CHAPTER 3

Style Sheets
CSS

As we have learned, HTML markup can be used to indicate both the *semantics* of a document (e.g., which parts are elements of lists) and its *presentation* (e.g., which words should be italicized). However, as noted in the previous chapter, it is advisable to use markup predominantly for indicating the semantics of a document and to use a separate mechanism to determine exactly how information contained in the document should be presented. *Style sheets* provide such a mechanism. This chapter presents basic information about *Cascading Style Sheets* (CSS), a style sheet technology designed to work with HTML and XML documents.

CSS provides a great deal of control over the presentation of a document, but to exercise this control intelligently requires an understanding of a number of features. And, while you as a software developer may not be particularly interested in getting your web page to look "just so," many web software developers are members of teams that include professional web page designers, some of whom may have precise presentation requirements. Thus, while I have tried to focus on what I consider key features of CSS, I've also included a number of finer points that I believe may be more useful to you in the future than you might expect on first reading.

While CSS is used extensively to style HTML documents, it is not the only style-related web technology. In particular, we will study the Extensible Stylesheet Language (XSL)—which is used for transforming and possibly styling general XML documents—in Chapter 7.

3.1 Introduction to Cascading Style Sheets

Before getting into details, let's take a quick look at an XHTML document that uses simple style sheets to define its presentation. Specifically, let's consider once again the "Hello World!" document of Figure 2.1, but with the addition of two `link` elements in the `head` of the document (`CSSHelloWorld.html`, shown in Fig. 3.1). Notice that the body of this document is identical to that of Figure 2.1. However, viewing this document in Mozilla 1.4 produces the result shown in Figure 3.2, which is quite different from the way Mozilla displayed the original "Hello World!" document (Fig. 2.2).

The difference between the two browser renderings, of course, has to do with the `link` element, which imports a *style sheet* located at the URL specified as the value of its `href` attribute. In this example, the style sheet is written in the CSS language, as indicated by the MIME type value of the `type` attribute. The `style1.css` file contains the lines

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      CSSHelloWorld.html
    </title>
    <link rel="stylesheet" type="text/css" href="style1.css"
          title="Style 1" />
    <link rel="alternate stylesheet" type="text/css" href="style2.css"
          title="Style 2" />
  </head>
  <body>
    <p>
      Hello World!
    </p>
  </body>
</html>
```

FIGURE 3.1 HTML source for "Hello World!" using style sheets.

```
body  { background-color:lime }
p     { font-size:x-large; background-color:yellow }
```

The first line simply says that, for rendering purposes, the `body` element of the document should be treated as if it contained the attribute `style="background-color:lime"`. The second line is similar, except that it specifies a style that should be applied to every `p` element of the document. The second line also specifies values for two different style *properties*, `font-size` and `background-color`. We'll learn details about these and many other style properties later in this chapter, but for now their meaning should be clear from their names and the effects shown in Figure 3.2.

The file `style2.css` contains the single line

```
p     { font-size:smaller; letter-spacing:1em }
```

This says that `p` elements should be set in a smaller than normal font size and that there should be space between adjacent letters. However, this style is not applied to the document

FIGURE 3.2 Browser rendering of `CSSHelloWorld.html`.

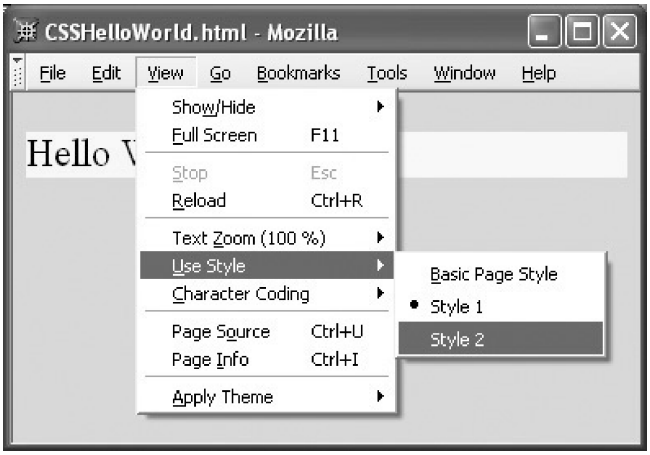

FIGURE 3.3 Selecting the style sheet to be used by Mozilla.

rendered in Figure 3.2, because this style sheet is specified as an *alternate style sheet* by the `rel` (relationship) attribute of the `link` element that imports this sheet. A style sheet such as the one in `style1.css`, which is referenced by a `link` element having a `rel` with value `stylesheet` as well as a `title` specification, is known as a *preferred style sheet*. An alternate sheet can be selected by the user, as illustrated in Figure 3.3. Notice that the values of the `title` attributes of the `link` tags are displayed in the Use Style menu along with the default Basic Page Style; preferred and alternate style sheet `link` elements must always contain `title` attribute specifications. After the alternate style sheet is selected, the page renders in the second style, as shown in Figure 3.4. (Alternate style sheets are not used often at the time of this writing, because the user interface for IE6 does not support user selection of alternate style sheets.)

Now that we have some understanding of what a style sheet is, we will discuss some of the major features of CSS.

3.2 Cascading Style Sheet Features

The key property of style sheet technology is that it can be used to separate the presentation of information from the information content and semantic tagging. The content and

FIGURE 3.4 Browser rendering of `CSSHelloWorld.html` using style sheet from `style2.css`.

TABLE 3.1 Possible Values for `media` Attribute Defined by HTML 4.01 Standard

Value	Media Type
`all`	All types (default)
`aural`	Speech synthesizer
`braille`	Tactile device generating braille characters
`handheld`	Handheld device, such as a cell phone or PDA
`print`	Printer
`projection`	Projector, such as one used to display a large monitor image on a screen
`screen`	Computer monitor
`tty`	Fixed-width character output device
`tv`	Television (monitor with low resolution and little or no scrolling)

semantics of the "Hello World!" page did not change in the previous example: it consisted of a single paragraph containing some text. Put another way, the `body` elements of the `HelloWorld.html` and `CSSHelloWorld.html` files will have exactly the same abstract syntax tree. But by changing the style sheet used by the browser to display this tree, we can achieve different presentations of the same information.

There are significant advantages to having such a separation between the information contained in a document and its presentation. First, it allows the information in the document to be presented without change in a variety of ways. We have already seen an example of this feature with user-selectable alternative style sheets. But CSS can do even more than this. For example, the `link` element defines a `media` attribute that can be used to define the types of media for which a style sheet is designed, such as for display on a monitor or output to a printer (see Table 3.1 for a complete list of media types defined by the HTML 4.01 standard). So, for example, if we had used the `link` elements

```
<link rel="stylesheet" type="text/css" href="style1.css"
      media="screen, tv, projection" />
<link rel="stylesheet" type="text/css" href="style2.css"
      media="handheld, print" />
```

then the style sheet of `style1.css` would be used for display on monitors, televisions, and projectors, the style sheet of `style2.css` for output to handheld devices and printers, and the browser's default style sheet for all other forms of output. (The example file `CSSHelloWorldPrint.html` demonstrates this feature: try loading it into your browser and then printing it.) You'll notice that the `title` attribute does not appear in the `link` elements in this example. This is because these style sheets cannot be selected by the user, but instead will apply regardless of user actions. Such style sheets are called *persistent* and can be recognized by their lack of a `title` attribute specification in the `link` element referencing the style sheet.

From a developer's perspective, another useful feature of using style sheets is that it is relatively easy to give all of the elements on a page a consistent appearance. That is, if we want all of the `h1` headers on a page to have a certain size, we can accomplish this easily

using a style sheet. Furthermore, if at a later time we wish to change the size of the headers, we need only make the change in that one style sheet. More generally, if we use a single style sheet for all of the pages at a site, then all of the site pages will have a consistent style, and one that can be changed with little work.

In addition to these properties, which apply to any style sheet language—including older print-oriented style sheet languages—the cascading quality of CSS makes it particularly appealing for use with web documents. As we will learn, both the document author and the person viewing the document can specify aspects of the document style as it is displayed by the browser (or other user agent displaying the document). For example, a user may instruct their browser to display all HTML documents using only a white background, regardless of the setting of the `background-color` property in style rules supplied by the page author. This can be an important feature to, for example, a user who because of an eyesight limitation needs high contrast between text and its background.

It should also be noted that, though I am going to cover CSS in the context of providing style for HTML documents, it can also be used with non-HTML XML documents (Section 7.10 contains an example).

So, there are many reasons to learn about style sheet technology in general, and CSS in particular. We'll start by covering some of the core CSS syntactic elements. After that, we'll study the cascading aspects of CSS in more detail. Finally, we'll consider details of a number of specific style properties and apply CSS to the blogging case study.

3.3 CSS Core Syntax

As with HTML, there are several W3C-recommended versions of CSS. At the time of this writing, there are technically two W3C recommendations for CSS: CSS level 1 [W3C-CSS-1.0] and CSS level 2 [W3C-CSS-2.0] (often referred to as CSS1 and CSS2). Work is also underway on CSS level 3, and several specialized versions of CSS for limited devices, such as cell phones, low-cost printers, and televisions, are in various stages of standardization.

Although CSS2 has been a W3C recommendation since 1998, at this time no widely used browser implements the entire recommendation. Recognizing this fact, the W3C has been developing CSS 2.1, which is largely a scaled-back version of CSS2 that attempts to capture those features of CSS2 that are—as of the time of the recommendation's official publication—implemented by multiple browsers. Using the February 2004 candidate version of CSS 2.1 [W3C-CSS-2.1] as a guide, in this chapter I will specifically focus on key aspects of CSS2 that are implemented in both IE6—the latest generally-available version of Internet Explorer at the time of the writing—and Mozilla 1.4. For the most part, the basic CSS syntax is the same for both levels 1 and 2, so much of what is presented should also be compatible with older browsers. Furthermore, just as browsers generally ignore HTML elements that they do not recognize, they also generally ignore CSS style properties that they do not recognize. So, if you use CSS as described in this chapter, almost all browsers should be able to display your document (although some older ones may not style it properly). It will of course be advisable for you to monitor the progress of the CSS 2.1 and CSS 3 recommendations so that you can use newer style sheet features as they become widely available; see the References section (Section 3.12) for more on this.

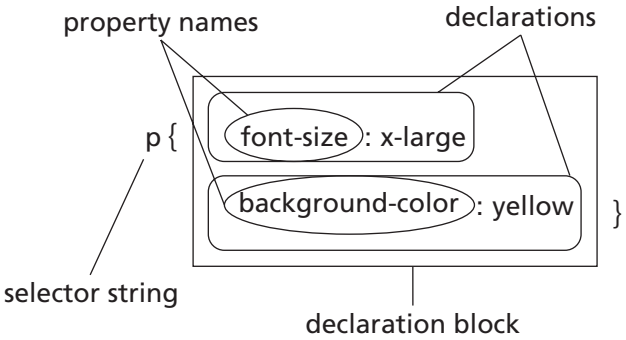

FIGURE 3.5 Parts of a single ruleset-type style rule.

One other word of warning is that versions of the Internet Explorer browser before IE6 supported style sheets but deviated from the CSS recommendation in several ways. Even in IE6, these deviations will be present unless you use a document type declaration such as the one for XHTML 1.0 Strict used in our examples. At the time of this writing, IE5 is still used on a substantial number of machines, although its usage is dwindling rapidly. So, if you develop real-world CSS style sheets in the near term, you may need to deviate somewhat from the material presented in this chapter. However, the concepts taught here are similar to those found in IE5, and as time goes on the details presented here should apply to the bulk of browsers in use. Again, see Section 3.12 for more information.

A CSS style sheet consists of one or more style rules (sometimes called *statements*). Each line in the `style1.css` file in Section 3.1 is an example of a rule. This form of rule is called a *ruleset* and consists of two parts: a *selector string* followed by a *declaration block*, which is enclosed in curly braces (`{` and `}`) (see Fig. 3.5). The declaration block contains a list (possibly empty) of *declarations* separated by semicolons (;) (the final declaration can also be followed by a semicolon, and many style sheet authors follow this convention). The selector string indicates the elements to which the rule should apply, and each declaration within the declaration block specifies a value for one style property of those elements. While the example shows one rule per line, it is syntactically legal to split a rule over several lines or (though not recommended) write multiple rules on a single line. No special character is needed to mark the end of a rule (no semicolon as in Java), due to the use of the braces to distinguish the parts of the rule.

We'll have much more to say about the properties that may be set within declarations in a later section. For the moment, the properties that we use, such as `color` (text color) and `font-style`, should be fairly self-explanatory. Before considering other properties, we will focus on selector strings.

3.3.1 Selector Strings

In the following paragraphs, we will be referring to an example style sheet and HTML document shown in Figure 3.6 and Figure 3.7, respectively. Notice that comments are written using the Java-style multiline syntax; HTML-style comments are not recognized in

```
/* Headers have dark background */

h1,h2,h3,h4,h5,h6 { background-color:purple }

/* All elements bold */

* { font-weight:bold }

/* Elements with certain id's have light background */

#p1, #p3 { background-color:aqua }

/* Elements in certain classes are italic, large font,
   or both */

#p4, .takeNote { font-style:italic }
span.special { font-size:x-large }

/* Hyperlink ('a' element) styles */

a:link { color:black }
a:visited { color:yellow }
a:hover { color:green }
a:active { color:red }

/* Descendant selectors */

ul span { font-variant:small-caps }
ul ol li { letter-spacing:1em }
```

FIGURE 3.6 Style sheet file `sel-demo.css` used to demonstrate various types of CSS selectors.

CSS, nor are Java end-of-line (`//`) comments. A browser rendering of this HTML document using the given style sheet is shown in Figure 3.8.

Probably the simplest form of selector string, which we have already seen, consists of the name of a single element type, such as `body` or `p`. A rule can also apply to multiple element types by using a selector string consisting of the comma-separated names of the element types. For example, the rule

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

says that any of the six heading element types should be rendered with a purple background. Therefore, in our example document, the markup

```
<h1>Selector Tests</h1>
```

has a purple background when displayed in the browser.

In the preceding style rule, each of the *selectors* (comma-separated components of the selector string) was simply the name of an element type. This form of selector is called a *type selector*. Several other forms of selector are also defined in CSS. One is the *universal*

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Selectors.html
    </title>
    <link rel="stylesheet" type="text/css" href="sel-demo.css" />
  </head>
  <body>
    <h1>Selector Tests</h1>
    <p id="P1" class="takeNote">
      Paragraph with id="P1" and class="takeNote".
    </p>
    <p id="p2" class="special">
      Second paragraph.  <span class="takeNote special cool">This span
      belongs to classes takeNote, special, and cool.</span>

      <ul>
        <li>Span's within this list are in <span>small-cap</span>
            style.</li>
        <ol>
          <li>This item spaces letters.</li>
        </ol>
      </ul>
    </p>
    <p id="p3">
      Third paragraph (id="p3") contains a
      <a href="http://www.example.net">hyperlink</a>.
      <ol>
        <li>This item contains a span but does not display it in
          <span>small caps</span>, nor does it space letters.</li>
      </ol>
    </p>
  </body>
</html>
```

FIGURE 3.7 HTML document used to demonstrate various types of CSS selectors.

selector, which is denoted by an asterisk (`*`). The universal selector represents every possible element type. So, for example, the rule

```
* { font-weight:bold }
```

specifies a value of `bold` for the `font-weight` property of every element in the document.

Another form of selector is the *ID selector*. Recall that every element in an XHTML document has an associated `id` attribute, and that if a value is assigned to the `id` attribute for an element then no other element's `id` can be assigned the same value. If a selector is preceded by a number sign (`#`), then it represents an `id` value rather than an element type name. So, for example, if a document contains the markup

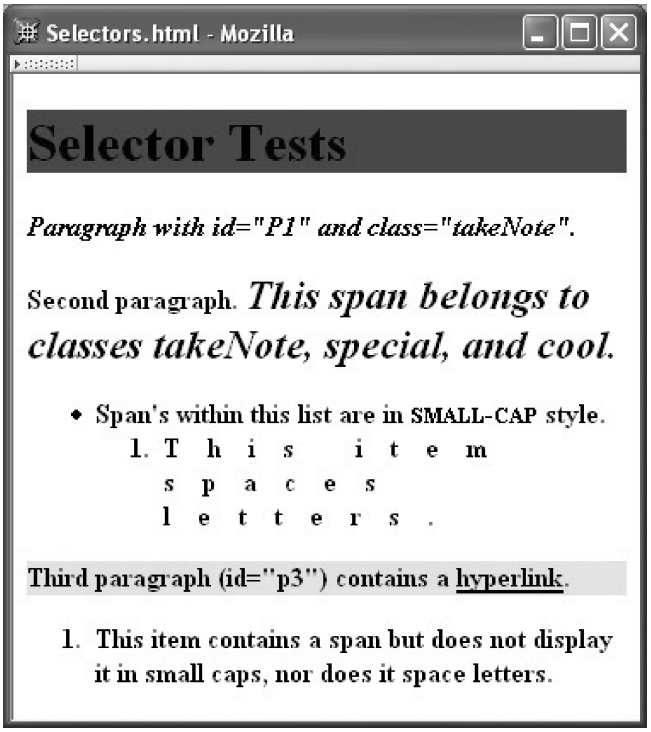

FIGURE 3.8 Browser rendering of `Selectors.html` after applying style sheet `sel-demo.css`.

```
<p id="p3">
  ...
</p>
```

then the following rule will cause this paragraph (and another element with `id` value `p1`, if such an element exists) to be displayed with an aqua background:

```
#p1, #p3 { background-color:aqua }
```

Note that `id` values are case-sensitive, so this rule will not apply to an element that has an `id` value of `P1`. This is why the first paragraph in Figure 3.8 does not have a background color.

Another HTML attribute that is frequently used with style sheets is `class`. This attribute is used to associate style properties with an element as follows. First, the style sheet must contain one or more rulesets having *class selectors*, which are selectors that are preceded by a period (`.`), such as `.takeNote` in the rule

```
#p4, .takeNote { font-style:italic }
```

Then any element that specifies `takeNote` (without the leading period) as the value of its `class` attribute will be given the properties specified in the declaration block of the corresponding style rule. Thus, the first paragraph of the example is displayed in an italic font. An element can be assigned to multiple style classes by using a space-separated list of class names as the value of the `class` attribute. For example, a `span` element with start tag

```
<span class="takeNote special cool">
```

will be affected by any rules for for the `takeNote`, `special`, and `cool` classes. Thus, the second sentence of the second paragraph of the example is italicized, since it belongs to the `takeNote` class, among others. If a class name does not correspond to a class selector in any of the style rules for a document (for example, `.cool` is not used as a class selector in `sel-demo.css`), then that class value is ignored.

Note that, like `id` values, `class` values are case sensitive and cannot begin with a decimal digit. However, unlike `id`, multiple elements can have the same value for their `class` attributes. All but a few elements, such as `html`, `head`, and elements that appear as content of `head`, have the `class` attribute.

