CHAPTER

5

Managing Multiple Styles: The Cascade

As you create increasingly complex style sheets, you'll sometimes wonder why a particular element on a page looks the way it does. CSS's inheritance feature, as discussed in Chapter 4, creates the possibility that any tag on a page is potentially affected by any of the tags that wrap around it. For example, the `<body>` tag can pass properties on to a paragraph, and a paragraph may pass its own formatting instructions on to a link within the paragraph. In other words, that link can inherit CSS properties from *both* the `<body>` and the `<p>` tag—essentially creating a kind of Frankenstyle that combines parts of two different CSS styles.

Then there are times when styles collide—the same CSS property is defined in multiple styles, all applying to a particular element on the page (for example, a `<p>` tag style in an external style sheet and another `<p>` tag style in an internal style sheet). When that happens, you can see some pretty weird stuff, like text that appears bright blue, even though you specifically applied a class style with the text color set to red. Fortunately, there's actually a system at work: a basic CSS mechanism known as the *cascade*, which governs how styles interact and which styles get precedence when there's a conflict.

NOTE This chapter deals with issues that arise when you build complex style sheets that rely on inheritance and more sophisticated types of selectors like descendent selectors (page 88). The rules are all pretty logical, but they're about as fun to master as the tax code. If that's got your spirits sagging, consider skipping the details and doing the tutorial on page 117 to get a taste of what the cascade is and why it matters. Or jump right to the next chapter, which explores fun and visually exciting ways to format text. You can always return to this chapter later, after you've mastered the basics of CSS.

How Styles Cascade

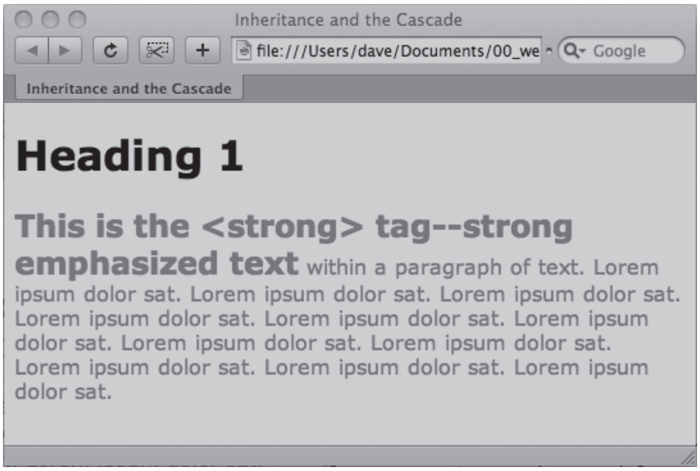
The *cascade* is a set of rules for determining which style properties get applied to an element. It specifies how a web browser should handle multiple styles that apply to the same tag and what to do when CSS properties conflict. Style conflicts happen in two cases: through inheritance when the same property is inherited from multiple ancestors, and when one or more styles apply to the same element (maybe you've applied a class style to a paragraph and also created a `<p>` tag style, so both styles apply to that paragraph).

Inherited Styles Accumulate

As you read in the last chapter, CSS inheritance ensures that related elements—like all the words inside a paragraph, even those inside a link or another tag—share similar formatting. It spares you from creating specific styles for each tag on a page. But since one tag can inherit properties from *any* ancestor tag—a link, for example, inheriting the same font as its parent `<p>` tag—determining why a particular tag is formatted one way can be a bit tricky. Imagine a font family applied to the `<body>` tag, a font size applied to a `<p>` tag, and a font color applied to an `<a>` tag. Any `<a>` tag inside of a paragraph would inherit the font from the body and the size from the paragraph. In other words, the inherited styles combine to form a hybrid style.

The page shown in Figure 5-1 has three styles: one for the `<body>`, one for the `<p>` tag, and one for the `<strong>` tag. The CSS looks like this:

```
body { font-family: Verdana, Arial, Helvetica, sans-serif; }
p { color: #F30; }
strong { font-size: 24px; }
```


Inheritance and the Cascade
file:///Users/dave/Documents/00_we
Google
Inheritance and the Cascade

Heading 1

This is the <strong> tag--strong emphasized text within a paragraph of text. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat. Lorem ipsum dolor sat.

FIGURE 5-1

Thanks to inheritance, it's possible for multiple styles to affect the appearance of one tag. Here the `<strong>` tag has a specific color, font family, and font size, even though only a single property is applied directly to that tag. The other two formatting options were inherited from the tag's ancestors: the `<body>` and the `<p>` tags.

