IN THE PREVIOUS CHAPTER, I showed you how XHTML markup provides a structural hierarchy to your document. We saw that elements have a basic set of styles applied to them by the browser, and that each element is either displayed as a block (stacked) or inline (side-by-side) element. If correctly marked up, the document will then automatically lay out in a useable way down the page—an effect know as document flow. However, “useable” doesn’t mean “beautiful,” so CSS enables creative people like you to change the browser’s default styles and apply additional styles to create a more functional and aesthetically pleasing result for your visitors. This chapter will teach you the mechanics of CSS, and by the end of this chapter, you’ll be ready to create your own styles for the piece of sample XHTML markup we studied in Chapter 1.
The Three Ways to Style Your Document

There are three ways to add CSS to your Web pages: *inline*, *embedded*, and *linked* from a separate CSS style sheet. The only one that really makes any sense in terms of developing Web sites is to link your XHTML pages to a CSS style sheet, but we will examine the other two as well, as they can be useful while creating your pages.

A style sheet can be linked to an infinite number of XHTML pages, which helps ensure a consistent look from page to page and allows edits made to a style to be instantly reflected across an entire site.

**Inline Styles**

Inline styles (also known as *local styles*) are added to a tag using the XHTML *style* attribute, like this:

```html
<p>This paragraph simply takes on the browser’s default paragraph style.</p>

<p style="font-size: 25pt; font-weight:bold; font-style: italic; color:red;"">By adding inline CSS styling to this paragraph, we can override the default styles.</p>

<p>And now we are back to a regular default paragraph without any inline styles.</p>
```

which looks like this (see Figure 2.1)

![Inline Styles example - Windows Internet Explorer](image-url)
Here are some things you need to know about inline styles:

- Their scope is very restricted. An inline style only affects the tag to which it is attached.

- The practice of using inline styles is simply another way of putting presentational markup directly on the tags, as we did in days of yore. Adding inline styles everywhere is as bad for the portability and editability of your markup as adding deprecated HTML attributes, such as FONT. Inline styles should be generally avoided.

- On those rare occasions when you need to override a style in just one specific instance and there is no better way to do it, you can create an inline style and not feel too guilty about it. That said, you could almost always avoid using inline styles by adding a unique ID or class to the tag in question and then writing a corresponding style in your style sheet.

- Using an inline style is a good way to try out a style before you move it into the style sheet (see “Linked Styles” on the next page). Just remember to clear out the `style` attribute entirely once you achieve the effect you want and then cut and paste just the style itself into the style sheet. If you have it in the markup, that inline style will always override whatever change you try to make to that particular tag from the style sheet, and you can spend hours trying to fix the style sheet when the problem is, in fact, hidden in the markup.

- Inline styles override the same styles you define with embedded styles (described next), which override global styles you define in style sheets. (See “The Cascade” later in this chapter for details on this.)

### Embedded Styles

You can place a group of CSS styles in the head of your XHTML document. These are known as *embedded styles* (or *page styles*) because they are part of the page (or embedded in it). Embedded styles work like this:

```html
<head>
<title>Embedded Styles example</title>
<meta http-equiv="Content-type" content="text/html; charset=iso-8859-1" />  
<meta http-equiv="Content-Language" content="en-us" />
</head>
```
The style tag tells the browser it is about to encounter code other than XHTML; the tag’s attribute states that the code is CSS.

Here’s what you need to know about embedded (or page) styles:

- The scope of embedded styles is limited to the page that contains the styles.

- If you are only publishing a single page with these particular styles, you can embed the styles in the head of the document, although you are not truly separating the styles from the content; they are still in the same document. You will become familiar with embedded styles as you follow along with the hands-on single-page examples in this chapter.

- If you are working up multiple styles for a complex layout such as a form, sometimes it’s easier to write the styles as embedded styles in the head of the document so that you don’t have to constantly switch between the markup and the style sheet. Then, once everything is working, you can move the styles into the main style sheet and replace the styles in the header with a link to the style sheet.

- Page styles override style sheet styles, but they lose out to attributes you define in inline styles.

- If you are sending an XHTML page to someone for a critique, it’s considerate to embed the CSS styles in the page, so the reviewer only has to open the page and everything works; however, for a Web site of any scale, there is really only one way to manage the CSS and that is in a style sheet that can be linked to all of the site’s pages.

### Linked Styles

Ideally, you place styles in a separate document (a style sheet) that links to multiple pages so that the styles have global (site-wide)
scope. The styles defined in this style sheet can then affect every page of your site, not just a single page or a single tag. This is the only method of the three that truly separates the presentational styles from the structural markup. If you centralize all your CSS styles in a style sheet in this way, Web site design and editing become much easier.

For example, if you need to make changes that affect the whole site ("The client wants all the paragraph text to be blue, not black.") doing so can be as quick and painless as modifying one CSS style. This is certainly much easier than the pre-CSS task of modifying every FONT attribute of every paragraph tag in every page of the site.

You can link your style sheet to as many XHTML pages as you want with a single line of code in the head of each XHTML page:

```html
<link href="my_style_sheet.css" media="screen" rel="stylesheet" type="text/css" />
```

Then the styles are applied to each page's markup as the page loads.

Note that, in the above link tag, the media attribute is defined as "screen", meaning the style sheet is designed for the screen, which currently means Web browsers. (Certain user agents look for particular media attributes that best suit their display capabilities; possibilities here include: all, projection, handheld, print and aural. See a full list on the W3 Schools site (www.w3schools.com/css/css_mediatypes.asp).

A browser reads a style sheet where the link tag media attribute is all or screen. But by adding a second link tag with the media attribute of "print", you can offer a second style sheet that the browser will use when printing. A style sheet for printing might hide navigational and other elements that don't make sense when the content goes to paper.

If you create a second style sheet for printing, its link tag might look like this

```html
<link href="my_style_sheet_print.css" media="print" rel="stylesheet" type="text/css" />
```

So now that you know what style sheets are, let's look at how you write style sheet rules, and how concepts like Inheritance, Specificity, and the Cascade control how these rules affect your markup.
Anatomy of a CSS Rule

Let's start learning about how CSS is written by looking at a simple CSS rule. For example, here's a rule that makes all text in all paragraphs of your document red

```css
p {color:red;}
```

So if you have this XHTML markup

```html
<p>This text is very important</p>
```

then it will be red.

