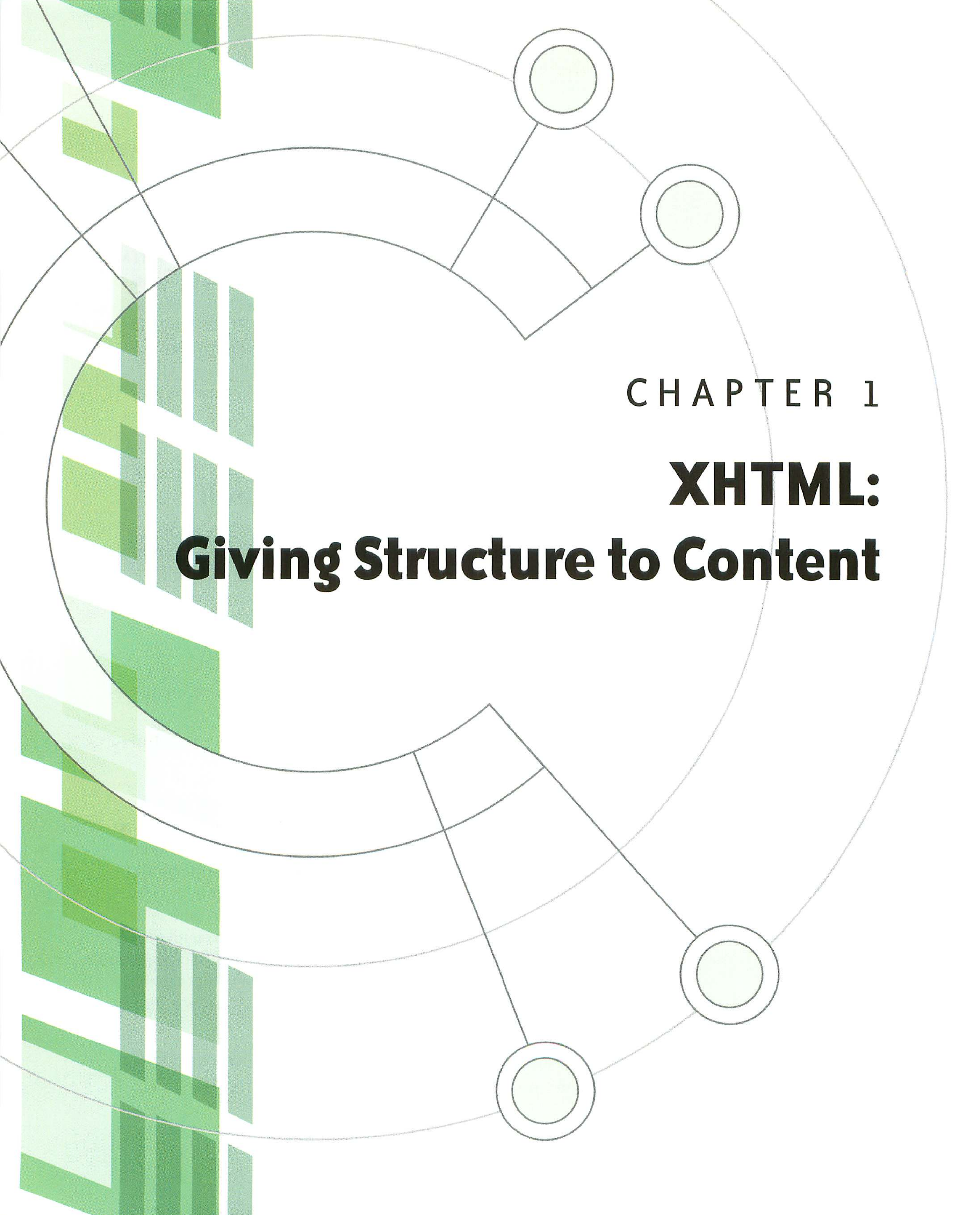

CHAPTER 1

# XHTML: Giving Structure to Content

**STYLIN' WITH CSS** is all about building standards-compliant Web sites in the most efficient, streamlined way possible, while making those sites accessible to the widest audience and easy to update. Web standards are simply a set of recommendations by the World Wide Web Consortium. If all browser manufacturers and all Web programmers followed them, so the theory goes, all Web pages would look and behave the same in every browser. Nice idea, but hard to realize.

When I wrote the first edition of *Stylin'* in late 2004, the Web standards movement was gaining massive momentum. Today, most new Web sites are being programmed to meet Web standards, and the Web is a better place because of it.

The Web standards advocates who drove this movement have made the Web a better and more predictable place. They worked with browser manufacturers to ensure that new browsers interpret the three primary interface programming languages (XHTML, CSS, and JavaScript) in the ways recommended by the W3C, instead of each one using custom tags and features in the self-interest of competitive advantage.

# Web Standards

By following best Web standards practices, Web developers like you and me can be very close to achieving a consistent display and performance of our sites for all our users. For example, you might expect Microsoft Internet Explorer to be the best, most Web standards-compliant browser, yet despite its current dominance, that is still not the case.

*Every so often I'll mention CSS2 or CSS3. These terms simply refer to a specific version of the CSS standard. As with any technology, CSS continues to be refined. CSS2 is now almost fully implemented in most browsers; CSS3 has been defined for some time, but is supported only partially by Firefox and Opera and barely at all by IE7. More on both later.*

Several other browsers do a good job of interpreting CSS, according to the W3C recommendations; the latest versions of Firefox and Opera on PC, and Safari and Firefox on Macintosh, all render XHTML styled with CSS2 in a remarkably consistent manner, but Microsoft Internet Explorer 6 (IE6) has numerous unimplemented features and buggy implementations of others.

Microsoft Internet Explorer 7 (IE7), released in October 2006, is a big improvement with regard to Web standards over IE6, and I had hoped that there would be rapid switchover from IE6 to IE7. However, according to thecounter.com, IE6 was still used by about 50 percent of all Web surfers as of July 2007.

