

HTML Mastery: Semantics, Standards, and Styling

Paul Haine



HTML Mastery: Semantics, Standards, and Styling

Copyright © 2006 by Paul Haine

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-765-1

ISBN-10 (pbk): 1-59059-765-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor **Assistant Production Director**
Chris Mills Kari Brooks-Copony

Technical Reviewer **Production Editor**
Ian Lloyd Ellie Fountain

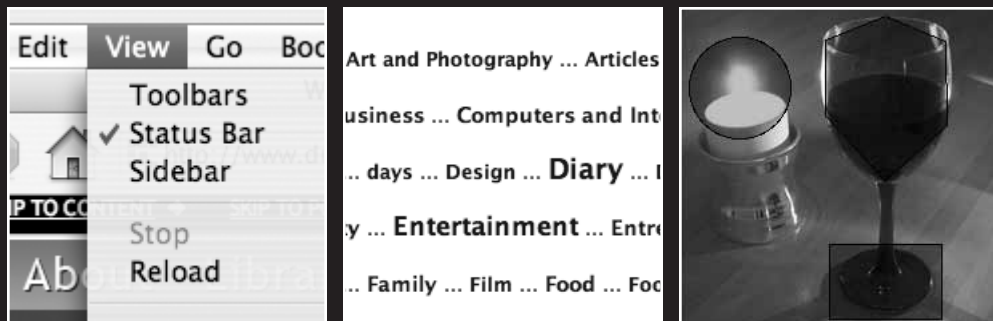
Editorial Board **Composer**
Steve Anglin, Ewan Buckingham, Gary Cornell, Jason
Lynn L'Heureux
Gilmore, Jonathan Gennick, Jonathan Hassell, James
Huddleston, Chris Mills, Matthew Moodie, Dominic
Proofreader
Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Linda Seifert

Project Manager **Indexer**
Elizabeth Seymour Julie Grady

Copy Edit Manager **Interior and Cover Designer**
Nicole Flores Kurt Krames

Copy Editors **Manufacturing Director**
Nicole Flores, Ami Knox Tom Debolski

2 USING THE RIGHT TAG FOR THE RIGHT JOB



When building websites, it's very easy to get by on only a few tags: a heading here and there, some paragraphs and lists, and a sprinkling of `` and `` and a few `div` and `span` to add some body to the `<body>`. However, this approach ignores the many other tags available that can allow you to enhance your pages with scripts and styles without needing to clog up the works with classes and semantically meaningless `` tags. In this chapter, we'll examine a number of these additional tags and how to use them.

I've divided this chapter into four loosely related sections:

- **Document markup:** This catchall section covers paragraphs, headers, lists, links, addresses, deletions and insertions, and quotes.
- **Presentational elements:** This section covers elements that do not have any semantic meaning, such as `<i>` and `<tt>`.
- **Phrase elements:** In this section, we'll look at inline elements that convey semantic meaning, such as `<cite>`, `<kbd>`, and `<acronym>`.
- **Images and other media:** The `` and `<object>` tags, image maps, CSS background images, and embedded media are examined here.

Throughout the chapter you'll find examples of how you can take advantage of these semantics and structures to add functionality and styling to your web pages using unobtrusive DOM Scripting and CSS.

Due to their complexity, tables and forms (and all related markup) are covered in their own chapters later in the book.

Document markup

First, let's look at the general category of document markup elements, including paragraphs, line breaks, and headings; how to display contact information, quotes, lists, and links; and how to mark up changes to your documents.

Paragraphs, line breaks, and headings

Perhaps the markup you've used most often when writing web pages is `<p>`. There isn't much to be said about `<p>`: it is simply used to mark up a paragraph. Yet this humble element is often abused by WYSIWYG software as a quick and dirty spacer. You have likely seen markup such as the following before, where an author has pressed the Enter key a few times:

```
<p>&nbsp;</p>  
<p>&nbsp;</p>  
<p>&nbsp;</p>  
<p>&nbsp;</p>
```

This is a prime example of (X)HTML being co-opted into acting in a presentational manner. We find here multiple, pointless paragraphs, with a nonbreaking space entity inside due to some browsers not displaying empty elements, but the effect should really be achieved with CSS. A quick way of adding some space beneath your content is to enclose the content in a `<div>`, like this:

```
<div id="maincontent">
  <p>Your content here.</p>
</div>
```

2

Then add some padding to the bottom of the `#maincontent` section with CSS:

```
#maincontent { padding-bottom: 3em; }
```

I use `em` as a unit of measurement here rather than `px`, so that the spacing beneath the paragraphs of content will scale appropriately when users change the text size in their browsers.

Similarly, the `
` tag for line breaks is often used to add a few lines of space here and there when it should be used simply to insert a single carriage return (e.g., when formatting a poem or code samples—but then perhaps you should be using `<pre>`, which is discussed further in the section “The `<hr>`, `<pre>`, `<sup>`, and `<sub>` elements”).

