CHAPTER 2

XHTML and CSS Basics

Chapter 1 briefly introduced you to XHTML and CSS, and in this chapter we'll show you how you can author markup and style sheets to create your own web pages. You'll become familiar with the essential components of XHTML documents and how they should be correctly assembled. As you know, you must adhere to some standards when authoring a document for the web, and we're going to be following the rules of XHTML 1.0 Strict throughout this book. XHTML is an updated reformulation of HTML, with just a few more stringent rules to obey, and we'll point out the differences between the two languages in this chapter.

Later in the chapter, we'll guide you through the essentials of CSS so you can use it to visually style your web pages. XHTML provides the structure that supports the content of your web pages, while CSS provides the polish to make your content more attractive and memorable. Designing websites with CSS isn't possible without some solid bedrock of markup underneath, so let's begin at the beginning.

The Parts of Markup: Tags, Elements, and Attributes

The linchpin of XHTML—as well as other markup languages—is the *tag*. Tags are the coded symbols that separate and distinguish one portion of content from another while also informing the browser of what type of content it's dealing with. A user-agent can interpret the tags embedded in an XHTML document and treat different types of content appropriately. Most of the tags available in XHTML have names that describe exactly what they do and what type of content they designate, such as headings, paragraphs, lists, images, quotations, and so on.

Tags in XHTML are surrounded by angle brackets (< and >) to clearly distinguish them from ordinary text. The first angle bracket (<) marks the beginning of the tag, immediately followed by the specific *tag name*, and the tag ends with an opposing angle bracket (>). For example, this is the XHTML tag that indicates the beginning of a paragraph:

```
<p>
```

Notice that the tag name is written in lowercase, which is a requirement of XHTML; tag names are not case-sensitive in HTML (and many web authors write them in uppercase to make their markup more readable), but they must be lowercase in XHTML (that's one of those more stringent rules that separates XHTML from HTML).

Most tags come in matched pairs: one *opening tag* (also called a *start tag*) to mark the beginning of a segment of content and one *closing tag* (also called an *end tag*) to mark its end. For example, the beginning of a paragraph is indicated by the opening tag, `<p>`, and the paragraph ends with a `</p>` closing tag; the slash after the opening bracket is what distinguishes it as a closing tag. A full paragraph would be marked up as follows:

```
<p>Hello, world!</p>
```

These twin tags and everything between them forms a complete *element*, and elements are the basic building blocks of an XHTML document. Some elements don't require a closing tag in older versions of HTML—the appearance of a new opening tag implies that the previous element has ended and a new one is beginning. But in XHTML, *all* elements must end with a closing tag . . . almost all, that is.

Some tags indicate *empty elements*, which are elements that do not, and in fact *cannot*, hold any contents. Empty elements don't require a closing tag but are instead "self-closed" in XHTML with a *trailing slash* at the end of a single tag that represents the entire element. For example, the following tag represents a line break, an empty element that forces the text that follows it to wrap to a new line when a browser renders the document (you'll learn more about this element in Chapter 4):

```
<br />
```

The space before the trailing slash isn't strictly required, but it will help older browsers interpret the tag correctly—without that space, some rare, old browsers fail to notice the tag's closing bracket. Some empty elements are also known as *replaced elements*; the element itself isn't actually rendered by a graphical browser but is instead replaced by some other content. Empty elements in HTML should not include a trailing slash.

An element's opening tag can carry *attributes* to provide more information about the element—specific properties that element should possess. An attribute consists of an *attribute name* followed by an *attribute value*, like so:

```
<p class="greeting">Hello, world!</p>
```

This paragraph includes a `class` attribute with a value of "greeting," making it distinct from other paragraphs that don't include that attribute (you'll learn more about the `class` attribute later). An attribute's name and its value are connected by an equal sign (`=`), and the value is enclosed in quotation marks. All attribute values must be quoted in XHTML, using either single quotes (`' . . . '`) or double quotes (`" . . . "`) so long as both of them match (quoting a value like `" . . . '` wouldn't be valid). Quoting attribute values was optional in HTML but is required in XHTML Strict. Some attributes don't require a value in HTML (an attribute without a value is called a *minimized attribute*), but all attributes must have a value in XHTML—minimizing attributes isn't allowed. Like tag names, attribute names must be lowercase in XHTML but aren't case-sensitive in HTML. Attribute values are never case-sensitive, especially since some values might need to use capital letters. Even so, it's not a bad idea to use lowercase wherever practical, for consistency's sake.

An element's opening tag can include several attributes separated by spaces, and attributes must appear *only* in an opening tag (or an empty element's lone, self-closing tag). Some elements require specific attributes, while others are optional—it all depends on the individual element, and you'll be learning about all of them throughout the rest of this book, including which attributes each element may or must possess.

Figure 2-1 illustrates the components of an element.

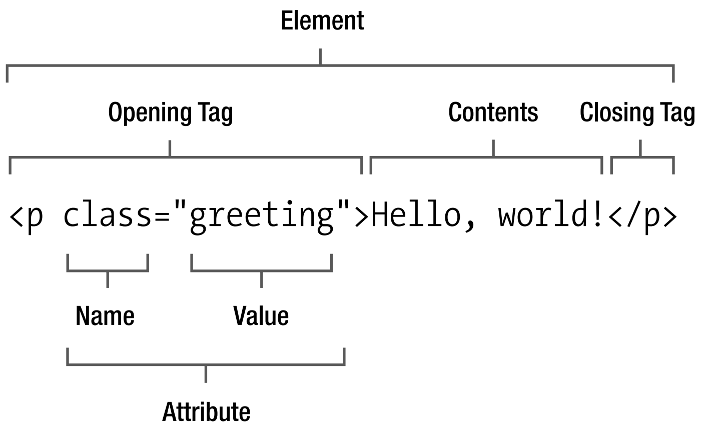

Figure 2-1. *The basic components of an XHTML element*

Block-Level and Inline Elements

The entire range of elements can be divided into two basic types: block-level and inline. A *block-level* element is one that contains a significant block of content that should be displayed on its own line, to break apart long passages of text into manageable portions such as paragraphs, headings, and lists. An *inline* element usually contains a shorter string of text and is rendered adjacent to other text on the same line, such as a few emphasized words within a sentence.