## Even Today, IDWIMIE

*I'll always indicate if this applies only to IE6 or to both IE6 and 7.*

Anyway, as a result of IE6's refusal to die a rapid death, I still sometimes have to mention a CCS feature and tell you IDWIMIE (pronounced id-wimmy)—It Doesn't Work In Microsoft Internet Explorer.

For some of Internet Explorer's (and other older browsers) shortcomings, there are workarounds known as *hacks*—the nonstandard use of CSS to fool particular browsers into seeing or ignoring certain styles. It's tedious and time-consuming to create hacks, but as long as IE6 is around, the hacks must continue.

For us Web site designers and the visitors to the sites we create, Web standards offer the prospect of sites displaying and behaving consistently in every browser, on every platform. We're not there yet, but the days of every browser supporting a different feature set, with all the resultant inconsistencies that can make cross-browser/cross-platform Web development slow and frustrating, are, it seems, almost over. Web standards are clearly here to stay.

*Strictly speaking, XHTML and CSS aren't programming languages, but mechanisms for marking up and styling content respectively, so I am using the term "language" in a general way here.*

## Content, Structure, and Presentation

So, following the W3C's Web standards recommendations, *Stylin'* shows you how to publish *content* by defining its structure with XHTML and then defining its *presentation* with CSS.

1. **Content** is the collective term for all the text, images, videos, sounds, animations, and files (such as PDF documents) that you want to deliver to your audience.

2. **XHTML** (e**X**tensible **H**yper**T**ext **M**arkup **L**anguage) enables you to define *what* each element of your content is. Is it a heading or a paragraph? Is it a list of items, a hyperlink, or an image? You determine this by adding XHTML *markup* to your content. Markup comprises tags (the tag name is enclosed in angle brackets `< >`) that identify each element of your content. You create an *XHTML element* (hereafter just called an *element*) by either surrounding a piece of content with an opening and a closing tag like this

```
<p>This tag defines the text content as a paragraph</p>
```

or, for content that is not text (an image, in this example), by using a single tag

```
<img src="images/fido.gif" alt="a picture of my dog" />
```

This chapter focuses on XHTML and how to use it, but the most important thing to know right now is this: XHTML defines a document's *structure*.

*This img tag also has two attributes for the image source and alternative text respectively—see the sidebar "What Are Attributes?" to learn more.*

3. **CSS** (**C**ascading **S**tyle **S**heets) enable you to define *how* each marked-up element of your content is presented on the page. Is that paragraph's font Helvetica or Times? Is it bold or italicized? Is it indented or flush with the edge of the page? CSS controls the formatting and positioning of each of the content elements. To format the size of the text in a paragraph, I might write

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

which would make the text 12 pixels high. Almost this entire book is dedicated to teaching you CSS, but the most important thing to know right now is this: CSS defines a document's *presentation*.

### What Are Attributes?

Attributes can be added to a tag and can help further define that tag. Each attribute comprises two parts: the attribute name and the attribute value, in the format `name="value"`. For example, this image tag

```
<img src="images/fido.gif" alt="a picture of my dog" />
```

has two attributes: the image source, which has the value `"images/fido.gif"` that defines its relative location on the server, and an alternative text description, which has the value `"a picture of my dog"` that appears on-screen if the image fails to load, or that could be read aloud by a screen reader. Both these attributes are part of the structure of the document.

Before Web standards, it was common practice to load up tags with additional presentational attributes, such as text sizes and colors. Now, we can move all presentational information into the style sheet and thereby greatly reduce the complexity of our markup and use only attributes that define document structure.

Providing a means of separating a document's structure from its presentation was the core objective in the development of Web standards, and it is key to development of content that is both portable (can be displayed on multiple devices) and durable (ready for the future).

### The Top 10 Benefits of Standards-Based Coding

You may be wondering "Why should I bother to change the way I have been marking up pages for years?" Here are 10 great reasons to adopt standards-based coding practices:

1. **Deliver to multiple user agents.** The same piece of marked-up content is readily deliverable in a wide variety of user agents, the collective name for devices that can read XHTML, such as browsers, handhelds like smartphones, cell phones with browsers, and screen readers that read text for the sight impaired. You simply create a different style sheet for each device type, or let the XHTML display as is.
2. **Improve performance.** Pages are much lighter (smaller in file size) and therefore download faster, because your content only needs minimal structural markup. We can now replace all of the presentational markup we used to load into the tags in every page of a site with a single style sheet. As you will see, a single style sheet can define the presentation of an entire site, and the user's browser only needs to download it once.
3. **Serve all browsers.** With a little effort, you can have your pages degrade nicely in older browsers, so all users get the best experience possible with their available technology.
4. **Separate content and presentation.** You can modify, or entirely change, either the content or the presentation (read: design) of your site without affecting the other.

*(continued on next page)*

**The Top 10 Benefits of Standards-Based Coding** *(continued)*

5. **Build fluid pages.** It's easier to code for varying quantities of dynamic content within your pages. For example, it's much easier to create pages that can accommodate varying numbers of items in a given listing or menu of your e-commerce store.
6. **Confirm your code is correct.** Validation services for XHTML and CSS can be used during development to report instantly on errors in your coding. This provides faster debugging, and the assurance that a page is truly completed when it both displays correctly on-screen and passes validation.
7. **Streamline production.** Production is more efficient. It's too easy for you (the designer) to be sidetracked into content management, because you are the only person who knows where the content goes in the mass of presentational markup. You end up being the one to add it—a tedious job and probably not what you were hired to do. By adopting standards-based practices, you can provide simple markup rules to the content team and work in parallel on the presentational aspects, knowing their content and your design will marry seamlessly down the line.
8. **Distribute content more easily.** Distributing your content for third-party use is much easier because the content is separate from any specific presentation rules, and in many cases, simply not feasible otherwise.
9. **Make it accessible.** It's easier to make your site accessible and meet legal requirements, such as the Americans with Disabilities Act, Section 508, known colloquially as ADA 508.
10. **Do less work.** You write less code, and it's a whole lot quicker and easier to get the results you want and modify your work over time.