A CSS rule is made up of two parts: the selector, which states which tag the rule selects, (or, as I like to say, which rule the selector targets)—in this case, a paragraph—and the declaration, which states what happens when the rule is applied—in this case, the text displays in red. The declaration itself is made up of two elements: a property, which states what is to be affected—here, the color of the text—and a value, which states what the property is set to—here, red. It's worth taking a good look at this diagram (Figure 2.2) so that you are absolutely clear on these four terms; I'll be using them extensively as we move forward.

While a `p` tag in the XHTML markup is enclosed in angle brackets, in the CSS style, you just write the tag name without the angle brackets.
Writing CSS Rules

CSS demands absolute accuracy; a missing semicolon can cause CSS to ignore an entire rule.

You may be wondering what other values properties such as font size and color may have. For example, you might want to know if you can specify a color using RGB (red, green, blue) instead of a color name. (The answer is yes, you can.) For now, just hang in there while I focus on showing you how selectors work. Then, later in this chapter, I’ll show you the declaration part of the rules.

This basic structure of the selector and the declaration can be extended in three ways:

Multiple declarations can be contained within a rule.

```css
p {color:red; font-size:12px; line-height:15px;}
```

Now our paragraph text is red, 12 pixels high, and the lines are 15 pixels apart. (Pixels are, of course, the tiny dots that make up your screen display.)

Note that each declaration ends with a semicolon to separate it from the next. The last semicolon before the closing curly bracket is optional, but I always add it so that I can tack on more declarations later without having to remember it.

Multiple selectors can be grouped. If, say, you want text for tags `h1` through `h6` to be blue and bold, you might laboriously type this

```css
h1 {color:blue; font-weight:bold;}
```

```css
h2 {color:blue; font-weight:bold;}
```

```css
h3 {color:blue; font-weight:bold;}
```

and so on. But you can avoid this kind of repetition by grouping selectors in a single rule like this

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

Just be sure to put a comma after each selector except the last. The spaces are optional, but they make the code easier to read.

Multiple rules can be applied to the same selector. If, having written the previous rule, you decide that you also want just the `h3` tag to be italicized, you can write a second rule for `h3`, like this

```css
h3 {font-style: italic;}
```
Targeting Tags Within the Document Hierarchy

If you have forgotten what the document hierarchy is since the end of the last chapter, you might want to reread “Document Hierarchy: Meet the XHTML Family” in Chapter 1 now so that I can avoid the redundancy of repeating myself repeatedly and redundantly.

Using Contextual Selectors

If you write a rule where you simply use the tag name as the selector, then every tag of that type is targeted. For example, by writing

```
.p {color:red;}
```

every paragraph would have red text.

But what if you only want one particular paragraph to be red? To target tags more selectively, you use contextual selectors. Here’s an example

```
div p {color:red;}
```

Now only paragraphs within `div` tags would be red.

As you can see in the example above, contextual selectors use more than one tag name (in this case, `div` and `p`) in the selector. The tag closest to the declaration (in this case the `p` tag) is the tag you are targeting. The other tag or tags state the containing ancestor tag(s) in which the target tag must be contained for it to be affected by the rule. Let’s look at this idea in detail.

We’ll work with this bit of sample markup

```
<h1>Contextual selectors are <em>very</em> selective.</h1>
<p>This example shows how to target a <em>specific</em> tag using the document hierarchy.</p>
<p>Tags only need to be descendants <span>in the <em>order stated</em> in the selector</span>; other tags can be in between and the selector still works.</p>
```
Note that the first paragraph contains an `em` element; the second paragraph's `em` element is nested inside a `span` tag. **Figure 2.3** shows how this code looks with just the browser's default styling.

**Figure 2.4** shows the markup's hierarchy.

This hierarchy diagram illustrates which tag is nested inside which. If you write this style

```css
em {color: green;}
```

for the markup on the previous page by adding it between the `style` tags in the head of your document, all the text in `em` tags would turn green (**Figure 2.5** on the next page).
FIGURE 2.5 In this example, all text within em tags is green.

Contextual selectors are very selective.

This example shows how to target a specific tag using the document hierarchy.

Tags only need to be descendants in the order stated in the selector; other tags can be in between and the selector still works.

But what if you want to be more selective? Let's say you only want the em text within the paragraphs to be green. If this is the case, you would write a rule like this:

```html
p em {color:green;}
```

which would result in Figure 2.6.

FIGURE 2.6 By adding a contextual selector, you cause the rule to affect only paragraphs, not the heading.

Contextual selectors are very selective.

This example shows how to target a specific tag using the document hierarchy.

Tags only need to be descendants in the order stated in the selector; other tags can be in between and the selector still works.

Because you preceded the em with a p in the selector, only em tags within p tags are now targeted by the rule; the em tag in the h2 tag
is no longer affected. Note that, unlike the group selectors you saw earlier, contextual selectors have spaces, not commas, between the selectors.

Remember, rules with contextual selectors are applied only to the last tag listed, and then only if the selectors that precede it appear in this same order *somewhere* in the hierarchy above it. It doesn’t matter how many tags appear in between.

Because of this, the em tag within the span tag is affected by this rule. Even though it is not an immediate child of the p tag, the rule still applies, because it is a descendant of the p tag. Here’s an example of how you can state multiple tags in the selector to make the targeting even more specific:

```
p span em {color:green;}
```

This results in Figure 2.7.

![Figure 2.7](image)

Your rule now states that only an em within a span within a p tag is selected; you set a very specific context in which the rule works, and only one tag meets this criterion. In a contextual selector like this, you can list as many selectors as you need to ensure that the tag you want to modify is targeted.

However, things get more difficult if you want to target the word “specific” only; as you saw in Figure 2.5, the rule `p em {color:green;}` selects the em tags inside both the paragraphs, and you simply can’t target just this particular tag with a standard contextual selector. What you need here is a *child selector.*
Working with Child Selectors

In Chapter 1, I mentioned that a child tag is a direct descendant of an enclosing tag. If you want to write a rule so that the tag you’re targeting has to be a child of a particular tag, then you can do that too, using the \( > \) symbol, like this

\[
p > \text{em} \{ \text{color:green;} \}
\]

Now you have successfully targeted the word “specific” without affecting the other \text{em} text, because “specific” is contained in an \text{em} tag that is a child of the \text{p} tag, but the words “order stated” are not (Figure 2.8).

Before you drop this book in your haste to start using child selectors in your CSS, it’s important to know that, at the time of writing, IDWIMIE6; Internet Explorer for Windows 6 simply ignores them (although Internet Explorer 7 for Windows does implement them). However, there are work-arounds if you find yourself in situations where only a child selector will do. As you will see shortly, classes and IDs let you target any individual tag you want, but to use them, you’ll need a little extra markup.