Many nonempty, block-level elements can contain other block-level elements, and all can contain text and inline elements. A nonempty, inline element, on the other hand, can contain only text or other inline elements. For example, the `em` element is inline and is used to add emphasis to the text within it, while the `p` element is block-level and designates a paragraph of text. Because `em` is inline, it cannot contain block-level elements, so the following example is wrong and invalid:

```
<em><p>Hello, world!</p></em>
```

You'll find out which elements are block-level and which are inline as you progress through this book, exploring each element in greater detail.

Nesting Elements

Elements can be *nested* like Russian nesting dolls, each one residing within its containing element. They must be nested correctly, with each closing tag appearing in the correct order to close an inner element before you close its container. The following markup is an example of an improperly nested set of elements:

```
<p><em>Hello, world!</p></em>
```

The opening `<em>` tag occurs after the opening `<p>` tag, but the closing `</p>` tag occurs *before* the closing `</em>` tag. To ensure correct nesting of elements, always close them in the reverse order in which they were opened:

```
<p><em>Hello, world!</em></p>
```

White Space

When you create your XHTML documents as plain text, you're free to format them however you want. Line breaks and indentions can help make your markup more readable as you work, as you'll see in most of the markup examples in this book. Indenting nested, block-level elements can make it easier to see where a particular element opens and closes, and thus you're less likely to run into nesting problems or forget to end an element with the correct closing tag.

Web browsers ignore any extra line breaks and carriage returns, collapsing multiple spaces into a single space. To illustrate, here's a bit of markup with a lot of extra space:

```
<p>


Wide      open
               spaces   !
      </p>
```

This is a rather extreme example—one you'd probably never commit yourself—but it serves to demonstrate how all of those spaces are collapsed when a browser renders the document. Although the spaces and returns are intact in the markup, your visitors would see something like this:

```
Wide open spaces!
```

Sometimes you may want to preserve extra spaces, tabs, and line breaks in your content—when you're formatting poetry or presenting computer code on your pages, for instance. The `pre` element can delineate passages of preformatted text in just such cases, and you'll learn more about that element in Chapter 4.

Standard Attributes

We'll be listing each element's required and optional attributes as they're covered individually throughout this book. But some common attributes can be assigned to practically any element (and are almost always optional). To spare you the repetition, we'll cover those attributes here, divided into a few categories.

Core Attributes

These attributes include general information about the element and can be validly included in the opening tag of almost any element:

- `class`: Indicates the class or classes to which a particular element belongs. Elements that belong to the same class may share aspects of their presentation, and classifying elements can also be useful in client-side scripting. A class name can be practically any text you like but can be made up only of letters, numbers, hyphens (-), and underscores (_); other punctuation or special characters aren't allowed in a `class` attribute. Any number of elements may belong to the same class. Furthermore, a single element may belong to more than one class, with multiple class names separated by spaces in the attribute value.

- `id`: Specifies a unique identifier for an element. An ID can be almost any short text label, but it must be unique within a single document; more than one element cannot share the same identifier. The `id` attribute cannot contain any punctuation or special characters besides hyphens (-) and underscores (_). The first character in an ID must be a letter; it cannot begin with a numeral or any other character.

- `style`: Specifies CSS properties for the element. This is known as *inline styling*, which you'll learn more about later in this chapter. Although the `style` attribute is valid with most elements, it should almost always be avoided because it mixes presentation with your content.

- `title`: Supplies a text title for the element. Many graphical browsers display the value of a `title` attribute in a "tooltip," a small, floating window displayed when the user's cursor lingers over the rendered element.

Internationalization Attributes

Internationalization attributes contain information about the natural language in which an element's contents are written (such as English, French, Latin, and so on). They can be included in almost any element, especially those that contain text in a language different from the rest of the document's content.

- `dir`: Sets the direction in which the text should be read, as specified by a value of `ltr` (left to right) or `rtl` (right to left). This attribute usually isn't needed, since a language's direction should be inferred from the `lang` and `xml:lang` attributes.
- `lang`: Specifies the language in which the enclosed content is written. Languages are indicated by an abbreviated language code such as `en` for English, `es` for Spanish (Español), `jp` for Japanese, and so on. You can find a listing of the most common language codes at `http://webpageworkshop.co.uk/main/language_codes`.
- `xml:lang`: Also specifies the language in which the enclosed content is written. This is the XML format for the `lang` attribute, as it should be used in XML documents. XHTML documents are both XML and HTML (depending on how the server delivers them), so both the `lang` and `xml:lang` attributes may be applied to an element, both with the same language code as their value.

Focus Attributes

When some elements—especially links and form controls—are in a preactive state, they are said to have *focus* because the browser's "attention" is concentrated on that element, ready to activate it. You can apply these focus attributes to some elements to enhance accessibility for people using a keyboard to navigate your web pages:

- `accesskey`: Assigns a keyboard shortcut to an element for easier and quicker access through keyboard navigation. The value of this attribute is the character corresponding to the access key. The exact keystroke combination needed to activate an access key varies between browsers and operating systems.
- `tabindex`: Specifies the element's position in the tabbing order when the Tab key is used to cycle through links and form controls.

■ **Note** Numerous *event attributes* are available for client-side scripting, including `onclick`, `ondblclick`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, and `onmouseup`. Each of these events occurs when the user performs the indicated action upon the element. However, use of such *inline event handlers* is strongly discouraged, so we won't be covering these optional attributes in any detail. Scripted behavioral enhancements are best separated from the document's content and structure, just as presentation should be separated. Chapter 10 offers a general introduction to client-side scripting.

Adding Comments

It's often useful to embed comments in your documents. They're notes that won't be displayed in a browser but that you (or someone else) can read when viewing the original markup. Comments can include background on why a document is structured a particular way, instruction on how to update a document, or a recorded history of changes. Comments in XHTML use a specialized tag structure:

```
<!-- Use an h2 for subheadings -->
<h2>Adding Comments</h2>
```

A comment starts with `<!--`, a set of characters the browser recognizes as the opening of a comment, and ends with `-->`. Web browsers won't render any content or elements that occur between those markers, even if the comment spans multiple lines. Comments can also be useful to temporarily "hide" portions of markup when you're testing your web pages.

```
<!-- Hiding this for testing
<h2>Adding Comments</h2>
End hiding -->
```

Although a browser doesn't visibly render comments, the comments are still delivered along with the rest of the markup and can be seen in the page's source code if a visitor views it. Don't expect comments to remain completely secret, and don't rely on them to permanently remove or suppress any important content or markup.