# The Times They Are A-Changing

Web standards are now quite widely adopted. Designers are moving away from using tables to lay out their pages and using clean structural markup, free of nested tables, spacers, and numerous `<br>` (line breaks) and ` ` (non-breaking spaces). These techniques were used only to force everything into the right place and had no meaning with respect to the content.

Here's an example of the old way of doing things:

## The Way We Were...

Take a look at this classic example of how Web sites were coded before Web standards became widely adopted. This is a snippet of markup from the Microsoft home page, July 1, 2004.

```
<table cellpadding="0" cellspacing="0" width="100%"
height="19" border="0" ID="Table5">

<tr>

<td nowrap="true" id="homePageLink"></td>
```

```
<td><span class="ltsep">|</span></td>

<td class="lt0" nowrap="true" onmouseenter="mhHover('localTo
olbar', 0*2+2, 'lt1')" onmouseleave="mhHover('localToolbar',
0*2+2, 'lt0')">

<a href="http://go.microsoft.com/?LinkID=508110">MSN Home</a>
</td>

<td><span class="ltsep">|</span></td>

<td class="lt0" nowrap="true" onmouseenter="mhHover('localTo
olbar', 1*2+2, 'lt1')" onmouseleave="mhHover('localToolbar',
1*2+2, 'lt0')">

<a href="http://go.microsoft.com/?linkid=317769">Subscribe</a>
</td>

<td><span class="ltsep">|</span></td>

<td class="lt0" nowrap="true" onmouseenter="mhHover('localTo
olbar', 2*2+2, 'lt1')" onmouseleave="mhHover('localToolbar',
2*2+2, 'lt0')">

<a href="http://go.microsoft.com/?linkid=317027">Manage Your
Profile</a></td>

<td width="100%"></td>

</tr>

</table>
```

All of this code produces just one row of buttons on this page (**Figure 1.1**).

FIGURE 1.1 It takes nearly 1,000 characters of code to create the three links above the picture.

The essential code is in green—247 characters out of 956, or less than 26 percent. The remaining 74 percent is just gooey chocolate sauce. Except for the `href` attributes, everything inside the tags is *presentation* and could all be ripped out and converted into a few brief definitions in a style sheet. The table is not used to display data; its purpose is solely to line everything up. The rest of the code is mostly concerned with making rollovers work. Each link requires the following information: a class to identify it to JavaScript, a forced `nowrap` attribute to keep the words on the link together, and two JavaScript function calls—yeah, on *every* link. (As an aside, rollovers are easy to create with CSS and require two simple CSS styles, as you will see later.) Note also that a table cell that contains a nested span with a class is required to display each tiny vertical line between the links.

Microsoft has recently made a decent effort to make its site more standards-compliant, but if your site's source (sauce?) code resembles the above, then read on. In the chapter on Interface Components, you will see how to create a similar navigation element with no more markup than a simple un-ordered list. After applying a few CSS rules, the result is a lightweight, easy-to-read, and most of all, semantically meaningful element that works on any XHTML-capable device, regardless of screen size, or even its capability to read CSS.

## The Future Just Happened

Today, with so many browsers and other devices standardizing around XHTML and CSS, noncompliant Web sites are finding that it is difficult to deliver their existing content on these newer devices and browsers. Have you seen your home page on a handheld computer lately?

Although bringing your current Web site into the modern age may take a substantial amount of work, you can console yourself that by following the new Web standards, you can do it once and do it right. If you are starting a new site, you can do it right the first time.

In *Stylin'*, you will learn to future-proof your site by separating the content from the presentation. You do this by marking up your content with XHTML, and then, using a single line of code, you link these pages to a separate file called a *style sheet*, which contains the presentation rules that define how the markup should be displayed.

The power of this church-and-state separation is that you can create different style sheets for browsers, for handheld devices, for printing, for screen readers used by the visually impaired, and so

*You can define which type of device a style sheet relates to using the* `link` *tag's* `media` *attribute.*

on. Many of today's user agents (types of devices) look for their specific type of style sheet. For example, smartphones such as RIM's Blackberrry and Palm's Treo look for a style sheet defined for use by a handheld device—and if it is present, use it, thus enabling you to provide a modified or entirely different presentation of the same XHTML on these small-screen devices.

Each style sheet causes the content to be presented in the best possible way for that use, but you only ever need *one* version of the XHTML content markup. As you will see, an XHTML page can automatically select the correct style sheet for each device or environment in which it is displayed. In this way, your write-once, use-many content becomes truly portable, flexible, and ready for whatever presentational requirements the future may bring its way. Note, however, that like any great vision of the future, there are still some current realities that we need to deal with.

# XHTML and How To Write It

*If you want more than this rather simplistic description of XML, check out the XML tutorial at the SpiderPro Web site (www.spiderpro.com/bu/buxmlm001.html).*

Because CSS is a mechanism for styling XHTML, you can't start using CSS until you have a solid grounding in XHTML. And what, exactly, is XHTML? XHTML is a reformulation of HTML as XML—didja get that? Put (very) simply, XHTML is based on the free-form structure of XML, where tags can be named to actually describe the content they contain; for example, `<starname>`Madonna`</starname>`. This very powerful capability of XML means that in XHTML, where the tag set is already defined for you, there is both a set of custom tags for your XHTML content and a second document, known as a *DTD* (document type definition), to explain how to handle those tags to the device that is interpreting the XHTML. By becoming XML-compliant, XHTML has been able to transcend the limitations of HTML and can be expanded over time, and can be shared as real-time Web services between other data systems. The DOCTYPE tag at the start of every XHTML Web page makes this critical association between the markup and the DTD.

XML has been almost universally adopted in business, and the fact that the same X (for eXtensible) is now in XHTML emphasizes the unstoppable movement toward the separation of presentation and content.

The rest of this chapter is dedicated to the latest, completely reformulated, totally modern, and altogether more flexible version of HTML.