ID and class selectors can also be prefixed by an element type name, which restricts the selector to elements of the specified type. For example, the style rule

```
span.special { font-size:x-large }
```

applies only to `span` elements that have a `class` value of `special`. So, in our example, the second paragraph itself is not set in the extra large (`x-large`) font size, but the second sentence of that paragraph is displayed using the extra large font, because the sentence is contained in a span with `class` value `special`. Also, an asterisk can be used in place of an element name in such a prefix, and (as with the universal selector) represents the set of all element names. In other words, the selectors `*.takeNote` and `.takeNote` are equivalent.

In addition to ID and class selectors, several predefined *pseudo-classes* are associated with `a` (anchor) elements that have an `href` attribute (source anchors). Table 3.2 lists these pseudo-class selectors. Figure 3.8 shows a link that has not been visited recently, and is therefore displayed in black. Positioning the cursor over that link without clicking the mouse button will cause the link to change to green, and clicking and holding the mouse button will change the color to red. If the link is visited, then the next time `Selectors.html` is

TABLE 3.2 Pseudo-Classes Associated with a Element Type

Selector	Associated a Elements
`a:visited`	Any element with `href` corresponding to a URL that has been visited recently by the user
`a:link`	Any element that does not belong to the `a:visited` pseudo-class
`a:active`	An element that is in the process of being selected; for example, the mouse has been clicked on the element but not released
`a:hover`	An element over which the mouse cursor is located but that does not belong to the `a:active` pseudo-class

loaded into the browser the link will be yellow. A fine point is that the current CSS 2.1 draft recommendation [W3C-CSS-2.1] allows a browser to ignore a pseudo-class style rule that would change the positioning of any elements within the browser. Color changes are therefore good choices as declarations for a rule that uses a pseudo-class selector, while even a seemingly innocuous declaration involving boldfacing should be used with caution (since boldfacing can increase the width of text and therefore move other elements).

Finally, a selector may be specialized so that it holds only within the content of certain element types. For example, the rule

```
ul span { font-variant:small-caps }
```

says that the text within a `span` element that is in turn part of the content of an unordered, or bulleted, list (`ul` element) should be displayed using a small-cap font form. Such a selector is known as a *descendant selector*. Notice that only the span within the bulleted list item in Figure 3.8 is displayed in the small-cap format.

Class selectors can also be included in the ancestor list; for example, the selector

```
.special span
```

would apply to any `span` element within the content of any element belonging to the class `special`. More generally, a white-space-separated list of element and/or class names may be used as a selector, representing a chain of elements each of which must be a descendant of the element to its left in order for the selector to apply. For example, the rule

```
ul ol li { letter-spacing:1em }
```

applies only to an `li` element within an `ol` (ordered, or numbered, list) element that is within a `ul` element. Thus, the numbered item in the second paragraph displays in the letterspaced format, because this paragraph's numbered list is contained within a bulleted list; but the numbered list in the third paragraph does not use this format, because it is not contained within a bulleted list.

3.3.2 At-Rules

So far, we have covered the ruleset form of style rules. The other form of rule is called an *at-rule*. The only at-rule that is widely supported and used at the time of this writing is the rule beginning with `@import`. This rule is used to input one style sheet file into another one. For example, a style sheet such as

```
@import url("general-rules.css");
h1, h2 { background-color: aqua }
```

will first read in rules from the file `general-rules.css` before continuing with the other rule in this style sheet. The `url()` function is used to mark its string argument as a URL. Single quotes can be used for this argument rather than double quotes; in fact, the quotes are not required at all. The URL can be absolute or relative. If it is a relative URL, like the one

shown in this example, then it will be taken as relative to the URL of the file containing the import at-rule, rather than relative to the HTML document. The `@import` rule must end with a semicolon, as shown. Also, all `@import` rules must appear at the beginning of a style sheet, before any ruleset statements.

3.4 Style Sheets and HTML

So far, the style sheets we have used have been stored in files and included in an HTML document through the use of a `link` element. Such style sheets are known as *external* style sheets. Another option is to embed a style sheet directly in an HTML document as the content of the HTML `style` element, which can appear any number of times in the `head` content of a document. For example, an XHTML document might contain the following markup:

```
<head>
  <title>InternalStyleSheet.html</title>
  <style type="text/css">
    h1, h2 { background-color:aqua }
  </style>
</head>
```

As you would expect, this will have the same effect as if the given style rule had been contained in an external style sheet and included in the HTML document via a `link` element. A style sheet that is included in the content of a `style` element is known as an *embedded* style sheet.

I have two notes of caution about using embedded style sheets. First, if any XML special character, such as less-than (<) or ampersand (&), appears in the style rules, then the character must be replaced by the appropriate entity or character reference. On the other hand, such references should *not* be used in an external style sheet, because an external style sheet is not an XML document and therefore is not processed like one. Second, the HTML 4.01 specification suggests enclosing the content of a `style` element within an SGML comment, for example,

```
<style type="text/css">
  <!--
  h1, h2 { background-color:aqua }
  -->
</style>
```

This was suggested because some older browsers did not recognize the `style` element. Such a browser would ignore the `style` start and end tags but would still attempt to process the content of the element, as discussed in Chapter 2. Therefore, a `style` element could produce strange behavior in such browsers. To circumvent this problem, CSS was defined so that the SGML comment start and end delimiters <!-- and --> are ignored by style sheet processors (the delimiters themselves are ignored, but the content within the delimiters is not ignored). So an older browser would ignore both the `style` tags and the content in a `style` element

written as shown, while a style-cognizant HTML 4.01 browser would process the `style` element as if the comment delimiters were not present.

However, using SGML comment delimiters in embedded style sheets is not recommended in XHTML, as XHTML parsers are allowed to strip out comments and their content regardless of what elements may contain the comments. So, in an XHTML-compliant browser an embedded style sheet enclosed within comment delimiters may be ignored. Given that almost all browsers in use today recognize the `style` element, and given this potential difficulty in XHTML browsers, I suggest that you not use SGML comment delimiters within `style` elements.

The `media` attribute described earlier can be used with the `style` element as well as with `link` elements, and therefore applies to both external and embedded style sheets. However, the `rel` attribute applies only to the `link` element, not to `style`. So an embedded style sheet is treated much the same as a persistent external style sheet: it cannot be selected or deselected by the browser user, but instead always applies to the document.

As we learned in the previous chapter, most HTML elements have a `style` attribute that can be used to define style properties for the element. Technically speaking, the value of a `style` attribute is not a style sheet, since it is not a set of style rules but is instead essentially a single list of style declarations that applies to a single document element. In fact, the use of style sheets is recommended over the use of `style` attributes, for a number of reasons. One reason is ease of coding: if you want all of the paragraphs in your document to have the same style applied, it is much easier to accomplish this by writing a single style rule than by adding a `style` attribute specification to every `p` element. Similarly, it is generally much easier to modify the style of a document that uses style sheets to define style than it is to modify one that uses `style` attributes. A `style` attribute value also cannot vary automatically with media type. This last observation is a special case of the more general recommendation that since markup is designed to carry structural and semantic information, it is generally best to keep all style information out of the body of an HTML document. All that said, there are times when the `style` attribute is convenient (e.g., to make an image cover an entire table cell, as in Section 2.7). So, while you shouldn't necessarily avoid its use altogether, try to use the `style` attribute wisely.

3.5 Style Rule Cascading and Inheritance

Before describing in detail many of the key CSS style properties, it will be helpful to understand two concepts: cascading of style sheet rules and element inheritance of style properties.

3.5.1 Rule Cascading

The style sheet of Figure 3.6 contains the rule

```
* { font-weight:bold }
```

which applies to every element of the HTML document. It also contains the rule

```
#p1, #p3 { background-color:aqua }
```

As we have seen, both of these rules applied to an element with `id` attribute value `p3`. That is, if multiple rules apply to an element, and those rules provide declarations for different properties, then all of the declarations are applied to the element. But what would happen if the rule

```
#p3 { font-weight:normal }
```

also appeared in a style sheet for the document? Which rule would apply to the `font-weight` property of the `p3` element?

This is one example of a more general question: For every property of every element on a page, the browser must decide on a value to use for that property. How does it determine this value if multiple style declarations apply to that property of that element? Furthermore, what should the browser do if no declaration at all directly applies to that element property? We'll deal with the first question in this subsection, and the second question in the next.

In order to choose between multiple declarations that all apply to a single property of a single element, the browser (or other user agent) applies *rule cascading*, a multistage sorting process that selects a single declaration that will supply the property value. The very first step of this process involves deciding which external and embedded style sheets apply to the document. For example, if alternate external style sheets are available, only one will apply, and rules in the other alternate style sheets will be ignored. Similarly, if a media type is specified for an embedded or external style sheet and that type is not supported by the user agent rendering the page, then that style sheet's rules will be ignored.

Once the appropriate external and embedded style sheets have been identified, the next stage of the sorting process involves associating an origin and weight with every declaration that applies to a given property of a given element. The *origin* of a style sheet declaration has to do with who wrote the declaration: the person who wrote the HTML document, the person who is viewing the document, or the person who wrote the browser software that is displaying the document. Specifically, the origin of a declaration is one of the following:

- *Author:* If the declaration is part of an external or embedded style sheet or is part of the value specified for the `style` attribute of the given element, then it originated with the author of the document that is being styled.
- *User agent:* A browser or other user agent may define default style property values for HTML elements. In the Mozilla 1.4 **View**|**Use Style** menu, this is the style sheet represented by the "Basic Page Style" option. Appendix A of the CSS 2.0 recommendation [W3C-CSS-2.0] contains an example user agent style sheet.
- *User:* Most modern browsers allow users to provide a style sheet or to otherwise indicate style preferences that are treated as style rules.

In Mozilla 1.4, the user style rules can be defined in two ways. First, under the **Edit**|**Preferences**|**Appearance** category, the Fonts and Colors panels allow a user to select various style options, which will be treated as user style rules. Second, the user can explicitly create a style sheet file that the browser will input when it is started. However, this is not an easy-to-use feature in Mozilla 1.4: you must create a file with a certain filename (`userContent.css`) and place it in a certain directory (the `chrome` subdirectory of the

directory specified by the Cache Folder field of **Edit| Preferences | Advanced | Cache**). Similar features are provided in IE6 under the **General** tab of the **Tools | Internet Options** window. The Colors and Fonts buttons allow the user to set style options, and a style sheet file can be read into IE by clicking the Accessibility button, checking the checkbox in the User Style Sheet panel, and selecting the file.

In addition to an origin, every author and user style declaration has one of two *weight* values: normal and important. A declaration has important weight if it ends with an exclamation mark (`!`) followed by the string `important` (or similar strings: case is not important, and there may be white space before or after the exclamation mark). So the rule

```
p { text-indent:3em; font-size:larger !important }
```

gives important weight to the declaration of the `font-size` property. A declaration without the `!important` string—such as the declaration of the `text-indent` property in that example—would have normal weight. All user-agent declarations can also be considered to have normal weight.

Once the origin and weight of all declarations applying to an element property have been established, they are prioritized (from high to low) as follows:

1. Important declaration with user origin
2. Important declaration with author origin
3. Normal declaration with author origin
4. Normal declaration with user origin
5. Any declaration with user agent origin

That is, we can think of each declaration as falling into one of five priority bins. We then look through the bins, starting with the first, until we find a nonempty bin. If that bin has a single declaration, the declaration is applied to the element property and we are done. Otherwise, there are multiple declarations in the first nonempty bin, and we continue to the next sorting stage in order to select a single declaration from among the candidates within this bin.

Before getting to this next stage, you may be wondering why important user declarations have higher priority than author declarations while normal-weight user declarations have lower priority. The reason is accessibility. If a visually impaired web user must have high contrast between text and background along with large bold fonts in order to read text on a monitor, that user can be accommodated by writing declarations with important weight, regardless of the page author's design decisions. On the other hand, a user who is merely stating style preferences will generally not want their default preferences to override those of a web site author who made specific style choices for his or her web site. One significant change between CSS1 and CSS2 was the adoption of the sort order just listed, which is also supported by the major modern browsers.

Now we return to the case in which the top nonempty bin of the weight-origin sort contains multiple style declarations for a single element property. The next step is to sort these declarations according to their *specificity*. First, if a declaration is part of the value of a `style` attribute of the element, then it is given the highest possible specificity value

(technically, in CSS2 this specificity value can be overridden, but that feature does not seem to be widely implemented by current browsers). If a declaration is part of a ruleset, then its specificity is determined by the selector(s) for the ruleset. We begin by treating a ruleset with a comma-separated selector string as if it were multiple rulesets each with a single selector; that is, a ruleset such as

```
h1, #head5, .big { font-size:x-large }
```

is treated as the equivalent three rulesets

```
h1 { font-size:x-large }
#head5 { font-size:x-large }
.big { font-size:x-large }
```

Next, we conceptually place each ruleset in one or more bins, each bin labeled with a class of selectors. The bins we use for this purpose, from highest to lowest specificity, are:

1. ID selectors
2. Class and pseudo-class selectors
3. Descendant and type selectors (the more element type names, the more specific)
4. Universal selectors

A ruleset with a selector such as `li.special` would go in two bins, since this is both a class and a type selector. Now we select a ruleset from the first nonempty bin. If, say, two rulesets appears in this bin, we search lower bins for the first recurrence of either ruleset. If one of the rulesets recurs before the other, then it is chosen. So, for example, `li.special` would be chosen over `*.special`.

Even after this sorting process, two or more declarations may still have equally high weight-origin ranking and specificity. The final step in the style cascade is then applied, and is guaranteed to produce a single declaration for a given property of a given element. First, if there is a declaration in the `style` attribute for the element, then it is used. Otherwise, conceptually, all of the style sheet rules are listed in the order in which they would be processed in a top-to-bottom reading of the document, with external and imported style sheets inserted at the point of the `link` element or `@import` rule that causes the style sheet to be inserted. The declaration corresponding to the rule that appears farthest down in this list is chosen. As an example, if the file `imp1.css` contains the statements

```
@import url("imp2.css");
p { color:green }
```

and the file `imp2.css` contains the statement

```
p { color:blue }
```

and a document `head` contains the markup

```
<title>StyleRuleOrder.html</title>
<style type="text/css">
  p { color:red }
</style>
<link rel="stylesheet" type="text/css" href="imp1.css" />
<style type="text/css">
  p { color:yellow }
</style>
```

then the style rulesets are effectively in the order

```
p { color:red }
p { color:blue }
p { color:green }
p { color:yellow }
```

and `p` elements will be displayed with yellow text. Notice that since the import at-rules must always come at the beginning of a style sheet, any imported rules can always be overridden by rules in the body of the style sheet causing the import. This is sensible, since rules imported from a file are presumably meant to be of a reusable, general-purpose nature and therefore should be subject to revision for a specific task.

Finally, certain (often deprecated) HTML attributes other than `style` can be used to affect the presentation of an HTML document. For example, the `height` attribute of the `img` element type can affect presentation. But `img` also has a `height` style property that can be set to achieve the same effect. If both are defined for an `img` element, which should take precedence: the attribute or the style property? The general answer is that any CSS style declaration takes precedence over style-type declarations made via HTML attribute specifications. More specifically, the browser or user agent treats non-CSS attribute styling as if an equivalent CSS style rule had been inserted at the very beginning of the author (normal weight) style sheet with a specificity lower than that for the universal selector. So any important-weight user style rule as well as any style rule written by the document author will take precedence over style rules derived from attributes such as `height`, which in turn will take precedence over normal-weight user and user-agent style rules.

The style cascade is summarized in Figure 3.9. We're now ready to tackle the other question posed earlier: if a property of an element has no associated style declarations, how is the value of the property determined? The answer is that the value is inherited from ancestors of the element, as discussed next.

3.5.2 Style Inheritance

While cascading is based on the structure of style sheets, inheritance is based on the tree structure of the document itself. That is, conceptually an element *inherits* a value for one of its properties by checking to see if its parent element in the document has a value for that property, and if so, inheriting the parent's value. The parent may in turn inherit its property value from its parent, and so on. Put another way, when attempting to inherit a property value, an element (say with `id` value `needValue`) will search upward through its tree of ancestor elements, beginning with its parent and ending either at the root `html` element or

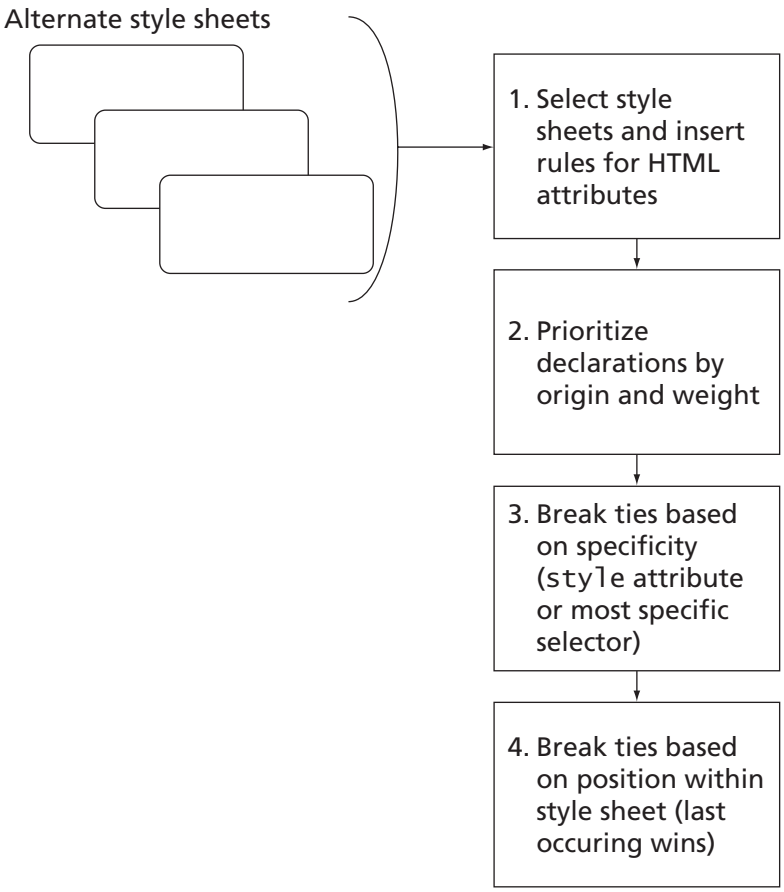

FIGURE 3.9 Steps in the CSS cascade.

at the first element that has a value for the property. If the search ends at an element with a value for the property, that value will be used by `needValue` as its property value. If no ancestor element has a value for the property, then as a last resort the property will be given a value specified for each property by the CSS specification [W3C-CSS-2.0] and known as the property's *initial value*. This terminology makes sense if you think of each element property as having its initial value assigned when the document is first read and then having this value changed if either the cascade or the inheritance mechanism supplies a value.