The XHTML Document

So far, we've been using the words *document* and *page* repeatedly, and you might think those terms are interchangeable. But generally speaking, when we refer to a *document,* we're talking about the plain-text file that contains the XHTML source code, while a *page* is the visible result when a graphical web browser renders that document. A document is what you author, and a page is what you (and the visitors to your website) will see and use.

An XHTML document must conform to a rigid structure to be considered valid and well formed, with a few required components arranged in a precise configuration. Listing 2-1 shows the basic skeleton of a well-formed document, with all the required pieces in their proper places.

Listing 2-1. *A Basic XHTML Document*

```
<!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>My first web page</title>
  </head>
  <body>
    <p>XHTML is easy!</p>
  </body>
</html>
```

As simple as it seems, this is actually a complete, valid, well-formed document. Every web page you create will begin with a framework just like this. Next, we'll discuss a few of the components in a bit more detail.

The Doctype

An XHTML document begins with a *Document Type Declaration* (*doctype*, for short), a required component that—as the name suggests—declares what type of document this is and the set of standardized rules the document intends to follow. Each "flavor" of XHTML has its own corresponding doctype.

- *XHTML 1.0 Strict*:

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

- *XHTML 1.0 Transitional*:

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

- *XHTML 1.0 Frameset*:

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

The doctype declaration is a sort of tag, but despite its enclosing angle brackets, it's not an element in XHTML, so it doesn't require a closing tag or trailing slash. In fact, it's not truly part of the document's markup at all; it merely relays information about the document to the user-agent so it can determine what kind of document it's dealing with and render the page according to the proper rules.

The doctype must appear in your XHTML documents exactly as we've shown here, complete with capitalization and quotes, though it doesn't have to be broken onto two lines. Other versions of HTML have their own doctypes, but we'll be using XHTML 1.0 Strict throughout this book. For a more exhaustive investigation into the parts of a doctype declaration, see Brian Wilson's informative explanation at `http://www.blooberry.com/indexdot/html/tagpages/d/doctype.htm`.

Doctype Switching: Compliance Mode vs. Quirks Mode

When a web browser downloads an HTML or XHTML document, it must make a number of programmed assumptions in order to parse the document's markup and apply the presentation suggested by the author's CSS. The earliest browsers that supported CSS did so largely according to their own rules, rather than following the standardized specifications. This was a major stumbling block in the adoption of CSS and web standards in general. A page might be rendered perfectly in one graphical browser and appear completely broken in another.

As browsers improved their support of CSS—that is, moved toward better compliance with web standards—they were faced with a dilemma. Many websites had already been designed with built-in dependencies on the inconsistent, inaccurate renderings of older browsers. Suddenly opting to follow the rules could cause millions of web pages to seem "broken" in the latest version of a web browser when they looked just fine the day before. The site didn't change overnight; only the browser's method of rendering it did.

This dilemma inspired the introduction of the *doctype switch*. When a document includes a full, correct doctype, a modern browser can assume the entire document is well formed and authored according to web standards. The browser can then render the page in a mode intended to comply with the established standards for markup and CSS, a mode known as *compliance mode* or *strict mode*. If the doctype is missing, incomplete, or malformed, the browser will assume it's dealing with an outdated document and revert to its loose and tolerant rendering mode, known as *quirks mode* because it's intended to adjust to the various quirks of nonstandard and improperly constructed markup (it's also sometimes called *compatibility mode*). Older browsers lack a built-in doctype switch and so are forever locked in their outdated quirks modes.

To correctly invoke compliance mode in modern web browsers, a complete doctype must be included as the very first line of text in a document; only white space is allowed to appear before it. Any markup, text, or even comments appearing before the doctype declaration will throw most modern browsers into quirks mode, with often-unpredictable results. Designing websites with CSS is considerably easier and the results are more consistent when the document is rendered in compliance mode. Hence, including a complete and correct doctype is essential. And because a doctype is already a required part of a valid web document, modern browsers will always render your pages in compliance mode if you build your documents correctly.

Peter-Paul Koch offers additional information and opinions on quirks mode at his aptly named website, Quirks Mode (`http://www.quirksmode.org/css/quirksmode.html`). To find out just how documents are rendered differently in quirks mode, see Jukka Korpela's article "What Happens in Quirks Mode?" (`http://www.cs.tut.fi/~jkorpela/quirks-mode.html`).

THE XML DECLARATION

To be honest, a proper XHTML document should include an *XML declaration* before the doctype. This special declaration indicates that the document has been encoded as XML and optionally specifies the XML version and the document's character encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Internet Explorer for Windows is far and away the most common graphical web browser for the most common computer operating system on the planet today, dominating 70% to 90% of the web-using world, depending on which statistics you believe. And, unfortunately, Internet Explorer doesn't recognize an XML declaration, instead reverting to quirks mode when any text appears before the doctype.

Including an XML declaration in your XHTML documents, while absolutely correct, would simply result in a vast number of your site's visitors seeing your pages displayed in an outdated rendering mode, a sure-fire recipe for frustration when you attempt to achieve consistent cross-browser presentation with CSS. So we recommend against adding an XML declaration, though its omission might make some XML purists cringe.

The html Element

The actual markup begins after the doctype with the `html` element, which acts as a container for the entire document. This is known as the *root element*, the one from which all other elements sprout and grow. The `html` element has no other properties of its own; it's strictly a container that defines where the document begins and ends. Any elements or content that appear outside this element (apart from the doctype, which isn't an element) will make the entire document invalid.

Required Attributes

- `xmlns`: A URL specifying an XML namespace, which is `http://www.w3.org/1999/xhtml` for XHTML documents

Optional Attributes

There are no optional attributes for the `html` element.

