), which, in HTML and XHTML, establishes a link from one document to another. Anchors are always anchors, of course, but some anchors refer to pages that have already been visited, while others refer to pages that have yet to be visited. You can't tell the difference by simply looking at the HTML markup, because in the markup, all anchors look the same. The only way to tell which links have been visited is by comparing the links in a document to the user's browser history. So, there are actually two basic types of anchors: visited and unvisited. These types are known as *pseudo-classes*, and the selectors that use them are called pseudo-class selectors.

To better understand these classes and selectors, consider how browsers behave with regard to links. The Mosaic convention designated that links to pages you hadn't visited were blue, and links to already visited pages were red (the red became purple in succeeding browsers such as Internet Explorer). So, if you could insert classes into anchors, such that any anchor already visited would have a class of, say, "visited," then you could write a style to make such anchors red:

```
a.visited {color: red;}
```

```
<a href="http://www.w3.org/" class="visited">W3C Web site</a>
```

However, such an approach requires that the classes on anchors change every time you visit a new page, which is a little silly. Instead, CSS defines pseudo-classes that make the anchors to visited pages act as though they have classes of “visited”:

```
a:visited {color: red;}
```

Now any anchor that points to a visited page will be red, and you don’t even have to add class attributes to any of the anchors. Note the colon (:) in the rule. The colon separating the a and the visited is the calling card of a pseudo-class or pseudo-element. All pseudo-class and -element keywords are preceded by a colon.

Link pseudo-classes

CSS2.1 defines two pseudo-classes that apply only to hyperlinks. In HTML and XHTML 1.0 and 1.1, these would be any a elements having an href attribute; in XML languages, they would be any elements that act as links to another resource. Table 2-1 describes these two pseudo-classes.

Table 2-1. Link pseudo-classes

Name	Description
:link	Refers to any anchor that is a hyperlink (i.e., has an href attribute) and points to an address that has not been visited. Note that some browsers may incorrectly interpret :link to refer to any hyperlink, visited or unvisited.
:visited	Refers to any anchor that is a hyperlink to an already visited address.

The first of the pseudo-classes in Table 2-1 may seem a little bit redundant. After all, if an anchor hasn’t been visited, then it must be unvisited, right? If that’s the case, all we should need is the following:

```
a {color: blue;}  
a:visited {color: red;}
```

Although this format seems reasonable, it’s actually not quite enough. The first of the rules shown here applies not only to unvisited links, but also to target anchors such as this one:

```
<a name="section4">4. The Lives of Meerkats</a>
```

The resulting text would be blue because the a element will match the rule a {color: blue;}, as shown above. Therefore, in order to avoid applying your link styles to target anchors, use the :link pseudo-class:

```
a:link {color: blue;} /* unvisited links are blue */  
a:visited {color: red;} /* visited links are red */
```

As you may have already realized, the :link and :visited pseudo-class selectors are functionally equivalent to the body attributes link and vlink. Assume that an author wants all anchors to unvisited pages to be purple and anchors to visited pages to be silver. In HTML 3.2, this could be done as follows:

```
<body link="purple" vlink="silver">
```

In CSS, the same effect would be accomplished with:

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

In the case of the CSS pseudo-classes, of course, you can apply more than just colors. Let's say you want visited links to be italicized and to have, in addition to their silver color, a strikethrough line, as shown in Figure 2-20.



Figure 2-20. Applying multiple styles to a visited link

This is simply done with the following styles:

```
a:visited {color: silver; text-decoration: line-through; font-style: italic;}
```

This is a good place to revisit class selectors and show how they can be combined with pseudo-classes. For example, let's say you want to change the color of links that point outside your own site. If you assign a class to each of these anchors, it's easy:

```
<a href="http://www.mysite.net/">My home page</a>
<a href="http://www.site.net/" class="external">Another home page</a>
```

In order to apply different styles to the external link, all you need is a rule like this:

```
a.external:link, a.external:visited {color: maroon;}
```

This rule will make the second anchor in the preceding markup maroon, while the first anchor will remain the default color for hyperlinks (usually blue).

The same general syntax is used for ID selectors as well:

```
a#footer-copyright:link{font-weight: bold;}
a#footer-copyright:visited {font-weight: normal;}
```

Although `:link` and `:visited` are very useful, they're also static—they typically don't change the styling of a document after its initial display. Other pseudo-classes that aren't quite so static are available in CSS2.1; we'll review them next.