Figure 3.10 shows the source of an HTML document that illustrates inheritance. Notice that the style sheet for this document contains `font-weight` declarations for both the `body` and `span` element types. So for `span` elements, the `font-weight` is specified by an author rule, and no value will be inherited for this property. For other elements within the `body`, though, there is no author rule, and assuming that there is also no user or user-agent rule, the `font-weight` property value will be inherited from the `body` element. Therefore,

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Inherit.html
    </title>
    <style type="text/css">
      body { font-weight:bold }
      li { font-style:italic }
      p { font-size:larger }
      span { font-weight:normal }
    </style>
  </head>
  <body>
    <ul>
      <li>
        List item outside and <span>inside</span> a span.
        <p>
          Embedded paragraph outside and <span>inside</span> a span.
        </p>
      </li>
    </ul>
  </body>
</html>
```

FIGURE 3.10 HTML document demonstrating inheritance.

as shown in Figure 3.11, the word "inside" (which is the content of two `span` elements) appears with a normal font weight, while all other text is boldfaced. However, since there are no other property declarations for the two `span` elements, these elements do inherit other property values from their ancestors. The first `span` inherits italicization from its parent `li` element, while the second inherits a larger font size from its `p` element parent and italicization from its `li` element grandparent. The `p` element similarly inherits italicization from its `li` parent.

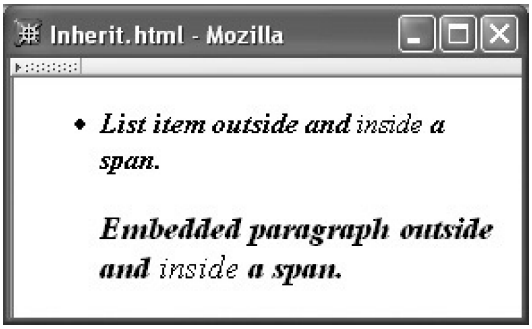

FIGURE 3.11 Rendering of document demonstrating inheritance.

A few final points should be mentioned about inheritance. First, while many CSS properties are inheritable, a number of other properties are not. In general, your intuition about whether or not a property will be inherited should be correct. For example, the `height` property of an element is not inherited from its parent, which is sensible, since often the parent has many children on many lines and therefore has a greater height than any one child. As I cover specific CSS properties in later sections of this chapter, you should assume that each property is inherited unless I explicitly say otherwise. Of course, you can consult the CSS specifications [W3C-CSS-2.0] if in doubt or for information about inheritance of properties not covered in this chapter.

A second point about inheritance has to do with exactly which of several possible property values is inherited. The value contained in a style declaration for a property is known as the *specified value* for the property. This value can be either relative or absolute. An *absolute value* is a value that can be understood without the need for any context, such as the value `2cm` (two centimeters). A *relative value*, on the other hand, cannot be understood without knowing some context. For example, the property declaration `font-size:larger` uses the relative value `larger` to set the size of the font of an element. Exactly what this value is relative to is discussed in Section 3.6.3. For now, it's sufficient to know that the browser must perform a calculation—which depends on the particular relative value—to obtain a *computed value* for the property. In the case of the font-size value `larger`, this calculation might involve multiplying the base font size by a factor such as 1.2 to obtain the computed font size. If the specified value is absolute, then the computed value is identical to the specified value. Finally, the computed value may not be suitable for immediate use by the browser. For example, a specified font size—relative or absolute—may not be available for the font currently in use, so the browser may need to substitute the closest available font size. The value actually used by the browser for a property is known, appropriately enough, as the *actual value*.

In terms of inheritance, the computed value is normally inherited for a property, not the specified or actual value. The one exception to this among the properties discussed in this chapter is `line-height`; its inheritance properties will be described in detail in Section 3.6.4.

A final note about inheritance is that the CSS2 recommendation allows every style property to be given the value `inherit`, whether or not the property is inherited normally. When this value is specified for a property, the computed value of the property is supposed to be obtained from its parent. However, you should be aware that this inheritance feature is not supported by IE6, and therefore should be used with care if at all. I am mentioning it mainly because it appears often in the CSS2 recommendation. Since this value can be used for every CSS2 property, I will not mention it explicitly when listing possible values for properties in the following sections.

We are now ready to begin learning about many of the available CSS2 properties. We'll begin with a number of text properties.

3.6 Text Properties

In this section, we will cover many of the CSS properties related to the display of text. Specifically, we will learn about how to select a font and how to modify text properties such as color. We'll also cover in some detail how browsers determine the spacing between

lines of text and how document authors can influence this spacing. Later sections will cover some other aspects of text, such as alignment, once we have covered necessary background material.

One note before beginning: CSS defines a `direction` property that can be thought of as defining the default direction in which text is written. It takes two possible keyword values: `ltr` indicates a left-to-right language, and `rtl` indicates right-to-left. This property affects the default behavior of many other CSS properties as well as some of their initial values. For example, the initial value for the `text-align` property, used to specify how a paragraph of text should be aligned, is `left` if `direction`'s value is `ltr` and is `right` otherwise. For simplicity, I will assume left-to-right languages throughout this chapter; if there is an asymmetry between left and right for a property (such as the initial value of `text-align`, which gives preference to `left`), simply switch the roles of left and right if you use a right-to-left language.

3.6.1 Font Families

Figure 3.12 is a browser rendering of an HTML document that displays characters using four different font families (we'll learn later how to write a document such as the one that generated this figure). A *font family* is a collection of related fonts, and a *font* is a mapping from a character (Unicode Standard code point) to a visual representation of the character (a *glyph*). Each glyph is drawn relative to a rectangular *character cell* (also known as the character's *content area*), which is shown shaded for each character in the figure. The fonts within a font family differ from one another in attributes such as boldness or degree of slantedness, but they all share a common design. The font families used in this example are, in order of use, Jenkins v2.0, Times New Roman®, Jokewood, and Helvetica™; they illustrate well how different font family designs can be from one another. (The Jenkins and Jokewood fonts may not be available on your machine, so this example may not appear the same in your browser as it does in Fig. 3.12.)

The font family to be used for displaying text within an HTML element is specified using the `font-family` style property. For example, the start tag

```
<p style="font-family:'Jenkins v2.0'">
```

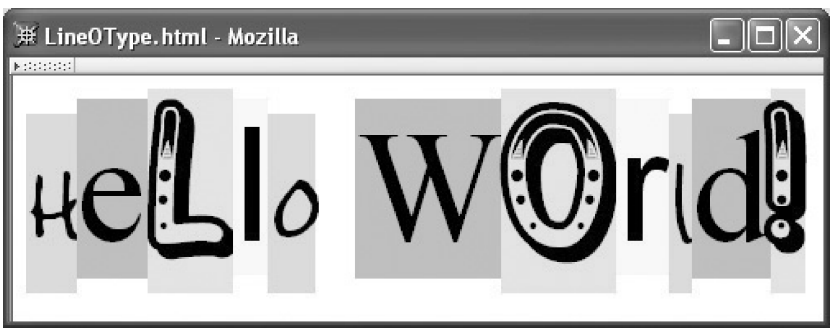

FIGURE 3.12 Rendering of document illustrating four different font families.

indicates that the text within the paragraph started by this tag should use the Jenkins v2.0 font (unless a child element specifies a different font). Some font family names must be quoted and/or special characters contained in the names must be escaped; for simplicity, I recommend that you always quote font family names. Either single or double quotes can be used, which is especially convenient when the declaration appears within a `style` attribute as shown.

Most end-user computers contain files describing a variety of font families. However, there is no guarantee that a font family that you would like to display in an HTML document you are authoring will be available on all of the client machines viewing your document. Although IE6 has a mechanism for downloading fonts from the Web for use within an HTML document, this facility is not included in the current version of CSS 2.1 [W3C-CSS-2.1]. Instead, a recommended mechanism for specifying a font family in CSS is to use a comma-separated list of font families as the value of the `font-family` property, such as

```
font-family:"Edwardian Script ITC","French Script MT",cursive
```

The browser will attempt to use the first family specified (`Edwardian Script ITC` in this example), but if that family is not available on the browser's system, then the browser will proceed through the list until it finds a family that is available. The last element in the list (`cursive` in this example) should be the name of a *generic* font family. The generic font families defined by CSS are listed in Table 3.3. Unlike normal font family names, the names of generic families are CSS keywords and therefore must not be quoted within a `font-family` declaration.

The browser will attempt to associate a reasonable font family available on the user's system with each generic name. In Mozilla 1.4, the user can specify the actual font family associated with each generic family through a preference setting as illustrated in Figure 3.13.

TABLE 3.3 CSS Generic Font Families

Font Family	Description
`serif`	A *serif* is a short, decorative line at the end of a stroke of a letter. There are three serifs at the top of the W in Figure 3.12, for example. Most glyphs in a serif font family will have serifs, and such a family is typically *proportionately* spaced (different glyphs occupy different widths).
`sans-serif`	Usually proportionately spaced, but glyphs lack serifs, so they don't look as fancy as serif fonts.
`cursive`	Looks more like cursive handwriting than like printing.
`fantasy`	Glyphs are still recognizable as characters, but are nontraditional.
`monospace`	All glyphs have the same width. Since monospace fonts are often used in editors when programming, these font families are frequently used to display program code or other computer data.

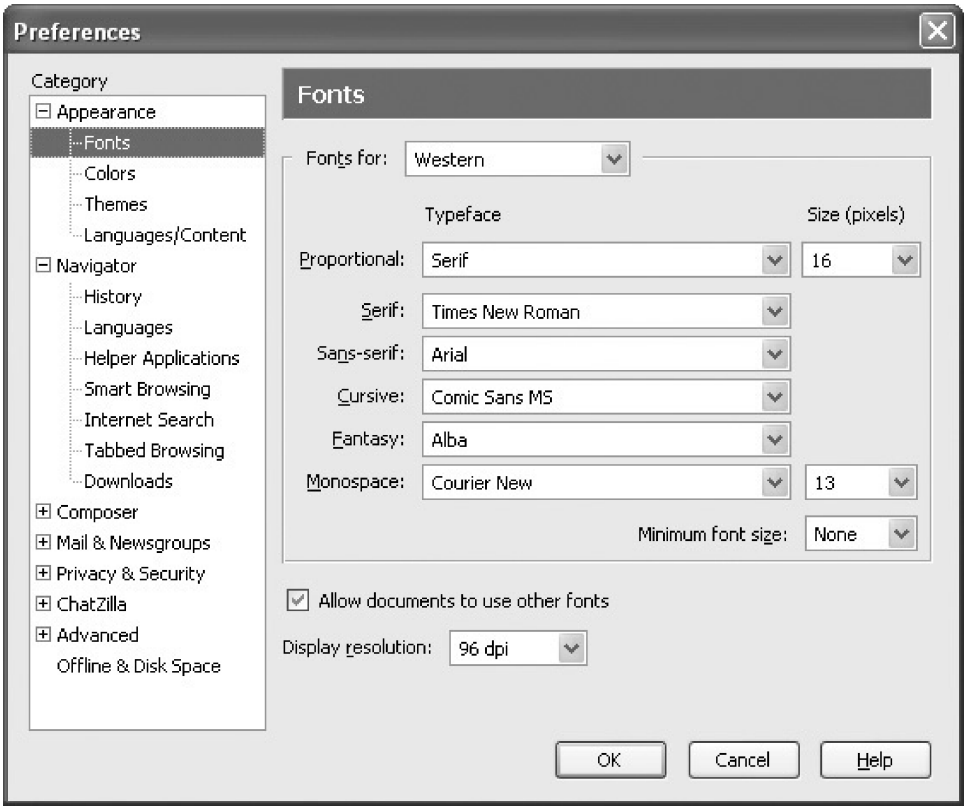

FIGURE 3.13 Example of associations of actual with generic font families in Mozilla.

3.6.2 Length Specifications in CSS

Font size is one of the key features used to distinguish between individual fonts within a font family. In CSS, the size of a font is specified using the `font-size` property. One type of value that can be assigned to `font-size` is a *CSS length*. In fact, CSS lengths can be assigned to many CSS properties, not just `font-size`. Therefore, we will cover length specification separately in this section before moving on to how to specify font properties such as size in CSS.

In CSS, a length value is represented either by the number `0` or by a number followed by one of the unit identifiers given in Table 3.4. Some example declarations involving length values are:

```
font-size:0.25in
font-size:12pt
font-size:15px
```

TABLE 3.4 CSS Length Unit Identifiers

Identifier	Meaning
`in`	Inch
`cm`	Centimeter
`mm`	Millimeter
`pt`	Point: 1/72 inch
`pc`	Pica: 12 points
`px`	Pixel: typically 1/96 inch (see text)
`em`	Em: reference font size (see text)
`ex`	Ex: roughly the height of the lowercase "x" character in the reference font (see text)

The first six units in Table 3.4 are, in practice, all related to one another by multiplicative scale factors. In particular, both Mozilla 1.4 and IE6 appear to maintain the relationships 1 in. = 2.54 cm = 25.4 mm = 72 pt = 6 pc = 96 px both on screen and when printing a document. Both also appear to use pixels to define all of the other units when displaying a document on a monitor. For example, if your monitor resolution is set to 1024 by 768 pixels and you specify a horizontal length as `1024px`, then this length will roughly correspond to the width of the monitor's display area. The display area will also be treated as if it were $1024/96 \approx 10.7$ in. across, regardless of its true physical width. Thus, the units `in`, `cm`, `mm`, `pt`, and `pc` are all only approximations on screen, and depending on the resolution may be off by 50% or more (all by the same factor). When printing, however, it appears that both browsers define `px` as 1/96 in. (or close to it) and define the other units accordingly.

Note that, despite the imprecisions, lengths defined using the first five units in Table 3.4 are absolute lengths in the sense defined earlier: they do not depend on other style property values. Such units are referred to as *absolute units*. The other three units are *relative units*. Technically, CSS defines `px` as a relative unit, since its physical value should depend on the medium displaying the document. However, as indicated, in practice it seems to be treated by typical browsers as an absolute unit. We'll see why `em` and `ex` are relative units in a moment.

Before defining the `em` and `ex` units formally, it will be helpful to understand several details about fonts. First, all character cells within a given font have the same height. However, generally speaking, this height is not exactly the same as the computed or even the actual value of the CSS `font-size` property. For example, in Figure 3.12, a single `font-size` value—72 pt—applies to all of the characters, yet obviously the character cells vary somewhat in height. Thus a combination of the font family and the `font-size` property determines the actual height of character cells. The `font-size` computed value is known as the *em height;* for most font families, the cell height is 10–20% greater than the em height.

Another feature that the font defines is the baseline height. The *baseline height* is the distance from the bottom of a character cell to an imaginary line known as the *baseline*, which is the line that characters appear to rest on. As shown in Figure 3.12, when a single line of text contains characters from different fonts, the characters are by default aligned

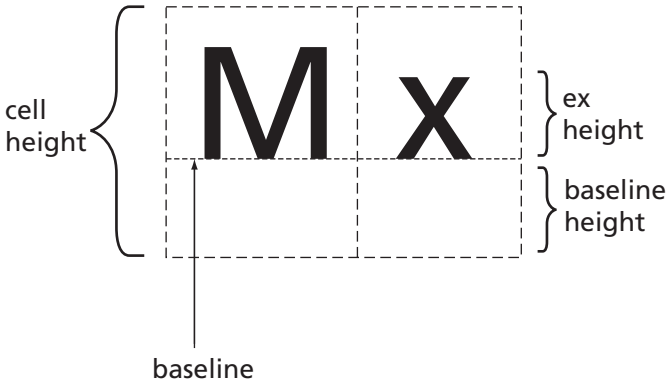

FIGURE 3.14 Some features and quantities defined by a font.

vertically by aligning their baselines. Thus, although we can see from the figure that the characters cells do not align vertically, the character glyphs themselves appear to all be written on a single horizontal line.

Yet another quantity defined by each font is the *ex height*. This quantity should be thought of as the font designer's definition of the height (above the baseline) of lowercase letters such as "x." Figure 3.14 illustrates this quantity and several other font features.

Now we can define the `em` and `ex` units. First, as noted in Table 3.4, these units are defined relative to a reference font (and are therefore relative units). With one exception explained in the paragraph after next, the reference font is just the font of the element to which the relative unit applies. So, for example, in the markup

```
<p style="width:20em">
```

the reference font is the font that applies (via a style rule or inheritance) to the `p` element.

Once the reference font is known, 1 em is simply the em height of the font, that is, the computed value of the `font-size` property of the reference font. So, continuing our example, if the computed value of the `p` element's `font-size` property is 0.25 in., then the computed value for its `width` property will be 5 in. Similarly, 1 ex is the ex height of the reference font.

The one exception when determining the reference font for these reference units is when one of them is used in a `font-size` declaration. In this case, the reference font is the font of the parent of the element to which the declaration applies. So in the markup

```
<div id="d1" style="font-size:12pt">
  <div id="d2" style="font-size:2em">
```

the reference font for the `div` with `id` attribute value `d2` will be `d1`'s font. Since the computed `font-size` for `d1` will be 12 pt (because absolute units are used), the computed `font-size` for `d2` will be 24 pt.

Now we are ready to more fully describe `font-size` and several other font properties.

3.6.3 Font Properties

The CSS font-size property, we now know, is used to specify the approximate height of character cells in the desired font within a font family. This property has an initial (default) value of medium, which is associated with a physical font size by the browser (these may vary with font family; Mozilla 1.4 defaults to 14 pt for proportional font families and 12 pt for monospace). A variety of other values can be specified for this property.

First, of course, a length value can be specified for font-size, using any of the length units described in the previous section. A second way that a font-size property may be specified is as a percentage of the computed font-size of the parent element. Since 1em represents the computed value of the parent's font-size, the following specifications are essentially equivalent:

```
font-size:0.85em
font-size:85%
```

Third, the font-size specification may be given using what is termed an *absolute size* keyword. One of these keywords is medium, the initial value for font-size. The remaining keywords are xx-small, x-small, small, large, x-large, and xx-large. The browser or other user agent creates a table of actual lengths corresponding to each of these size keywords. The CSS2 recommendation [W3C-CSS-2.0] is that each of these be approximately 20% larger than its next-smaller size.

Finally, the *relative size* keywords smaller and larger may be specified. Again, like the relative units em and ex, each of these keywords specifies the font size for the current element relative to the font size of its parent. These relative size keywords conceptually say "move one position in the font-size table." So, if the parent element has a font size of large, then a relative size specification of larger for its child is equivalent to an absolute size specification of x-large. If the parent font size is outside the range of the browser's font-size table, then an appropriate numerical font change (for example, 20%) is applied instead.

CSS also provides several other font style properties; three of the most commonly used are shown in Table 3.5. Several other font-related properties, including color, are covered later in this section.

TABLE 3.5 Additional Font Style Properties

Property	Possible Values
font-style	normal (initial value), italic (more cursive than normal), or oblique (more slanted than normal)
font-weight	bold or normal (initial value) are standard values, although other values can be used with font families having multiple gradations of boldness (see CSS2 [W3C-CSS-2.0] for details)
font-variant	small-caps, which displays lowercase characters using uppercase glyphs (small uppercase glyphs if possible), or normal (initial value)

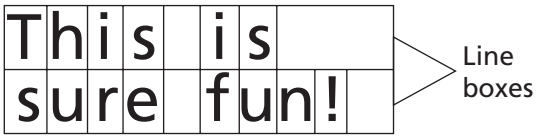

FIGURE 3.15 A box representing a p element that consists of two line boxes, each partially filled with character cells.

3.6.4 Line Boxes

We now want to consider how a browser determines where text should be rendered within an HTML element. We will assume in this section that the text is the content of an HTML p element, but the details are essentially the same for most other HTML block elements, such as div.

The simplest case is the one in which the content of the p element consists solely of text. In this case, we can think of the p element as being rendered as a rectangular box composed entirely of a stack of imaginary boxes called line boxes. Each *line box* contains one line of text. The height of a line box is precisely the height of a character cell in the p element's font. Character cells representing the text are placed side by side in the topmost line box until it contains as many words (strings of contiguous non-white-space characters) as possible; a single space (with width defined by the font) separates each word. When the first line box can hold no more words, this process continues with the second line box, and so on until all of text of the p element has been added to line boxes. There will be just enough line boxes to contain the text, so the height of the box representing the p element will be the number of line boxes multiplied by the height of a character cell. Figure 3.15 illustrates the rendering of the text "This is sure fun!" using a monospace font within a p-element box that is only wide enough to hold 10 characters. Notice that the box consists entirely of two line boxes, and that neither of the line boxes in this case is completely filled by character cells.

The browser's default setting of the height of line boxes can be overridden by specifying a value for the p element's line-height property. The initial value of this property is normal, which as we have seen sets the height of line boxes equal to the height of a character cell (a typical value might be 1.15 em, or 15% greater than the computed value of font-size). Other legal values for this property are a CSS length (using any of the units defined earlier), a percentage (treated as a percentage of the computed value of the p element's font-size), or a number without units. In the final case, the number is treated as if its units are em, except in terms of inheritance (we deal with this in a moment). Thus, the following declarations are all equivalent in terms of their effect on the p element itself:

```
line-height:1.5em
line-height:150%
line-height:1.5
```

If the height of a line box is greater than the character cell height, then the character cells are vertically centered within the line box. The distance between the top of a character

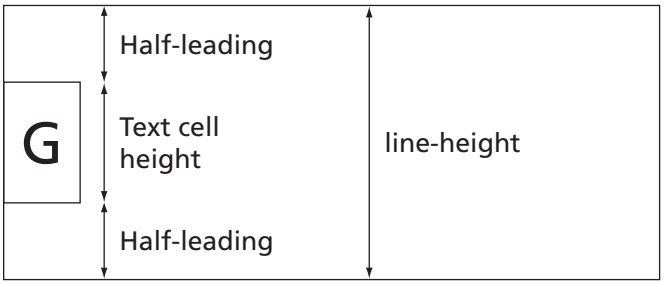

FIGURE 3.16 Default placement of text cell within a line box when the value of `line-height` exceeds the height of a text cell. An equal amount of space (half-leading) is inserted above and below the text cell.

cell and the top of a line box (which is the same as the distance between the bottom of a cell and the bottom of the line box) is sometimes called the *half-leading* (pieces of lead were often used to separate lines of type in early manual typesetting systems, hence the term). Thus, increasing the `line-height` value above its `normal` value not only increases the distance between lines, but actually moves the text of the first line down by the half-leading distance as well as increasing the distance between the last line of text and whatever follows the `p` element's box by the same distance (Fig. 3.16). We will learn how to override this default centering of text within tall line boxes later in this chapter.

A fine point about inheritance of this property: If `normal` or a number without units is specified as the value of `line-height`, then this specified value is inherited rather than the computed value. For any other specified value, such as `1.5em`, the computed value is inherited. An exercise explores this further.

Now that we have learned about the `line-height` property, I can describe a convenient property called `font`. This property is an example of a CSS *shortcut property*, which is a property that allows values to be specified for several nonshorthand properties with a single declaration. As an example of the use of the `font` shortcut, the declaration block

```
{ font: italic bold 12pt "Helvetica",sans-serif }
```

is equivalent to the the declaration block

```
{ font-style: italic;
  font-variant: normal;
  font-weight: bold;
  font-size: 12pt;
  line-height: normal;
  font-family: "Helvetica",sans-serif }
```

Notice that the `font` shortcut always affects all six of the properties shown, resetting those for which a value is not specified explicitly in the `font` declaration to their initial (default) values. The font size and font family (in this order) must be included in the specified value for `font`. If values for any of style, variant, and weight appear, they must appear

TABLE 3.6 Primary CSS Text Properties

Property	Values
`text-decoration`	`none` (initial value), `underline`, `overline`, `line-through`, or space-separated list of values other than `none`.
`letter-spacing`	`normal` (initial value) or a length representing additional space to be included between adjacent letters in words. Negative value indicates space to be removed.
`word-spacing`	`normal` (initial value) or a length representing additional space to be included between adjacent words. Negative value indicates space to be removed.
`text-transform`	`none` (initial value), `capitalize` (capitalizes first letter of each word), `uppercase` (converts all text to uppercase), `lowercase` (converts all text to lowercase).
`text-indent`	Length (initial value 0) or percentage of box width, possibly negative. Specify for block elements and table cells to indent text within first line box.
`text-align`	`left` (initial value for left-to-right contexts), `right`, `center`, or `justified`. Specify for block elements and table cells.
`white-space`	`normal` (initial value), `pre`. Use to indicate whether or not white space should be retained.

before the font size and may appear in any order among themselves. To specify a value for `line-height`, immediately follow the font size value by a slash (/) and the `line-height` value. For example,

```
{ font: bold oblique small-caps 12pt/2 "Times New Roman",serif }
```

is a valid `font` declaration that explicitly sets all six font properties.

3.6.5 Text Formatting and Color

Beyond font selection, several other CSS properties can affect the appearance of text. These are listed in Table 3.6. All of these properties except `text-decoration` are inherited. And, while not inherited, `text-decoration` automatically applies to all text within the element, while skipping nontext, such as images. The decoration uses the element's color value. Some of these properties may interfere with one another. For example, since text justification (lining up text with a straight edge on both left and right sides) generally involves inserting space between letters and/or words, specifying `justify` for `text-align` and also specifying values for `letter-spacing` and `word-spacing` may not produce the results you expect. As usual, see the CSS2 recommendation [W3C-CSS-2.0] for details on such special cases.

Finally, as we learned in early examples in this chapter, the `color` property is used to specify the color for text within an element. There are many possible values for the `color` property, which we now cover. It should be noted that these values can also be specified for several other CSS properties, as discussed later.

CSS2 color properties can be assigned several types of values. The most flexible type is a numerical representation of the color. In particular, three numerical values are used

TABLE 3.7 Alternative Formats for Specifying Numeric Color Values

Format	Example	Meaning
Functional, integer arguments	`rgb(255,170,0)`	Use arguments as RGB values.
Functional, percentage arguments	`rgb(100%,66.7%,0%)`	Multiply arguments by 255 and round to obtain RGB values (at most one decimal place allowed in arguments).
Hexadecimal	`#ffaa00`	The first pair of hexadecimal digits represents the red intensity; the second and third represent green and blue, respectively.
Abbreviated hexadecimal	`#fa0`	Duplicate the first hexadecimal digit to obtain red intensity; duplicate the second and third to obtain green and blue, respectively.

to specify a color, representing intensities of red, green, and blue to be mixed together in order to simulate the desired color (the typical human eye can be "tricked" into perceiving light from multiple sources at various intensities and wavelengths as if it were from a single source with a single intensity and wavelength). The specific color model used involves specifying an integer between 0 and 255, inclusive, for each of the intensities of red, green, and blue, in that order (early Web pages used a limited range of intensities due to hardware limitations of many computers in use at the time, but most machines today can reliably display any of these intensities). Such an integer is known as an *RGB value*. Many readily available software tools, including Microsoft Paint, provide visual maps from colors to RGB values. Four different formats can be used to specify these three values, as shown in Table 3.7. All of the examples in this table specify the same color value. (A word of caution: it's easy to forget the leading `#` for the third and fourth formats.)

Many of our earlier style sheet examples used a second, more convenient way to specify common colors: many color values have a standard name associated with them. A list of the 16 colors named in CSS2 and their associated RGB values is given in Table 3.8. The current CSS 2.1 specification also adds `orange` (`#ffa500`) to the list. Furthermore, Mozilla 1.4 supports all and IE6 supports almost all (there are some exceptions containing `gray` or `grey`) of the 147 color names recognized as part of the W3C's Scalable Vector Graphics recommendation [W3C-SVG-1.1]. This provides 130 color names in addition to those of CSS 2.1, from `aliceblue` through `yellowgreen` (see `http://www.w3.org/TR/SVG11/types.html#ColorKeywords` for a complete list).

Finally, color values can be specified by referencing colors set for other purposes on the user's system. For example, the keyword `Menu` represents the color used for menu backgrounds, and `MenuText` the color used for text within menus. This can be useful, for example, if you plan to provide menus within your page and want them to use colors familiar to your users, regardless of user selected menu color preferences. A full list of these so-called system color names is provided in Section 18.2 of the CSS2 specification [W3C-CSS-2.0]. However, be advised that the draft for CSS3 current at the time of this writing deprecates such system color names.

TABLE 3.8 CSS2 Color Names and RGB Values

Color Name	RGB Value
black	#000000
gray	#808080
silver	#c0c0c0
white	#ffffff
red	#ff0000
lime	#00ff00
blue	#0000ff
yellow	#ffff00
aqua	#00ffff
fuchsia	#ff00ff
maroon	#800000
green	#008000
navy	#000080
olive	#808000
teal	#008080
purple	#800080

We have referred throughout this section to the notion of a box corresponding to a p element. In the next section, we begin to make this concept more precise.

3.7 CSS Box Model

In this section we define a number of CSS properties that relate to the boxes that a browser renders corresponding to the elements in an HTML document. In subsequent sections we will learn how browsers position these boxes relative to the browser client area and relative to one another.

3.7.1 Basic Concepts and Properties

In CSS, each element of an HTML or XML document, if it is rendered visually, occupies a rectangular area—a *box*—on the screen or other visual output medium. What's more, every box consists conceptually of a nested collection of rectangular subareas, as shown in Figure 3.17. Specifically, there is an innermost rectangle known as the *content area* that encloses the actual content of the element (line boxes or boxes for other elements, or both). *Padding* separates the content area from the box's *border*. There is then a *margin* surrounding the border. The content and margin edges are not displayed in a browser, but are drawn in Figure 3.17 for definitional purposes. Note the similarity between the CSS box model and the concept of a cell in HTML tables. However, as we will see, style properties in CSS provide finer-grained control over boxes than HTML provides for table cells.

Some other terminology related to the box model will also be helpful. The content and margin edges of an element's box are sometimes referred to as the *inner* and *outer* edges of the box, respectively. Also, as shown in Figure 3.18, the outer (margin) edges of a

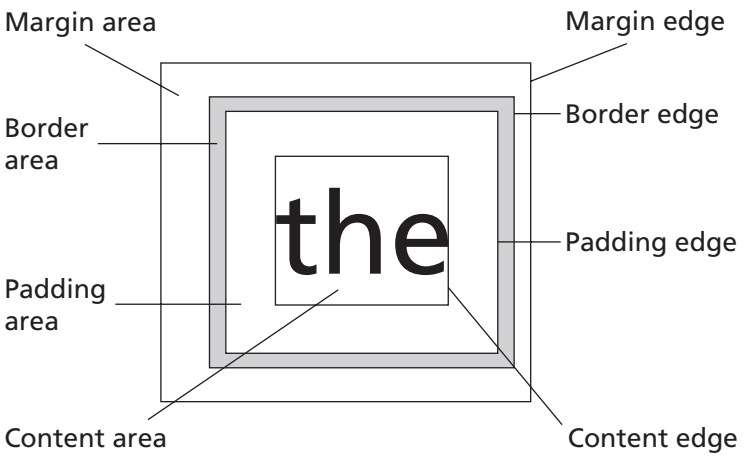

FIGURE 3.17 Definition of areas and edges in the CSS box model.

box define the *box width* and *box height*, while the inner (content) edges define the *content width* and *content height* of the box.

Figure 3.18 also gives the CSS property names corresponding to the 12 distances between adjacent edges in the box model. Notice that the border properties have the suffix `-width`. This suffix is used to distinguish border properties related to distances from other border properties that affect the color and style of borders (and have the suffixes `-color` and `-style`, respectively). Note that the same suffix is used for both horizontal and vertical distances, which can be confusing, since in the rest of the box model "width" normally refers to a horizontal distance.

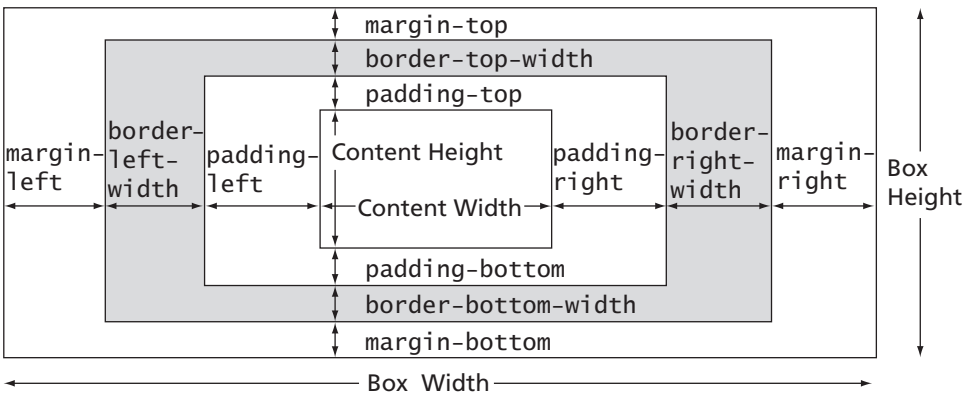

FIGURE 3.18 Definition of various lengths in the CSS box model.

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      SpanBoxStyle.html
    </title>
    <link rel="stylesheet" type="text/css" href="span-box-style.css" />
  </head>
  <body>
    <p>
      The <span>first span</span> and <span>second span</span>.
    </p>
  </body>
</html>
```

FIGURE 3.19 HTML document demonstrating basic box model style properties.

As a simple example of what can be done with what we have already learned (and a few other things that we will learn shortly about border-style property values), consider the following style sheet:

```
/* span-box-style.css */
/* solid is a border style (as opposed to dashed, say). */
span { margin-left: 1cm;
       border-left-width: 10px;
       border-left-color: silver;
       border-left-style: solid;
       padding-left: 0.5cm;
       border-right-width: 5px;
       border-right-color: silver;
       border-right-style: solid }
```

and assume that this style sheet is contained in a file named `span-box-style.css`, as indicated by the comment. Then the HTML document shown in Figure 3.19 will be rendered by a CSS2-compliant browser as illustrated in Figure 3.20. Note that for `span` elements, any margin, border, or padding distance that is not specified by an author or user style sheet is given the value 0.

FIGURE 3.20 Rendering of document demonstrating basic style properties.

3.7.2 Box Model Shorthand Properties

CSS2 defines a number of shorthand properties related to the box model. For example, the declaration

```
padding: 30px;
```

is shorthand for four declarations:

```
padding-top: 30px;
padding-right: 30px;
padding-bottom: 30px;
padding-left: 30px;
```

Table 3.9 lists a number of such shorthand properties as well as the properties already covered and gives for each property its allowable values. None of the properties in this table is inherited.

The `auto` value that can be used when setting margin widths has a meaning that depends on its context, so we will defer discussing it to the appropriate later sections. I'll try to answer other questions you may have about Table 3.9 here.

First, notice that five of the properties in Table 3.9 (`padding`, `border-width`, `border-color`, `border-style`, and `margin`) take from one to four space-separated values. Each of these properties is a shorthand for specifying values for the four

TABLE 3.9 Basic CSS Style Properties Associated with the Box Model

Property	Values
`padding-{top,right,bottom,left}`	CSS length (Section 3.6.2).
`padding`	One to four length values (see text).
`border-{top,right,bottom,left}-width`	`thin`, `medium` (initial value), `thick`, or a length.
`border-width`	One to four `border-*-width` values.
`border-{top,right,bottom,left}-color`	Color value. Initial value is value of element's `color` property.
`border-color`	`transparent` or one to four `border-*-color` values.
`border-{top,right,bottom,left}-style`	`none` (initial value), `hidden`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`.
`border-style`	One to four `border-*-style` values.
`border-{top,right,bottom,left}`	One to three values (in any order) for `border-*-width`, `border-*-color`, and `border-*-style`. Initial values are used for any unspecified values.
`border`	One to three values; equivalent to specifying given values for each of `border-top`, `border-right`, `border-bottom`, and `border-left`.
`margin-{top,right,bottom,left}`	`auto` (see text) or length.
`margin`	One to four `margin-*` values.

TABLE 3.10 Meaning of Values for Certain Shorthand Properties that Take One to Four Values

Number of Values	Meaning
One	Assign this value to all four associated properties (`top`, `right`, `bottom`, and `left`).
Two	Assign first value to associated `top` and `bottom` properties, second value to associated `right` and `left` properties.
Three	Assign first value to associated `top` property, second value to `right` and `left`, and third value to `bottom`.
Four	Assign first value to associated `top` property, second to `right`, third to `bottom`, and fourth to `left`.

associated properties that include `top`, `right`, `bottom`, or `left` in their names. For example, `border-style` is a shorthand for specifying values for `border-top-style`, `border-right-style`, `border-bottom-style`, and `border-left-style`. Table 3.10 shows the meaning of the values for these properties. We have just seen an example of such a shorthand declaration, when a single padding declaration was equivalent to four declarations. As another example, the style declaration

```
margin: 15px 45px 30px
```

is equivalent to

```
margin-top: 15px
margin-right: 45px
margin-left: 45px
margin-bottom: 30px
```

You may also have a question about the border styles listed in Table 3.9. There is no precise definition for most of these border styles, so their visual appearance may vary somewhat when displayed in different browsers. For example, Figure 3.21 shows two paragraphs `p1` and `p2` displayed in two different browsers using the following style sheet:

```
/* border-styles.css */
#p1 {
        background-color: yellow;
        border: 6px maroon;
        border-style: solid dashed dotted double
}

#p2 {
        border: 16px teal;
        border-style: groove ridge inset outset
}
```

Obviously, you may want to experiment with these border-style values in browsers you are targeting before using these values. Also note that both of the border style values

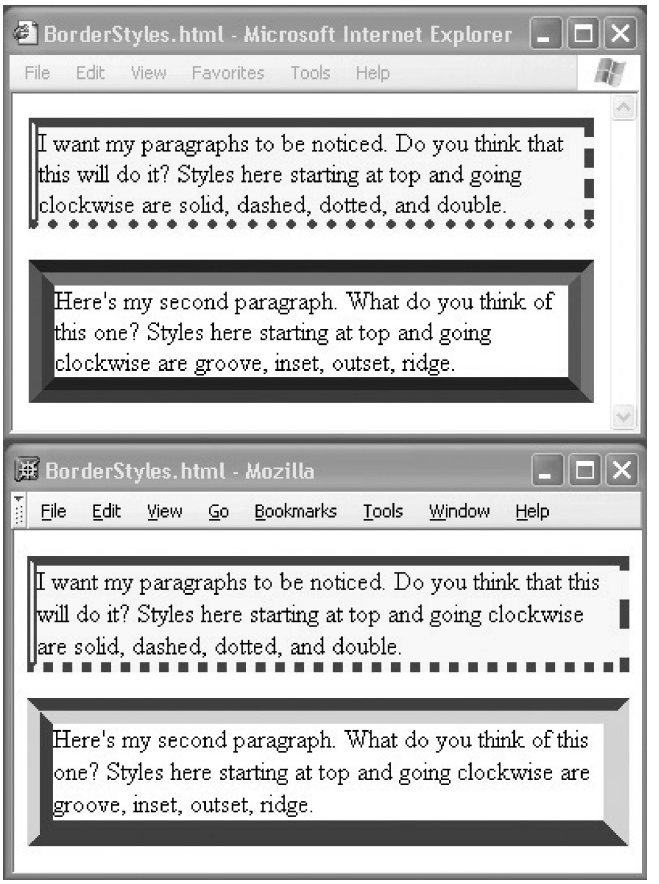

FIGURE 3.21 Illustration of some border styles in different browsers.