Standard Attributes

- `dir`
- `id`

- `lang`
- `xml:lang`

A *namespace* is where element and attribute names are specified for XML languages. XML is an extensible markup language, allowing authors to define their own customized elements and attributes. For example, an `animal` element with a `species` attribute could be useful for documents about animals, and such customized names could be defined in a special namespace. XHTML 1.0, on the other hand, has a *predefined* set of element and attribute names, and the correct URL of its namespace is `http://www.w3.org/1999/xhtml` (XHTML 1.1 and 2 can be extended with a custom namespace, but those versions of XHTML haven't yet been released as official standards). The namespace is declared in an XHTML document via the `xmlns` attribute of the root `html` element.

The standard `lang` and `xml:lang` attributes are optional for the `html` element (as they are for most other elements). Because this is the root element from which all other elements descend, the language declared here will be passed on to every other element in the document, so it's recommended to include them.

CONTENT TYPES

Web servers and clients rely on standardized *content types* to differentiate one type of content from another, in order to determine how the data should be processed. Plain, unformatted text is delivered with a content type of `text/plain`, a JPEG image is delivered with a content type of `image/jpeg`, an MPEG video uses `video/mpeg`, and so on. Most of this goes on automatically, behind the scenes between the server and the client, and a web author usually doesn't need to be concerned with content types. Content types are also known as *Internet media types* or *MIME types* (MIME stands for Multipurpose Internet Mail Extensions, but the standard is used on the web as well).

HTML documents use a content type of `text/html`, so both the server and the client know exactly what that document is and how it should be handled. However, we've said before that XHTML is a reformulation of HTML following the stringent rules of XML. But the truth is that XHTML *is* XML, and should most correctly be served as such with a content type of `application/xml+xhtml`. Unfortunately, many popular web browsers (most notably Internet Explorer for Windows, the most dominant browser in the world) don't correctly interpret XHTML documents served with the correct content type. Those browsers, unable to cope with XHTML delivered as XML, will fail to render the document. This is simply unacceptable for most web authors since the overwhelming majority of the browsing public would be unable to see and use their sites.

Furthermore, devices that parse XML are required to stop processing the document on the first error they encounter. A single validation error would make the entire web page fail if it was being treated as true XML. As much as we might want to keep our documents strictly valid, it's simply not always possible, especially when third-party software and content management systems are involved. Alas, delivering XHTML documents with the correct content type is rarely practical at this time.

Luckily, XHTML documents can optionally be served with a content type of `text/html`, just as other versions of HTML are. This effectively means XHTML is treated as if it were HTML 4.01, sacrificing some of the power of XML for the sake of wider compatibility with web browsers. You still gain some benefits from using XHTML, ensuring your documents are well formed and forward compatible, but for all intents and purposes you're simply writing HTML 4.01 with a few tighter constraints.

And the Rest . . .

The rest of the document consists of the `head` and `body` elements; the `head` element contains information about the document itself (including the required `title` element), while the `body` element contains all the content that will ultimately be rendered by a browser, to be seen and used by your visitors. These elements are covered in detail in the next two chapters (in fact, Chapter 3 is devoted entirely to the `head` element).

All in all, the basic structure of an XHTML document is quite simple, requiring only a doctype, a root element, a head with a title, and a body.

The Document Tree

It's helpful to visualize the structure of an XHTML document as an inverted tree, with all the elements represented as connected branches. The tree begins with the root element at the top and all other elements descending downward, making it more like a family tree than the leafy, wooden sort. Because of this, genealogy terms are often used to refer to the relationships between elements. Figure 2-2 shows the family tree of a simple document.

In the diagram, the tree begins with the root element, which has two *child* elements: the `head` and the `body`. That `body` element has two children of its own: a level-one heading (the `h1` element, covered in Chapter 4) and a `p` element for a single paragraph (also covered in Chapter 4). Those two elements are *siblings* of each other, sharing the `body` element as their common *parent*. They're also *descendants* of the `html` element, which is their *ancestor*. The paragraph contains an `em` element and an `a` element, sibling children of their parent paragraph, descended from the ancestral `body` and `html` elements.

We'll use these terms—children, siblings, parents, descendants, and ancestors—often throughout this book.

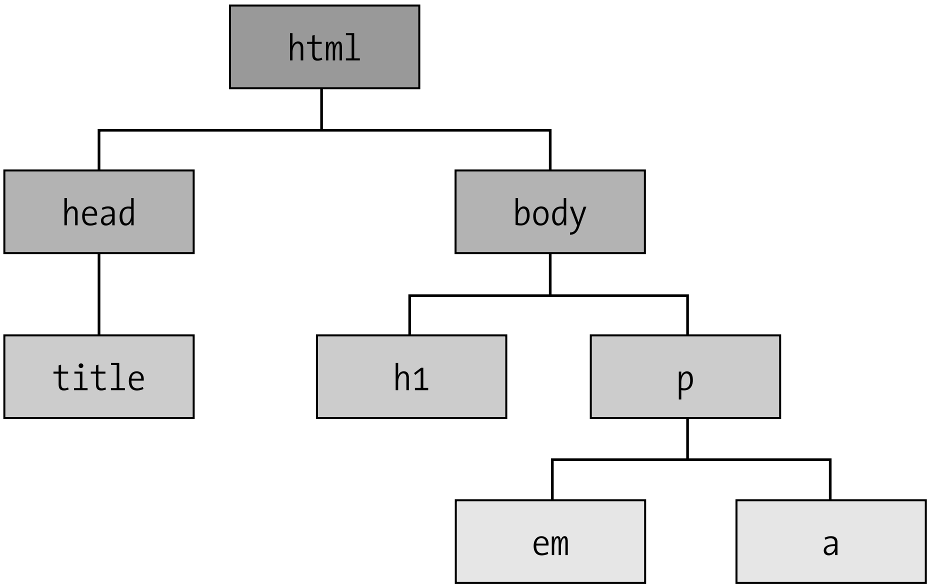

Figure 2-2. *A simple document tree*

CSS Fundamentals

CSS can add style to your pages, enhancing and improving the presentation of your content. The structure is supplied by XHTML—each element designates a different portion of content, and attributes pass along more information about those elements. CSS acts as another layer to influence the presentation of those XHTML elements when they're rendered. Colors, fonts, text sizes, backgrounds, and the arrangement of elements on the page are all presentational aspects of your content, and all can be controlled through artful application of CSS.