The `<strong>` tag is nested inside a paragraph, which is inside the `<body>` tag. That `<strong>` tag inherits from both of its ancestors, so it inherits the font-family property from the body and the color property from its parent paragraph. In addition, the `<strong>` tag has a bit of CSS applied directly to it—a 24px font size. The final appearance of the tag is a combination of all three styles. In other words, the `<strong>` tag appears exactly as if you'd created a style like this:

```
strong {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  color: #F30;
  font-size: 24px;
}
```

Nearest Ancestor Wins

In the previous example, various inherited and applied tags smoothly combined to create an overall formatting package. But what happens when inherited CSS properties conflict? Think about a page where you've set the font color for the body tag to red and the paragraph tag to green. Now imagine that within one paragraph, there's a `<strong>` tag. The `<strong>` tag inherits from both the `body` and `p` tag styles, so is the text inside the `<strong>` tag red or green? Ladies and gentleman, we have a winner: the green from the paragraph style. That's because the web browser obeys the style that's closest to the tag in question.

In this example, any properties inherited from the `<body>` tag are rather generic. They apply to all tags. A style applied to a `<p>` tag, on the other hand, is more narrowly defined. Its properties apply only to `<p>` tags and the tags inside them.

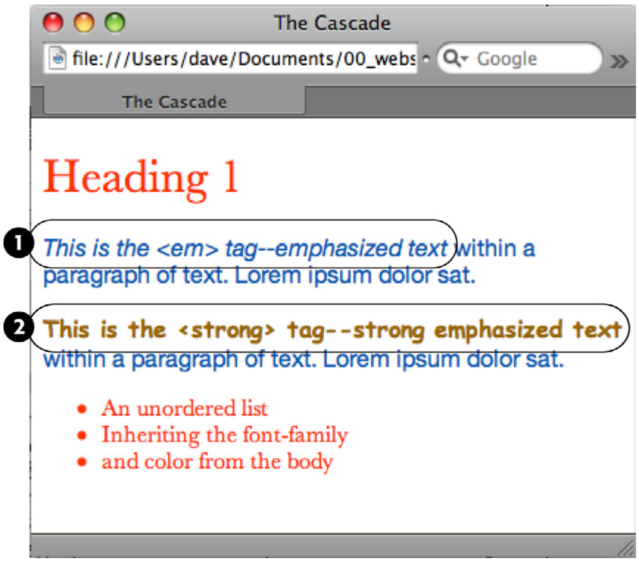
In a nutshell, if a tag doesn't have a specific style applied to it, then, in the case of any conflicts from inherited properties, the nearest ancestor wins (see Figure 5-2, number 1).

Here's one more example, just to make sure the concept sinks in. If a CSS style defining the color of text were applied to a `<table>` tag, and another style defining a *different* text color were applied to a `<td>` tag inside that table, then tags inside that table cell (`<td>`) such as a paragraph, headline, or unordered list would use the color from the `<td>` style, since it's the closest ancestor.

The Directly Applied Style Wins

Taking the "nearest ancestor" rule to its logical conclusion, there's one style that always becomes king of the CSS family tree—any style applied directly to a given tag. Suppose a font color is set for the body, paragraph, *and* strong tags. The paragraph style is more specific than the body style, but the style applied to the `<strong>` tag is more specific than either one. It formats the `<strong>` tags and only the `<strong>` tags, overriding any conflicting properties inherited from the other tags (see Figure 5-2, number 2). In other words, properties from a style specifically applied to a tag beat out any inherited properties.

This rule explains why some inherited properties don't appear to inherit. A link inside a paragraph whose text is red still appears browser-link blue. That's because browsers have their own predefined style for the <a> tag, so an inherited text color won't apply.



Heading 1

This is the <em> tag--emphasized text within a paragraph of text. Lorem ipsum dolor sat.

This is the <strong> tag--strong emphasized text within a paragraph of text. Lorem ipsum dolor sat.

- An unordered list
- Inheriting the font-family
- and color from the body

FIGURE 5-2

Here's how web browsers figure out which properties to display when inherited properties conflict: The <em> tag in the first paragraph (1) inherits the font family and color from both the <body> tag and the paragraph. But since the body and paragraph have different fonts and colors applied to them, the <em> tag uses the font and color specified for its closest ancestor—the <p> tag. When a style applies directly to a tag—the font-family and color are specified for the <strong> tag (2)—browsers ignore conflicting inherited properties.