Dynamic pseudo-classes

CSS2.1 defines three pseudo-classes that can change a document's appearance as a result of user behavior. These dynamic pseudo-classes have traditionally been used to style hyperlinks, but the possibilities are much wider. Table 2-2 describes these pseudo-classes.

Table 2-2. Dynamic pseudo-classes

Name	Description
:focus	Refers to any element that currently has the input focus—i.e., can accept keyboard input or be activated in some way.
:hover	Refers to any element over which the mouse pointer is placed—e.g., a hyperlink over which the mouse pointer is hovering.
:active	Refers to any element that has been activated by user input—e.g., a hyperlink on which a user is clicking during the time the mouse button is held down.

As with :link and :visited, these pseudo-classes are most familiar in the context of hyperlinks. Many web pages have styles that look like this:

```
a:link {color: navy;}
a:visited {color: gray;}
a:hover {color: red;}
a:active {color: yellow;}
```

The first two rules use link pseudo-classes, and the last two employ dynamic pseudo-classes. :active is analogous to the alink attribute in HTML 3.2, although, as before, you can apply color changes and any style you like to active links.



The order of the pseudo-classes is more important than it might seem at first. The usual recommendation is “link-visited-hover-active,” although this has been modified to “link-visited-focus-hover-active.” The next chapter explains why this particular ordering is important and reasons you might choose to change or even ignore the recommended ordering.

Notice that the dynamic pseudo-classes can be applied to any element, which is good since it’s often useful to apply dynamic styles to elements that aren’t links. For example, using this markup:

```
input:focus {background: silver; font-weight: bold;}
```

you could highlight a form element that is ready to accept keyboard input, as shown in Figure 2-21.

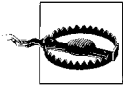
Name	<input type="text" value="Eric Meyer"/>
Title	<input type="text" value="Standards Evang"/>
E-mail	<input type="text"/>

Figure 2-21. Highlighting a form element that has focus

You can also perform some rather odd feats by applying dynamic pseudo-classes to arbitrary elements. You might decide to give users a “highlight” effect by way of the following:

```
body *:hover {background: yellow;}
```

This rule will cause any element that's descended from the body element to acquire a yellow background when it's in a hover state. Headings, paragraphs, lists, tables, images, and anything else found inside the body will be changed to have a yellow background. You could also change the font, put a border around the element being hovered, or anything else the browser will let you do.



Internet Explorer for Windows through IE6, the most recent version as of this writing, does not permit dynamic pseudo-classes to select any elements other than hyperlinks.

Real-world issues with dynamic styling

Dynamic pseudo-classes raise some interesting issues and peculiarities. For example, it's possible to set visited and unvisited links to one font size and make hovered links a larger size, as shown in Figure 2-22:

```
a:link, a:visited {font-size: 13px;}
a:hover {font-size: 20px;}
```



Figure 2-22. Changing layout with dynamic pseudo-classes

As you can see, the user agent increases the size of the anchor while the mouse pointer hovers over it. A user agent that supports this behavior must redraw the document while an anchor is in hover state, which could force a reflow of all the content that follows the link.

However, the CSS specifications state that user agents are not required to redraw a document once it's been rendered for initial display, so you can't absolutely rely on your intended effect actually happening. I strongly recommend that you avoid designs that depend on such behavior.

Selecting a first child

Another static pseudo-class, `:first-child`, is used to select elements that are the first children of other elements. This particular pseudo-class is easily misunderstood, so an extended example is in order. Consider the following markup:

```
<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
```

```

<li>Turn key <strong>clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p>
Do <em>not</em> push the brake at the same time as the accelerator.
</p>
</div>

```

In this example, the elements that are first children are the first p, the first li, and the strong and em elements. Given the following two rules:

```

p:first-child {font-weight: bold;}
li:first-child {text-transform: uppercase;}

```

you get the result shown in Figure 2-23.

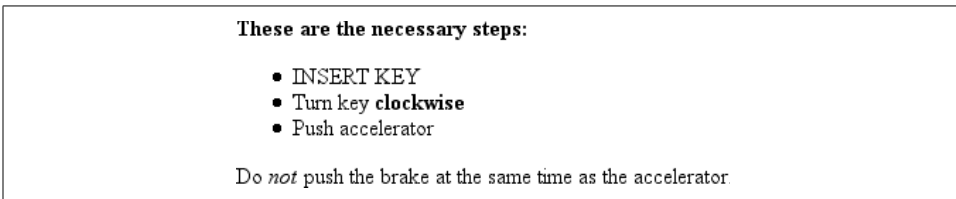


Figure 2-23. Styling first children

The first rule boldfaces any p element that is the first child of another element. The second rule uppercases any li element that is the first child of another element (which, in HTML, must be either an ol or a ul element).