Anatomy of a CSS Rule

If elements are the building blocks of markup, the building block of CSS is the *rule*. It's a set of instructions that a browser can follow to alter the appearance of XHTML elements based on the presentational values you supply. A CSS rule consists of a few component parts, diagrammed in Figure 2-3.

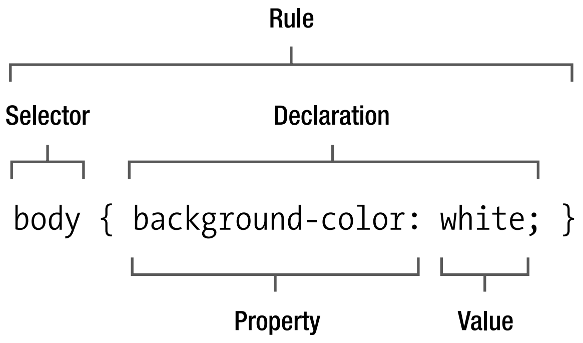

Figure 2-3. *The components of a rule in CSS*

The *selector* is the part of the rule that targets an element that will be styled. Its scope can be very broad, affecting every instance of a particular element or very narrow and specific, affecting only a few elements or even just one. We'll cover the different kinds of selectors in the next section of this chapter.

A *declaration* comprises two more parts: a *property* and a *value*. The property is that aspect of an element's presentation that is being modified, such as its color, its width, or its placement on the page. Dozens of properties are available in the CSS language, and you'll become familiar with many of them in the pages of this book.

The property value delivers the specific style that should be applied to the selected element. The values accepted depend on the particular property, and some properties accept multiple values, separated by spaces.

Declarations reside in a set of curly braces (`{` and `}`), and multiple declarations can apply to the same selector, thus modifying several aspects of an element's presentation in the course of a single rule. A property and its value are separated by a colon (`:`) and the declaration ends with a semicolon (`;`). That semicolon is important to separate multiple declarations, but if there's only one declaration in the rule or if it's the last declaration in a series, the terminating semicolon is optional. It's not a bad idea to get in the habit of including a semicolon at the end of every declaration, even when there's only one, just to play it safe.

If your CSS doesn't conform to this basic structure and syntax—if you forget the closing brace or the colon separating a property from its value, for example—the entire rule or even the entire style sheet might fail. Just like XHTML, a style sheet should be well formed and properly constructed. The W3C provides a CSS validation service (`http://jigsaw.w3.org/css-validator/`) that can help you catch goofs and glitches in your style sheets.

CSS Selectors

A selector, as its name implies, selects an element in your XHTML document. A few different types of selectors are available, with varying levels of specificity to target a large number of elements or just a few. *Specificity* is a means of measuring a given selector's scope, in other words how many or few elements it selects. CSS is designed so that more specific selectors override and supersede less specific selectors. Specificity is one of the more nebulous and hard-to-grasp concepts in CSS but is also one of the most powerful features of the language. We'll cover the rules of specificity in more detail later, but let's first introduce the selectors.

Universal Selector

The universal selector is merely an asterisk (*) acting as a "wild card" to select any and all elements in the document. For example, this rule:

```
* { color: blue; }
```

would apply a blue foreground (text) color to *all* elements. Headings, paragraphs, lists, cells in tables, and even links—all would be rendered in blue because the universal selector selects the entire universe. This is the least specific selector available, since it's not specific at all.

Element Selector

An element selector selects all instances of an element, specified by its tag name. This selector is more specific than the universal selector, but it's still not very specific since it targets every occurrence of an element, no matter how many of them there may be. For example, the rule:

```
em { color: red; }
```

gives every `em` element the same red foreground color, even if there are thousands of them in a document. Element selectors are also known as *type selectors*.

Class Selector

A class selector targets any element that bears the given class name in its `class` attribute. Because a `class` attribute can be assigned to practically any element in XHTML, and any number of elements can belong to the same class, this selector is not extremely specific but is still more specific than an element selector. In CSS, class selectors are preceded by a dot (`.`) to identify them. For example, this rule will style any elements belonging to the "info" class, whatever those elements happen to be:

```
.info { color: purple; }
```

ID Selector

An ID selector will select only the element carrying the specified identifier. Practically any element can have an `id` attribute, but that attribute's value may be used only once within a single document. The ID selector targets just one element per page, making it much more specific than a class selector that might target many. ID selectors are preceded by an octothorpe (`#`). (This is often called a *hash*, *number sign*, or *pound*, but *octothorpe* is the character's proper name. It also sounds cool and will impress people at dinner parties.) The following rule would give the element with the ID "introduction" a green foreground color:

```
#introduction { color: green; }
```

Pseudo Class Selector

A pseudo class is somewhat akin to a class selector (and is equal to classes in specificity), but it selects an element in a particular state. It's preceded by a colon (`:`), and only a few pseudo classes are available:

```
:link { color: blue; }
:visited { color: purple; }
:active { color: red; }
:hover { color: green; }
:focus { color: orange; }
```

The `:link` pseudo class selects all elements that are hyperlinks (which you'll learn much more about in Chapter 6). The `:visited` pseudo class selects hyperlinks whose destination has been previously visited (recorded in a web browser's built-in history). The `:active` pseudo class selects links in an active state, during that interval while they're being activated (while clicking a mouse or pressing the Enter or Return key). The `:hover` pseudo class selects any element that is being "hovered" over by a user's pointing device. Although any element can be in a hover state, this most commonly applies to links (though some older browsers supported this pseudo class only for links and no other elements). The `:focus` pseudo class selects any element in a focused state. Some browsers don't support `:focus`, most notably Internet Explorer 6 for Windows. However, Internet Explorer does (incorrectly) treat the `:active` pseudo class as if it were `:focus`, but only for links and not any other elements.

Descendant Selector

One of the most useful and powerful selectors in the CSS arsenal, a descendant selector can be assembled from two or more of the basic selector types (universal, element, class, pseudo class, and ID), separated by spaces, to select elements matching that position in the document tree. These are also called *contextual selectors* because they target elements based on their context in the document. For example:

```
#introduction em { color: yellow; }
```

That rule will color any `em` element within the element with the `id` value `introduction` yellow. Descendant selectors allow for very precise selection of just the elements you want to target, based on the structure of your XHTML document. This more elaborate example:

```
#introduction .info p * { color: pink; }
```

would select all elements that are descendants of a `p` element that is a descendant of an element with the class `info` that is a descendant of the element with the ID `introduction`. You can see how the scope of a descendent selector can be very narrow indeed, targeting only a few elements that meet the selector's criteria.