NOTE You can learn how to overcome preset styles for the <a> tag and change link colors to your heart's content. See page 115.

One Tag, Many Styles

Inheritance is one way that a tag can be affected by multiple styles. But it's also possible to have multiple styles apply *directly* to a given tag. For example, say you have an external style sheet with a <p> tag style and attach it to a page that has an internal style sheet that *also* includes a <p> tag style. And just to make things really interesting, one of the <p> tags on the page has a class style applied to it. So for that one tag, three different styles directly format it. Which style—or *styles*—should the browser obey?

The answer: It depends. Based on the types of styles and the order in which they're created, a browser may apply one or more of them at once. Here are a few situations in which multiple styles can apply to the same tag:

- **The tag has both a tag selector and a class style applied to it.** For example, a tag style for the <h2> tag, a class style named `.leadHeadline` and this HTML: `<h2 class="leadHeadline">Your Future Revealed!</h2>`. Both styles apply to this <h2> tag.

NOTE Hold onto your hat if you're worried about what happens when these multiple styles conflict; details to follow.

- **The same style name appears more than once in the style sheet.** There could be a group selector (page 84), like `.leadHeadline`, `.secondaryHeadline`, `.newsHeadline`, and the class style `.leadHeadline` in the same style sheet. Both of these rules define how any element with a class of `leadHeadline` looks.
- **A tag has both a class and an ID style applied to it.** Maybe it's an ID named `#banner`, a class named `.news`, and this HTML: `<div id="banner" class="news">`. Properties from both the `banner` and `news` styles apply to this `<div>` tag.
- **There's more than one style sheet containing the *same* style name attached to a page.** The same-named styles can arrive in an external style sheet and an internal style sheet. Or, the same style can appear in multiple external style sheets that are all linked to the same page.
- **There are complex selectors targeting the same tag.** This situation is common when you use descendent selectors (page 88). For example, say you have a div tag in a page (like this: `<div id="mainContent">`), and inside the div is a paragraph with a class applied to it: `<p class="byline">`. The following selectors apply to this paragraph:

  ```
  #mainContent p
  #mainContent .byline
  p.byline
  .byline
  ```

If more than one style applies to a particular element, then a web browser combines the properties of all those styles, *as long as they don't conflict*. An example will make this concept clearer. Imagine you have a paragraph that lists the name of the web page's author, including a link to his email address. The HTML might look like this:

```
<p class="byline">Written by <a href="mailto:jean@cosmofarmer.com">Jean Graine
de Pomme</a></p>
```

Meanwhile, the page's style sheet has three styles that format the link:

```
a { color: #6378df; }
p a { font-weight: bold; }
.byline a { text-decoration: none; }
```

The first style turns all `<a>` tags powder blue; the second style makes all `<a>` tags that appear inside a `<p>` tag bold; and the third style removes the underline from any links that appear inside an element with the `byline` class applied to it.

All three styles apply to that very popular `<a>` tag, but since none of the properties are the same, there are no conflicts between the rules. The situation is similar to the inheritance example (page 104): the styles combine to make one überstyle containing all three properties, so this particular link appears powder blue, bold, *and* underline-free.

NOTE Your head will really start to ache when you realize that this particular link's formatting can also be affected by inherited properties. For example, it would inherit any font family that's applied to the paragraph. A few tools can help sort out what's going on in the cascade. (See the box on page 110.)

Specificity: Which Style Wins

The previous example is pretty straightforward. But what if the three link styles above each had a *different* font specified for the `font-family` property? Which of the three fonts would a web browser pay attention to?

As you know if you've been reading carefully so far, the cascade provides a set of rules that helps a web browser sort out any property conflicts; namely, *properties from the most specific style win*. But as with the styles listed above, sometimes it's not clear which style is most specific. Thankfully, CSS provides a formula for determining a style's *specificity* that's based on a value assigned to the style's selector—a tag selector, class selector, ID selector, and so on. Here's how the system works:

- A tag selector is worth **1 point**.
- A class selector is worth **10 points**.
- An ID selector is worth **100 points**.
- An inline style (page 44) is worth **1,000 points**.

NOTE The math involved in calculating specificity is actually a bit more complicated than described here. But this formula works in all but the weirdest cases. To read how web browsers actually calculate specificity, visit *www.w3.org/TR/css3-selectors/#specificity*.