So until Internet Explorer 6 becomes insignificant in usage, you’ll mainly use child selectors to create variations in your style sheet to work around Internet Explorer’s various nonstandards-compliant quirks, or in ways that cause IE6 to display a different but still acceptable result. We will use them in this way in later chapters.
Adding Classes and IDs

So far you’ve seen that when you have a rule with a selector that simply states a tag name such as \texttt{p} or \texttt{h1}, the rule is applied to every instance of that tag. You’ve also seen that to be more specific in the selection process, you can use contextual selectors to specify tags within which target tags must be contained.

However, you can also target specific areas of your document by adding IDs and class attributes to the tags in your XHTML markup. IDs and classes give you a second approach to styling your document—one that can operate without regard for the document hierarchy.

**SIMPLE USE OF A CLASS**

Here’s a piece of markup that illustrates how you might use a class

\[
\text{<h1 class="specialtext">This is a heading with the <span>same class</span> as the second paragraph</h1>}
\]

\[
\text{<p>This tag has no class.</p>}
\]

\[
\text{<p class="specialtext">When a tag has a class attribute, we can target it <span>regardless</span> of its position in the hierarchy.</p>}
\]

Note that I’ve added the class attribute \texttt{specialtext} to two of these tags. Let’s now apply these styles to this markup where \texttt{specialtext} is formatted as bold (Figure 2.9).

\[
\text{p \{font-family: Helvetica, sans-serif;\}}\]

\[
\text{.specialtext \{font-weight:bold;\}}\]

![Image](http://www.stylinwith/)

**Figure 2.9** Here I use a class selector to bold two different tags.
These rules result in both paragraphs displaying in the Helvetica font (or the browser’s generic sans-serif font if Helvetica is not available) and the paragraph with the specialtext class displaying in Helvetica bold. The text in the h1 tag remains in the browser’s default font (usually Times), but it is bold because it also has the specialtext class. Note that the span, a tag that has no default attributes, doesn’t affect anything because I didn’t explicitly style it.

**CONTEXTUAL CLASS SELECTORS**

If you only want to target one paragraph with the class, you create a selector that combines the tag name and the class, like this (Figure 2.10)

```css
p {font-family: Helvetica, sans-serif;}
.specialtext {font-weight:bold;}

p.specialtext {color:red;}
```

This is another kind of contextual selector because the class must be in the context of a paragraph for the rule to be applied.

You can go one step further and write the following (Figure 2.11)

```css
p {font-family: Helvetica, sans-serif;}
.specialtext {font-weight:bold;}

p.specialtext span {font-style:italic;}
```
FIGURE 2.11 By adding a second selector, you can be very specific about which tag is styled.

Now the word “regardless” is bold and italicized because it is in a span tag that is in a paragraph with the specialtext class, as the rule specifies. If you also want this rule to target the span in the h1 tag, you can modify it in one of two ways. The easiest way is not to associate the class with any specific tag (Figure 2.12).

.specialtext span {font-style:italic;}

FIGURE 2.12 With a less specific selector, the headline's span text is also selected.

The words “same class” in the headline are now also italicized. By deleting the p from the start of the selector, you remove the requirement for the class to be attached to any specific tag, so now both span tags are targeted. The rule states that the span tag can be a
descendant of any tag with the specialtext class because no tag is specified.

The benefit of this approach is that you can use a class without regard for the tag to which it belongs, so you are escaping the inherent constraints of the hierarchy when you do this.

The downside is that other tags that you don’t intend to style might also be affected because this modified rule is less specific than it was. So, say you later added a span inside another tag that also had the specialtext class, such as this one:

```
<div class="specialtext">In this div, the span tag <span>may or may not</span> be styled.</div>
```

The text within the span would be italicized also, which may or may not be the desired effect (Figure 2.13).

If you don’t want to style this new div’s span, you can adopt a second, more focused, group selector approach, like this (Figure 2.14)

```
p.specialtext span, h1.specialtext span {font-style:italic;}
```

Now only the two tags in question are targeted and your new tag is not affected. Your grouped rules don’t target that span because it’s descended from a div, whereas if you use the more simple and less specific .specialtext span approach, it is targeted.

Although this may seem like a lot to think about when you are styling a four-line example like this, when you are working on a style sheet that might be dozens or hundreds of lines long, you need to keep these considerations in mind, as we do in later chapters.
MUTLIPLE CLASSES

One final note on classes is that you can apply multiple classes to a single tag, like this

```html
<div class="specialtext featured">In this div, the span tag <span>may or may not</span> be styled.</div>
```

You can see that both the `specialtext` and the `featured` class names go in the same set of quotes with a space between them, which looks a little strange at first, but please refer any questions about this to the W3C. You will see uses for multiple classes in examples in later chapters.

Introducing IDs

IDs are written in a similar way to classes, except you use a `#` (hash symbol) to indicate them in your CSS instead of the class’s . (period)

If a paragraph is marked up with an ID, like this

```html
<p id="specialtext">This is the special text</p>
```

then the corresponding contextual selector looks like this

```css
p#specialtext {some CSS rules here}
```

Other than this, IDs work in the same way as classes, and everything in our previous discussion of classes applies equally to IDs. So what’s the difference?
The Difference Between IDs and Classes

So far, I have shown aspects of classes and IDs that might make them seem to be interchangeable—we have used them both to identify a specific tag within our markup. However, an ID is more powerful than a class, rather like the queen is more powerful than a pawn in a game of chess. (You will see just how true this is when you look at the concept of rule specificity in the "The Cascade" section later in this chapter.) This is because, according to the rules of XHTML, only a single instance of a particular ID (such as id="mainmenu") can be in a page, but a class (such as class="strongparagraph") can appear many times.

So, if you want to identify a unique piece of your page’s markup, such as the main navigation menu to which you want to target a special set of CSS rules, you might use an ID on a div (division element) that encloses the menu’s elements.

To identify a number of special paragraphs in a page that all require the same variation of styling from the basic paragraph, you would use a class.

As an aside, you also use an ID to enable JavaScript to be targeted at a tag (for example, to activate a DHTML animation when the user mouses over a link). You JavaScript jocks might like to know that the id attribute replaces the deprecated name attribute (which the XHTML validator flags as invalid) for this purpose. It’s especially important that you make sure JavaScript-related IDs appear only once in a page, or the JavaScript may behave unpredictably.

Don’t Go Crazy with Classes

Generally, you should use IDs and classes sparingly; the right kind of use is putting them on the divs that contain the main sections of your markup, and then accessing the tags within them with contextual selectors that begin with the ID or class’s name.