Combining Selectors

You can combine two or more selector types, such as an element and an ID or an ID and a class. These combinations can also narrow down the specificity of your selectors, seeking out only the elements you want to style and leaving others alone. This rule:

```
p.info { color: blue; }
```

selects only paragraphs (`p` elements) belonging to the `info` class. Another element in that class would be overlooked, and other paragraphs *not* belonging to the `info` class are also left untouched.

Combining selectors within a descendant selector can target elements with surgical precision:

```
p#introduction a.info:hover { color: silver; }
```

This rule would apply only to hovered links (`a` elements) belonging to the `info` class that are descendants of the paragraph with the ID `introduction`.

Grouping Selectors

You can group several selectors together as part of a single rule so the same set of declarations can apply to numerous elements without redundantly repeating them. A comma separates each selector in the rule:

```
p, h1, h2 { color: blue; }
```

The previous rule applies the same color value to every instance of the `p`, `h1`, and `h2` elements. The more complex set of selectors in this rule:

```
p#introduction em, a.info:hover, h2.info { color: gold; }
```

will target all `em` elements descended from the paragraph with the ID `introduction` and all hovered links with the class `info` as well as `h2` elements (a second-level heading) in the `info` class (remember that different types of elements can belong to the same class).

Grouping and combining selectors is a great way to keep your style sheets compact and manageable.

Advanced Selectors

The selectors you've seen so far are all part of CSS 1, the first standardized version of CSS introduced way back in 1996. This version of CSS is very well supported in today's generation of graphical web browsers, so you can use all of these selectors with fair confidence that most of your visitors will see their intended effect.

Since CSS 1, newer versions have come about, including CSS 2.1 and CSS 3. These updates to the CSS specifications have introduced a number of new and exciting selectors:

- *Attribute selectors* target an element bearing a particular attribute and even an attribute with a specified value.
- *Pseudo element selectors* target elements that don't specifically exist in the markup but are implied by its structure, such as the first line of a paragraph or the element immediately before another element.
- *Child selectors* select an element that is an immediate child of another element and not its other descendants.
- *Adjacent sibling selectors* target elements that are immediate siblings of another element, sharing the same parent in the document.

Unfortunately, CSS 2.1 and CSS 3 haven't yet been released by the W3C as official recommendations, though you can see them in their draft status at the W3C website to learn about these selectors and how they work (`http://www.w3.org/Style/CSS/`). These advanced selectors are already supported by many of the latest graphical browsers, but not all of them (and even some modern browsers don't support all of these selectors). Such advanced CSS features should be used with care combined with intensive cross-browser testing. For the purposes of this book, we'll stick with the CSS 1 selectors we've covered here, and they're all you'll need for most of what you may want to accomplish.

Specificity and the Cascade

As we mentioned earlier, each type of selector is assigned a certain level of specificity, measuring how many possible XHTML elements that selector might influence. Examine these two CSS rules, one with an element selector and the other with a class selector:

```
h2 { color: red; }
.title { color: blue; }
```

and this snippet of XHTML, an `h2` element classified as a title:

```
<h2 class="title">Specificity and the Cascade</h2>
```

The first rule selects all h2 elements, and the second rule selects all elements belonging to the title class. But the element shown fits *both* criteria, causing a conflict between the two CSS rules. A graphical browser must choose one of the two rules to follow to determine the heading's final color. In CSS, a more specific selector trumps a less specific selector. Because a class selector is more specific than an element selector, the second rule has greater specificity, and the heading is rendered in blue.

Modern web browsers follow a complex formula to calculate a selector's specificity, which can be rather confusing to noncomputers like us. Thankfully, you'll rarely need to calculate a selector's numeric specificity value if you just remember these few rules:

- A universal selector isn't specific at all.
- An element selector is more specific than a universal selector.
- A class or pseudo class selector is more specific than an element selector.
- An ID selector is more specific than a class or pseudo class.
- Properties in an inline style attribute are most specific of all.

Specificity is also cumulative in combined and descendant selectors. Each of the base selector types carries a different weight in terms of specificity—a selector with two classes is more specific than a selector with one class, a selector with one ID is more specific than a selector with two classes, and so on. The specificity algorithm is carefully designed so that a large number of less specific selectors can never outweigh a more specific selector. No number of element selectors can ever be more specific than a single class, and no number of classes can ever be more specific than a single ID. Even if you assembled a complex selector made up of hundreds of element selectors, another rule with just one ID selector would still override it.

Understanding specificity will allow you to construct CSS rules that target elements with pinpoint accuracy. For a more in-depth explanation of how specificity is calculated by web browsers, see the W3C specification for CSS 2.1 (http://www.w3.org/TR/CSS21/cascade.html#specificity) along with Molly Holzschlag's more approachable clarification at http://www.molly.com/2005/10/06/css2-and-css21-specificity-clarified/.

At this point you might be wondering what happens when two selectors target the same element and also have the same specificity. For example:

```
.info h2 { color: purple; }
h2.title { color: orange; }
```

If an h2 element belonging to the title class is a descendant of another element in the info class, both of these rules should apply to that h2. How can the browser decide which rule to obey? Enter the cascade, the *C* in CSS.

Assuming selectors of equal specificity, style declarations are applied in the order in which they are received, so later declarations override prior ones. This is true whether the declarations occur within the same rule, in a separate rule later in the same style sheet, or in a separate style sheet that is downloaded after a prior one. It's this aspect of CSS that gives the language its name: multiple style sheets that *cascade* over each other, adding up to the final presentation in the browser. In the earlier example, the `h2` element would be rendered in orange because the second rule overrides the first.

For another example, the following rule:

```
p { color: black; color: green; }
```

contains two declarations, but paragraphs will be rendered in green because that declaration comes later in the cascade order.

The sometimes-complex interplay between specificity and the cascade can make CSS challenging to work with in the beginning, but once you understand the basic rules, it all becomes second nature. You'll learn more about the cascade order later in this chapter, but first we'll explain how you can attach style sheets to your XHTML documents.