Heading tags are used to denote different sections of your web page or document, and they allow various user agents (such as screen reading software or some web browsers such as Opera) to easily jump between those sections. There are six heading levels, `<h1>` through to `<h6>`, with `<h1>` being considered the most important heading and `<h6>` the least important (see Figure 2-1). Having six heading levels available to you means that you should never need to write `<div id="heading">` or `<p>heading</p>`.

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

Figure 2-1.
The six heading levels,
in all their unstyled glory

So, with all these exciting different headings to choose from, where should you start and what should you consider the “most important” heading? This largely depends on the content of your site. If you are building a blog or a similar article-based website, then the title of each blog entry or article could be your `<h1>`. Alternatively, you may decide that your website brand (e.g., eBay, Amazon.com, etc.) should be considered the most important heading, and set that to be your `<h1>`.

It’s possible to automatically generate a rudimentary table of contents (TOC) for a single document by using JavaScript to pull out the headers in a page, as described by JavaScript guru Peter-Paul Koch at www.quirksmode.org/dom/toc.html. Although this rudimentary TOC lacks the permanence and scope of a TOC for a collection of documents, it does have the advantage of dynamically updating when you insert or remove a header.

Contact information

The `<address>` tag is used for marking up contact information—nothing more, nothing less. The HTML specification is very specific about the `<address>` tag’s use: it is not for displaying contact information for just anybody or anything; rather, it is used to display contact information related to the document itself. For example, say you are maintaining a list of contact details for a society membership directory. In this case, use of the `<address>` tag—once per each member—would not be appropriate. However, if you also had on that page the webmaster’s contact details, or the contact details of whoever maintains the directory, then the `<address>` tag *would* be appropriate to use for just that person. It doesn’t have to specifically describe a physical, brick-and-mortar location—the example in the HTML specification, for instance, contains links to personal pages and a date:

```
<address>
  <a href="../People/Raggett/">Dave Raggett</a>,
  <a href="../People/Arnaud/">Arnaud Le Hors</a>,
  contact persons for the <a href="Activity.html">
W3C HTML Activity</a><br />
  Date: 1999/12/24 23:37:50
</address>
```

The `<address>` element is a block-level element, and as such it can contain only inline elements—no paragraphs, lists, or divs. This element is also of use when using the hCard microformat, which I discuss in Chapter 5.

Quotes

HTML provides you, the web author, with two distinct ways of marking up quotations, one for block-level quotes and the other for inline quotes. However, I only recommend the use of the former method: the `<blockquote>` element.

Block quotes

`<blockquote>` is another tag that has historically been used for its presentational effect rather than its semantic meaning or structural relevance. Indeed, this misuse is even referenced in the HTML specification:¹

“We recommend that style sheet implementations provide a mechanism for inserting quotation marks before and after a quotation delimited by BLOCKQUOTE in a manner appropriate to the current language context and the degree of nesting of quotations.

*“However, as some authors have used BLOCKQUOTE merely as a mechanism to indent text, in order to preserve the intention of the authors, user agents should **not** insert quotation marks in the default style.*

“The usage of BLOCKQUOTE to indent text is deprecated in favor of style sheets.”

As you can see, the `<blockquote>` tag has been so commonly used to indent text instead of acting as a container for quotes that the W3C recommends against user agents automatically including quote marks and has taken the unusual step of actually *deprecating* this misuse. How this can be enforced I can only guess.

The `<blockquote>` tag also allows for a `cite` attribute to allow the author to reference the source of the quote (usually in the form of a URL, but it can be any form of citation, such as the name of another author or the title of a movie). User agent support for utilizing this hidden information is currently poor. While Firefox will allow you to right-click a `<blockquote>` element, select Properties, and see the citation information in a pop-up alert box (see Figure 2-2), most other user agents simply ignore it.

This is a blockquote with *joelblade.com* as the cite attribute value.



Figure 2-2. Firefox displays the `cite` and `title` attributes of a blockquote. It’s not an intuitive method of displaying the information, but at least it’s there.

1. See www.w3.org/TR/html4/struct/text.html#h-9.2.2.

We can retrieve the citation information and display it on the client side in at least two ways. The first is with CSS, and the second is with some DOM Scripting.

Here's a CSS way:

```
blockquote[cite]:after {
    content: "Source: " attr(cite);
}
```

This is an example of an **attribute selector**, part of the CSS2 specification. It means that if a `<blockquote>` element with a `cite` attribute is found, the value of that attribute should be displayed after the contents of the quote. There are some fairly severe downsides to this method, though. First, it will work only in modern browsers. It won't work in Internet Explorer 6 or below, and while Internet Explorer 7 *will* support attribute selectors, it *won't* support the `:after` and `:before` pseudo-elements or the `content` property, so again this method will fail. Second, Firefox and other Mozilla browsers will not allow you to select, click on, or otherwise engage with the content (see Figure 2-3). As far as these browsers are concerned, this generated content is much like the bullet character in an unordered list—it doesn't exist in any meaningful fashion. This may not be a problem if your citation is a name, but if it's a URL, then ideally it ought to be clickable or at the very least selectable.