What you want to avoid is what Jeffrey Zeldman describes as “classitis—the measles of markup,” where you add a unique class or ID to just about every tag in your markup and then rewrite a rule for each one. This is only one step removed from loading up your markup with FONT tags and other extraneous markup. The good doctor Zeldman has cured me and many others of this nasty affliction. If you are already in the habit of slapping classes on every tag, as most of us do when we enthusiastically jump into CSS, take a look at the markup sample in Chapter 1 in light of what you just read in this chapter. You’ll see that you can target styles at every tag quite easily without adding any more IDs or classes.

If you use IDs to identify only the main sections of your markup—and use those classes for occasional tags that can’t be specifically targeted with contextual tag-based selectors—you won’t go far wrong. This approach has the added benefit of making your style sheet simpler too.
In summary, you can use multiple id attributes in a page, but each one must have a unique value (name) to identify it. You can apply a particular class name to as many tags as needed.

**Specialized Selectors**

Although not an official CSS category, these “specialized” selectors let you target markup in ways different from the selectors we have seen so far. With the exception of the * (star) selector, the following selectors effectively examine the markup and apply themselves if certain conditions are true—for example, if one particular type of tag follows another type of tag sequentially. These selectors offer some powerful capabilities, but are not well supported by older browsers, especially IE, including IE6. IE7, however, is much improved in this regard. If you want to test any browser’s capability to support pseudo-classes, I have put up a test page at www.stylinwithcss.com/chapter_2/code/pseudo_tests.htm. This page will allow you to see at a glance how well the browser you are using supports these “specialized” selectors.

**THE UNIVERSE SELECTOR**

The * universal selector (commonly known as the star selector) means “anything,” so if you use

* {color:green;}

in your style sheet, all type will be green, except where you specify it to be different in other rules. Another interesting use for this selector is as the inverse of the child selector—a not-a-child selector, if you will.

p * em {font-weight:bold;}

Here, any em tag that is at least a grandchild of the p tag, but not a child, is selected; it doesn’t matter what the em’s parent tag is. (The star selector is supported by all browsers, old and new.)

**THE ADJACENT SIBLING SELECTOR**

This rule selects a tag that follows a specific sibling tag (sibling tags are at the same level in the markup hierarchy—that is, they share the same parent tag). Here’s an example

h1 + p {font-variant:small-caps}
Applying this rule to this markup

```html
<div>
  <h1>All about siblings selectors</h1>
  <p>There must be at least two selectors, with a + sign before the last one.</p>
  <p>The targeted selector will only be affected if it is a sibling to, and preceded by, the one before the + sign.</p>
</div>
```

results in what is shown in Figure 2.15, because only the first paragraph is preceded by a sibling h1 tag.

As you can see, the p tag that follows the h1 meets the condition of the rule, so it is in small caps. The second p tag, which is not adjacent to the h1, is unaffected. This is a good way to have the first item in a list be bold, for example (ul + li {font-weight:bold;}). (Adjacent sibling selectors work in SCBs and IE 7, but not IE6).

**ATTRIBUTE SELECTORS**

Attribute selectors use the attributes of the tag. This is another way to target different CSS at similar elements; as long as there is some difference between attributes on the tags you are trying to target, you can apply different rules to those tags. However, this interesting capability is of limited use, since neither IE 6 nor even the new,
improved IE 7 supports them. So for now, we can only use attribute selectors to enhance the experience of viewers with SCBs.

This rule

```css
img[title] {border: 2px solid blue;}
```

causes any `img` with a `title` attribute, like this

```html
<img src="../images/Windsor-castle_walls.jpg" title="Windsor-castle walls" alt="Windsor-castle walls" />
```

to have a blue, two-pixel border around it; it doesn’t matter what the value of the `title` attribute is, just that there is one. You might use such a style to indicate to the user that if he points at this image, a tooltip (pop-up text generated by the title attribute) displays. It’s common practice to duplicate the `alt` and `title` attribute values—the `<alt>` tag text displays if the image does not load, or can be read by a screen reader, and the `title` causes a tooltip to appear if the user points at the image.

You can also be specific about what the attribute’s value should be. For example, the rule

```css
img[alt="Dartmoor-view of countryside"] {border:3px green solid;}
```

only puts the border around the image if the image’s `alt` attribute is "Dartmoor-view of countryside"; in other words, if the image markup looks something like this

```html
<img src="../images/dartmoor-view.jpg" title="Dartmoor-view of countryside" alt="Dartmoor-view of countryside" />
```

This selector is made more useful by the fact that it lets you specify just the first characters of the attribute value, but the “common” part of the attribute must be separated from the “different” part of the attribute with a hyphen. So, if you have carefully written your `img` tags with attributes like these (note the hyphens)

```html
<img src="../images/dartmoor-cottage.jpg" title="Dartmoor-small cottage" alt="Dartmoor-small cottage" />
```

```html
<img src="../images/dartmoor-view.jpg" title="Dartmoor-view of countryside" alt="Dartmoor-view of countryside" />
```

then you can select them by adding the pipe symbol (usually typed with `Shift-|`) into your rule, like this

```css
img[alt="Dartmoor"] {border:3px blue solid;}
```
By the way, this rule would also select this example

```html
<img src="../images/dartmoor-view.jpg" title="Dartmoor"
alt="Dartmoor"/>
```

even though this example’s alt tag doesn’t have the hyphenated extension to the value.

**Figure 2.16** is a screenshot in Firefox displaying these code examples.

![Firefox screenshot showing attribute selectors](image)

**Figure 2.16A.** ...but IE 7 has no attribute selector capabilities.

---

**Summary of Selectors**

So far, you’ve seen that you can target CSS rules in several ways: by using tag selectors, by using class and ID selectors, by using selectors that are a combination of both, and even by selecting based on the attributes that are attached to the tag.

One common aspect of these selectors is that they all are targeting *something* in the markup—a tag name, a class, an ID, an attribute,
or an attribute value. But what happens if you want some kind of styling to happen when some event occurs, such as the user pointing at a link? In short, you want a way to apply rules based on events. And after all this buildup, you know I’m going to tell you there’s a way to do that.

Pseudo-Classes

Named for the fact that they are classes that aren’t actually attached to tags in the markup, pseudo-classes cause rules to be applied to the markup when certain events occur. The most common event that occurs is that the user points at or clicks on something. With the newer browsers (sadly, not Internet Explorer 6 or earlier; at least not without adding the special JavaScript function, hover.HTC), it’s easy to make any on-screen object respond to a rollover, which is the act of moving the pointer over something, also known as hovering. For example, the :hover pseudo-class can cause a border to appear around an image when the mouse rolls over the image.