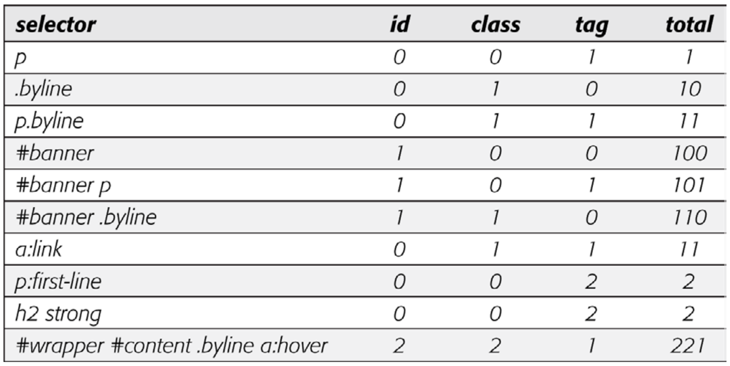
The bigger the number, the greater the specificity. So say you create the following three styles:

- A tag style for the `<img>` tag (specificity = 1)
- A class style named `.highlight` (specificity = 10)
- An ID style named `#logo` (specificity = 100)

Then, say your web page has this HTML: `<img id="logo" class="highlight" src="logo.gif" />`. If you define the same property—such as the border property—in all three styles, then the value from the ID style (`#logo`) always wins out.

NOTE A pseudo-element (like `::first-line` for example) is treated like a tag selector and is worth 1 point. A pseudo-class (`:link`, for example) is treated like a class and is worth 10 points. (See page 68 for the deal on these pseudo-things.)

Since descendent selectors are composed of several selectors—`#content p`, or `h2 strong`, for example—the math gets a bit more complicated. The specificity of a descendent selector is the total value of all of the selectors listed (see Figure 5-3).



selector	id	class	tag	total
p	0	0	1	1
.byline	0	1	0	10
p.byline	0	1	1	11
#banner	1	0	0	100
#banner p	1	0	1	101
#banner .byline	1	1	0	110
a:link	0	1	1	11
p:first-line	0	0	2	2
h2 strong	0	0	2	2
#wrapper #content .byline a:hover	2	2	1	221

FIGURE 5-3

When more than one style applies to a tag, a web browser must determine which style should "win out" in case style properties conflict. In CSS, a style's importance is known as specificity and is determined by the type of selectors used when creating the style. Each type of selector has a different value, and when multiple selector types appear in one style—for example, the descendent selector `#banner p`*—the values of all the selectors used are added up.*

NOTE Inherited properties don't have any specificity. So even if a tag inherits properties from a style with a large specificity—like `#banner`—those properties will always be overridden by a style that directly applies to the tag.

The Tiebreaker: Last Style Wins

It's possible for two styles with conflicting properties to have the same specificity. ("Oh brother, when will it end?" Soon, comrade, soon. The tutorial is coming up.) A specificity tie can occur when you have the same selector defined in two locations. You may have a `<p>` tag selector defined in an internal style sheet and an external style sheet. Or two different styles may simply have equal specificity values. In case of a tie, the style appearing last in the style sheet wins.

Here's a tricky example using the following HTML:

```
<p class="byline">Written by <a class="email" href="mailto:jean@cosmofarmer.
com">Jean Graine de Pomme</a></p>
```

In the style sheet for the page containing the above paragraph and link, you have two styles:

```
p .email { color: blue; }
.byline a { color: red; }
```

Both styles have a specificity of 11 (10 for a class name and 1 for a tag selector) and both apply to the `<a>` tag. The two styles are tied. Which color does the browser use to color the link in the above paragraph? Answer: Red, since it's the second (and last) style in the sheet.

FREQUENTLY ASKED QUESTION

Get a Little Help

My head hurts from all of this. Isn't there some tool I can use to help me figure out how the cascade is affecting my web page?

Trying to figure out all the ins and outs of inherited properties and conflicting styles confuses many folks. Furthermore, doing the math to figure out a style's specificity isn't your average web designer's idea of fun, especially when there are large style sheets with lots of descendent selectors.

All current web browsers have built-in help in the form of an inspector. The fastest way to inspect an element on a page and all the CSS that affects it is to right-click (Control-click on a Mac) the element (the headline, link, paragraph, or image), and choose Inspect Element from the contextual menu. A panel will open (usually beneath the web page) showing the page's HTML, with your selected element's HTML highlighted. (To get this to work in Safari, you first need to turn on the Show Develop Menu option in the Preferences window→Advanced.)