## XHTML—The Rules

Correctly written XHTML markup gives you the best chance that your pages will display correctly in a broad variety of devices for years to come. The clean, easy-to-write, and flexible nature of XHTML produces code that loads fast, is easy to understand when editing, and prepares your content for use in a variety of applications.

You can easily determine if your site complies with Web standards—if your markup is *well-formed* with *valid* XHTML, and your style sheet is *valid* CSS.

*Well-formed* means that the XHTML is structured correctly according to the markup rules described in this chapter.

*Valid* means that that page only uses tags that are defined in the DTD (document type definition) that is associated with every modern Web page by the page's DOCTYPE tag (more on DOCTYPEs later). Certain tags that you may have been used to using in the past are now deprecated, meaning they still work but that a different, and usually more semantically correct, tag is now available for this purpose. To encourage you to use the newer tags, which unlike deprecated tags will still work in the future, deprecated tags are flagged as errors by the validator. You can check to see if your page meets these two criteria by uploading the page onto a Web server and then going to http://validator.w3.org and entering the page's URL.

*If you install the wonderful Developer's Toolbar into Firefox, you can easily validate pages on your local machine without uploading them to your web server. Download it from http://chrispederick.com/work/web-developer/*

Press Submit, and in a few seconds you are presented with either a detailed list of the page's errors or the very satisfying "This Page Is Valid XHTML 1.0 Strict!" message (**Figure 1.2**). CSS can be validated in the same way at http://jigsaw.w3.org/css-validator.

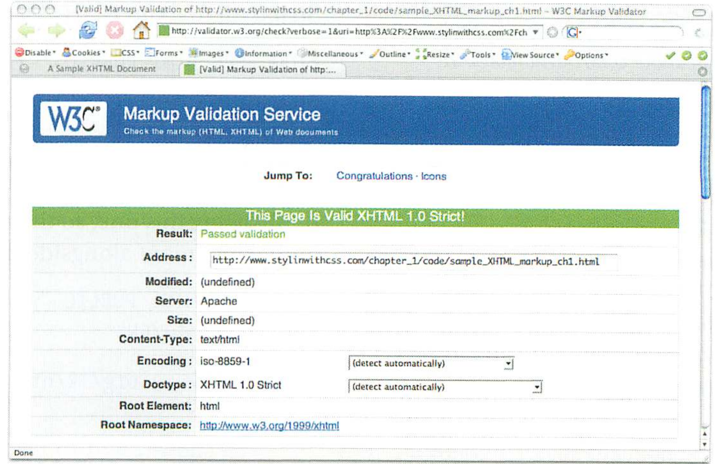


FIGURE 1.2 This nice message from the W3C validator almost guarantees that your page will display meaningfully on any XHTML-capable device.

## Does My Page Have To Validate?

The W3C validator (jigsaw.w3c.com) exists to enable you to ensure your pages are *valid* (only use elements and attributes as defined by the DOCTYPE's DTD) and *well-formed* (tags are structured correctly). It's definitely good practice to attempt to write pages that pass validation, and some would say that your pages *must* pass validation. What's undeniable is that the validator can instantly find common errors in your code that might otherwise take you hours to find.

However, just because a page doesn't validate doesn't necessarily mean it won't display the way you intend in current Web browsers. What future devices or other non-browser devices might do with such a page is not so certain.

My recommendation is to validate every page you create and take heed of the errors the validator displays. Close any tags it determines are left open, for example, and recode tags that it determines are being incorrectly nested (such as block elements inside inline elements). In short, you do want to ensure that your page is *well-formed*. However, *valid* is another story.

For example, in order to use Peter-Paul Koch's excellent JavaScript code that reveals additional parts of a form as the user makes selections (see http://www.quirksmode.org/dom/usableforms.html), you must add a `rel` attribute to each of the divs that hold the elements to be revealed. `rel` tags are not valid attributes for divs, and the page no longer passes validation, but the document can still be well-formed, so I am OK with letting that error go in exchange for added functionality. Some purists insist that every page must pass validation, but if it fails because of minor "valid" errors like this and is still well-formed, I think it's OK to ignore that advice.

Looking forward to the email I'm going to get on this one...

Here's the complete (and mercifully, short) list of the coding requirements for XHTML compliance:

*For a list of deprecated tags that you should abandon and replace with their XHTML equivalents, refer to the About.com Web site (http://webdesign.about.com/od/htmltags/a/bltags_deprctag.htm).*

*There are other flavors of DOCTYPES, and you can read about them at http://www.oreillynet.com/pub/a/javascript/synd/2001/08/28/doctype.html?page=1.*

1. **Declare a DOCTYPE.** The DOCTYPE goes before the opening `html` tag at the top of the page and tells the browser whether the page contains HTML, XHTML, or a mix of both, so that it can correctly interpret the markup. There are three main DOCTYPEs that let the browser know what kind of markup it is dealing with:

   **Strict:** All markup is XHTML-compliant.

   ```
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
   ```

   **Transitional:** This states that the markup is a mix of XHTML and deprecated HTML. Many well-established sites are currently using this one, so their old HTML code can exist happily in the document alongside the XHTML they are now adding.