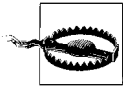
The most common error is to assume that a selector like p:first-child will select the first child of a p element. However, remember the nature of pseudo-classes, which is to attach a sort of phantom class to the element associated with the pseudo-class. If you were to add actual classes to the markup, it would look like this:

```

<div>
<p class="first-child">These are the necessary steps:</p>
<ul>
<li class="first-child">Insert key</li>
<li>Turn key <strong class="first-child">clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p>
Do <em class="first-child">not</em> push the brake at the same time as the
accelerator.
</p>
</div>

```

Therefore, if you want to select those em elements that are the first children of another element, you write em:first-child. This selector allows you to, for example, style the first list item in a list, the first paragraph of a div, or the first td in a table row.



Internet Explorer for Windows through IE6, the most recent version as of this writing, does not support `:first-child`.

Selecting based on language

For situations where you want to select an element based on its language, you can use the `:lang()` pseudo-class. In terms of its matching patterns, the `:lang()` pseudo-class is exactly like the `|=` attribute selector. For example, to italicize any element in the French language, you would write:

```
*:lang(fr) {font-style: italic;}
```

The primary difference between the pseudo-selector and the attribute selector is that the language information can be derived from a number of sources, some of which are outside the element itself. As CSS2.1 states:

In HTML, the language is determined by a combination of the “lang” attribute, the META element, and possibly by information from the protocol (such as HTTP headers). XML uses an attribute called `xml:lang`, and there may be other document language-specific methods for determining the language.

Therefore, the pseudo-class is a bit more robust than the attribute selector and is probably a better choice in most cases where language-specific styling is needed.

Combining pseudo-classes

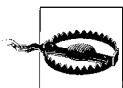
With CSS2.1, you can combine pseudo-classes in the same selector. For example, you can make unvisited links red when they’re hovered, but visited links maroon:

```
a:link:hover {color: red;}  
a:visited:hover {color: maroon;}
```

The order you specify doesn’t actually matter; you could also write `a:hover:link` to the same effect. It’s also possible to assign separate hover styles to unvisited and visited links that are in another language—for example, German:

```
a:link:hover:lang(de) {color: gray;}  
a:visited:hover:lang(de) {color: silver;}
```

Be careful not to combine mutually exclusive pseudo-classes. For example, a link cannot be both visited and unvisited, so `a:link:visited` doesn’t make any sense. User agents will most likely ignore such a selector and thus effectively ignore the entire rule.



Internet Explorer for Windows through IE6, which is the most recent version as of this writing, does not correctly recognize combined pseudo-classes. As with class-value combinations, it will pay attention to the last of the combined pseudo-classes. Thus, given `a:link:hover`, it will pay attention to the `:hover` but not the `:link` portion of the selector.

Pseudo-Element Selectors

Much as pseudo-classes assign phantom classes to anchors, pseudo-elements insert fictional elements into a document in order to achieve certain effects. Four pseudo-elements are defined in CSS2.1: styling the first letter, styling the first line, and styling before and after elements.

Styling the first letter

The first pseudo-element styles the first letter, and only that letter, of a block-level element:

```
p:first-letter {color: red;}
```

This rule causes the first letter of every paragraph to be colored red. Alternatively, you could make the first letter of each h2 twice as big as the rest of the heading:

```
h2:first-letter {font-size: 200%;}
```

The result of this rule is shown in Figure 2-24.

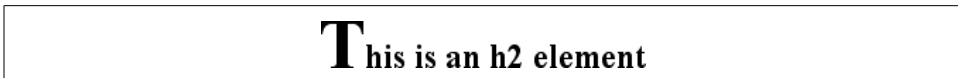


Figure 2-24. The `:first-letter` pseudo-element in action

As I mentioned, this rule effectively causes the user agent to respond to a fictional element that encloses the first letter of each h2. It would look something like this:

```
<h2><h2:first-letter>T</h2:first-letter>his is an h2 element</h2>
```

The `:first-letter` styles are applied only to the contents of the fictional element shown in the example. This `<h2:first-letter>` element does *not* appear in the document source. Instead, its existence is constructed on the fly by the user agent and is used to apply the `:first-letter` style(s) to the appropriate block of text. In other words, `<h2:first-letter>` is a pseudo-element. Remember, you don't have to add any new tags. The user agent will do it for you.