On the right side of the panel, you'll see the styles applied to the element. There's usually a "computed" style—the sum total of all the CSS properties applied to the element through inheritance and the cascade or the element's "Frankenstyle." Below that you'll find the style rules that apply to the element, listed in order of most specific (at the top) to least specific (at the bottom).

In the listing of styles, you'll probably see some properties crossed out—this indicates that the property either doesn't apply to the element or that it's been overridden by a more specific style. For a couple of short tutorials on using Chrome's Developer's Tools for analyzing CSS, visit *https://developers.google.com/chrome-developer-tools/docs/elements-styles* and *http://webdesign.tutsplus.com/tutorials/workflow-tutorials/faster-htmlcss-workflow-with-chrome-developer-tools/*.

Now suppose the style sheet looked like this instead:

```
.byline a {color: red; }
p. email { color: blue; }
```

In this case, the link would be blue. Since `p .email` appears after `.byline` in the style sheet, its properties win out.

What happens if you've got conflicting rules in an external and an internal style sheet? In that case, the placement of your style sheets (within your HTML file) becomes very important. If you first add an internal style sheet by using the `<style>` tag (page 45) and *then* attach an external style sheet farther down in the HTML by using the `<link>` tag (page 48), then the style from the external style sheet wins. (In effect, it's the same principle at work that you just finished reading about: *The style appearing last wins.*) The bottom line: Be consistent in how you place external style sheets. It's best to list any external style sheets first, and only use an internal style sheet when you absolutely need one or more styles to apply to a single page.

NOTE Any external style sheets attached with the `@import` rule have to appear before internal styles within a `<style>` tag, and before any styles in an external style sheet. See page 48 for more information on external and internal style sheets.

TROUBLESHOOTING MOMENT

Overruling Specificity

CSS provides a way of overruling specificity entirely. You can use this trick when you absolutely, positively want to make sure that a particular property can't be overridden by a more specific style. Simply insert `!important` after any property to shield it from specificity-based overrides.

For example, consider the two following styles:

```
.nav a { color: red; }
a { color: teal !important; }
```

Normally, a link inside an element with the class of nav would be colored red since the `.nav a` style is more specific than the a tag style.

However, including `!important` after a property value means that property always wins. So in the above example, all links on the page—including those inside an element with the nav class—are teal.

Note that `!important` works on an individual property, not an entire style, so you need to add `!important` to the end of each property you wish to make invincible. Finally, when two styles both have `!important` applied to the same property, the more specific style's `!important` rule wins.