Anchor Link Pseudo-Classes

Pseudo-classes are most commonly used with hyperlinks (a tags), enabling things like a change in their color or causing their underlining to be removed when rolled over.

There are four pseudo-classes for anchor links, since links always are in one of these four states:

- **Link.** The link is just sitting there looking like a link and waiting for someone to click on it.
- **Visited.** The user has clicked on the link at some point in the past.
- **Hover.** The link is currently being pointed at (rolled over).
- **Active.** The link is currently being clicked.

Here are the corresponding pseudo-class selectors for these states (using the a selector with some sample declarations):

```css
a:link {color:black;}
```

```css
a:visited {color:gray;}
```

```css
a:hover {text-decoration:none;}
```

```css
a:active {color:navy;}
```
First, let’s save the debate about appropriate link colors and behavior for later and simply observe that, according to the declarations above, links are initially black (and underlined by default). When the mouse rolls over them (the hover state), the underlining is removed, and they stay black, because no color is defined here for the hover state. When the user holds the mouse down on the link (the active state), it turns navy, and forever after (or more accurately, until the browser’s history of the visit to the link’s URL expires or is deleted by the user), the link displays in gray. When using these pseudo-class selectors, you have complete control over the look and behavior of the four states of links.

And that’s all very nice, but the real power comes when you start using these anchor link pseudo-classes as part of contextual selectors. Then you can create different looks and behaviors for various groups of links in your design—navigation, footers, sidebars, and links in text, for example. We’ll explore using these pseudo-classes for styling of links and other things to the point of tedium (or perhaps, ecstasy) later in the book, but for now, let’s note the following and then move on:

**You don’t have to define all four of these states.** If you just want to define a link and a hover state, that’s fine. Sometimes it doesn’t make sense to have links show as having been visited.

**A browser may skip some of these rules if you don’t state them in the order shown above: link, visited, hover, active.** The mnemonic “LoVe-HA!” is an easy, if cynical, way to remember this.

**You can use any element with these pseudo-classes, not just a, to create all kinds of rollover effects.** For example

```css
p:hover {background-color:gray;}
```

This code will, well, I don’t think I even need to tell someone as smart as you what is apt to happen to your paragraph when you roll over it.

Link (a) pseudo-classes are supported by all browsers tested for this book—IE5 and later. As mentioned before, be aware that IE6 does not support hovers on anything except links without a special JavaScript file called `hover.htc` being linked as an Internet Explorer behavior to the page. IE7 does support hover on any element, but only in pages with the Strict DOCTYPE. All a bit confusing, but we will see how to make hover work on any element in any of our tested browsers in a later chapter.
Other Useful Pseudo-Classes

The purpose of pseudo-classes is to simulate classes being added to your markup when certain conditions occur. Not only can they be used to provide a response to user actions such as pointing and clicking, but they can also be applied based on certain conditions being true in your markup.

:FIRST-CHILD

x:first-child

This pseudo-class selects the first-child element with the name x. For example, if this rule

\[
\text{div.weather strong:first-child \{color:red;\}}
\]

is applied to this markup

\[
\text{<div class="weather">}
\text{It's <strong>very</strong> hot and <strong>incredibly</strong> humid.}
\text{</div>}
\]

then very is red and incredibly is not.

See Figure 2.17.

(SCBs and IE7 support :firstchild.)

:FOCUS

x:focus

An element such as a text field of a form is said to have focus when the user clicks it; that’s where the characters appear when the user types. For instance, the code

[Image of a website with text: It's very hot and incredibly humid.]

[Caption: FIGURE 2.17 First-child selectors enable you to target the first tag of a particular type with a set of tags.]
input:focus {border: 1px solid blue;}

puts a blue border around such a field when the user clicks it. (IE6, IE7 and Safari do not support :focus.)

## Pseudo-Elements

Pseudo-elements provide the effect of extra markup elements magically appearing in your document, although you don’t actually add any extra markup. Here are some examples.

This pseudo-class

```css
x:first-letter
```

For example:

```css
p:first-letter {font-size:300%; float:left;}
```

enables you, for example, to create a large drop-cap effect at the start of a paragraph.

This pseudo-class

```css
x:first-line
```

enables you to style the first line of (usually) a paragraph of text. For example,

```css
p:first-line{font-variant:small-caps;}
```

results in the first line, not surprisingly, being in small capital letters. If you have a liquid layout where the line length changes as the browser window is sized, words automatically change format as required so that only the first line is styled in this way. (All SCBs and IE7 support :first-letter and :first-line.)

These two pseudo-classes

```css
x:before and x:after
```

cause specified text to be added in before and after an element, so this markup

```html
<h1 class="age">25</h1>
```

and these styles

```css
h1.age:before {content:"Age: "}
```

```css
h1.age:after {content:" years old."}
```
Don't make critical features of your site dependent on pseudo-classes and pseudo-elements because they are not supported by IE6 and only partially supported by IE7 (and these two browsers are currently used by about 70 percent of your visitors). Instead, use these selectors to enhance the user experience in browsers that support them—for example, use the :focus selector to add a strong border around the form field in which a user is currently typing. Users that don't get this enhancement still have an acceptable experience.

There are four other pseudo-classes. The first is :lang, which is applied to elements with a specific language code, and the other three are :left, :right, and :first, which apply to paged media (print) rather than content displayed in browsers. They are little used and unevenly or not at all supported by browsers, so I am not covering them here.

result in text that reads "Age: 25 years old." Note that the spaces added inside the quoted content strings ensure proper spacing in the resultant output. These two selectors are especially useful when the tag's content is being generated as a result of a database query; if all the result contains is the number, then these selectors allow you to provide that data point with some meaningful context when you display it for the user. (IE7 does not support :before and :after.)

CSS3

Most modern browsers support CSS2, which defined a slew of new features that were added to the initial CSS recommendations in the mid-nineties. CSS3 is the latest upgrade to CSS. Actually, it's been around since about 2000, and the specification was refined over the next five years, but browser developers have been very slow in adopting CSS3.

The objective of CSS3 is to move into yet more presentation control of documents to CSS. As we move into the world of CSS, you can, you can, you can... get rid of all the visual controls that are available to our print design colleagues through programs such as Adobe InDesign and QuarkXPress.

The CSS3 spec is so extensive that it has been divided into several modules, which include a Color Module, Backgrounds and Borders, and Multi-Column Layout. You can go to http://www.css3.info to learn all about the various CSS3 modules.