`hidden` and `none` effectively eliminate a border from a box element, but `hidden` behaves slightly different within HTML table elements in certain circumstances (refer to the CSS2 specification [W3C-CSS-2.0] for details).

Finally, you probably have noticed that shorthand properties make it possible to declare multiple values for a single property within a single declaration block. For example, the value of `border-top-style` can be specified by a direct declaration of this property as well as by declarations for the `border-top` and `border` shorthand properties. If multiple declarations within a single declaration block apply to a property, the last declaration takes precedence over any earlier declarations. So, for example, in the declaration block

```
{ border: 15px solid;
  border-left: 30px inset red;
  color: blue }
```

the border on the top, right, and bottom will be 15-px-wide solid blue, while the left border will be a 30-px-wide red inset style. This is because the first declaration sets all four borders to 15 px wide and solid, with the border color set to its initial value, which for border colors is the value specified for the element's color property (blue in this case). The second declaration effectively overrides these values for the `border-left` property.

3.7.3 Background Colors and Images

The `background-color` property specifies the color underlying the content, padding, and border areas of an element's box. The background color in the border area will normally be covered by the border itself, but will be visible if the border color is specified as `transparent` or partly visible if the border style is `dashed`, `dotted`, or `double` (see Fig. 3.21). Notice that the margin area is not affected by the background color. The margin area is always transparent, which allows the background color of the parent element to be seen in the margin area. Strictly speaking, the `background-color` property's value is not inherited; however, the initial value of `background-color` is `transparent`, and the background color of an element will be visible through transparent areas of child elements. In other words, for CSS box model purposes, we should think of the browser as rendering parent elements first and then rendering the nontransparent portions of the child elements over top of the parents.

A related property that is used in many Web pages is `background-image`. The acceptable values for this property are `none`, the initial value, or a URL specified using the same `url()` functional notation used with the `@import` style rule. By default, the image found at the specified URL will be *tiled* over the padding and content areas of the element to which this property is applied (such as the `body` element of an HTML document). *Tiling* simply means that if an image is too small to cover the element, either from left to right or from top to bottom or both, then the image is repeated as needed.

Like `background-color`, `background-image` is not inherited. Conceptually, the element to which the background image will be applied is first drawn, including its background color if any. Then the background image is drawn over top of the element, with the element showing through any transparent areas of the image. Finally, any child elements are drawn over top of the background image. The positioning of a background image and whether it is tiled or not can be specified using various CSS properties; see the CSS2 specification [W3C-CSS-2.0] for complete details.

This concludes our discussion of the basic CSS box model. We next turn to considering how this model relates to some specific HTML elements.

3.8 Normal Flow Box Layout

In a browser's standard rendering model (which is called *normal flow processing*), every HTML element rendered by the browser has a corresponding rectangular box that contains the rendered content of the element. The primary question we address in this section is where these boxes will be placed within the client area of the browser.

3.8.1 Basic Box Layout

First, recall that the `html` element is the root element of an HTML document. The browser generates a box corresponding to this element, which is called the *initial containing block*. The CSS2 recommendation does not specify the lengths (margin, padding, etc.) or dimensions (width and height) of this box, but instead leaves it to each browser to choose values for these parameters. In both IE6 and Mozilla 1.4, by default the border, margin, and padding lengths are all zero, so the inner (content) and outer edges of the box coincide. As for dimensions of the initial containing block box, if the browser's horizontal and vertical scrollbars are not active, then the box coincides with the browser's client area, and therefore has the same dimensions. On the other hand, if either scrollbar is active, then the the outer edges of the initial containing block are located at the edges of the underlying area over which the browser can be scrolled. Conceptually, it is as if the document is drawn on an imaginary *canvas*. The browser client area acts as a *viewport* through which all or part of the canvas is viewed. The initial containing block's height is the total height of this canvas, or the height of the browser's client area if that is greater than the canvas height. The width of the initial containing block is defined similarly. Figure 3.22 illustrates the relationship

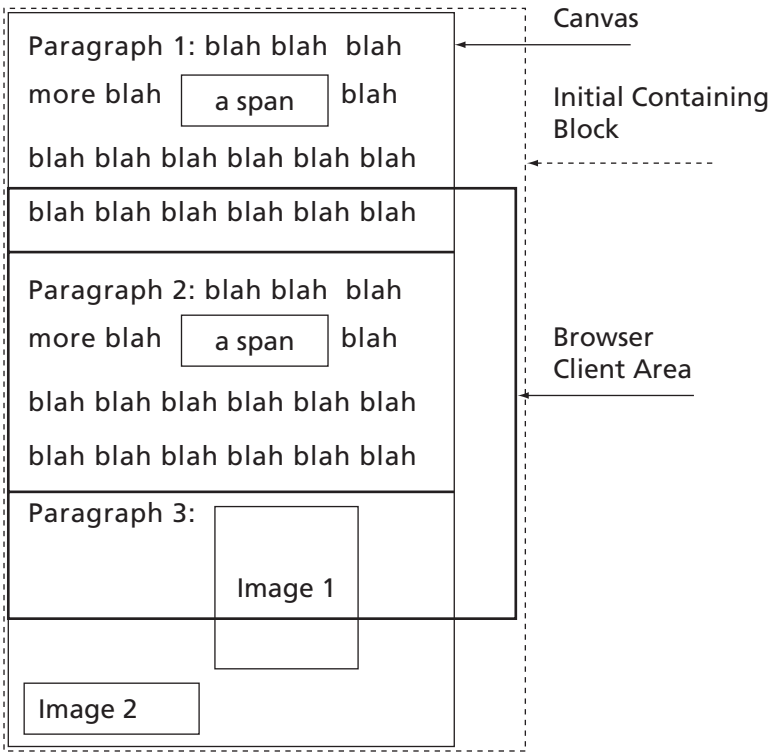

FIGURE 3.22 Initial containing block box when canvas is taller than client area but client area is wider than canvas.

between the canvas, client area, and initial containing block box when the canvas is taller than the client area but not as wide. Note that if the browser window is resized, the initial containing block's box will be resized automatically as needed.

All other CSS boxes within the client area are laid out (either directly or indirectly) relative to the initial containing block box. For an HTML document, the first such box to be added to the client area is the one corresponding to the `body` element. Because the `body` element is contained within the `html` element, the box corresponding to the `body` element is placed within the initial containing block box (which corresponds to the `html` element). This is the default behavior for all boxes: if one HTML element is part of the content of a second HTML element, then the box corresponding to the first element will be contained within the content area of the box for the second element. This default behavior is known as *normal flow* processing of boxes. Thus, if normal flow processing is used for an entire HTML document, all of the boxes corresponding to elements within the `body` element of the document will be contained within the box generated for the `body`, which in turn will be contained within the initial containing block box. In essence, in normal flow processing, the block corresponding to the `body` element is the canvas on which boxes for all other elements will be drawn.

By default, the `body` box will be placed so that its left, top, and right outer edges coincide with the left, top, and right inner (content) edges of the initial containing block. If the width of the browser window is changed, then the width of the `body` box may change as well, since the width of the initial containing block can change automatically when the browser width changes. The height of the `body` box, on the other hand, is determined by its content. You might think of the box as starting with the height of its content area set to 0. Then, as the browser generates boxes corresponding to elements contained within the `body`, it increases this height so that it is just sufficient to contain all the generated boxes. The height when this process is done determines the final height of the content area of the `body` element's box (the overall height of the box also depends on the values of style properties such as `margin-top`).

Similar rules apply to the default placement of boxes within the `body` box. That is, the first child element's box will be placed so that its left, top, and right outer edges coincide with the corresponding content edges of the `body` box. The height of this box will then be determined by generating boxes for all of the elements contained within the first element and laying these boxes out within this first child box (by recursively applying the layout rules being described). The second child element's box will be placed so that its top outer edge coincides with the bottom outer edge of the first child box (this isn't quite correct; see Section 3.8.3 for more details). The left and right edges of this second child box will also coincide with the left and right content edges of the `body`. The second child is then filled with all of its descendants' boxes. This process continues with the remaining children of the `body`.

Figure 3.23 is an HTML document that illustrates the layout concepts discussed thus far (I have used an embedded style sheet in this document and several others in this chapter for ease of reading, but in practice I would probably have used an external style sheet). Figure 3.24 shows how Mozilla 1.4 renders this document (the IE6 rendering is similar, although my copy of IE6 incorrectly draws the initial containing block's border so that it always coincides with the client area, regardless of how the browser window is sized).

```
 <!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      BlockBoxes.html
    </title>
    <style type="text/css">
      html, body { border:solid red thin }
      html { border-width:thick }
      body { padding:15px }
      div { margin:0px; padding:15px; border:solid black 2px }
      .shade { background-color:aqua }
      .topMargin { margin-top:10px }
    </style>
  </head>
  <body>
    <div id="d1">
      <div id="d2">
        <div id="d3" class="shade"></div>
      </div>
      <div id="d4" class="shade topMargin"></div>
    </div>
  </body>
</html>
```

FIGURE 3.23 HTML document containing nested `div` elements.

Figure 3.24 shows the borders of a number of boxes. The outermost border (thick red border at the edges of the browser's client area) is for the initial containing block box generated by the `html` element. The thin red border immediately inside the `html` element's box belongs to the `body` element's box. You'll notice that the `body` border does not touch the `html` border. This is because the Mozilla 1.4 user agent style sheet specifies a nonzero margin value (apparently about 8 px in Mozilla 1.4 and 10 px in IE6) for the `body` box, and the embedded author style sheet does not override this value. Inside the `body` block box there is a box with a medium-width black border generated by the `div` with ID `d1`. Inside this box are two child boxes, one for each of the `div` children of `d1` (`d2` and `d4`). Finally, the first of these child elements (`d2`) itself has a child `div` with id `d3`, which generates its own box. The boxes for the `div` elements `d3` and `d4`, which have no content, are given a background color in Figure 3.24.

3.8.2 The `display` Property

The layout rules described so far only apply to HTML elements that CSS recognizes as *block elements*. These are elements for which the CSS `display` property has the value `block`. Of the elements covered in Chapter 2, standard user agent style sheets will define the following HTML elements as block elements: `body`, `dd`, `div`, `dl`, `dt`, `fieldset`, `form`, `frame`, `frameset`, `hr`, `html`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `ol`, `p`, `pre`, and `ul`. You may recall from the last chapter that

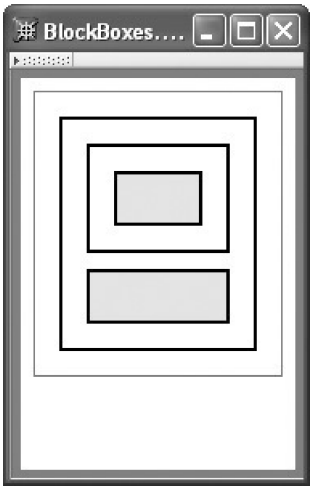

FIGURE 3.24 Nested boxes.

we informally introduced the concept of block elements as those elements for which the browser essentially generates a new line before and after the element. Now we see that what actually happens is that the browser stacks the boxes for these elements one on top of the next.

CSS defines a number of other possible values for the `display` property. Many of these values are associated with specific HTML elements. For example, there is a `list-item` value that is intended to be used as the `display` value for `li` elements, a `table` value for HTML `table` elements, and a `table-row` value for `tr` elements. In fact, nearly every element associated with the HTML table model has its own value for the `display` property (`td` and `th` share the `table-cell` value). We will not discuss these values further; see [W3C-CSS-2.0] for details.

In addition to these and other somewhat specialized values for `display`, there is another value that is shared by a number of HTML elements: `inline`. Again, recall from the previous chapter that inline HTML elements are those that do not interrupt the flow of a document by starting a new line as block elements do. Examples of inline elements were `span` and `strong`. In a typical browser, all of the HTML elements discussed in the last chapter except the block elements listed at the begining of this subsection, the `li` element, and table-related elements will be treated as having the value `inline`, which is the initial value for the `display` property.

As you might expect, the rules for laying out the boxes for elements with a `display` value of `inline` (which I'll refer to as *inline boxes*) are different from those for laying out boxes for elements with a `display` value of `block` (*block boxes*). In fact, how content is laid out within inline and block boxes also differs. We'll cover some more details concerning block boxes in the next few sections and then look more closely at inline boxes.

Before leaving this section, let me mention that an author style sheet can override the default value of an element's `display` property just as any other default property value

can be overridden. For example, suppose that an HTML document has a large number of consecutive p elements but that for some reason we would like—with a minimal amount of change to the document—to have all of these separate paragraphs in the document displayed as one (long) paragraph. We can accomplish this by adding to the document the style rule

```
p { display:inline }
```

Obviously, this style rule significantly changes the expected semantics of the p element, so a rule such as this should be used with some caution.

3.8.3 Margin Collapse

Earlier I said that, roughly speaking, consecutive block boxes are positioned one on top of the next. I'll now explain why this isn't exactly the case.

When two consecutive block boxes are rendered (the first on top of the second), a special rule called *margin collapse* is used to determine the vertical separation between the boxes. As the name implies, two margins—the bottom margin of the first (upper) box and the top margin of the second (lower) box—are collapsed into a single margin.

Specifically, let m_1 represent the value of margin-bottom for the top box, and let m_2 represent the value of margin-top for the lower box. Without margin collapse, the distance between the borders of these boxes would be $m_1 + m_2$. With border collapse, the distance will instead be $\max(m_1, m_2)$ (see Fig. 3.25).

3.8.4 Block Box Width and Height

Each block element has a width property (not inherited) that defines the width of the content area of the element's block box. The initial value of this property is auto, which produces the width-defining behavior described earlier: the box will automatically be stretched horizontally so that its left and right outer edges align with the left and right content edges of its parent box. As an example, if the browser window shown in Figure 3.24 is widened, the block boxes displayed in the content area will also become wider (Fig. 3.26).

More precisely, if the value of width is auto, and if a value other than auto is specified for both margin-left and margin-right (the initial value for these properties is 0), then for display purposes width will be given the value

```
width = parent's displayed content width -
        (margin-left + border-left-width + padding-left +
         padding-right + border-right-width + margin-right)
```

This value is not in any way associated with the width property itself, as a specified or computed value is. Instead, it is used by the browser strictly for display purposes. Such a value is sometimes referred to as a property's *used value*.

In addition to auto, a length value can be specified as the value of the width property of a block element. The length value can use any of the units described in Section 3.6.2. Furthermore, the specified length value can be a *percentage*, which is a number (integer or decimal) immediately followed by a percent sign (%). In the case of the width property,

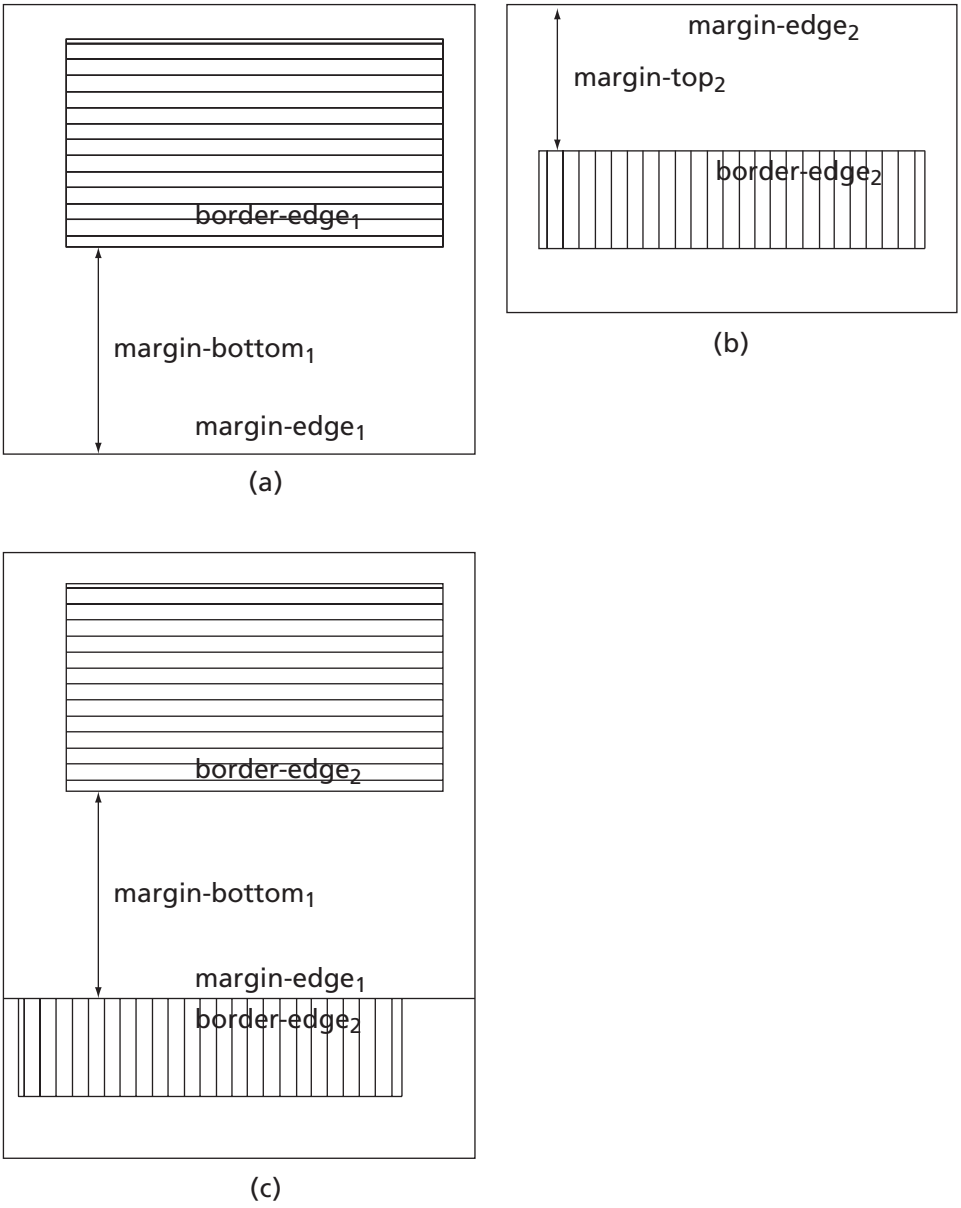

FIGURE 3.25 (a) A block box (only margin and border edges are shown). (b) A second block box with `margin-top` smaller than `margin-bottom` of first box. (c) First and then second boxes rendered, illustrating margin collapse.