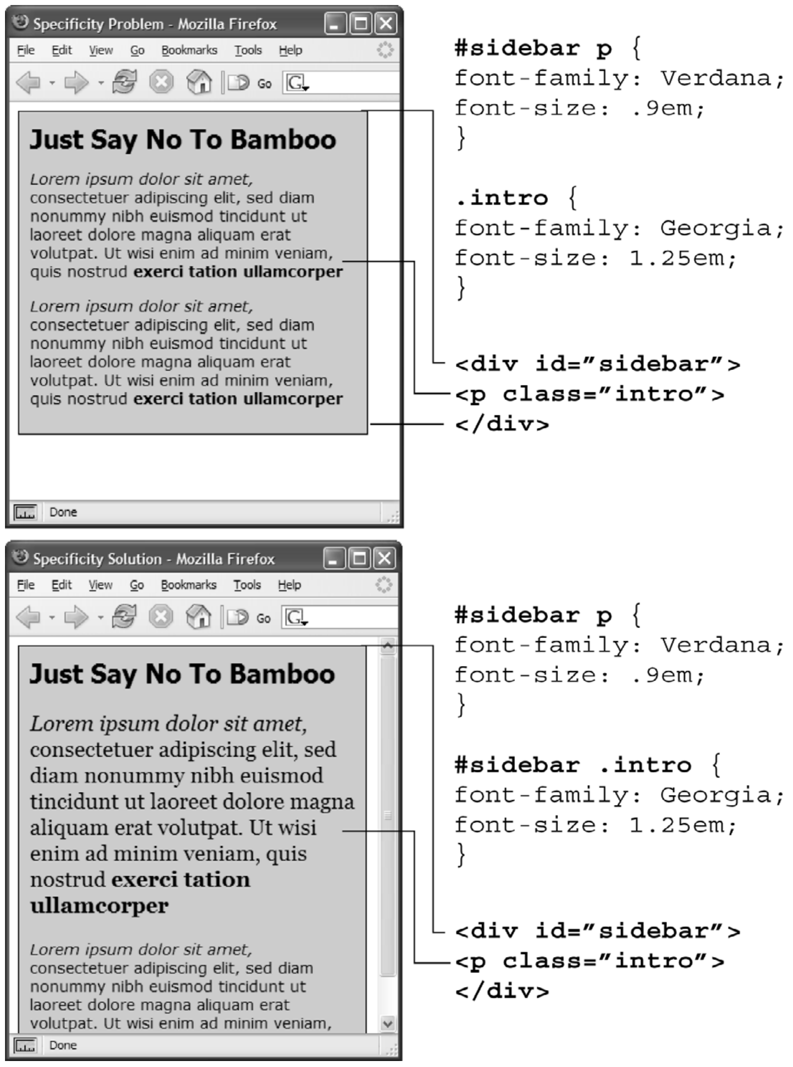
Controlling the Cascade

As you can see, the more CSS styles you create, the greater the potential for formatting snafus. For example, you may create a class style specifying a particular font and font size, but when you apply the style to a paragraph, nothing happens! This kind of problem is usually related to the cascade. Even though you may think that directly applying a class to a tag should apply the class's formatting properties, it may not if there's a style with greater specificity.

You have a couple of options for dealing with this kind of problem. First, you can use `!important` (as described in the box above) to make sure a property *always* applies. The `!important` approach is a bit heavy-handed, though, since it's hard to predict that you'll never, ever, want to overrule an `!important` property someday. Read on for two other cascade-tweaking solutions.

Changing the Specificity

The top picture in Figure 5-4 is an example of a specific tag style losing out in the cascade game. Fortunately, most of the time, you can easily change the specificity of one of the conflicting styles and save `!important` for real emergencies. In Figure 5-4 (top), two styles format the first paragraph. The class style—`.intro`—isn't as specific as the `#sidebar p` style, so `.intro`'s properties don't get applied to the paragraph. To increase the specificity of the class, add the ID name to the style: `#sidebar .intro`.


Specificity Problem - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Done

Just Say No To Bamboo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud **exerci tation ullamcorper**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud **exerci tation ullamcorper**

```
#sidebar p {
font-family: Verdana;
font-size: .9em;
}

.intro {
font-family: Georgia;
font-size: 1.25em;
}

<div id="sidebar">
<p class="intro">
</div>
```

Specificity Solution - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Done

Just Say No To Bamboo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud **exerci tation ullamcorper**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam,

```
#sidebar p {
font-family: Verdana;
font-size: .9em;
}

#sidebar .intro {
font-family: Georgia;
font-size: 1.25em;
}

<div id="sidebar">
<p class="intro">
</div>
```

FIGURE 5-4

Even though a class is applied to a specific tag—like the first paragraph in the top image—its properties may not always have an effect. In this case, the paragraph is inside a `<div>` tag with an ID of `#sidebar`, so the descendent selector `#sidebar p` is more specific than the `.intro` class. The solution: Make the `.intro` class more specific by adding the ID before it—`#sidebar p.intro`—as in the bottom example.

However, simply tacking on additional selectors to make a style's properties "win" can lead to what's been called *specificity wars* where you end up with style sheets containing very long and convoluted style names like: `#home #main #story h1`. In fact, as you'll read on page 114, you should try to avoid these types of styles and aim to keep your selectors as short as possible.

NOTE If you're into math, the `#sidebar p` style has a specificity of 101 (100 for the ID, and 1 for the tag selector), while the `.intro` style has a specificity of 10 (10 points for a class selector). Since 101 is greater than 10, `#sidebar p` takes precedence. Changing `.intro` to `#sidebar .intro` changes its specificity to 110.

Selective Overriding

You can also fine-tune your design by *selectively* overriding styles on certain pages. Say you've created an external style sheet named *styles.css* that you've attached to each page in your site. This file contains the general look and feel for your site—the font and color of `<h1>` tags, how form elements should look, and so on. But maybe on your home page, you want the `<h1>` tag to look slightly different than the rest of the site—bolder and bigger, perhaps. Or the paragraph text should be smaller on the home page, so you can wedge in more information. In other words, you still want to use *most* of the styles from the *styles.css* file, but you simply want to override a few properties for some of the tags (`<h1>`, `<p>`, and so on).