**This is a blockquote with *joelblade.com* as the cite attribute value.
Source: <http://joelblade.com>**

Figure 2-3. Mozilla browsers can display the source with CSS, but you can't click or select the link.

The second method uses DOM Scripting, which has the advantage of working in a wider range of browsers, but the disadvantage of only working where JavaScript is present. So, for example, someone reading the text in a newsfeed (via RSS, Atom, etc.) might not see the JavaScript-generated content. (But it will degrade gracefully, at least; users without JavaScript will just see the quote.) The script to achieve this could look something like this:

```
function betterBlockquotes()
{
    if (document.getElementsByTagName) {
        quotes = document.getElementsByTagName("blockquote");
        for(i=0; i<quotes.length; i++)
        {
            citation = quotes[i].getAttribute('cite');
            citelink = document.createElement('a');
            citelink.setAttribute('href', citation);
            citelink.appendChild(document.createTextNode('Source: '));
            p = document.createElement('p');
            p.appendChild(citelink);
            quotes[i].appendChild(p)
        }
    }
}
window.onload = betterBlockquotes;
```


The preceding script is fairly minimal. It could be enhanced by pulling out the title attribute value of the block quote and using that for the link text, or it could include class attributes on the generated elements to allow for specific styling.

This method loops through a document, picking out all the blockquotes, and then it sifts through them to find those with `cite` attributes. When it finds a blockquote with a `cite` attribute, the script creates a new `<a>` element and drops the `cite` attribute value into the `a`'s `href` attribute value. It creates the required `'Source: '` text fragment (it's generated by the script, as we don't want that showing up if there isn't any citation), and then finally creates a new `<p>` element and inserts it into the document just after the blockquote itself. The result looks exactly the same as the preceding CSS example, and despite not appearing in the original (X)HTML markup, it can still be styled as desired (see Figure 2-4).

2

This is a blockquote with *joelblade.com* as the cite attribute value.

Source

Figure 2-4. Using JavaScript allows us to create a clickable link.

Inline quotes

While the `<blockquote>` tag should be used for block-level quotes, there also exists a `<q>` tag for inline quotes. I'll warn you now: I'm going to conclude by recommending you *don't* use the `<q>` tag, so if you want to save yourself a bit of time, you can skip the next few paragraphs and pick up again where I start discussing lists. If, however, you like to know the details anyway, read on.

What is *supposed* to happen when you use the `<q>` tag is that the browser automatically includes typographically correct quote marks at the beginning and end of the quote, meaning that you, the author, should *not* include them. Furthermore, with judicious use of the `lang` attribute, those quotes should be displayed in the style appropriate to the specified language (e.g., some European languages will use chevrons or *guillemets*: « and » instead of " and "). Also, browsers should display an awareness of nested quotes (in English, if a quote begins with the double-quote character, then quotes within that quote should use the single-quote character and vice versa). So, for example, the following fragment of HTML:

```
<p><q>This is a quote that has a <q>nested quote</q>
as part of it.</q></p>
```

should display as follows:

“This is a quote that has a ‘nested quote’ as part of it.”

While this may all sound very exciting, the reality is more humdrum. Not only is modern browser support inconsistent—for instance, Firefox (and other Mozilla browsers) will correctly nest double and single quotes, while Safari will not—but Internet Explorer does not generate any quotes at all, which makes things even more problematic.² Do you add the quotes yourself and end up with two sets of quotes in modern browsers? Do you leave the quotes out and allow modern browsers to get on with things, but ignore the most popular browser in the world? The question of what to do about Internet Explorer’s poor standards support is a question that you will face often. In the case of `<q>`, I suggest simply not using it at all. The inconsistent support for `<q>` features has essentially rendered it useless—you can’t even use the CSS2 quotes property (as in `q {quotes: none}`) to remove generated quotes, because Safari doesn’t support that property, and although there are various other solutions involving JavaScript, CSS, or a combination of the two, they all fail in one situation or another.³ Urgh.

So, to summarize how best to use `<q>`: just don’t. Let’s move on.

Lists

Three list types are available in current (X)HTML versions: unordered lists ``, ordered lists ``, and definition lists `<dl>`.

Two other forms of lists, `<menu>` and `<dir>`, have been deprecated, and the W3C recommends using an unordered list in their place.