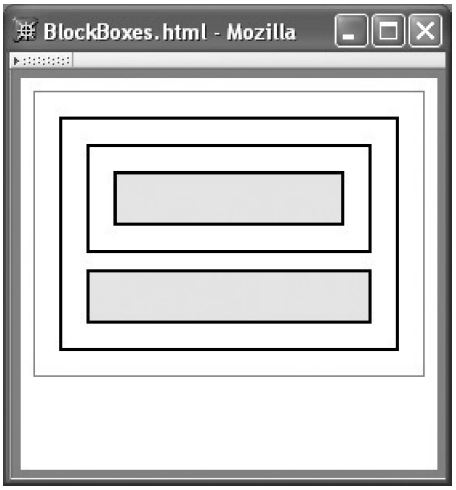

FIGURE 3.26 `BlockBoxes.html` displayed in a wider window.

this represents a percentage of the width of the parent element's content area, or more precisely, a percentage of the used value associated with the content width. For example, the declaration

```
width:50%
```

says that the width of the content area of a box should be half the (used) width of the content area of its parent box. Percentages can also be used with many other CSS properties that take a length value, although the length to which the percentage is applied varies from property to property. See the CSS2 recommendation [W3C-CSS-2.0] for details regarding properties not explicitly mentioned in this chapter.

You might expect (I did initially) that the percentage `width` declaration would cause an associated element to be centered within the parent box. However, this is not the case by default. Instead, the element will appear left-justified within its parent box. In essence, when only the `width` is specified for an element, the browser computes a used value for the `margin-right` property of the element's box so that the overall width of the box (sum of the element width plus left and right margins, borders, and paddings) is equal to the width of its parent's content area. The `margin-left`, however, is unchanged. To center an element, in addition to specifying a value for the element's `width` property, the value `auto` should be specified for both the `margin-left` and `margin-right` properties of the element. The browser will then use a single value for both margins, with the used value being computed so that the borders (but not necessarily the content) of the box will be centered within the content area of the parent box.

So, for example, assume that we create an HTML document `BlockBoxesWidth.html` from the earlier `BlockBoxes.html` example (Fig. 3.23) by adding the following two rules to the embedded style sheet:

```
#d3 { width:50% }
#d4 { width:50%; margin-left:auto; margin-right:auto }
```

Note that these rules will override the declarations having element selector `div`, due to the higher specificity of ID selectors. The new `BlockBoxesWidth.html` document will be rendered as shown in Figure 3.27. Notice that although both of the shaded boxes have specified widths of `50%`, their actual widths are different because the percentage is applied to parent boxes that have different content widths. Also, keep in mind that the value of the `width` property defines the width of the content area of a box. Thus, the shaded boxes are both wider than half the width of their parents' content areas, because each box includes a total of 30 px of horizontal padding (15 px for each side) in addition to the content area.

In general, the value `auto` can be specified for any combination of `width`, `margin-left`, and `margin-right`. For example, if for a given box `margin-left` is `auto`, `width` is a specified length, and `margin-right` is 0, then the box will be right-justified within its containing block. See the CSS2 recommendation [W3C-CSS-2.0] for details on how CSS interprets other possible combinations for values of these three properties.

Block boxes also have a `height` property (not inherited) with an initial value of `auto`. As with the `width` property, the default block box height calculation described earlier can be overridden by specifying a value (length with units or percentage) for the block element's `height` property. If a percentage is specified, it is interpreted as a percentage of the value (if any) specified for the parent block's `height` property. If no value was specified for the parent's `height`, then the percentage specification is essentially ignored and treated as a specification of `auto`.

We're now ready to consider how inline boxes are rendered within a block box.

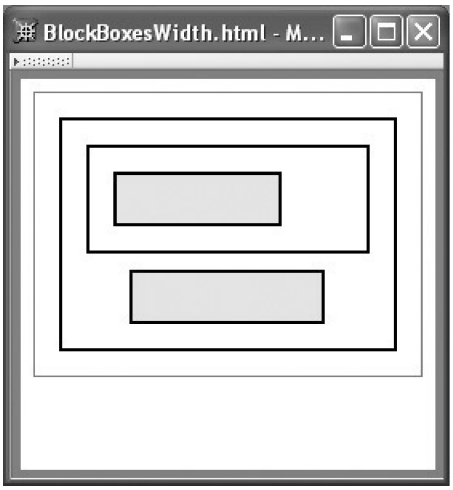

FIGURE 3.27 Rendering when widths of shaded boxes are specified as percentages. Lower box is centered because left and right margins are `auto`.

3.8.5 Simple Inline Boxes

Until now, we have thought of block boxes as either containing text (or more precisely, a stack of line boxes containing character cells) or containing other block boxes. But a block element can also contain inline elements, such as `span` and `strong`, and the browser will generate inline boxes corresponding to these elements. These inline boxes will be added to line boxes within the containing block box, much like text characters. In fact, we have already seen an example of this in Figure 3.20. In this section, we will look more closely at how browsers lay out *simple inline boxes*, that is, boxes for inline elements that contain only text or that are of type `img`. We'll briefly consider more complex inline elements in the next section.

We will first consider simple inline boxes consisting of text, generated by elements such as

```
<span style="font-size:36pt">BIG</span>
```

The height of the content area of such a box will be determined exactly as the height of a line box is determined: the height of a character cell (in the inline element's font) will be used unless the inline element's `line-height` property has a value other than `normal`, in which case this value will determine the content area height. Character glyphs are then added to the content area of the inline box as if they were being added to a line box, with half-leading added if needed to center the character cells vertically. Note that this process defines a baseline for the inline box: it is at the baseline height (as defined by the inline element's font) above the bottom edge of the content area of the inline box. We therefore now have a box that has a well-defined height (the height of the content area), width (the overall width of the box, including left and right padding, border, and margin lengths), and baseline height. These are essentially the same characteristics that a character cell has, so the browser can add this inline box to a line box as if it were a character cell, vertically aligning the baseline of the inline box with the baseline of the line box. If the inline box is too long to fit within the current line box, it may be broken on word boundaries into a sequence of shorter inline boxes that will each be added to a separate line box. If the top or bottom of an inline box extends beyond the corresponding edge of the line box, the line box height will automatically be expanded as needed to contain the inline box. If the line box height is extended upward, then the line box will be moved down within the containing block box by the same amount so that the line boxes within the block box will still effectively be stacked one on top of the other (Fig. 3.28).

You probably noticed that there is an asymmetry in how the height and width of the "character" representation of an inline box are determined. Specifically, the height of this "character" is determined by the content height of the inline box, but the width is determined by the overall box width. To illustrate, suppose we change the `d3` element of the document of Figure 3.23 as follows:

```
<div id="d3" class="shade">
  Here are
  <span style="border:dotted silver 10px">some</span>
  lines of text.
</div>
```

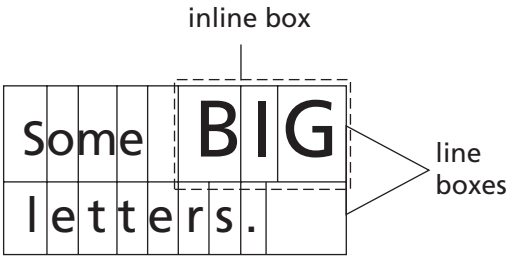

FIGURE 3.28 Two line boxes, the top box containing an inline box with a larger font size than the text elsewhere in the line boxes. Notice that the baselines are aligned in the top line box and that the line boxes stack despite having different heights.

Then the document will be rendered as in Figure 3.29. Notice that the word "some" is moved to the right to make room for the border, but the line height is unchanged, and in fact the border overlaps somewhat the text in the second line box.

The other type of simple inline element is an `img`. An `img` element is similarly treated, for rendering purposes, as a character to be added to a line box. However, the height and width of the "character" are the values specified for the `height` and `width` properties of the element (or, if these properties are not specified, the values of the `height` and `width` attributes, or, if these values are also not provided, then values contained within the image file itself). The baseline height of an image is always considered to be 0. Therefore, the bottom of the image will coincide with the baseline of the line box. As with inline boxes, if the top of an `img` box extends past the top of the line box, then the height of the line box

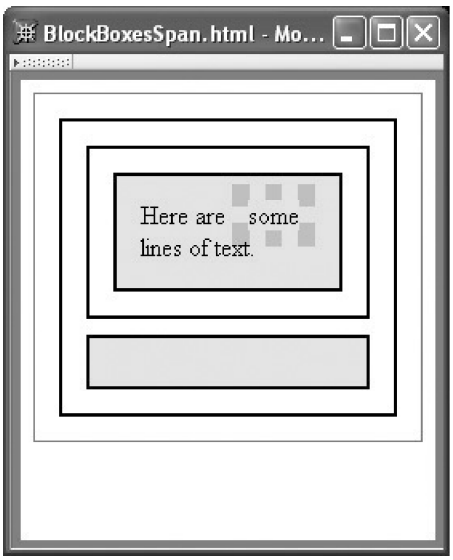

FIGURE 3.29 A span element with a border is added to the text.

will be increased to fit. Unlike other inline elements, the border, margin, and padding of an `img` element are considered part of the height of the image for purposes of determining the height of a line box containing the image.

The default vertical placement of an inline box within a line box can be overridden by specifying a value for the `vertical-align` property (not inherited) of the element generating the inline box. The initial value of `vertical-align` is `baseline`, which produces the default behavior described. Some other possible values are `text-bottom`, which aligns the bottom of the inline box with where the bottom of any character cell written into the line box would be located; `text-top`, which is similar except it aligns the top of the inline box with the top of the location for character cells; and `middle`, which aligns the vertical midpoint of the inline box with the *character* middle of the line box, a location that is one-half the ex height above the baseline of the line box. The CSS2 recommendation specifies several additional keyword values for `vertical-align`, but my copy of IE6 does not seem to handle them properly, so I will not cover them here.

In addition to these keywords, the value specified for `vertical-align` can be a length or a percentage (of the value of the `height` of an `img` element or the `line-height` of any other inline element). For both percentage and length specifications, a positive value indicates that the inline box should be raised by the specified distance relative to the default baseline position, and a negative value indicates that it should be lowered.

3.8.6 Nested Inline Boxes

How are text and boxes laid out within an inline element? We will consider the standard case for HTML, in which an inline element contains only text and other inline elements (see the CSS specification [W3C-CSS-2.0] for details on how a block box is handled within an inline box).

Actually, the layout of inline boxes and text within an inline box is essentially identical to the layout of inline boxes and text within a line box. In particular, the content area of the containing inline box is treated as a line box that initially has a height and baseline location defined by the font and `line-height` properties of the corresponding inline element. Characters and boxes are then added to this content area just as they would be to a line box, including having the vertical alignment of boxes determined by the `vertical-align` property. One difference is that the height of the content area is not adjusted to contain inline boxes whose top or bottom edges extend beyond the respective edges of the content area. However, when these boxes are eventually transferred from the content area to a line box, that line box's height will be adjusted. For example, changing the `d3` element of the document of Figure 3.23 to

```
<div id="d3" class="shade">
  Here are some lines of text.
  This
  <strong style="line-height:3">bold</strong>
  word has a line-height of 3.
</div>
```

FIGURE 3.30 Effect of setting `line-height` on an inline element (the word "bold").

causes the browser to increase the height of the line box containing the word "bold," as shown in Figure 3.30. However, the heights of other line boxes are not affected.

3.9 Beyond the Normal Flow

What we have described so far is the default way in which a browser will format an HTML document. In this section, we'll learn that there are several CSS alternatives to the normal flow processing we have seen so far that can be used to control the position of boxes within a browser window. Three alternatives to normal flow positioning supported by both Mozilla 1.4 and IE6 are: relative positioning, in which the position is altered relative to its normal flow position; float positioning, in which an inline element moves to one side or the other of its line box; and absolute positioning, in which the element is removed entirely from the normal flow and placed elsewhere in the browser window.

We'll look at each of these three positioning schemes in detail in Sections 3.9.2–3.9.4. First, though, we'll briefly learn about the CSS properties used to indicate whether or not a box should use an alternative positioning scheme.

3.9.1 Properties for Positioning

The type of positioning for an element is defined by specifying two style properties. The `position` property takes the value `static` (the initial value) to indicate normal flow, `relative` and `absolute` to represent the respective flow positionings, or `fixed`, which is a special type of absolute positioning discussed in the exercises. The `float` property can be set for elements with either `static` or `relative` specified for `position`. Possible values for

FIGURE 3.31 HTML document using relative positioning to nudge text to the left.

`float` are `none` (the initial value), `left`, or `right`. The latter two values indicate that the element's box should move to the far left or far right side of the current line box, respectively. Neither `position` nor `float` is an inherited property.

Any element with a `position` value other than `static` is said to be *positioned*. If the `position` value of a positioned element is `absolute` or `fixed`, then it is said to be *absolutely positioned*; otherwise it is *relatively positioned*. Four (noninherited) properties apply specifically to positioned elements: `left`, `right`, `top`, and `bottom`. Each of these properties takes a value that is either a length, a percentage, or the keyword `auto` (the initial value). The meaning of these properties is explained for each positioning scheme in the following Sections 3.9.2–3.9.4.

3.9.2 Relative Positioning

Relative positioning is useful when you want to nudge a box a bit from the position where the browser would normally render it, and you want to do so without disturbing the placement of any other boxes. That is, all other boxes are positioned as if the nudged box had never moved.

For example, suppose that you were asked to produce the rendered HTML document shown in Figure 3.31. Notice that the first letter of each of the words "Red," "Yellow," and "Green" has a background that is partly shaded and partly not. This is not an effect that we would expect to produce in the normal flow processing model. But with relative positioning, it's easy: we use a style rule

```
.right { position:relative; right:0.25em }
```

and wrap each word to be moved in a `span` that specifies `right` for the value of its `class` attribute. As a side benefit, we get a little more separation between each word and the shaded box to its right than we would have had in normal flow processing, since the locations of these boxes is not affected by the relative shifting of the words.

Notice that for relatively positioned boxes, a positive value for the `right` property moves the box to the left by the specified amount. You can think of this as adding space to the right margin of the box. Recall that the initial value of `left` is `auto`; in this example, the corresponding computed value for the `left` property will be −0.25 em. Alternatively, if the style rule had been

```
.right { position:relative; left:-0.25em }
```

then the browser would have displayed the same rendering and `left` and `right` would have had the same computed values as they did with the original style rule. If, for some reason, both `left` and `right` have specified values other than `auto`, then the value of `left` will be used for the positioning, and the computed value of `right` will be set to the negative of the value of `left` (assuming `direction` is `ltr`). Similar rules apply to `top` and `bottom`, with `top` "winning" if both properties have non-`auto` values.

3.9.3 Float Positioning

Float positioning is often used when embedding images within a paragraph. For example, recall that the HTML markup

```
<p>
  <img
      src="http://www.w3.org/Icons/valid-xhtml10"
      alt="Valid XHTML 1.0!" height="31" width="88"
      style="float:right" />
  See
  <a href="http://www.w3.org/TR/html4/index/elements.html">the
    W3C HTML 4.01 Element Index</a>
  for a complete list of elements.
</p>
```

is part of the document displayed in Figure 2.13. The `float:right` declaration causes the image to be treated specially in several ways. First, the image is not added to a line box. Instead, the widths of one or more line boxes are shortened in order to make room for the image along the right content edge of the box containing the line boxes and image (the block box generated by the `p` element, in this case). The first shortened line box is the one that would have held the image if it had not been floated. Subsequent line boxes may also be shortened if necessary to make room for the image. Line boxes below the floated box extend to the full width of the containing block, producing a visual effect of text wrapping around the floated block (Fig. 3.32).

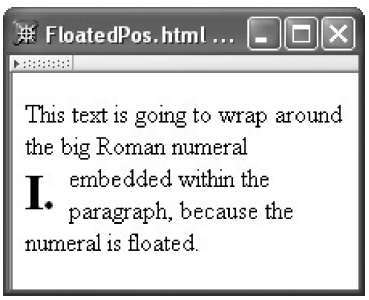

FIGURE 3.32 Wrapping of text around a floated box.

The markup used to generate this figure includes the following:

```
<style type="text/css">
  .bigNum { float:left; font-size:xx-large; font-weight:bold }
</style>
...
<p>
  This text is going to wrap
  around the
  <span class="bigNum">I. </span>
  big Roman numeral
  embedded within the paragraph, because the numeral is floated.
</p>
```

Notice that, unlike a relatively positioned box, the words "the" and "big" are not separated by the width of the floated `span` that separates these words in the source document. In other words, portions of this document that are part of the normal flow are formatted as if the floated element were not present at all (except for its effect on the width of line boxes). We say that float boxes are *removed from the normal flow* to indicate that making them float has an impact on how normal flow elements are rendered.

One small detail about floated boxes is that a floated inline box becomes a block box for display purposes; that is, an inline box's `display` property will have a computed value of `block` if the box is floated. This means, for example, that values can be specified for the `height` and `width` of a floated inline element.

For more details on float positioning, such as what happens when multiple floated boxes touch one another or when floated inline boxes extend below their containing block, see the CSS2 specification [W3C-CSS-2.0].

3.9.4 Absolute Positioning

Absolute positioning offers total control over the placement of boxes on the canvas. This power should be used with care: while you can create interesting visual effects this way, any information conveyed by these effects will generally not be available to users who are accessing the document in other ways (text-based browsing, speech synthesis, etc.). That said, there are certain times when it is useful to be able to place a box exactly where you want it.

For example, suppose that you would like to be able to easily add marginal notes to the left of paragraphs in an HTML document, as shown in Figure 3.33. Specifically, you'd like to be able to embed each note within the paragraph to which it applies, as in

```
<p>
  This second paragraph has a note.
  <span class="marginNote">This note is pretty long, so
  it could cause trouble...</span>
</p>
```

Then you would like the browser to automatically place the note next to the paragraph, starting vertically at the top of the paragraph.