One approach is to simply create an internal style sheet listing the styles that you want to override. Maybe the *styles.css* file has the following rule:

```
h1 {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 24px;
  color: #000;
}
```

You want the `<h1>` tag on the home page to be bigger and red. So just add the following style in an internal style sheet on the home page:

```
h1 {
  font-size: 36px;
  color: red;
}
```

In this case, the `<h1>` tag on the home page would use the font Arial (from the external style sheet) but would be red and 36 pixels tall (from the internal style).

TIP Make sure you attach the external style sheet *before* the internal style sheet in the `<head>` section of the HTML. This ensures that the styles from the internal style sheet win out in cases where the specificity of two styles are the same, as explained on page 114.

Another approach would be to create one more external style sheet—*home.css* for example—that you attach to the home page in addition to the *styles.css* style sheet. The *home.css* file would contain the style names and properties that you want to overrule from the *styles.css* file. For this to work, you need to make sure the *home.css* file appears *after* the *styles.css* file in the HTML, like so:

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

TIP Another way to fine-tune designs on a page-by-page basis is to use different class names for the <body> tag of different types of pages—for example .review, .story, .home—and then create descendent selectors to change the way tags on these types of pages look. This technique is discussed on page 67.

Avoiding Specificity Wars

As mentioned on page 61, many web designers these days avoid ID selectors in favor of classes. One reason: ID selectors are very powerful, and, therefore, require more power to override. This often leads to specificity wars in which style sheets get loaded with unnecessarily long-winded and complicated selectors. This problem is best explained by example. Say, for instance, your page has this snippet of HTML:

```
<div id="article">
<p>A paragraph</p>
<p>Another paragraph</p>
<p class="special">A special paragraph</p>
</div>
```

You decide that you want to color the paragraphs inside the article div red, so you create a descendent selector like this:

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

But you want that one paragraph with the class of special to be blue. If you simply create a class selector, you won't get what you want.

```
.special { color: blue; }
```

As you read on page 108, when determining which properties to apply to a tag, a web browser uses a simple mathematical formula to deal with style conflicts: browsers give an ID selector a value of 100, a class selector a value of 10, and a tag selector a value of 1. Because the selector #article p is composed of one ID and one element (a total of 101 specificity points), it overrides the simple class style—forcing you to change the selector:

```
#article .special {color: blue; }
```

Unfortunately, this change causes two more problems. First, it makes the selector longer, and second, now that blue color is applied only when the special class appears inside something with an ID of article. In other words, if you copy the HTML <p class="special">A special paragraph</p> and paste it elsewhere in the page, it will no longer be blue. In other words, the use of the ID makes your selectors both longer and less useful.

Now look what happens if you simply replace all IDs with classes. The previous HTML would change to:

```
<div class="article">
<p>A paragraph</p>
<p>Another paragraph</p>
<p class="special">A special paragraph</p>
</div>
```

And you could change the CSS to this:

```
.article p { color: red; }
p.special { color: blue; }
```

The first style—`.article p`—is a descendent selector worth 11 points. The second style `p.special` is also worth 11 points (one tag and one class) and means "apply the following properties to any paragraph with the special class." Now if you cut that HTML and paste it anywhere else on the page, you'd get the blue styling you're after.

This is just one example, but it's not hard to find style sheets with ridiculously long selectors like `#home #article #sidebar #legal p` and `#home #article #sidebar #legal p.special`.

There's basically no reason to use IDs. They don't provide anything that you can't have with a simple class selector or tag selector, and their powerful specificity can only lead you to unnecessarily complex style sheets.

NOTE For a more detailed discussion of why you should avoid ID selectors, visit *http://csswizardry.com/2011/09/when-using-ids-can-be-a-pain-in-the-class*.

Starting with a Clean Slate

As discussed on page 96, browsers apply their own styles to tags: for example, `<h1>` tags are bigger than `<h2>` tags, and both are bold, while paragraph text is smaller and isn't bold; links are blue and underlined; and bulleted lists are indented. There's nothing in the HTML standard that defines any of this formatting: Web browsers just add this formatting to make basic HTML more readable. However, even though browsers treat all tags roughly the same, they don't treat them identically.