The differences between the list types are fairly minimal and straightforward:

- An **unordered list** should be used when your content is (as you would expect) not in any particular order.
- An **ordered list** should be used when your content *is* in an order of some kind: alphabetical, numerical, and so on.
- A **definition list** is designed for associating names or terms with values or other data—any items that have a direct relationship with one another, such as a glossary of terms. Though it is a *definition* list, the W3C also suggests using such a list to mark up dialogue,⁴ with each definition term as a speaker and the definition description as the spoken words, so we can consider allowed usage of this element to be fairly liberal.

2. I’m referring to the Windows version of Internet Explorer, by the way. Internet Explorer on the Mac deserves a special mention for being the only browser—even now—that gets almost all of the `<q>` display properties correct. It even changes the quote character depending on the value of the `lang` attribute. Say what you like about the browser, but it was ahead of its time.

3. See www.alistapart.com/articles/qtag.

4. See www.w3.org/TR/REC-html40/struct/lists.html#h-10.3.

Unordered and ordered lists

Unordered and ordered lists consist of an opening `` or ``, respectively, followed by any number of list item—``—elements, and then finally a closing `` or ``. The opening and closing tags can contain only list items, but list items can contain anything, including paragraphs, divs, headers, and yet more lists. So long as they're all contained within a single list item, it's perfectly valid and well-formed markup. As far as differences between (X)HTML versions go, in HTML you don't have to close list items with ``, but in XHTML you do. Figure 2-5 shows the default styling of both list types.

Sadly, lists do not have any interesting or obscure attributes that have not been deprecated in strict doctypes. The `start` and `value` attributes used to be available to us. `start` allowed authors to start the numbering of an ordered list at a number other than 1, which was useful if you needed to interrupt an ordered list, such as in the case of a list of search results split over several pages. `value` allowed authors to give a specific list item an out-of-sequence number. It may be possible to reproduce the functions of these attributes with CSS in the future using CSS counters,⁵ but as yet very few browsers will support this. If you need to use these attributes, consider using a Transitional doctype, otherwise your document(s) will fail validation tests. There also existed a `compact` attribute, intended to inform browsers that the list was a brief, compact list and should be rendered with that in consideration, but with no browser support, this attribute has also fallen by the wayside.

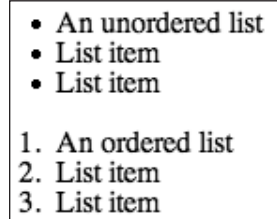
- 
- An unordered list
 - List item
 - List item
1. An ordered list
 2. List item
 3. List item

Figure 2-5. An unordered list and an ordered list with their default styling

Thankfully, though the markup may not be exciting, it is at least flexible, and with CSS you can display a list in a wide variety of ways: horizontally, vertically, expanding/collapsing, or as an image map (see the section “Image maps” later in the chapter). For instance, take the following navigation menu:

```
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/contact/">Contact</a></li>
  <li><a href="/about/">About</a></li>
  <li><a href="/archive/">Archive</a> </li>
</ul>
```

We can turn this into a horizontal menu very easily:

```
li {
  float: left;
}
```

We can achieve the same effect by turning the `` from a block into an inline container with use of `display: inline`; but a block-level container will offer us more styling options in the future. Floating the list items left will cause the list to look as shown in Figure 2-6.

5. See www.w3.org/TR/REC-CSS2/generate.html#counters.

- HomeContactAboutArchive

Figure 2-6. An unordered list with the list items floated left

Simple! But clearly quite ugly and a bit unusable at this stage, so let's tidy things up a little:

```
li {
  border: 1px solid;
  float: left;
  list-style-type: none;
  padding: 0.2em 1em;
}
```

By adding a border, removing the list bullet and adding a touch of padding, we get the list shown in Figure 2-7.

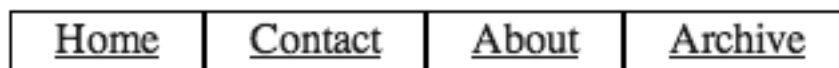


Figure 2-7. An unordered list with the list items floated left, bordered, and padded

As you can see, we already have a very serviceable horizontal menu. We could jazz this up even more with some styling of the anchor tags, giving them a background-color, using `display: block` to allow them to fill the whole list item area, changing their background with `:hover`, and so on.

Russ Weakley, a co-chair of the Web Standards Group, has created a huge collection of different list styles (more than 70 at the current count) available at <http://css.maxdesign.com.au>, and the article titled "CSS Design: Taming Lists" by Mark Newhouse at www.alistapart.com/articles/taminglists is also well worth a look. To help take some of the pain out of creating lists of links, it's also worth trying out Accessify's List-O-Matic (<http://accessify.com/tools-and-wizards/developer-tools/list-o-matic>), an online list-builder that lets you select from a variety of prebuilt styles.