Attaching Style Sheets to Your Documents

To style your pages with CSS, you'll also need to connect your style sheets to your documents. When a graphical browser downloads the XHTML document and parses it for rendering, it will automatically seek out CSS rules to instruct it on how the various elements should be presented. You can include style sheets with your documents in a few ways, each with its own benefits and some drawbacks.

Inline Styles

You can include CSS declarations within the optional `style` attribute of each element in your markup. Inline styles aren't constructed as rules, and there is no selector because the properties and values are attached directly to the element at hand, as in Listing 2-2. An inline style is the most specific of all because it applies to exactly *one* element and no others.

Listing 2-2. *An Example of Inline Styles*

```
<h2 style="color: red;">Good eats for hungry geeks</h2>

<p style="color: gray;">Our fresh pizzas, hearty pasta dishes, and
succulent desserts are sure to please. And don't forget about our
daily chalkboard specials!</p>
```

However, you should avoid using inline styles. They mix presentation with your structural markup, thus negating one of the primary advantages of using CSS. They're also highly redundant, forcing you to declare the same style properties again and again to maintain consistent presentation. Should you ever want to update the site in the future—changing all your headings from red to blue, for example—you would need to track down every single heading in every single document to implement that change, a daunting task on a large and complex website.

Still, an inline style might be an efficient approach on a few rare occasions, but those occasions are very few and far between, and another solution is always preferable; inline styles should be a last resort only when no other options are available.

Embedded Style Sheets

You can embed style rules within the head element of your document, and those rules will be honored only for the document in which they reside. An embedded style sheet (sometimes called an *internal style sheet*) is contained within the style element, shown in Listing 2-3 and covered in greater detail in Chapter 3.

Listing 2-3. *An Example of an Embedded Style Sheet*

```
<!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>Spaghetti and Cruft : Our Menu</title>
    <style type="text/css">
      h2 { color: red; }
      p { color: gray; }
    </style>
  </head>
  <body>
    <h2>Good eats for hungry geeks</h2>

    <p>Our fresh pizzas, hearty pasta dishes, and succulent
    desserts are sure to please. And don't forget about our
    daily chalkboard specials!</p>
  </body>
</html>
```

Embedding a style sheet in the head of your document does further separate presentation from your structured content, and those rules will be applied throughout that document, but it isn't an efficient approach if you're styling more than one page at a time. Other documents within the same website would require embedded style sheets of their own, so making any future modifications to your site's presentation would require updating every single document in the site.

External Style Sheets

The third and best option is to place all your CSS rules in a separate, external style sheet, directly connected to your documents. An external style sheet is a plain-text file that you can edit using the same text editing software you use to create your XHTML documents, saved with the file extension `.css`. This approach completely separates presentation from content and structure—they're not even stored in the same file. A single external style sheet can be linked from and associated with any number of XHTML documents, allowing your entire website's visual design to be controlled from one central file. Changes to that file will propagate globally to every page that connects to it. It's by far the most flexible and maintainable way to design your sites, exercising the true power of CSS.

An XHTML document links to an external style sheet via a `link` element in the document's `head`, and you'll learn more about that in the next chapter. For now, Listing 2-4 shows a simple example.

Listing 2-4. *Linking to an External Style Sheet*

```
<!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>Spaghetti and Cruft : Our Menu</title>
    <link rel="stylesheet" type="text/css" href="styles.css" />
</head>
  <body>
    <h2>Good eats for hungry geeks</h2>

    <p>Our fresh pizzas, hearty pasta dishes, and succulent
    desserts are sure to please. And don't forget about our
    daily chalkboard specials!</p>
  </body>
</html>
```

When a graphical browser downloads and begins processing the document, it will follow that link to retrieve the external style sheet and process it as well, automatically following its rules to render the page. An external style sheet is downloaded only once and then *cached* in the browser's memory for use on subsequent pages, keeping your documents lighter and improving the speed of your entire website.

The Cascade Order

You're not limited to a single style sheet; several different CSS files can be linked to from one document, with each style sheet having its own `link` element in the document's `head`. Depending on the complexity of your site, you might have one style sheet containing general rules for the entire site while pages within a certain section can link to a second style sheet to define specific styles for that subset of pages. You might also prefer to break your styles apart based on their purpose: for example, one style sheet defining colors and backgrounds and another style sheet defining your page layout.

You can also combine all three methods—inline, embedded, and external—to style your web pages, although it's rarely advisable. If just one page on your site needs some additional rules, you might choose to include an embedded style sheet within that document alone. You may even, very rarely, want to call out one element for special treatment and use an inline style for just that element. In almost every case, external style sheets are the best approach: they eliminate presentational markup, improve a site's performance, and are much easier to maintain.

With so many CSS rules being dictated from so many different sources, some overlap is to be expected. You already have specificity on your side, with more specific selectors overruling general selectors. But specificity alone isn't enough to resolve all the potential style conflicts a graphical browser might run into when trying to render a web page. Where specificity fails, the *cascade order* steps in to sort things out.

CSS rules are applied to content in the order in which they are received; later rules override previous rules. Separate style sheets are downloaded in a particular order as well. In the case of external style sheets, their order is indicated by the order in which the `link` elements appear in the document; rules in later linked style sheets override rules in previously linked style sheets. Rules embedded in a document's `style` element are processed after all external style sheets. If more than one style sheet is embedded in a document—each in its own `style` element—later embedded style sheets override previous ones. Inline declarations in an element's `style` attribute are applied even after embedded style sheets.

In addition to *author style sheets*, every modern graphical web browser has its own built-in style sheet to define the default presentation of various elements. When you view a web page without any of the author's CSS applied, you're simply seeing it rendered with the *browser style sheet*, which comes first in the cascade order so all the author's styles override those defaults. To complicate matters just a bit further, most web browsers allow the end user to attach their own customized style sheets—known as a *user style sheet*—which comes second in the cascade order, thus overriding the browser's default styles but not the author's.

To break it down, the cascade order for multiple style sources is as follows:

1. Browser style sheet
2. User style sheet
3. External author style sheets (in the order in which they're linked)
4. Embedded author style sheets (in the order in which they occur)
5. Inline author styles

And don't forget, the cascade works within each style sheet as well. To remember how the cascade works, follow this rule of thumb: *the style closest to the content wins*. Whichever value is declared last will be the one applied when the content is rendered.