   ```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
   Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
   ```

**Frameset:** This is the same as transitional but in this case frames, which are deprecated under XHTML, are OK, too.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
```

*If you copy a DOCTYPE or namespace from some other site, make sure that the URL is absolute (that is, it starts with http:// followed by a complete path to the document). Some sites (including W3C, of course) host their own DOCTYPE and namespace files, so they can use relative URLs to them. But if you use these URLs as is, with a different server that doesn't host these files, your pages may behave unpredictably because the URLs aren't pointing at anything.*

It is important to specify a DOCTYPE. Browsers that don't see a DOCTYPE in the markup assume that the site page was coded for browsers developed long before Web standards. My recommendation is that if you are building a site from scratch, and can therefore avoid deprecated or abandoned tag attributes, such as FONT and COLOR, use the XHTML Strict DOCTYPE listed previously.

When encountering a page without a DOCTYPE, many browsers go into what is known as *Quirks mode*, a backwards-compatibility feature supported by Mozilla, Internet Explorer 6 for Windows, and Internet Explorer 5 for Macintosh.

In Quirks mode, the browser functions as if it has no knowledge of the modern DOM (document object model) and pretends it has never heard of Web standards. This ability to switch modes depending on the DOCTYPE, or lack thereof, enables browsers to do the best possible job of interpreting the code of both standards-compliant and noncompliant sites.

*You can learn more about Quirks mode at the Dive into Mark Web site (http://diveintomark.org/archives/2002/05/29/quirks_mode).*

Note that for some weird reason, the `DOCTYPE` tag does not need to be closed with a slash and `DOCTYPE` is always in caps. This entirely contradicts XHTML rules 4 and 7 below. Go figure.

2. **Declare an XML namespace.** Note this line in your new `html` tag. Here's an example:

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:
lang="en">
```

When a browser is handling an XHTML page and wants to know what's in the DTD, which lists and defines all the valid XHTML tags, here's where it can find it: buried away on the servers of the WC3.

In short, the DOCTYPE and namespace declarations ensure that the browser interprets your XHTML code as you intended.

*Because the DOCTYPE (and the XML namespace and content type discussed in items 2 and 3) are a pain to type, they are in the page templates on the Stylin' Web site (www.stylinwithcss.com). You can use these as a starting point for your own XHTML documents. Just pick whichever of the three (Strict, Transitional, or Frames) DOCTYPEs you want to use.*

3. **Declare your content type.** The content type declaration goes in the head of your document, along with any other meta tags you may add. The most common is

```
<meta http-equiv="Content-type" content="text/html;
charset=iso-8859-1" />
```

This simply states what character coding was used for the document. ISO-8859-1 is the Latin character set, used by all standard flavors of English. If your next site is going to be in Cyrillic or Hebrew, you can find the appropriate content types on Microsoft's site (http://msdn.microsoft.com/workshop/author/dhtml/reference/charsets/charset4.asp).

4. **Close every tag, whether enclosing or nonenclosing.** Enclosing tags have content within them, like this

```
<p>This is a paragraph of text inside paragraph tags. To be XHTML-compliant, it must, and in this case does, have a closing tag.</p>
```

Nonenclosing tags do not go around text but still must be closed, using space-slash at the end, like this

```
<img src="images/siamese.jpg" alt="My cat" />
```

The space before the slash isn't required in modern browsers, but I always add it as it's easier to see that the tag is correctly closed.

5. **All tags must be nested correctly.** If a tag opens before a preceding one closes, it must be closed before that preceding one closes. For example:

```
<p>It's <strong>very important</strong> to nest tags correctly.</p>
```

Here, the `strong` tag is correctly placed inside the `<p>`; it closes before the containing `p` tag is closed. A tag enclosed inside another in this way is said to be *nested*.

This is wrongly nested

```
<p>The nesting of these tags is <strong>wrong.</p></strong>
```

Multiple elements can be nested inside a containing element; a list nests multiple `li` elements inside a single `ul` or `ol` element, like this:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

Because CSS relies on proper nesting in order to target styles to elements, you have to get this right. Use the W3C validator for confirmation that all tags are correctly nested and therefore your document is well-formed.

6. **Inline tags can't contain block level tags.** Block-level elements, such as `p` (paragraph) and `div` (division), automatically organize themselves one under the next down the page. If you have two paragraphs, the second paragraph appears by default under the previous one—no line breaks are required. By contrast, inline tags, such as `a` (anchor, a hyperlink) and `em` (emphasis, usually displayed as italics) occur in the normal flow of text, and don't force a new line.

   We discuss block and inline elements in detail later in Chapter 4, but for now, just remember that if you nest a block element, such as a paragraph `p`, inside an inline element, such as a link `a`, your code won't validate.

   Also, some block-level elements can't contain other block-level elements either; for instance, an `h1-6` (heading) tag can't contain a paragraph. Besides using validation, you can let common sense be your guide to avoid these problems. You wouldn't put an entire paragraph inside a paragraph heading when you are writing on paper or in Word, so don't do illogical things like that in your XHTML either, and you won't go far wrong.

*The use of `a` as the tag name for a link comes from the fact that a link that jumps to another location within the same page is know as an anchor. The same tag can be used to jump to a different page; an `a` tag used for this purpose is now universally referred to as a link. Of course, there is an XHTML `link` tag, which is used to associate a style sheet with a page, so don't get confused here. Remember, a "hyperlink" that the user clicks to jump to a new location is technically known as an anchor and always uses the `a` tag, even though everyone refers to this mechanism as a link.*

7. **Write tags entirely in lowercase.** Self-explanatory—no capital letters at all. I've always done this myself, but if you haven't, the days of `P` are over. Now it has to be `p`.

8. **Attributes must have values and must be quoted.** Some tags' attributes don't need values in HTML, but in XHTML, all attributes must have values. For example, if you previously used the `select` tag to create a pop-up menu in an HTML form, and wanted to have one menu choice selected by default when the page loaded, you might have written something like this

```
<SELECT NAME=ANIMALS>
<OPTION VALUE=Cats>Cats</OPTION>
<OPTION VALUE=Dogs SELECTED>Dogs</OPTION>
</SELECT>
```

   which would have given you a drop-down menu with Dogs displayed by default.

   The equivalent valid XHTML is this

*Quoted attribute values don't have to be lowercase, but it's good practice to write everything lowercase—then you can't go wrong. The only time I don't follow this guideline is with alt tags, where the attribute value—a text string—might appear on-screen.*

```
<select name="animals">
<option value="cats">Cats</option>
<option value="dogs" selected="selected">Dogs</option>
</select>
```

Note that in this revised version, all the tag and attribute names are in lowercase and all the attribute values are in quotes.