So, already you can see that a simple list can be displayed in a different way from its default style. It's possible to use CSS to create some quite dynamic behavior with lists (though in most cases JavaScript is also required for compatibility with Internet Explorer). As documented by Eric Meyer (<http://meyerweb.com/eric/css/edge/menus/demo.html>), browsers that supported the `:hover` pseudo-class on any element (Firefox et al.) could use that to display nested lists as pop-out menus. The CSS to accomplish this is very simple:

```
li ul {display: none;}
li:hover > ul {display: block;}
```

This means that any `` that is a descendent element of an ``—that is, a nested list—should not be displayed. The second line says that any `` that is a child element of an

 that is being hovered over should display as normal. In compliant browsers, the end result looks like Figure 2-8.

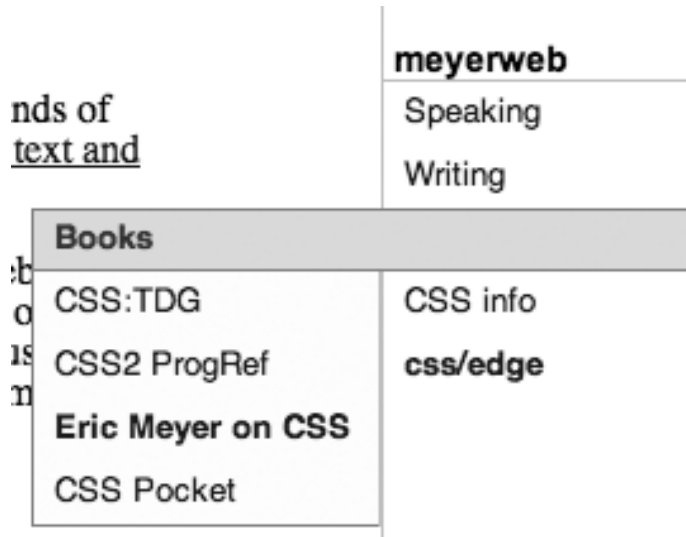


Figure 2-8. A pure CSS nested menu

All very neat. As mentioned, though, Internet Explorer 6 and below won't have a clue what to make of your `li:hover`,⁶ so JavaScript is required. Patrick Griffiths's Suckerfish Dropdowns script (www.htmldog.com/articles/suckerfish/dropdowns) provides both a CSS solution and a JavaScript solution that are pretty robust (multiple nested menus are catered for) and very easy to implement, requiring only the inclusion of a small script and the addition of a class selector to your CSS file.

The definition (is this)

The definition list consists of an opening <dl>, followed by a definition term (<dt>), and then any number of definition descriptions (<dd>). A typical definition list looks like this:

```
<dl>
  <dt>Bottle</dt>
  <dd>A receptacle having a narrow neck, usually no handles,
  and a mouth that can be plugged, corked, or capped.</dd>
  <dd>To hold in; restrain: "bottled up my emotions."</dd>
  <dt>Rocket</dt>
  <dd>A vehicle or device propelled by one or more rocket engines,
  especially such a vehicle designed to travel through space.</dd>
</dl>
```

6. Internet Explorer 7 will support `:hover` on all elements, so it is likely that these pure CSS menus will work without scripting.

Most browsers would display the preceding code in a similar way to that shown in Figure 2-9.

| | |
|---------------|--|
| Bottle | A receptacle having a narrow neck, usually no handles, and a mouth that can be plugged, corked, or capped. To hold in; restrain: <i>bottled up my emotions</i> . |
| Rocket | A vehicle or device propelled by one or more rocket engines, especially such a vehicle designed to travel through space. |

Figure 2-9. A definition list, with the definition terms on the left and the definition descriptions indented

Definition lists are, as noted, fairly flexible. As long as there is a direct relationship between the term and the definition(s), many constructs can be represented using this list. For instance, a photograph as the term could have descriptions including information about both the photographer and the camera. In addition, a definition list could be used to display a schedule for a series of presentations at a conference, with the title of the presentation as the definition term and the descriptions including details of the presenting author and the date and time. A definition list could also be used in an online shopping application to describe product details, and so on.

Although definition lists are flexible in use, you should bear the following caveat in mind: a definition term cannot contain any block-level elements—no paragraphs, headers, or lists—which means that terms cannot be given differing levels of importance in the same way that headings can (<h1>, <h2>, etc.). A definition description, however, can contain any element or series of elements, so long as they're well-formed.

Links

Links are most likely up there alongside paragraphs and headers as among the first pieces of HTML you ever learned, but we can still plumb the depths of obscurity and poke around at a few unused and potentially useful attributes. Strictly speaking, the <a> tag is not a link; it's an **anchor**, which can either link to a new file, point to a named anchor elsewhere (either on the same page or on a different page), or point to any element that has an `id` attribute. There's also the <link> tag, which is used solely in the head of a document. You may have used it already for linking style sheets to your pages, but it can also be used to provide extra navigational information, as you'll see shortly.