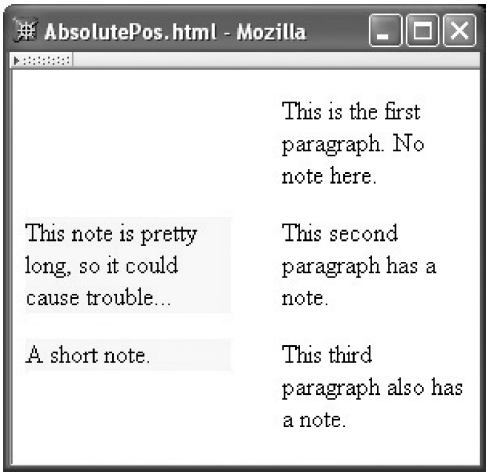

FIGURE 3.33 Absolute positioning used to create marginal notes.

This can be done easily using absolute positioning. When a box is absolutely positioned, as indicated by specifying `absolute` for the `position` property, the `left`, `top`, `right`, and `bottom` properties can be used to place the box relative to a containing block. The *containing block* for purposes of absolute positioning is defined as follows. First, we locate the nearest positioned ancestor of the absolutely positioned element (recall that a positioned element has `position` value other than `static`). If this ancestor is a block element (which we will assume; see the CSS2 recommendation for other possibilities), then the containing block is formed by the padding edge of the element's box, *not* the content edge as you might expect (the next example will show why this is a good choice of edge). If there is no positioned ancestor, then the initial containing block is used as the containing block.

Similar to relative positioning, specifying a value such as `10em` for the `left` property of an absolutely positioned box tells the browser to position the left outer edge of that box 10 ems to the right of the left (padding) edge of the containing block. Positive values for the other three positioning properties have similar effects, while negative values for these properties have the opposite effects (e.g., a negative value of `left` moves the box to the left rather than to the right). Like floats, if the box of an inline element is positioned absolutely, the box becomes a block box, and therefore can have its width set explicitly.

In our marginal note application, we would like each note to be positioned starting vertically at the top of the paragraph containing the note and horizontally to the left of the paragraph. This means that we want the paragraphs containing notes to be positioned, so that they can act as containing blocks for absolutely positioned elements. Also, we want to leave room next to paragraphs for the notes. So we will use the style rule

```
p { position:relative; margin-left:10em }
```

In relative positioning, if no value is specified for `left` or the other positioning properties, then the element is not moved. So the `position:relative` declaration has no visible effect. Instead, it marks `p` elements as positioned, making them eligible to act as containing blocks for absolutely positioned elements.

The rule for the `marginNote` class is longer, but not particularly difficult to understand. The rule is

```
.marginNote { position:absolute;
              top:0; left:-10em; width:8em;
              background-color:yellow }
```

This says to put any box belonging to the `marginNote` class and that is contained within a `p` element where we have said we would like it placed: beginning to the left of the top line of the paragraph. Notice that, for the given style rules, the outer left edge of a `marginNote` box will coincide with the outer left edge of the `p` element containing the `marginNote` element, regardless of the padding value of the `p` element (and assuming that the `p` element has no border). You can now see an advantage to positioning the box relative to the padding edge of the containing element rather than the content edge: we did not have to add in the padding distance in order to place our note in the margin.

As with float positioning, elements that are absolutely positioned are removed from the normal flow. This can be seen from the fact that there is no additional space between the second and third paragraphs in the figure. In contrast with float boxes, however, the normal flow will not flow around absolute boxes. In fact, absolute boxes will not flow around one another, either. For example, if we widen the browser window so that the second paragraph fits on a single line, the two marginal note boxes collide and the second obscures some of the text of the first (Fig. 3.34). This is another reason to use absolute positioning with care.

3.9.5 Positioning-Related Properties

A few additional CSS properties deserve mention in relation to positioning. The first of these is related to the overlay phenomenon that we have just discussed with absolute positioning.

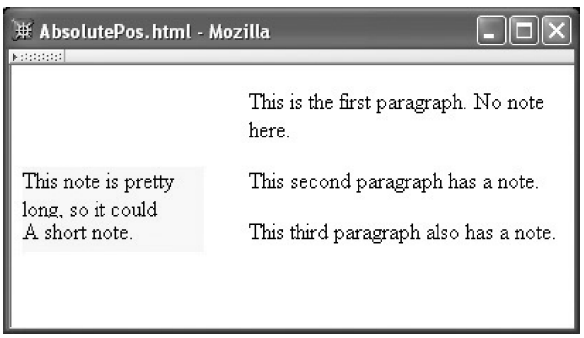

FIGURE 3.34 Absolutely positioned boxes can obscure one another.

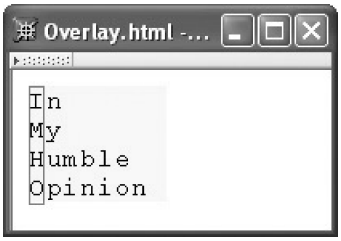

FIGURE 3.35 An overlay of one box on top of another.

There may be times when you want several boxes to overlie one another, at least partially, but you want to be certain that they overlie in a certain order. For example, you may want one box to be drawn first, then a second box to be drawn as an overlay over top of the first box, possibly obscuring some or all of the first box. Figure 3.35 illustrates this: a box containing text is drawn first, and then an empty box (basically just a border) is drawn over top of the text box.

The `z-index` property can be used to define a drawing order for positioned boxes. In its simplest form, the root element of each positioned box is assigned an integer `z-index` value (this assumes that normal flow is followed for all of the descendants of a positioned box). If this is done, then drawing will proceed as follows. First, the box with the most negative `z-index` value (if any) will be drawn. Other boxes with negative `z-index` values will then be drawn on top of this box, proceeding in ascending order, until the box with the negative value closest to 0 has been drawn. At this point, all of the elements that are not positioned are drawn. Finally, all elements with positive `z-index` values are drawn, again in ascending order. The full definition of `z-index`, including how ties are broken between elements with the same `z-index` value (or no value at all) is contained in Section 9.9.1 of the CSS2 specification. But for most purposes, the simple use of `z-index` described should be sufficient to guarantee the drawing order you want.

To produce the effect shown in Figure 3.35, I used the following style rules:

```
#text { position:absolute; top:10px; left:10px;
        font-family:"Courier",monospace; letter-spacing:0.1ex;
        background-color:yellow;
        z-index:1 }
#overlay { position:absolute; top:10px; left:10px;
           width:1.1ex; height:4.5em;
           border:solid red 1px;
           z-index:2 }
```

The first rule is applied to a `div` containing the text, and the second to an empty `div`. The key item to notice is that the `z-index` value of the second `div` is greater than that of the first, so the second `div` is drawn on top of the first.

We discussed the `display` property earlier, but it has a keyword value that we did not cover. Specifying `none` for the value of `display` tells the browser to, for display purposes, treat the element and all of its descendants as if they did not exist. In other words, the

element is effectively removed from the normal flow and is also not displayed elsewhere. This is sometimes used with scripting to allow portions of a document to be easily added to or removed from the browser window.

The final property, `visibility`, is related. If the value of this property is `hidden`, then the element and its children—except those that specify `visible` for this property—will not be rendered. However, much as with relative positioning, the space occupied by the element will remain rendered. In other words, whether an element is visible or not does not affect the rendering of other nondescendant elements. Like `display`, this property is generally used in scripting contexts.

3.10 Some Other Useful Style Properties

While we have covered a significant portion of the CSS2 specification, we have also omitted a number of details and quite a few properties. A few of the remaining items are covered in this section.

3.10.1 Lists

The `list-style-type` property can be used to vary the default styles used for bulleted and numbered list items. In HTML, this property normally only applies to the `li` element type. However, it is inherited, so can be set on a parent `ol` or `ul` element in order to affect the style of all of that element's children. For bulleted lists, the values `disc`, `circle`, and `square` may be specified. For numbered lists, some of the normal values are `decimal` (1, 2, ...), `lower-roman` (i, ii, ...), `upper-roman` (I, II, ...), `lower-alpha` (a, b, ...), and `upper-alpha` (A, B, ...). A value of `none` can also be specified to indicate that no *marker* (leading bullet or number) should be generated for an `li` element.

A related `li` element type property is `list-style-image`, which has an initial value of `none`. If a URI is specified for this property (using the `url("...")` syntax described in Section 3.3), and if an image can be loaded from this URI, then this image will be used in place of the normal marker as specified by `list-style-type`. Once again, this property is inherited and is often set on parent elements rather than directly on `li` elements.

The `list-style-position` property can be used to change the location of the marker relative to the content of an `li`. A browser normally generates two boxes for an `li`: a special box to hold the marker, and a block box to hold the element content. If `list-style-position` has its initial value of `outside`, the marker box is outside the content block box. However, if the value is specified as `inside`, then the box generated for the marker will be the first inline box placed within the content block box. The visual effect in this case will be that the first line of the list item content is indented to make room for the marker.

Finally, the shortcut property `list-style` can be used to specify values for any or all of the mentioned properties, in any order.

3.10.2 Tables

For the most part, the box model properties discussed in this chapter, such as `border-style` and `padding`, can be used with elements related to tables (`table`, `td`, etc.), although their effect on table elements may vary slightly from their effect with other boxes. Furthermore,

the values top, bottom, middle, and baseline may be specified for the vertical-align property of td and th elements. A top value causes the top of the content of the cell to coincide with the top of the row containing the cell, bottom makes the bottom of the content coincide with the row bottom, and middle centers the content within the row. If multiple cells specify baseline, then the baselines of their first lines of text will be aligned with one another. If baseline is specified for a cell containing a single img element, then the bottom of the image is treated as the baseline of the cell for alignment purposes. The baseline cells are displayed as high as possible within the row while keeping the content of all cells within the row.

CSS2 also specifies two different models for how borders should be handled: a *separate* model in which each cell has its own border, and a *collapse* model in which adjacent cell borders may be collapsed into a single border. The user agent style sheets for both Mozilla 1.4 and IE6 apparently specify the separate model as the default. You can override the default by assigning a value of collapse or separate to the border-collapse style property of a table element. Full details of CSS support for tables are well covered in Chapter 17 of the CSS2 specification, which should not be hard to understand if you have mastered the material in this chapter. So I will not cover tables further here.

3.10.3 Cursor Styles

CSS specifies a number of different cursor styles that can be used. The initial value for the cursor property is auto, which allows the browser to choose a cursor style as it deems appropriate. Mozilla 1.4, for example, will display a text cursor when the mouse is over text, a pointing finger when over a hyperlink, an arrow and hourglass when a link is clicked and a new document is loading, and an arrow in most other contexts. Other keywords that can be used to specify a value for the cursor property include default (often an arrow), text (used over text that may be selected), pointer (often used over hyperlinks), and progress (often used when a link is clicked). Some other keywords produce cursors that would normally be seen outside the browser client area, such as move (used to indicate window movement), various resizing arrows (e-resize, ne-resize, sw-resize, and other compass points), wait (program-busy, often an hourglass), and help (often an arrow with a question mark).

Like some other properties, cursor is normally used by scripts running within the browser, a topic covered in Chapter 5.

3.11 Case Study

We'll now create a style sheet suitable for our blogging application and also modify our previous view-blog document, formatting it using style properties rather than tables. For colors, fonts, and to a lesser extent spacing, our style sheet will be similar to the Oldstyle style sheet available as part of W3C's Core Styles project (http://www.w3.org/StyleSheets/Core/), giving us an opportunity to see some real-world styling (and something that looks much better than anything I would have produced). Ultimately, the style sheet and markup changes presented in this section will transform the view-blog page from that shown in Figure 2.31 to that shown in Figure 1.12.

```
/* The W3C Core Styles, Copyright © 1998 W3C (mit, inria, Keio). All
Rights Reserved. W3C liability, trademark, document use and software
licensing rules apply. See
http://www.w3.org/Consortium/Legal/ipr-notice.html

This is a modified version of the Oldstyle style sheet available
at http://www.w3.org/StyleSheets/Core/
A list of modifications made to the original Oldstyle style sheet is
at http://www.mathcs.duq.edu/~jackson/webtech/OldstyleMods.txt */
        /* Elements */
body    {
        background-color: #fffff5;
        font-family:    "Verdana", "Myriad Web", "Syntax", sans-serif;
        font-size-adjust: .58;
        margin:         1.5em 5%;
        }
p       {
        margin-top:     .75em;
        margin-bottom:  .75em;
        }
h1      {
        font-family:    "Georgia", "Minion Web", "Palatino",
                        "Book Antiqua", "Utopia", "Times New Roman",
                        serif;
        font-size-adjust: .4;
        font-size:      2.0em;
        font-weight:    600;
        color:          #C00;
        }
hr      {
        height:         1px;
        background-color: #999;
        border-style:   none;
        }
```

FIGURE 3.36 Style rules for nonanchor elements.

The style sheet rules we create will all be stored in a file named `style.css`. Each of the HTML documents for the application will be modified to include this style sheet file using markup such as

```
<link rel="stylesheet" href="style.css" type="text/css" />
```

in the `head` element of the document. Note that the relative URL used in the `href` attribute assumes that the HTML files and `style.css` file all reside within the same directory.

Figure 3.36 shows the first portion of the `style.css` file. The four rules shown each have a selector string that is a type selector. The first rule states that the background color of the body will be slightly off-white (recall that white is `#ffffff`). The default font family (unless overridden by another element) will be Verdana or, if Verdana is not available to the browser, one of three other font families listed (the final option is the generic sans-serif

```
        /* Hyperlinks */
a       {
        font-weight:    bold;
        text-decoration: none;
        }
a:link  {
        color:          #33F;
        background:     #fffff5;
        }
a:visited {
        color:          #93C;
        background:     #fffff5;
        }
a:active {
        color:          black;
        background:     #ccf;
        }
a:hover {
        color:          #fffff5;
        background:     #33F;
        }
```

FIGURE 3.37 Style rules for anchor elements.

family). The `font-size-adjust` property, which is not supported by IE6, has an effect if the first font family is not available. Given an appropriate specified value, the size of the selected font family is scaled so that its ex height is roughly the same as that of the first font family. This should make the letters appear to be about the same size regardless of the font actually used. Finally, notice that the left and right margins of the body are set at 5% of the width of the browser window, providing side margins that change as the window width changes.

The `p` and `h1` rules are reasonably straightforward, although the `h1` rule does use a numeric value for its `font-weight` property. This value corresponds to two steps bolder than `normal` and one step lighter than `bold`. The `hr` rule turns off the border, which the user agent style sheets for both IE6 and Mozilla 1.4 apparently turn on, and instead displays only a 1-pixel-high gray line. Note the use of both three-digit and six-digit color values.

Figure 3.37 shows the style rules related to hyperlinks (anchor elements). The first rule makes links bold and removes the underlining that would normally be associated with links. The remaining pseudo-class rules change the text and background colors of a hyperlink depending on its status, as described earlier.

So far, we have been slightly adapting the W3C Oldstyle Core style for our purposes. We next want to create a number of style rules specifically for the view-blog document. Recall that this document has three overall segments: an image above two segments, the blog entries on the left, and some navigation hyperlinks on the right. It is natural to lay out these segments by creating `div` elements and positioning them using CSS. Figure 3.38 shows the structure of the body of the new document (still called `index.html`).

The corresponding style rules are given in Figure 3.39. The first two rules center the top image and the body (main portion) of the document, which contains the blog entries and navigation links. It also fixes a width for the body portion of the document. This value

```
<div class="imgcentered">
  <!-- Banner image -->
  <img src="banner.gif" width="438" height="120"
       alt='"My Own Blog!" Banner' />
</div>
<div class="bodycentered">
  <div class="leftbody">
    <!-- Blog entries -->
    <div class="entry">
      ...
    </div>
    <hr />
    <div class="entry">
      ...
    </div>
  </div>
  <div class="rightbody">
    <!-- Side information -->
    ...
  </div>
</div>
```

FIGURE 3.38 Structure of the HTML document for the view-blog page using CSS rather than a table for layout.

```
        /* Classes for view-blog page */
.imgcentered {
        width:          438px;
        margin-left:    auto;
        margin-right:   auto;
        }
.bodycentered {
        width:          660px;
        margin-left:    auto;
        margin-right:   auto;
        }
.leftbody {
        width:          410px;
        float:          left;
        }
.rightbody {
        width:          230px;
        float:          right;
        }
.entry  {
        margin-top:     .75em;
        margin-bottom:  .75em;
        }
```

FIGURE 3.39 Style rules for `div` elements used for positioning.

```
<div class="datetime">AUGUST 9, 2005, 5:04 PM EDT</div>
<div class="entrytitle">I'm hungry</div>
<div class="entrybody">
  <p>
    Strange.  I seem to get hungry at about the same time
    every day.  Maybe it's something in the water.
  </p>
</div>
<hr />
```

FIGURE 3.40 Markup for a blog entry.

is narrow enough to be viewed without horizontal scrolling on almost any modern monitor, yet wide enough to display a reasonable number of words per line in the blog entries. The `div`'s for the entries and navigation links are then floated to the left and right, respectively, within this body `div`. Notice that the sum of the widths of these `div`'s is 20 px less than the width of the containing `div`, providing some visual separation between the blog entries and the navigation links. The final rule defines vertical spacing between blog entries, or more specifically, between blog entries and the horizontal rule separating them.

We can also use CSS to style the components of a blog entry. For example, the markup for the first entry of our example is shown in Figure 3.40, and Figure 3.41 gives style rules corresponding to this markup. Given the earlier discussion, these rules should not need any explanation.

Finally, let's use CSS to add a "displayed quote" feature, as illustrated in Figure 3.42. The basic idea is that if markup such as

```
.datetime {
        color:          #999;
        font-size:      x-small;
        }
.entrytitle {
        /* based on h2 of Oldstyle */
        font-family:    "Georgia", "Minion Web", "Palatino",
                        "Book Antiqua", "Utopia", "Times New Roman",
                        serif;
        font-size-adjust: .4;
        font-size:      1.75em;
        font-weight:    500;
        color:          #C00;
        margin-top:     .25em;
        }
.entrybody {
        font-size:      small;
        }
```

FIGURE 3.41 Style rules used for formatting components of a blog entry.

FIGURE 3.42 Example of a displayed quote (in a preview window, which suppresses the navigation links).

```
<span class='dquote'>It's more important than that.</span>
```

is included within text, then the content of the `span` will be displayed within the entry and also floated to the left of the enclosing text, enlarged, and enclosed within a three-sided, dotted border. This displayed-quote feature is not foolproof: if the `span` is included near the bottom of the text, then it might overlap with the next entry, since a floated element is taken out of the normal flow. But, if used carefully, it provides an interesting effect.

Figure 3.43 gives a suitable rule for producing the displayed-quote effect. One thing to notice is that the three-sided border was created using two declarations, and that the order of these declarations is important (the second rule overrides a portion of the first due to the cascade rules).

```
        /* For displaying a quote */
.dquote {
        float:          left;
        font-size:      larger;
        padding:        1px;
        margin-right:   5px;
        width:          10em;
        border-style:   dotted;
        border-left-style: none;
        border-color:   #999;
        }
```

FIGURE 3.43 Style rules for `span` element used to display a quote.