What I will say is that when I wrote the original version of *Stylin'* three years ago, the CSS3 specification was pretty much where it is today, and the fact remains that in virtually all browsers, it is very poorly supported. Attribute selectors are the only part of it that has really made inroads, and although useful, they represent a small part of the entire specification.

Is CSS3 so incredibly difficult to implement? Even though it seems that the whole notion of Web 2.0 is visually encapsulated by boxes with rounded corners, rounded corners on XHTML elements are a part of the CSS3 specification that only Mozilla can render using its special "-moz" selectors. A common work-around technique has been to wrap lots of spans around an element, each containing rounded-corner background graphics, to achieve this effect. We will look at more simple ways to create rounded corners without resorting to graphics later in the book.
Inheritance

Just like the money you hope you’ll get from rich Uncle Dick, inheritance in CSS involves passing something down from ancestors to descendants: the values of CSS properties. You may remember from our discussion on the document hierarchy in Chapter 1 that the body tag is the great-ancestor of them all—all CSS-targeted tags in your markup descend from it. So thanks to the power of CSS inheritance, if you style the body tag like this

```
body {font-family: verdana, helvetica, sans-serif; color: blue;}
```

then the text of every text element in your entire document inherits these styles and displays in blue Verdana (or in one of the other choices if Verdana is not available), no matter how far down the hierarchy it is. The efficiency is obvious; rather than specify the desired font for every tag, you set it once in this way as the primary font for the entire site. Then you only need font-family properties for tags that need to be in a different font.

Many CSS properties are inherited in this way, most notably text attributes. However, many CSS properties are not inherited because inheritance doesn’t make sense for them. These properties primarily relate to the positioning and display of box elements, such as borders, margins, and padding. For example, imagine that you want to create a sidebar with text in it. You might do this by writing a div (which you can think of as a rectangular box), which has a list of links inside it, and styling the div with a border, say a two-pixel red line. However, it makes no sense for every one of those list items within the div to automatically get a border too. And they won’t—border properties are not inherited. When we look at the box model in Chapter 4, we’ll look at inheritance in greater detail.

Also, you must be careful when working with relative sizes such as percentages and ems; if you style a tag’s text to be 80 percent and it’s descended from a tag whose text is also sized at 80 percent, its text size will be 64 percent (80 percent of 80 percent), which is probably not the effect you want. In Chapter 3, I’ll cover the pros and cons of absolute and relative text sizing.

In the examples that follow, we will be examining the effect that inheritance has on your styles as you write them, and how to make the most of inherited styles so that you write the minimum amount of CSS necessary to achieve the desired result.
The Cascade

OK, now we have enough information to have a meaningful discussion about one of the toughest aspects of CSS to get your head around—the Cascade. If this section gets to be too much, skip ahead and read the “Charlie’s Simple Cascade Summary” sidebar later in this chapter. This sidebar is a simplified, if slightly less accurate, version that will serve you until you have done some CSS coding and really need the details.

As its name suggests, the Cascade in Cascading Style Sheets involves styles falling down from one level of the hierarchy of your document to the next, and its function is to let the browser decide which of the many possible sources of a particular property for a particular tag is the one to use.

The Cascade is a powerful mechanism. Understanding it helps you write CSS in the most economical and easily editable way and enables you to create documents that are viewed as you mean them to be seen, while leaving appropriate control of aspects of the document’s display, such as overall font sizes, with users who have special needs.

Sources of Styles

Styles can come from many places. First, it’s not hard to accept that there must be a browser style sheet (the default style sheet) hidden away inside the browser, because every tag manifests styles without you writing any. For example, h1 tags create large bold type, em tags create italicized type, and lists are indented and have bullets for each item, all automatically. You don’t style anything to make this formatting happen.

If you have Firefox installed on your computer, search for the file html.css, and you can then see the Firefox default browser style sheet. Modify it at your peril.

Then there is the user style sheet. The user can create a style sheet, too, although very few do. This capability is handy, for example, for the visually impaired, since they can increase the baseline size of text or force text to be in colors that they can discern one from another. You can add a user style sheet to Internet Explorer for Windows (6 or 7) by selecting Tools > Internet options and clicking the Accessibility button. This capability, for example, enables visually impaired users to add a style like
body {font-size:200%;}

that doubles the size of all text—inherence at work again. This is why it is important to specify text in relative sizes, such as ems, rather than fixed sizes, such as points, so you don't override such changes. We will discuss this interesting topic more in Chapter 3.

Then there are author style sheets, which are written by you, the author. We have already discussed the sources of these: linked style sheets, embedded styles at the top of pages, and inline styles that are attached to tags.

Here's the order in which the browser looks at, or cascades through, the various locations:

- Default browser style sheet
- User style sheet
- Author style sheet
- Author embedded styles
- Author inline styles

The browser updates its settings for each tag's property values (if defined) as it encounters them while looking sequentially in each location. They are defined in the default browser style sheet, and the browser updates any that are also defined in the other locations. If, for example, the author style sheet style defines the `<p>` tag's font-family to be Helvetica but the `<p>` tag is also specified to be Verdana in an embedded (page) style, the paragraph will be displayed in Verdana—the embedded styles are read after the author style sheet. However, if there is no style for paragraphs in the user or author style sheet, they will display in Times, because that's the style defined in all browser default style sheets.

That's the basic idea of how the Cascade works, but in fact, there are several rules that control the Cascade.

## The Cascade Rules

In addition to the order in which styles are applied, you should know several rules about how the Cascade works.

**Cascade Rule 1:** Find all declarations that apply to each element and property. As it loads each page, the browser looks at every tag in the page to see if a rule matches it.
Cascade Rule 2: Sort by order and weight. The browser sequentially checks each of the five sources, setting any matched properties as it goes. If a matched property is defined again further down the sequence, the browser updates the value and does this repeatedly, if necessary, until all five possible locations of properties for each tag in that page have been checked. Whatever a particular property is set to at the end of this process, that’s how it is displayed.

In Table 2.1, we look at this process for a page with numerous p tags. Let’s assume, for the sake of the example, that two of those p tags have inline styles that define their color as red. In this case, every p tag text is blue, except for ones with the inline color attribute—these are red.

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>TAG</th>
<th>PROPERTY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default style sheet</td>
<td>P</td>
<td>color</td>
<td>black</td>
</tr>
<tr>
<td>User style sheet</td>
<td>P</td>
<td>color</td>
<td>blue</td>
</tr>
<tr>
<td>Author style sheet</td>
<td>P</td>
<td>color</td>
<td>red</td>
</tr>
<tr>
<td>Author embedded styles</td>
<td>P</td>
<td>color</td>
<td>red</td>
</tr>
</tbody>
</table>

Of course, things aren’t quite that simple. There is also the weight of the declaration. You can define a rule as important, like this:

```css
p {color:red !important; font-size:12pt;}
```

The word `important` follows a space after the style you want to make important but before the `;` (semicolon) separator.