9. **Use the encoded equivalents for a left angle bracket and ampersand within content.** When XHTML encounters a left angle-bracket, `<` (also known as the less-than symbol), it quite reasonably assumes you are starting a tag. But what if you actually want that symbol to appear in your content? The answer is to encode it using an entity. An entity is a short string of characters that represents a single character; an entity causes XHTML to interpret and display the character correctly and not to confuse it with markup. The entity for the left angle-bracket/less-than symbol is `<`—remember `lt` stands for less than.

   Entities not only help avoid parsing errors like the one just mentioned, but they also enable certain symbols to be displayed at all, such as `©` for the copyright symbol (©). Every symbolic entity begins with an ampersand (`&`) and ends with a semicolon (`;`). Because of this, XHTML regards ampersands in your code as the start of entities, so you must also encode ampersands as entities when you want them to appear in your content; the ampersand entity is `&`.

   A good rule of thumb is that if a character you want to use is not printed on the keys of your keyboard (such as é, ®, ©, or £), you need to use an entity in your markup.

   There are some 50,000 entities in total, which encompass the character sets of most of the world's major languages, but you can find a shorter list of the commonly used entities at the Web Design Group site (www.htmlhelp.com/reference/html40/entities).

   And those are the rules of XHTML markup. They are relatively simple, but you must follow them exactly if you want your pages to validate (and you do).

*Various tools take your old HTML markup and convert it to XHTML. Of these, HTML Tidy is considered the best. The Infohound site (http://infohound.net/tidy) has an online version of HTML Tidy and links to downloadable versions and documentation. After the conversion is complete, you always have some final hand cleanup to do, but HTML Tidy and others can save you hours of work.*

## An XHTML Template

There are certain tags that must be in your Web page for it to be valid XHTML. As you learned from items 1, 2, and 3 above, you need to tell the browser whether your page is pure XHTML or also

contains deprecated tags, and what character encoding the page uses. No matter what content your page displays, these tags need to be present. You also need tags to indicate the head and body areas of the page. Dreamweaver will generate a "page template" containing all the required elements when you select New from the File menu—ready for you to add your page content. You can preset which DOCTYPE appears at the top by selecting: Edit > Preferences: New Document: Default Document Type (DTD) in your Preferences.

*When I use the term "page template" in this book, I am simply referring to a block of code such as the one shown to the right that forms the basis of an XHTML-compliant page, and not the notion of page templates that contain the nonchanging parts of page layouts as used by Dreamweaver and content management systems.*

To show you what this template looks like, here's the required code for a valid and well-formed page that uses the Strict XHTML 1.0 DOCTYPE:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!--the DOCTYPE-->

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

<title>XHTML 1.0 Strict template</title>

</head>

<body>

        <!--the content of your page goes here-->

</body>

</html>
```

*This template is in the Stylin' site download package. For more HTML and XHTML templates, see the Web Standards site (www.webstandards.org/learn/templates/index.html).*

To use this template, add the code for your page's content. You would also want to change the text between the title tags to something that describes the content of your page for both screen readers and search engines. See the sidebar "About Title Tags."

## About Title Tags

It's easy to miss the title of a page because it is displayed at the very top of the browser window, but title tags carry a lot of weight with search engines. The pages that get listed on page one of Google's results, for example, almost always have some or all of your search terms present in their titles, which are also displayed as the titles of each of the results. Make sure that your page title contains keywords that your users might use to search with and is written so that it entices clicks when it appears in search results. Don't waste your title tag with the useless and all-too-common "Welcome to our Home Page."

## Marking Up Your Content

Adopting Web standards means working in new ways. Start the development process by thinking about the structure of the content—what it means—rather than its presentation—what it looks like. That said, it's absurd to think you would begin programming without some sense of how the final page is gong to look when finished. I'll usually whip up (i.e., obsess over for days) an Adobe Fireworks comp to get an approval of the design from the client before I start, and I'll use that comp as a guide to what content needs to be in the markup. When actually marking up the page elements (as headings, paragraphs, images, etc.), so that I have something to style with my CSS, my focus is on "what is the most meaningful tag I can wrap around each piece of content?"

Once we cover the workings of CSS in the next chapter, then we can start looking at markup in terms of both using the right tag on each content element *and* ensuring that the elements are organized in a way that makes it easy to target them with your CSS rules.

Right now, we are going to focus on the individual XHTML tags and their semantic meaning, so that you can consider any piece of content that will appear on your page and select the most appropriate tag to mark it up. As we do this, we can also think about the concept of document flow.

## Document Flow—Block and Inline Elements

I mentioned previously that most XHTML tags fall into two broad categories in respect to the way they display on the page: block and inline. Block level elements such as headings `<h1>` through `<h6>` and paragraphs `<p>` will obligingly stack down the page with no line breaks required. They even have preset margins to create space between them. Inline elements have no such margins, and sit side by side across the page, only wrapping to the next line if there is not enough room for them to sit next to each other.

### A SIMPLE EXAMPLE OF XHTML MARKUP

In this first simple example of an XHTML page, the screenshot shows not only the stacking effect of block level elements, but also that inline elements, in this case `<a>` (a link) and `<strong>` (usually displays bold), can appear within block level elements and don't create new lines (**Figure 1.3** on the next page).