3.12 References

The primary reference for the material covered in this chapter is the CSS2 recommendation [W3C-CSS-2.0], and I have also consulted a version of the CSS 2.1 candidate recommendation [W3C-CSS-2.1] for guidance on which aspects of CSS2 seem most likely to find widespread support by browsers. The W3C home page for CSS, `http://www.w3.org/Style/CSS/`, contains links to all CSS recommendations as well as to CSS-related development software, books, tutorials, discussion groups, and other resources, including a CSS validator. Preliminary versions of CSS3 are also available at this site.

As mentioned earlier in Section 3.3, versions of Internet Explorer prior to IE6 did not fully support even CSS1, and IE6 also does not follow the CSS recommendation unless you use an appropriate document type declaration in your HTML document. See `http://msdn.microsoft.com/library/en-us/dnie60/html/cssenhancements.asp` for information on CSS in pre-IE6 versions of Internet Explorer as well as for details on how to turn on CSS-compliance in IE6. `http://www.mozilla.org/catalog/web-developer/css/` is Mozilla's documentation for CSS developers, which contains few details on Mozilla's support for CSS2 at the time of this writing (but see the next paragraph for other ways to learn about Mozilla's CSS support).

Going forward, as new browsers continue to emerge and older browsers become more rare, you will want to periodically acquaint yourself with emerging browser CSS capabilities. A preliminary set of CSS2 tests (and other helpful CSS information compiled by Eric Meyer, who has written extensively about CSS) is currently available at `http://www.meyerweb.com/eric/css/`. Presumably, a final CSS2 suite will eventually be available at the W3C site (a CSS1 suite is already available there). You can run such tests on various browsers yourself or rely on the results of tests performed by others. For example, as I wrote this chapter, I referred to the results of tests run by Christopher Hester (`http://www.designdetector.com/articles/CSS2TestSuiteFailures.php`) for information about CSS2 support (and lack thereof) in Mozilla and IE6. While this resource may or may not be up to date when you read this, a bit of Web searching for "CSS test suites" should provide the information you need.

Exercises

3.1. Practice writing simple style rules. In the following exercises, make use of the following declarations (one per line):

```
background-color: silver ;
font-size: larger ;
```

These will be referred to as "the background declaration" and "the text declaration," respectively.

- **(a)** Write CSS style rules that apply the background declaration to `div` elements and the text declaration to `strong` elements.
- **(b)** Write a single style rule that applies both the background and text declarations to both `p` and `em` elements.
- **(c)** Write a single style rule that applies the background declaration to HTML elements having a value of `Nevada` for their `id` attributes as well as to elements belonging to the `shiny` class.

(d) Write a style rule that applies the text declaration to `span` elements that belong to the `bigger` class.

(e) Write a style rule that applies the text declaration to `span` elements that are descendants of other `span` elements.

(f) Write a style rule that applies the background declaration when the cursor hovers over a hyperlink.

3.2. Create three external style sheets, using a different subset of the style rules you wrote for the previous exercise in each style sheet. Then write a complete XHTML 1.0 Strict document that uses all of your style rules.

(a) Your document should treat your style sheets as being of three different types:
- A non-persistent and preferred style sheet
- An alternate style sheet
- A style sheet used only if the XHTML document is printed

(b) Use the `@import` rule to have the first of your style sheets import the second, which imports the third. Your XHTML document should treat the first style sheet as a persistent style sheet.

3.3. Write an embedded style sheet (including the appropriate HTML tags) that sets the value of the `font-family` property to `Gill Sans Bold SmallCaps & OSF` for all elements of the document.

3.4. Assume that the author, user, and user agent style sheets for an HTML document are as follows:

- Author:
```
div  { color:blue }
p    { color:green;
       font-size:smaller !important }
.hmm { color:fuchsia }
```
- User:
```
p    { color:white;
       background-color:black;
       font-size:larger !important }
body { color:yellow }
```
- User agent:
```
body { color:black }
```

Assume that these are the only style rules for the document (i.e., no `style` attributes appear).

(a) What specified value will the browser use for the `color` property of `p` elements? For the `background-color` property of `p` elements? For the `font-size` property? Do any of your answers change if the `p` element belongs to the `hmm` class? Justify your answers.

(b) What specified value will the browser use for the `color` property of `div` elements? Does your answer change if the `div` element belongs to the `hmm` class? Does the value depend on which element type contains the `div`? Justify your answers.

(c) What `color` value will be given to a `ol` element that is a child of the `body` element, assuming that neither the `ol` element nor the `body` element belongs to the `hmm` class? Does your answer change if the `body` element (not the `ol` element) belongs to the `hmm` class? Justify your answers.

(d) Assume now that the user agent rule is changed to
```
*    { color:black }
```
and answer the previous question.

3.5. **(a)** Write a style rule to create a class named `quote`. This rule should set the top and bottom margins to `0` and the left and right margins to `4em`. The rule should contain a single (shortcut) declaration.

(b) Explain why `em` might be a better length unit to use for the task of indenting quoted text than `px` or one of the absolute length units.

3.6. Based on the textbook description of a typical browser's implementation of the CSS `px` (pixel) length measure, quantify how a `1px` length changes if a monitor's resolution is changed from 1024 by 768 to 1280 by 1024.

3.7. Picture "framing."

(a) Write a style rule that will place a nice "frame" around `img` elements. The "frame" should be brown. The inside edges of the "frame" should touch the outside edges of the image. There should be 10-px distance between adjacent images (either horizontally or vertically). See the left image in Figure 3.44.

(b) Modify your style rule to "mat" your images. In particular, there should now be a 3-px gap between the outside edges of your images and the inside edges of the "frames." This gap should be a tan color. See the right image in Figure 3.44.

3.8. Figure 3.22 shows a client area wider than the canvas. Explain how such a situation could occur in an HTML document.

3.9. The em and ex units are both related to the height of characters; there is no unit related to character width. Give a rationale for this difference.

3.10. Create an HTML document and CSS style sheet that together produce the stairstep effect shown in Figure 3.45. The right sides of the steps should line up in the middle of the document, regardless of the width of the browser window. Also, each step should have the same height as all other steps, regardless of the number of lines of text contained in the step (see the lowest step in the figure, for example).

3.11. Assume that an HTML document uses the following style sheet:

```
body { margin:5px; border:0; padding:2px }
div { margin: 3px; border:1px solid blue; padding:4px }
```

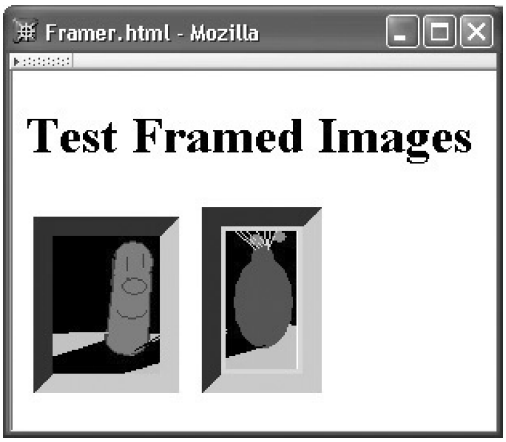

FIGURE 3.44 Two "framed" images. The right image is "matted." (Graphics courtesy of Ben Jackson.)

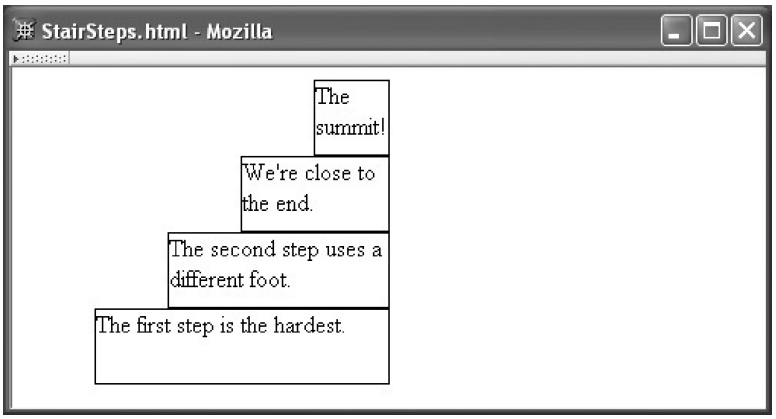

FIGURE 3.45 Stairsteps containing text.

Relative to the upper left corner of the initial containing block, for each of the three `div` elements in the following HTML `body` give the coordinates of the upper left corner of the content area of the block box generated by that `div`. Show/explain your work.

```
<body>
  <div id="div1">
    <div id="div2">
    </div>
  </div>
  <div id="div3"></div>
</body>
```

3.12. Assume that the `normal` value for the `line-height` property of a given font corresponds to the value `1.2em`. Also assume that the height of the baseline above the bottom of a character cell in this font is `0.2em`. If a value of `2em` is specified for `line-height`, what is the corresponding half-leading value? What is the height of the baseline above the bottom of a line box containing only text in the given font and given that the `line-height` value is `2em`?

3.13. Assume that the `line-height` value for a block box is `2em`.

(a) Assume that one of the line boxes within this block box contains an image with `height` equal to `1.5em` and a default value for `vertical-align`. If you knew the height of character cells as well as the height of the baseline within a character cell for the block box's font, how could you use this information to determine whether or not the height of the line box containing the image would need to be increased to make room for the image?

(b) If one of the line boxes contains an inline element having a specified font size twice the font size for the containing block box, will the baseline height of this line box necessarily be greater than it is in line boxes that contain only text in the default font? Explain.

3.14. **(a)** Identify at least three problems with the following style declaration:

```
font: 2em/12pt italic "Times New Roman" serif
```

(b) Rewrite the declaration so that it is syntactically correct.

(c) Assume that the corrected style declaration applies to an element E contained within an element to which the following declaration applies:

```
font-weight:bold
```

What will be the value of E's `font-weight` property?

3.15. The following HTML document produces an image followed by a label that is roughly vertically centered with respect to the image:

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      CenteredText.html
    </title>
    <style type="text/css">
      .labeledImage { font-size:100px }
      .image { vertical-align:middle; height:100px }
      .label { vertical-align:middle;
                        font-size:medium; font-weight:bold }
    </style>
  </head>
  <body>
    <div class="labeledImage">
      <img src="../images/CFP1.png" alt="cucumber"
        class="image" />
      <span class="label">A cucumber.</span>
    </div>
  </body>
</html>
```

(a) How would the rendered document differ if the `vertical-align` declaration were removed from the `label` rule? Why?

(b) How would the rendered document differ if the `vertical-align` declaration were removed from the `image` rule? Why?

(c) There is a large space between the image and the label when the document is rendered. Why is such a large space present, and how can a smaller space be displayed instead?

3.16. Consider the following markup:

```
<p style="font-size:12pt; line-height:1.15em">
  This paragraph has
  <span style="font-size:30pt">line-height:1.15em</span>
  which is "normal".
</p>
```

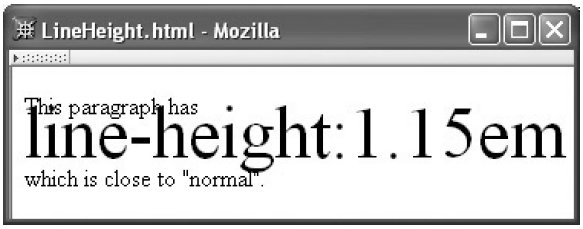

FIGURE 3.46 Rendering of markup with different font sizes.

The Mozilla browser rendering of this markup in a narrow window is shown in Figure 3.46. Why does text in the second line overlap the others? What small change to the markup would fix this problem?

3.17. Both the `vertical-align` property and relative positioning can be used to move an inline box vertically. Give a style rule for each approach that could be used to move an inline box up a distance of 1 cm. In addition to moving the box up, what other display change(s) might occur if the `vertical-align` approach is used?

3.18. Some Web pages create a navigation bar (navbar) on the left side of the page and the main content in a wider column on the right side of the page. Write style rules that could be used to wrap the content around the navbar. That is, at the top of the canvas the navbar and content should each be displayed side by side. However, lower on the canvas, when the bottom of the navbar is reached, the content area should extend across the entire width of the browser client area.

3.19. Assume that you want to lay out a number of playing card images so that they overlap one another, as shown at the top and bottom of Figure 3.47. Would it be easier to use absolute or relative positioning to accomplish this? Explain.

3.20. Write a style sheet that will cause the `li` elements within any `ol` element to be numbered in an outline style: the top-level `li` elements should use uppercase Roman numerals, the next level uppercase letters, the next level lowercase Roman numerals, then lowercase letters, and finally decimal numerals at the fifth level.

Research and Exploration

3.21. Create a document that displays two boxes. The first box should have a thin border and a width of 6 in. The second box should have an equivalent width in pixels, using the relation 1 in. = 96 px. Now answer the following questions using the browser(s) assigned by your instructor:

(a) Do the two boxes appear to be the same length when displayed by your browser?

(b) Measure the first box with a ruler. How many inches (or centimeters) across is it? If the width of the second box differs from the first, measure it as well.

(c) Print your document. Now what are the actual widths of the boxes?

3.22. Locate a Web site (or visit a site specified by your instructor) that displays the colors of the so-called "browser-safe color palette," a collection of colors that can reliably be

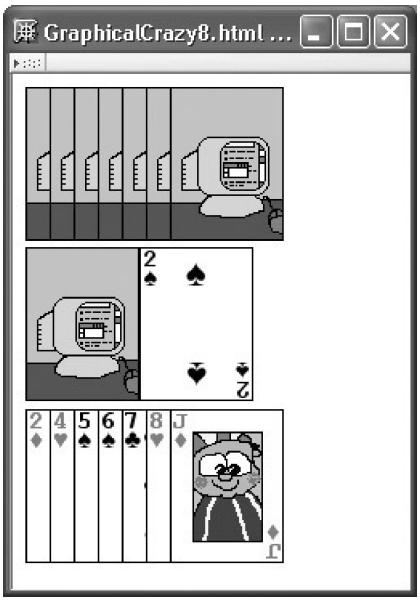

FIGURE 3.47 Overlapping images of playing cards (face card and card back images courtesy of Ben Jackson)

displayed even using video cards that are capable of showing only 256 different colors simultaneously.

(a) How many colors are contained in this palette?

(b) Of the colors red, orange, yellow, green, blue, purple, and brown, identify the color(s) that seem to have the most different shades in the browser-safe palette and the color(s) that have the fewest.

(c) Visit some popular Web sites (as directed by your instructor) and analyze their use of CSS color values by viewing document and style sheet sources. Which sites use the browser-safe palette and which do not?

3.23. Identify all of the color keywords containing "gray" or "grey" at `http://www.w3.org/TR/SVG11/types.html#ColorKeywords`. Create an HTML document and CSS style sheet that can be used to test a browser's support for these keywords. Use your document to test and report on the support for these color keywords provided by IE6 (or other browser(s) as assigned by your instructor).

Refer to the CSS2 recommendation [W3C-CSS-2.0]—or a later W3C recommendation as specified by your instructor—in order to answer the following questions.

3.24. Give a style sheet rule for the `body` element of a document that will cause a background image to be repeated across the vertical center of the browser client area. The image should remain in the center of the window even if the window is scrolled (see Fig. 3.48, in which "Draft . . . " is an image).

3.25. Describe what the `fixed` value for the `position` style property does when viewing a document in a browser. Give an example of how this feature might be useful. Test to see which browsers (as assigned by your instructor) support this value for `position` (IE6 does not).

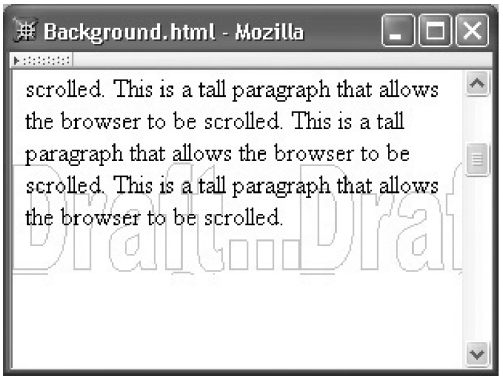

FIGURE 3.48 A background image containing "Draft . . . " is repeated across the center of the browser client area.

Projects

3.26. Create an external style sheet to be used with the HTML reference pages you wrote for Exercise 2.30 in order to accomplish the following (with minimal changes to your HTML source):

(a) Use a `seashell` background color for all pages (`seashell` is one of the SVG color keywords).

(b) Change the style of the bullets used in the short list of two hyperlinks (you choose the style).

(c) Define alternative colors for all four of the anchor pseudo-classes.

(d) Make table captions boldface, and table headers (`th` elements) normal weight but italicized.

(e) Use a 14-pt sans serif font for all `td` elements.

(f) Force text in cells that span multiple rows to be displayed at the top of the cell, rather than the default middle of the cell.

(g) On the definition pages, cause each term and its definition to be displayed in a box that has a 3-px solid yellow border. The box should occupy 75% of the width of the client area (or frame, if you completed the framed version of the earlier assignment) and should be centered.

(h) The W3C suggests the following markup be included in a Web page that is valid XHTML 1.0:

```
<p>
 <a href="http://validator.w3.org/check?uri=referer"><img
     src="http://www.w3.org/Icons/valid-xhtml10"
     alt="Valid XHTML 1.0!" height="31" width="88" /></a>
</p>
```

Use this markup on all of your valid XHTML 1.0 pages. Also include a `class` specification (of your choosing) in the `img` element. Then, without further change to this markup, write a style rule that will make these images appear on the right side of the browser client area and display the images at roughly half the height and width shown.

3.27. **(a)** Create an HTML document that renders similarly to Figure 3.47. Card images can be found in the `images/PlayingCards` directory of the example files download available at the Web site given in the Preface.

(b) Write a Java program that creates an HTML document such as the one just described. Your program should accept as input the number of cards to be held in each hand (the upper and lower parts of the figure represent hands held by players of a card game). The program should randomly select the images to display for all face-up cards (the right card in the second row and all of the cards in the lower hand). You will probably want to use the `java.util.Random` package for randomly selecting card images, but be careful not to display the same card image twice.

3.28. The following questions suggest extensions to the case study of Section 3.11. Implement a subset of the following requirements as specified by your instructor.

(a) Use CSS to style the comments document described in Exercise 2.33. First, link the comments document to the `style.css` file described in Section 3.11 (and available for download from the textbook Web site given in the Preface). Then add classes to `style.css` appropriate for styling various elements of the comments document, including each comment as a whole and the individual components of a comment: author name, comment heading, and comment body. Your class rules should center the comment heading over the comment body and right-justify the author name following the body. Finally, rewrite the document to use your new class definitions.

(b) Add a companion to the `dquote` class, named `drquote`, that is like `dquote` except that it floats text to the right instead of the left. Also, text within the floated box should be right-justified rather than left-justified, and the box border should be open on the right and closed on the left. Create an example document that demonstrates the use of your class.

(c) Use the validator at `http://jigsaw.w3.org/css-validator/` to ensure that the style sheet rules added in (a) and (b) are valid CSS. Turn in a copy of the Web page, showing your validation results.