!important

In some extremely rare cases where both specificity and the cascade may not be sufficient to apply your desired value, the special keyword `!important` (complete with preceding exclamation point) can force a browser to honor that value above all others. This is a powerful and dangerous tool and should be used only as a last resort to resolve conflicting styles beyond your control (for example, if you're forced to work with third-party markup that uses inline styles that you're unable to modify directly).

The `!important` directive must appear at the end of the value, before the semicolon, like so:

```
h1 { color: red !important; }
```

A value declared as `!important` is applied to the rendered content regardless of where that value occurs in the cascade or the specificity of its selector. That is unless another competing value is also declared to be `!important`; specificity and the cascade once again take over in those cases. There's one notable exception to be aware of: `!important` values in a user style sheet always take precedence, even overriding `!important` values in author style sheets. This gives the ultimate power to the user, which is only right; after all, it's their computer.

Formatting CSS

Like XHTML documents, external style sheet files are plain text. You're free to format your CSS however you like, just as long as the basic syntax is followed. Extra spaces and carriage returns are ignored in CSS; the browser doesn't care what the plain text looks like, just that it's technically well formed. When it comes to formatting CSS, the most important factors are your own preferences. Individual rules can be written in two general formats: *extended* or *compacted*.

Extended rules break the selector and declarations onto separate lines, which many authors find more readable and easier to work with. It allows you to see at a glance where each new property begins and ends, at the expense of a lot of scrolling when you're working with long and complex style sheets. Listing 2-5 shows a few simple rules in an extended format.

Listing 2-5. *CSS Rules in Extended Format*

```
h1, h2, h3 {
  color: red;
  margin-bottom: .5em;
}

h1 {
  font-size: 150%;
}

h2 {
  font-size: 130%;
}

h3 {
  font-size: 120%;
  border-bottom: 1px solid gray;
}
```

Compact formatting condenses each rule to a single line, thus shortening the needed vertical scrolling, but it can demand horizontal scrolling in your text editor when a rule includes many declarations in a row. Listing 2-6 demonstrates the same set of rules compacted to single lines and with unnecessary spaces removed.

Listing 2-6. *CSS Rules in Compacted Format*

```
h1,h2,h3{color:red;margin-bottom:.5em;}
h1{font-size:150%;}
h2{font-size:130%;}
h3{font-size:120%;border-bottom:1px solid gray;}
```

Another advantage of compacted rules is a slight reduction in file size. Spaces, tabs, and carriage returns are stored as characters in the electronic file, and each additional character adds another byte to the overall file size that must be downloaded by a client. A long style sheet might be a considerably larger file in an extended format because of all the extra space characters. In fact, you could choose to remove *all* excess spaces and place your entire style sheet on a single line for optimal compression, but that might be overkill and make your CSS much harder to work with. To reconcile maximum readability with minimal file size, some authors work with style sheets in an extended format and then automatically compress the entire thing to a single line when moving it to a live web server.

A few extra spaces in a compacted rule can at least make it easier to scan, spreading a one-line rule out a bit by including spaces between declarations and values. For lack of a better term, we'll call this format *semicompacted*, as shown in Listing 2-7.

Listing 2-7. *CSS Rules in Semicompacted Format*

```
h1, h2, h3 { color: red; margin-bottom: .5em; }
h1 { font-size: 150%; }
h2 { font-size: 130%; }
h3 { font-size: 120%; border-bottom: 1px solid gray; }
```

In the end, the choice is entirely yours, and you should author your style sheets in a way that makes sense to you.

CSS Comments

You can add comments to your style sheets for the same reasons you might use comments in XHTML: to make notes, to pass along instructions to other web developers, or to temporarily hide or disable parts of the style sheet during testing. A comment in CSS begins with /* and ends with */, and anything between those markers won't be interpreted by the browser. Just like comments in XHTML, CSS comments can span multiple lines.

```
/* These base styles apply to all heading levels. */
h1, h2, h3, h4, h5, h6 { color: red; margin-bottom: .5em; }
/* Adjust the size of each. */
h1 { font-size: 150%; }
h2 { font-size: 130%; }
h3 { font-size: 120%; }
/* Temporarily hiding these rules
h4 { font-size: 100%; }
h5 { font-size: 90%; }
h6 { font-size: 80%; }
End hiding */
```

Summary

This chapter has covered a lot of ground to get you up to speed on the inner workings of XHTML and CSS. You've seen how you can author XHTML documents, using tags to define elements and adding attributes to relay more information about them. Throughout the rest of this book, you'll become intimately familiar with most of the elements you'll use when you create your own web pages.

HTML was first introduced in the early 1990s, but the language has already undergone many changes in its short and bright career. XHTML is a stricter reformulation of earlier versions of HTML, with just a few rules that differentiate the two, as shown in Table 2-1.

Table 2-1. *HTML 4.01 Strict vs. XHTML 1.0 Strict*

HTML 4.01 Strict	XHTML 1.0 Strict
Tag and attribute names are not case-sensitive.	Tag and attribute names must be written in lowercase.
Some attributes can be minimized, and attribute values don't require quotes.	All attributes must have a specified value, and the value must be quoted.
Some elements don't require closing tags, and empty elements should not be closed with a trailing slash.	All elements must be closed, either with a closing tag for nonempty elements or with a trailing slash for empty elements.

The second part of this chapter gave you a crash course in CSS, unveiling the mechanics of this rich and powerful language. You learned about CSS selectors and how specificity and the cascade work together to give you great control over how your content is presented. You'll use XHTML to build the structure of your documents and then use CSS to apply a separate layer of polished presentation. In the following chapters, you'll see glimpses of how you can use CSS in different ways to create different visual effects. Chapter 9 will delve a bit deeper to show you a few ways to use CSS to lay out your pages by placing elements where you want them to appear on-screen, all without damaging their underlying structure.

From here on, we'll assume you've reached an understanding of the basic rules of syntax for authoring your own XHTML and CSS, and the rest of this book will dig into the real meat of markup. To get things rolling, Chapter 3 is a detailed examination of the `head` element, where you'll include vital information about the documents you create.