First, let's go through a quick refresher of the basics. An anchor tag linking to another document will use the `href` attribute, like so:

```
<a href="newpage.html">link</a>
```

To link to another anchor tag, you target the **fragment identifier**, which is set in the linked anchor tag via the `name` or `id` attribute. The linking is done like so:

```
<a href="newpage.html#parttwo">link</a>
```

If you want to link to an anchor tag or other identified element on the same page, there is no need to include the filename:

```
<a href="#parttwo">link</a>
```

Linking to an anchor or an identified element on the same page can have multiple uses. A common use is for a table of contents for a lengthy document with anchors scattered throughout, and sometimes a “back to the top of the page” link: `back to top`.⁷ Another common use is to create **skip links**, which are links that allow people to skip past long blocks of navigation links to get at the content. Skip links are usually included for users who navigate with a keyboard, or a mobile or screen-reading device, but sometimes also present visually as well for users who are zooming in and may not enjoy scrolling around. A typical skip link looks like this:

```
<!-- present near the top of the page -->
<a href="#maincontent">Skip Navigation</a>
<ul>
  <li><a href="/about/">About</a></li>
  <li><a href="/contact/">Contact</a></li>
  <li><a href="/help/">Help</a></li>
  and so on...
</ul>
<a name="maincontent" id="maincontent"></a>
<!-- beginning of your actual content -->
```

Use of these sorts of links can dramatically increase the usability of your website for certain groups of users, but there are several browser bugs and usability issues to consider.

A useful article that summarizes good use of skip links is Jim Thatcher’s “Skip Navigation” (www.jimthatcher.com/skipnav.htm).

An anchor tag that isn’t linking anywhere and is literally just an anchor point will include its fragment identifier as a value of either the `name` attribute or the `id` attribute, like so:

```
<h3><a name="parttwo">Part Two</a></h3>
```

It is valid to have an empty anchor tag of the form ``, but the HTML specification warns that some browsers may not recognize it.

So, what’s the difference between using the `name` and the `id` attribute? Well, the `name` attribute can only be applied to anchor tags for link-targeting purposes, whereas you can use the `id` attribute in any tag. So, if you give each of your heading tags a unique identifier, you can point your links directly to those instead of having to include an extra anchor. For instance, the preceding example can be simplified like this:

7. It’s a good idea to explicitly include an anchor or identified element with a `name/id` of `top`—some browsers will infer such location, but not all will do that.

```
<h3 id="parttwo">Part Two</h3>
```

If you *are* using an anchor, then bear in mind that if you give it an ID *and* a name, they must be identical. Furthermore, you cannot have an identical name and ID in separate elements on the same page.

You also have the option of using the hreflang attribute to specify the language of the resource designated by the href attribute, and the charset attribute to specify the character encoding, but it is unlikely that you will ever need to use these attributes.

Relationship issues

You can specify the relationship *type* of the link by using the rel attribute, and the reverse relationship with the rev attribute, both of which can be used in either <a> links or <link> links. The nature of these two attributes can be a little tricky to grasp, so let's consider an example. You may have encountered rel before when using the <link> tag to reference an RSS feed in the head of your web page, like this:

```
<link rel="alternate" type="application/rss+xml" ↪  
href="http://example.com/feed/" />
```

The preceding code means that “an alternative version of this document exists at http://example.com/feed/,” and user agents can spot that and find the RSS feed—most modern browsers will display a feed icon in the browser address bar, allowing the user to select, view, and subscribe to the feed. The rev attribute works the same way, but in reverse. So, using the preceding example, the document at the feed URL could have a rev attribute value of alternate, but this time it would mean “this document is an alternative version of the document located at http://example.com/.”

The rev and rel attributes can take any value, but the HTML specification lists several predefined types:

- **alternate:** As mentioned earlier, this value designates alternate versions of the document in which the link occurs. It can be used with the lang and hreflang attributes if the alternate version is a translation, and with the media attribute if the alternate version is designed for a different medium. For instance, if you were linking to a print style sheet, you would use the attribute/value media="print".⁸
- **stylesheet:** A rel attribute value of stylesheet informs the user agent that the linked document is a style sheet (rather obviously). You can use it in conjunction with alternate (as in rel="alternate stylesheet") to specify a range of alternative style sheets that the user agent can allow the user to select from (both Firefox and Opera have this functionality built in).

8. The media attribute has a number of valid values, but only a few are widely supported. The values available are screen, tty, tv, projection, handheld, print, Braille, aural, and all, which are aimed respectively at computer screens, terminals, televisions, projectors, handheld devices, printed pages, Braille tactile feedback devices, speech synthesizers, and all of the above. Of these, you are most likely to use screen and print, and possibly handheld (which has limited support among handheld browsers). Opera uses projection when in full-screen mode, so if you have specified a screen type for your main style sheet, you may wish to consider including projection: <link media="screen, projection" ... />.