This style defines the text’s red color as important, and therefore, it displays this way, even if it is declared as a different color further down the Cascade. Think hard and long before you force a particular style on the user with `important` rule definition, because you may be messing up someone’s personal style sheet, which may be set that way for a very good reason; be sure that it truly is important for such a style to dominate over any other possible style for that tag.
Charlie’s Simple Cascade Summary

You need to remember just three things in this simplified version of the Cascade rules. These are true for virtually every case.

**Rule 1:** Selectors with IDs override selectors with classes; these, in turn, override selectors with only tags.

**Rule 2:** If the same property for the same tag is defined in more than one location in the Cascade, inline styles override embedded styles, which override style sheet styles. Rule 2 loses out to Rule 1, though—if the selector is more specific, it overrides, wherever it is.

**Rule 3:** Defined styles override inherited styles, regardless of specificity. A little explanation is required for Rule 3. This markup

```html
<div id="cascadedemo">
  <p id="inheritancefact">Inheritance is <em>weak</em> in the Cascade</p>
</div>
```

and this rule, which has a high specificity,

```
2-0-4html body div#cascadedemo p#inheritancefact {color:blue;}
```

results in all the text, including the word *weak*, being blue because the *em* inherits the color from its parent, the *p* tag.

As soon as we add this rule for the *em*, even though it has very low specificity

```
0-0-1em {color:red}
```

the *em* text is red. The inherited style is overridden by the defined style for the *em*, regardless of the high specificity of the rule for the containing paragraph.

There, three simple cascade rules. That was much easier, wasn’t it?

---

**Cascade Rule 3:** Sort by specificity. Besides being very hard to pronounce, specificity determines just how specific a rule is. I tried to get you started on this idea by using the word *specific* in exactly this way many times while we were discussing selectors. As you saw, if a style sheet contains this rule

```css
p {font-size:12px;}
```

and this rule

```css
p.largetext {font-size:16px;}
```

then this markup

```html
<p class="largetext">A bit of text</p>
```

displays text 16 pixels high because the second rule is more specific—it overrides the first rule. This may seem intuitively obvious, but what happens to that bit of markup if you use these styles instead?
Both these rules match the tag, but the class overrides, and the text is 16 pixels. Here’s why: the numeric specificity of the tag selector is 1, but the class has a specificity of 1-0. Here’s how to calculate the specificity of any selector. There is a simple scoring system for each style that you plug into a three-value layout like this:

A - B - C

The dashes are separators, not subtraction signs. Here’s how the scoring works:

1. Add one to A for each ID in the selector.
2. Add one to B for each class in the selector.
3. Add one to C for each element name (tag name).
4. Read the result as a three-digit number. (It’s not really a three-digit number; it’s just that in most cases, reading the result as a three-digit number works. Just understand that you can end up with something like 0-1-12, and 0-2-0 is still more specific.)

So let’s look at the specificity of these examples

<table>
<thead>
<tr>
<th>Selector</th>
<th>Specificity</th>
</tr>
</thead>
<tbody>
<tr>
<td>.p.largeText</td>
<td>11</td>
</tr>
<tr>
<td>body #largeText</td>
<td>102</td>
</tr>
<tr>
<td>body p.largeText</td>
<td>113</td>
</tr>
<tr>
<td>body p.largeText ul.mylitli</td>
<td>114</td>
</tr>
</tbody>
</table>

Each example is a higher specificity than the previous one.

**Cascade Rule 4: Sort by order.** If two rules have exactly the same weight, the one furthest down the Cascade overrides.

And that, dear reader, is the Cascade and, yes, it is somewhat hard to understand, especially if you have not yet had much experience with CSS, but my simplified version of the Cascade rules (see the sidebar earlier in this chapter) applies in about 98 percent of cases. If you find that something isn’t behaving the way you want when you’re using this simplified version, refer to the above rules.
Rule Declarations

So far I've focused on how you use CSS rule selectors to target tags, but you haven't yet looked much at the other half of a CSS rule, the declaration. I've used numerous different declarations to illustrate the selector examples but have only explained them minimally. Now it's time to look at declarations in detail.

The diagram showing the structure of a CSS rule earlier in this chapter (Figure 2.2) shows that a declaration is made of two parts: a property and a value. The property states what aspect of the element is affected (its color, its height, and so on) and the value states what that property is set to (green, 12px, and so on).

Every element has a number of properties that can be set using CSS; these differ from element to element. You can set the font-size property for text, but not for an image, for example. In each subsequent chapter of this book, I use real-world examples to show you the properties you can set for different elements and the values you can set for those properties. Because there are only a few different types of CSS rule values, let's look at them now.

Values fall into three main types:

**Words.** For example, in `font-weight: bold`, *bold* is a type of value.

**Numerical values.** Numerical values are usually followed by a unit type. For example, in `font-size: 12px`, 12 is the numerical value and `px` is the unit type—pixels in this example.

**Color values.** Color values are written as `color: #336699`, where the color in this example is defined with a hexadecimal value.

There's not much I can tell you about word values that would make sense until you start using them, because they are specific to each element. Numerical and color values, however, can only be expressed in certain ways.

**Numerical Values**

You use numerical values to describe the length (and I use “length” generically to mean height, width, thickness, and so on) of all kinds of elements. These values fall into two main groups: absolute and relative.
Absolute values (Table 2.2) describe a length in the real world (for example, 6 inches), as compared to a relative measurement, which is simply a relationship with some other measurable thing (when you say “twice as long” that’s a measure relative to something else).

<table>
<thead>
<tr>
<th>Absolute Value</th>
<th>Unit Abbreviation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inches</td>
<td>in</td>
<td>height:6in</td>
</tr>
<tr>
<td>Centimeters</td>
<td>cm</td>
<td>height:40cm</td>
</tr>
<tr>
<td>Millimeters</td>
<td>mm</td>
<td>height:500mm</td>
</tr>
<tr>
<td>Points</td>
<td>pt</td>
<td>height:60pt</td>
</tr>
<tr>
<td>Picas</td>
<td>pc</td>
<td>height:90pc</td>
</tr>
<tr>
<td>Pixels</td>
<td>px</td>
<td>height:72px</td>
</tr>
</tbody>
</table>

*Examples are not equivalent lengths.