```
<!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>

<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

<title>Block and inline XHTML elements</title>

</head>

<body>

<h1>Here's a level 1 heading</h1>

<h2>Here's a level 2 heading</h2>

<p>This paragraph is very short and contains very little
text.</p>

<p>This paragraph is longer and contains not only
<strong>bold text</strong> but also <a href="#">a link</a>
that doesn't really link to anything.</p>

</body>

</html>
```

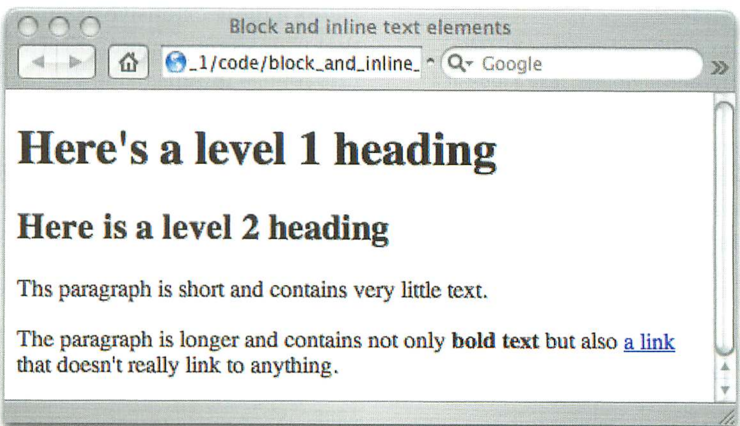


FIGURE 1.3 Example of the document flow of block and inline elements.

This example also illustrates XHTML's default document flow, which refers to the way the browser lays out block and inline elements. Document flow ensures meaningful layout of correctly marked-up content that has no author styling associated with it. As you will see later, there are all kinds of ways to use CSS to reorganize the default

document flow into a variety of layouts—multiple columns, for example—without having to pollute the markup with presentational tags and attributes.

### THE BROWSER'S INTERNAL STYLE SHEET

One interesting thing to note here is that each element by default has certain styles associated with it. As the screenshot shows, `h1` headings are already styled to be larger type than `h2` headings, and paragraph text is styled to be smaller than both of them. This is because a browser has a built-in style sheet that sets each element's default font size, color (type is black, links are blue, for example), display setting (block or inline), and usually many other settings, too.

When you use CSS to style an element, you are actually overriding the default settings for that element as set in the browser's style sheet. This makes the job easier, as you only need to change the styles that aren't already to your liking. However, if the browser doesn't read your style sheet for some reason, these default styles are your fallback, so it's worth making sure that your marked-up, but unstyled, page displays meaningfully in the browser before you start on the CSS. If it's well-formed XHTML, the default document flow pretty much guarantees it will.

*For a quick reference of XHTML tags and attributes, take a look at a listing at the Cookwood site (run by another Peachpit author, Elizabeth Castro, whose books I highly recommend) at http://www.cookwood.com/html/extras/xhtml_ref.html.*

### A MORE STRUCTURED XHTML PAGE

Here's a more extensive example (**Figure 1.4**) of a marked-up page that uses some common XHTML tags and also organizes the tags into logical groups using `div` tags based on their purpose in the page. We will look at more XHTML tags as we go on. I don't want to get into a full-blown rundown of each XHTML tag and its associated attributes here, as it would be a book in its own right, but I will show you examples of many different tags and their uses throughout this book.

*Highlighted code is the XHTML template shown earlier in the chapter..*

```
<!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" lang="en" xml:
lang="en">
<head>
<title>A Sample XHTML Document</title>
<meta http-equiv="Content-type" content="text/html;
charset=iso-8859-1" />
```

```
<meta http-equiv="Content-Language" content="en-us" />
</head>
<body>
<!--header-->
<div id="header">
<img src="images/stylin_logo1.gif" width="150" height="80"
alt="Stylin logo" />
  <h3>a New Riders book by Charles Wyke-Smith</h3>
</div>
<!--end header-->
<!--main content-->
<div id="contentarea">
  <h1>Welcome to XHTML</h1>
  <p>Good XHTML markup makes your content portable,
  accessible and future-proof. Creating XHTML-compliant pages
  requires following a few simple rules. Also, XHTML code
  can be easily validated online so you can ensure your code
  is correctly written.</p>
  <p>Here are the key requirements for successful validation
  of your XHTML code.</p>
  <ol>
    <li>Declare a DOCTYPE</li>
    <li>Declare an XML namespace</li>
    <li>Declare your content type</li>
    <li>Close every tag, enclosing or non-enclosing</li>
    <li>All tags must be nested correctly</li>
    <li>Inline tags can't contain block level tags</li>
    <li>Write tags in lowercase</li>
    <li>Attributes must have values and must be quoted</li>
    <li>Use encoded equivalents for left brace and
    ampersand</li>
  </ol>
```