Styling the first line

Similarly, `:first-line` can be used to affect the first line of text in an element. For example, you could make the first line of each paragraph in a document purple:

```
p:first-line {color: purple;}
```

In Figure 2-25, the style is applied to the first displayed line of text in each paragraph. This is true no matter how wide or narrow the display region is. If the first line contains only the first five words of the paragraph, then only those five words will be purple. If the first line contains the first 30 words of the element, then all 30 will be purple.

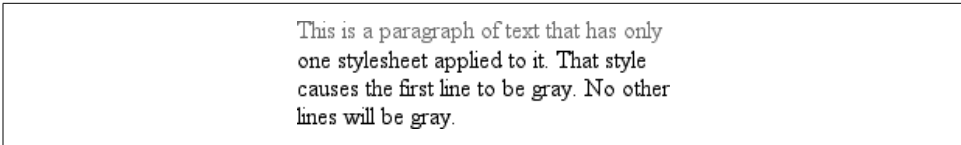


Figure 2-25. The `:first-line` pseudo-element in action

Since the text from “This” to “only” should be purple, the user agent employs a fictional markup that looks something like this:

```
<p><p:first-line>This is a paragraph of text that has only</p:first-line>
one stylesheet applied to it. That style
causes the first line to be purple. No other ...
```

If the first line of text were edited to include only the first seven words of the paragraph, then the fictional `</p:first-line>` would move back and occur just after the word “that.”

Restrictions on `:first-letter` and `:first-line`

In CSS2, the `:first-letter` and `:first-line` pseudo-elements can be applied only to block-level elements such as headings or paragraphs and not to inline-level elements such as hyperlinks. In CSS2.1, `:first-letter` applies to all elements. There are also limits on the CSS properties that may be applied to `:first-line` and `:first-letter`. Table 2-3 displays the limits.

Table 2-3. Properties permitted on pseudo-elements

<code>:first-letter</code>	<code>:first-line</code>
All font properties	All font properties
All color properties	All color properties
All background properties	All background properties
All margin properties	word-spacing
All padding properties	letter-spacing
All border properties	text-decoration
text-decoration	vertical-align
vertical-align (if float is set to none)	text-transform
text-transform	line-height
line-height	clear (CSS2 only; removed in CSS2.1)
float	text-shadow (CSS2 only)
letter-spacing (added in CSS2.1)	
word-spacing (added in CSS2.1)	
clear (CSS2 only; removed in CSS2.1)	
text-shadow (CSS2 only)	

In addition, all pseudo-elements must be placed at the very end of the selector in which they appear. Therefore, it would not be legal to write `p:first-line em` since the pseudo-element comes before the subject of the selector (the subject is the last element listed). The same rule applies to the other two pseudo-elements CSS2.1 defines.

Style before and after elements

Let's say you want to preface every `h2` element with a pair of silver square brackets as a typographical effect:

```
h2:before {content: "]]"; color: silver;}
```

CSS2.1 lets you insert *generated content*, and then style it directly using the pseudo-elements `:before` and `:after`. An example is shown in Figure 2-26.

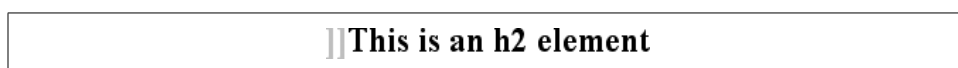


Figure 2-26. Inserting content before an element

The pseudo-element is used to insert the generated content and to style it. To place content after an element, use the pseudo-element `:after`. You could end your documents with an appropriate finish:

```
body:after {content: " The End.";}
```

Generated content is a separate subject, and the entire topic (including more detail on `:before` and `:after`) is covered in more detail in Chapter 12.

Summary

By using selectors based on the document's language, authors can create CSS rules that apply to a large number of similar elements just as easily as they can construct rules that apply in very narrow circumstances. The ability to group together both selectors and rules keeps style sheets compact and flexible, which incidentally leads to smaller file sizes and faster download times.

Selectors are the one thing that user agents usually have to get right because the inability to correctly interpret selectors pretty much prevents a user agent from using CSS at all. On the flip side, it's crucial for authors to correctly write selectors because errors can prevent the user agent from applying the styles as intended. An integral part of correctly understanding selectors and how they can be combined is an understanding of how selectors relate to document structure and how mechanisms—such as inheritance and the cascade itself—come into play when determining how an element is to be styled. This is the subject of the next chapter.