When writing CSS that relates to fixed-sized elements such as images, I use only pixels. It’s up to you, but pixels are also the only absolute unit that I use throughout this book, except in print style sheets—because paper is measured in inches, it makes sense to design print layouts with the same units.

Although the absolute units are pretty self-explanatory, the relative units (Table 2.3) warrant a little more explanation.

Em and ex are both measurements of type size. The em is derived from the width of the characters in a font, so its size varies, depending on which font you are using. Ex is the equivalent of the x-height of the given font (so named because it is the height of a lowercase x—in other words, the center bit without the ascenders and descendents that appear on characters such as p and d).

<table>
<thead>
<tr>
<th>Relative Value</th>
<th>Unit Abbreviation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Em</td>
<td>em</td>
<td>height:1.2em</td>
</tr>
<tr>
<td>Ex</td>
<td>ex</td>
<td>height:6ex</td>
</tr>
<tr>
<td>Percentage</td>
<td>%</td>
<td>height:120%</td>
</tr>
</tbody>
</table>

Percentages are useful for setting the width of containing elements, such as divs, to the proportion of the browser width, which is the one way to create “liquid” designs that smoothly change size as the
user resizes the browser window. Using percentages is also the right way to get proportional leading (pronounced like lead, the metal), which is the distance between the baseline of one line of text and the next in a multiple-line text block such as a paragraph. You will learn more about leading in Chapter 3.

Why You Should Use Ems to Specify Type Sizes

There are two important benefits to using a relative sizing method like ems to specify your font sizes:

- You can use inheritance to your advantage by declaring the body element to have a size of 1em, and this becomes a sizing baseline because it causes all other element's text to size relative to it. Because your content text always goes inside other elements, such as p and h4, you then simply write rules that state that the p tag is .8em, and that text links are .7em, for example. In this way, you establish proportional relationships between all the text elements of your design.

  Note that in Internet Explorer 6, when you set an em size for the body, paragraphs size in proportion automatically, but h1 thru h6 don't; you have to explicitly set some relative size for them (such as 1.1em for h1, .9em for h2, and so on), otherwise they remain fixed at their default sizes.

  If you later decide to increase the overall size of the text in your site, you can go back to the body tag and set its size to, say, 1.2em. Magically, all your text increases in size proportionally by the same amount (a fifth larger, in this case) because all the other tags inherit their size from the body tag.

- If you don't define font sizes with relative units, you effectively disable the font sizing capabilities available in the View menu of Internet Explorer (although other browsers can resize absolute font-size units), and therefore disenfranchise visually impaired users who rely on that capability to get your content to a size where they can read it. You need to check frequently during development to make sure that upping the font size in this way doesn't break your page's structure.

For these two reasons, I advise you to set all font sizes in ems rather than in absolute units, such as pixels. If you are designing a row of tabs in a fixed horizontal space, the layout has the potential to break if the text gets resized. If you're careful, however, and design with this possibility in mind, you can develop such components of your design so that they can accommodate larger type when the size is changed by the user.
Color Values

You can use several value types to specify color. Use whichever one of the following you prefer.

Hexadecimal (##RRGGBB and #RGB). If you already know languages like C++, PHP, or JavaScript, then you are familiar with hexadecimal (hex) notation for color. The format is this

#RRGGBB

In this six-character value, the first two characters define red, the next two green, and the next two blue. Computers use units of two to count, rather than base 10 like us mortals, and that's why hex is base 16 (2 to the power of 4), using the 16 numbers/letters 0–9 and A–F. A thru F effectively function as 10 through 15. Because color is represented by a pair of these base 16 numbers, there are 256 (16 × 16) possible values for each color, or 16,777,216 combinations (256 × 256 × 256) of colors. You definitely get the most color options by using hexadecimal, although you can get by with far less. You'd be hard pressed (to say nothing of your monitor) to discern the difference between two immediately adjacent hex colors. Don't forget the # (hash) symbol in front of the value.

So, for example, pure red is #FF0000, pure green is #00FF00, and pure blue is #0000FF.

You can also use the following shorthand hex format

#RGB

If you select a color where each pair has the same two letters, such as #FF322 (a strong red), you can abbreviate it to #F32.

Percentages RGB (R%, G%, B%). This is notation that uses a percentage of each color like this

R%, G%, B%

Acceptable values are 0% to 100%. Although this only yields a piddling one million color combinations (100 × 100 × 100), that's more than enough for most of us. Also, it's much easier to make a guess at the color you want in RGB compared with hex notation.

So, for example, 100%, 0%, 0% is max red, 0%, 100%, 0% is max green, and 46%, 76%, 80% is close to that dusky green-blue color I demonstrated in hex above.

Color Name (red). As you have seen from all the preceding color examples in the selector discussions, you can simply specify a color
by name, or keyword, to use the official term. However, there are
limitations. There is no W3C specification to say exactly how the
browser should render a color like olive or lime; basically, every
browser manufacturer assigns its own (presumably hex) values to
each color keyword. Also, only 16 colors are in the W3C spec, and,
therefore, you can be sure to find only these 16 in every browser.
Here they are, in alphabetical order:
aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive,
purple, red, silver, teal, white, yellow
Most modern browsers offer many more colors (usually 140), but if
you want to specify colors by name, you can only absolutely rely on
these 16.
I usually use hex colors because I program, and that’s how you do it
in the murky world of coding. To save you from struggling to mix up
colors yourself, visit http://www.bookmarkbliss.com/tools/book-
mark-bliss-10-tools-to-help-you-select-a-web-20-color-palette/,
which has fun tools to help you choose color palettes. Also, see the
sidebar “You Don’t Have to Limit Yourself to Web-Safe Colors”.
Now that you have a basic understanding of how CSS works, let’s
next look at how to style text

You Don’t Have to Limit Yourself to Web-Safe Colors
If you use Adobe Dreamweaver or other Web development tools, you are
used to picking colors from a Web-safe palette. This is a set of 216 colors
that only an engineer would have come up with, comprising mostly bright
and saturated colors, with limited choices in dark and pale colors. These
colors, you may (not) be interested to know, comprise twin hex pairs like this,
#3399CC or #FF99CC, and only use the values 0, 3, 6, 9, C, and F. So any color
you can come up with that meets these criteria is Web-safe. These colors
are a large subset of the 256 colors (40 are reserved for the system) that a
monitor driven by an 8-bit VGA card can display (remember 8-bit?), so for
years, we were told not to use any others. As of July 2007, fewer than .01 per-
world’s surfers still use 8-bit color, so you can confidently use any color in
your designs that you can create with the methods listed in this chapter.