- **start**: In a collection of documents, a `rel` attribute value of `start` indicates to user agents and search engines which document should be considered the starting point for the collection.
- **next**: This value indicates that the linked document is the next document in a collection of documents.
- **prev**: This value is similar to `next`, except it indicates the previous document rather than the next. If you're a completist, you could have links with both `rev/prev` and `rev/next` in them, but it would probably be enough to just have the `revs`.
- **contents**: This value refers to a document serving as a table of contents.
- **index**: This value refers to a document providing an index for the current document.
- **glossary**: This value refers to a document providing a glossary for the current document.
- **copyright**: This value refers to a document providing a copyright statement for the current document.
- **chapter**: This value refers to a document serving as a chapter in a collection of documents.
- **section**: This value refers to a document serving as a section in a chapter.
- **subsection**: This value refers to a document serving as a subsection in a section.
- **appendix**: This value refers to a document serving as an appendix in a collection of documents.
- **help**: This value refers to a help document that should relate to the website or web page—for instance, a collection of “further reading” links or an explanatory document listing FAQs.
- **bookmark**: This value refers to a bookmark (i.e., a starting point within a document or collection of documents).

These are just the link types predefined by the W3C; you can also define additional types of your own. For instance, you could label any links that point to external sources with a `rel="external"` attribute, and in his weblog Joe Clark advocates using the `rev` attribute to indicate that a zoom layout is available (<http://axxlog.wordpress.net/archives/2005/01/14/zoom-hack>).

*A **zoom layout** is a CSS layout specially formatted for low-vision users. It usually features large, light text on a dark background and a single-column design. See Joe Clark's article “Big, Stark & Chunky” at [www.alistapart.com/articles/lowvision](http://www alistapart.com/articles/lowvision) for more information.*

Some user agents already take advantage of these attributes. For example, Opera features a navigation bar that displays links with `rel` attributes in a fixed toolbar (see Figure 2-10) and will allow you to select alternate style sheets from a drop-down menu (as does Firefox).



Figure 2-10. Opera's rel-based navigation bar. The page here has a Contents link, as well as Previous and Next links.

Both Opera and Firefox rely on the alternate style sheet link to also have a title attribute, which is used to display the options to the user (see Figure 2-11). The attributes also provide web authors with extra hooks for CSS and JavaScript, as you will see later on.

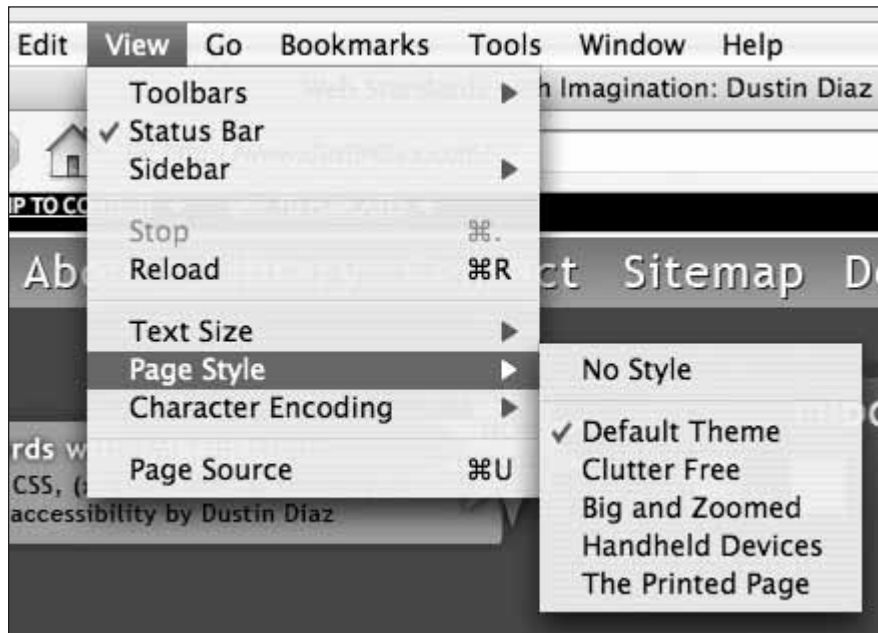


Figure 2-11. Changing styles in Firefox

Technorati (<http://technorati.com>), a weblog search and tracking engine, utilizes the rel attribute to help generate its tag-based navigation (see Figure 2-12).



Figure 2-12. A Technorati tag cloud, with the more popular tags appearing in a larger font

By adding `rel="tag"` to a link within a blog post, you are indicating that the resource indicated by the `href` of the link can be used as a general category page for the specific topic. For instance, if you are writing an article on Apple's iPod for your blog, you could include a link such as the following somewhere in the body of your article:

```
<a href="http://technorati.com/tag/ipod/" rel="tag">iPod</a>
```

When you do this, Technorati and other tag-aware services and aggregators can determine that your article belongs in the category of "ipod," and that the resource indicated can be considered as a collection of related articles. Your article can then be included in future collections of related articles.