```
  <a href="more.htm">more about these requirements</a> </div>
  <!--end main content-->
<!--navigation-->
<div id="navigation">
  <p>Here are some useful links from the Web site of the
  <acronym title="Word Wide Web Consortium">W<sup>3</sup>C
  </acronym> (World Wide Web Consortium), the guiding body of
  the Web's development.</p>
  <ul>
    <li><a href="http://validator.w3.org">W3C's XHTML
    validator</a></li>
    <li><a href="http://jigsaw.w3.org/css-validator/">W3C's
    CSS validator</a></li>
    <li><a href="http://www.w3.org/MarkUp/">XHTML Resources
    </a></li>
    <li><a href="http://www.w3.org/Style/CSS/">CSS
    Resources</a></li>
  </ul>
</div>
<!--end navigation-->
<!--footer-->
<div id="footer">
  <p>&copy; 2007 Charles Wyke-Smith.</p>
</div>
<!--end footer-->
</body>
</html>
```

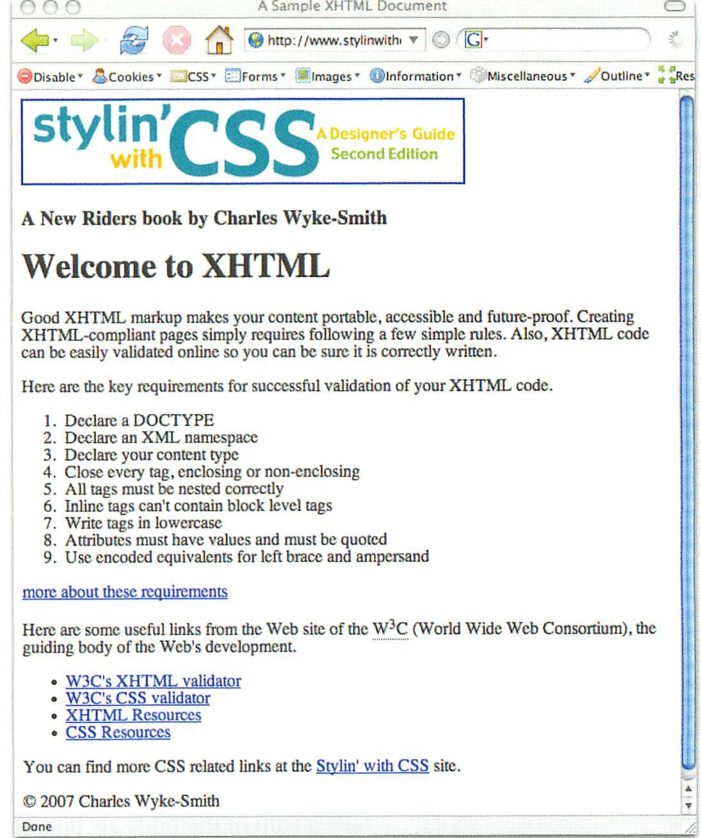


FIGURE 1.4 This is the above code rendered in the Firefox browser with the default browser styles. It's not beautiful, but it is useable.

*The default blue border around the header graphic indicates that the image is clickable (the `img` tag is enclosed in an `a` tag), and this rather ugly border can easily be removed with CSS, as we will see later.*

*Class attributes are similar to IDs in as much as they can be used to identify groups of tags, but while a class can appear many times within a page, an ID can appear only once. We will learn about the correct uses of classes and IDs in the next chapter. Also see the sidebar "Naming Classes and IDs."*

This page nicely illustrates the inherently structured document flow produced by correctly marked-up elements. You can see from the code that the markup has been divided with (aptly named) divs into four logical groups: header, content, navigation, and footer. The `id` attribute of these divs allows us to give each group a name that is unique to that page.

While it's helpful that these div IDs let us see at a glance where we are within the document structure, the primary purpose of organizing our tags within divs with IDs is to enable us to target sets of CSS rules at a specific group of tags, rather than the entire document. An element type, such as `p`, can be styled one way within one div and another way within another div. Let's start understanding how this works by looking at *document hierarchy*.

### Naming Classes and IDs

IDs and class attributes are identifiers you can add to your tags. You can add a class or an ID attribute to any tag, although most commonly, you add them to block-level elements. IDs and classes help you accurately target your CSS to a specific element or set of elements. I get into the uses for (and differences between) IDs and classes later, but for now, it's helpful to know that an attribute value must be a single word, although you can make compound words that the browser sees as single words using underscores, such as `class="navigation_links"`.

Because the browser can misinterpret attribute names made of bizarre strings of characters, my advice is to start the word with a letter, not a number or a symbol. Because the only purpose of a class or ID is to give an element a name that you can reference in your style sheet (or JavaScript code), the value can be a word of your own choosing. That said, it's good practice to name classes and IDs something meaningful such as `class="navigationbar"` rather than `class="deadrat"`. Although the deadrat class might provide a moment of levity during a grueling programming session, the humor may be lost on you when you are editing your code at some point in the future. Don't save time with abbreviated names either; call the ID `"footer"` rather than `"fr"` or you are apt to waste your time (or someone else's) later trying to figure out what you meant. Do yourself a favor and take the time to give classes and IDs unambiguous and descriptive names.

## Document Hierarchy: Meet the XHTML Family

Document hierarchy is the final XHTML concept we'll look at before we start looking at CSS. The document hierarchy is like a family tree or an organizational chart that is based on the nesting of a page's XHTML tags. A good way to learn to understand this concept is to take a snip of the body section of the markup we just discussed and strip out the content so that you can see the organization of the tags better. Here's the stripped-down header

```
<body>
  <div id="header">
    <img />
    <h3> </h3>
  </div>
<!-- remaining tags removed here for clarity -->
</body>
```

*Dreamweaver's Code View will automatically indent nested tags as shown above (Commands > Apply Source Formatting) to help you see the hierarchy more clearly.*

Now you can clearly see the relationships of the tags. For example, in the markup, you can see that the `body` tag contains (or nests) all the other tags. You can also see that the `div` tag (with the ID of `"header"`) contains two tags: an image tag and a head 3 tag.

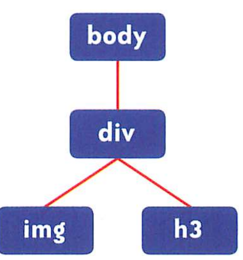


FIGURE 1.5 shows a conceptual way to represent a document's structure—with a hierarchy diagram.

When examining this hierarchical view (**Figure 1.5**), we can say that both the `img` tag and the `h3` tag are the *children* of the `div` tag, because it is the containing element of both. In turn, the `div` tag is the *parent* tag of both of them, and the `img` tag and the `h3` tag are *siblings* of one another because they both have the same parent tag. Finally, the `body` tag is an *ancestor* tag of the `img` and `h3` tags, because they are indirectly descended from it. In the same way, the `img` and `h3` tags (and the `div`, for that matter) are *descendants* of the `body` tag. To quote Sly Stone: "It's a family affair..."

In CSS, you write a kind of shorthand based on these relationships, for example

```
div#header img {some CSS styling in here}
```

Such a CSS rule only targets `img` tags inside of (descended from) the div with the ID of `"header"` (the `#` is the CSS symbol for an ID). Other `img` tags in the page are unaffected by this rule because they aren't contained within the `"header"` div. In this way, you can add a border around just this image or set its margin to move it away from surrounding elements.

We will get into learning to write CSS rules like this in greater detail in the next chapter, but the important concept to understand is that every element within the body of your document is a descendant of the `body` tag, and, depending on its location in the markup, the element could be an ancestor, a parent, a child, or a sibling of other tags in the document hierarchy.

By creating rules that use (and often combine) references to IDs, classes, and the hierarchy structure, you have means by which you can accurately dictate which CSS rules affect which XHTML elements, and this is exactly what you will learn to do next.