For example, Safari and Firefox use the `padding` property to indent bulleted lists, but Internet Explorer uses the `margin` property. Likewise, you'll find subtle differences in the size of tags across browsers and an altogether confusing use of margins among the most common web browsers. Because of these inconsistencies, you can run into problems where, for instance, Firefox adds a top margin, while Internet Explorer doesn't. These types of problems aren't your fault—they stem from differences in the built-in browser styles.

To avoid cross-browser inconsistencies, it's a good idea to start a style sheet with a clean slate. In other words, erase the built-in browser formatting and supply your own. The concept of erasing browser styling is called *CSS reset*. This section gives you a working introduction.

In particular, there's a core set of styles you should include at the top of your style sheets. These styles set a baseline for properties that commonly are treated differently across browsers.

Here's a bare-bones CSS reset:

```
html, body, div, span, object, iframe, h1, h2, h3, h4, h5, h6, p, blockquote,
pre, a, abbr, acronym, address, big, cite, code, del, dfn, em, img, ins, kbd,
q, s, samp,small, strike, strong, sub, sup, tt, var, b, u, i, center, dl, dt,
dd, ol, ul, li, fieldset, form, label, legend, table, caption, tbody, tfoot,
thead, tr, th, td, article, aside, canvas, details, embed, figure, figcaption,
footer, header, hgroup, menu, nav, output, ruby, section, summary, time, mark,
audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    vertical-align: baseline;
}

article, aside, details, figcaption, figure, footer, header, hgroup, menu,
nav, section {
    display: block;
}
body {
    line-height: 1.2;
}
ol {
    padding-left: 1.4em;
    list-style: decimal;
}
ul {
    padding-left: 1.4em
    list-style: square;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
```

NOTE The above CSS reset is adapted from Eric Meyer's well-known and influential CSS reset, which you can find at *http://meyerweb.com/eric/tools/css/reset*.

The first style is a very long group selector (page 84) that takes the most common tags and *zeros them out*—removing all the padding and margins, setting their base text size to 100%, and removing bold text formatting. This step makes your tags look pretty much identical (see Figure 5-5), but that's the point—you want to start at zero and then add your own formatting so that all browsers apply a consistent look to your HTML.

The second selector (`article, aside, details...`) is another group selector that helps older browsers correctly display the new HTML5 tags. The third selector (body) style sets a consistent `line-height` (space between lines in a paragraph). You'll learn about the `line-height` property in the next chapter.

NOTE You don't have to type all this code yourself. You'll find a file named *reset.css* in the *05* tutorial folder at *www.sawmac.com/css3* that contains a basic CSS reset file. Just copy the styles from this file and paste them into your own style sheets. Another approach to resets (discussed on page 551) is available in the Chapter 17 tutorial files inside the *17* folder.

The fourth and fifth styles (the `ol` and `ul` tag styles) set a consistent left margin and style (page 173 introduces list styling), and the last style makes adding borders to table cells easier (you'll learn why this style is useful on page 380).

Tutorial: The Cascade in Action

In this tutorial, you'll see how styles interact and how they can sometimes conflict to create unexpected results. First, you'll look at a basic page that has the CSS reset styles mentioned above plus a couple of other styles that provide some simple layout. Then, you'll create two styles and see how some properties are inherited and how others are overruled by the cascade. Then, you'll see how inheritance affects tags on a page and how a browser resolves any CSS conflicts. Finally, you'll learn how to troubleshoot problems created by the cascade.

To get started, you need to download the tutorial files located on this book's companion website at *www.sawmac.com/css3*. Click the tutorial link and download the files. All of the files are enclosed in a zip archive, so you'll need to unzip them first. (Go to the website for detailed instructions on unzipping the files.) The files for this tutorial are contained inside the folder named *05*.

Resetting CSS and Styling from Scratch

First, take a look at the page you'll be working on.

1. **In a web browser, open the file named *cascade.html* located in the *05* tutorial folder (see Figure 5-5).**

   The page doesn't look like much—two columns, one with a blue background and a lot of same-looking text. There are a few styles already applied to this file, so open the CSS up in a text editor and have a look.

2. **Using your favorite text or web page editor, open the file *styles.css* located in the *05* folder.**

   This file is the external style sheet that the *cascade.html* file uses. It has several styles already in it—the first group is the CSS reset styles discussed on the previous page. They eliminate the basic browser styles, which is why all of the text currently looks the same. (You'll create your own styles to make this page look great soon.)