We'll take a closer look at the use of `rel="tag"` in Chapter 5, so don't worry about it too much at this stage.

Targeting links

Something worth noting is that the `target` attribute, which is used on anchor tags to direct a link into either a new browser window or an alternative frame within the window, has been deprecated, so if you are writing strict XHTML, you must use JavaScript to replicate this behavior. This is where you can take advantage of the `rel` attribute: by using a simple bit of JavaScript, it is possible to pick up any links with a `rel` value of `external` and cause them to open in new windows.⁹ Here's a short script that does just that:

```
function popup()
{
  if (document.getElementsByTagName) {
    a = document.getElementsByTagName("a");
    for(i=0; i<a.length; i++)
    {
      if(a[i].getAttribute("rel") ==
&& a[i].getAttribute("rel") == "external")
      {
        a[i].onclick = function()
        {
          window.open(this.getAttribute('href'));
          return false;
        }
      }
    }
    else return false;
  }
  window.onload = popup;
```

9. If you want to style your external links in a different fashion from your regular links, then you would still need to add a class. Although you can use CSS attribute selectors to target links with a specific `rel` value (using `a[rel=external]`), these will not work in Internet Explorer.

What's happening here? Well, it's a JavaScript function that will collect all `<a>` tags in the (X)HTML document, and then loop through that collection picking out each `<a>` with a `rel` attribute value of `external`, set its `onClick` action to open in a new window, and finally instruct the user agent to not follow the link again in the existing window (with `return false`). This allows you to maintain a valid, XHTML Strict document without muddying up your markup with inline `onClick` attributes, and if you change your mind in the future, you only have to change (or remove) this script, instead of having to change every link.

The preceding script is by no means the only way to do achieve this effect. For a comprehensive, all-singing, all-dancing new window script, I suggest Roger Johnanson's script, available at www.456bereastreet.com/archive/200610/opening_new_windows_with_javascript_version_12/. It covers a wider range of situations and includes support for progamatically including a warning to users about the new window.

Also worth noting is the CSS `:target` pseudo-class, introduced as part of the as-yet incomplete CSS 3 specification (see www.w3.org/TR/css3-selectors), but supported already in Firefox and Safari (and likely to soon be supported in Opera version 9 and above). This pseudo-class allows you to style a targeted anchor or identified element. Though it's not supported in any version of Internet Explorer,¹⁰ it never hurts to include this type of extra treat for users of modern browsers. To see it in action, you should create a file containing a number of links, like so:

```
<p><a href="#one">link to paragraph one</a> | ➤  
<a href="#two">link to paragraph two</a> | ➤  
<a href="#three">link to paragraph three</a></p>
```

Then, underneath, create three paragraphs, each with a unique `id` matching those in your links:

```
<p id="one">paragraph one</p>  
<p id="two">paragraph two</p>  
<p id="three">paragraph three</p>
```

Finally, include the following line in the head of your document:

```
<style type="text/css">:target { background: yellow;}</style>
```

Now, in either Firefox or Safari, start clicking those links. The targeted paragraph gains a yellow background, which is useful to indicate to users where they should be looking if they've followed a link from a separate page or a list of contents at the start of a long document, and it saves them from having to hunt around for what they want.

10. You can replicate the `:target` functionality in Internet Explorer with JavaScript. See Patrick Griffiths and Dan Webb's "Suckerfish `:target`" article (www.html5dog.com/articles/suckerfish/target) for more information.

Accessible linking

You can enhance the accessibility and usability of your website with judicious use of the `tabindex` and `accesskey` attributes (on the `<a>` tags, but not the `<link>` tags) and the `title` attribute (available to use on all elements).

The `title` attribute gives extra information about the related element. In the case of anchor links, it should be used to give a description about what the link is about, but only if the link text does not already provide enough information. For instance, there's no need to write `Contacts`, as a screen-reading device is likely to read out both the title and the link text redundantly. A better title in this case might be "E-mail, telephone, and postal contact information for the board of directors," as this title allows you to have a short link (one that would fit in a narrow navigation list or similar) but still provide a detailed explanation. Some browsers display the `title` value as a tooltip, visible when users hover over the element with their mouse, while other browsers display the value in the status bar in lieu of the URL. In the case of `<link>` links, the `title` value is usually what is used to generate a menu of alternative style sheets and such to users.

The `tabindex` and `accesskey` attributes both only apply to elements that can take keyboard focus: anchor links and form fields (I discuss these attributes' use with form fields in Chapter 4). `tabindex` allows you to create a specific tabbing order for users who are using a keyboard instead of a mouse to navigate your site. You do this by giving an attribute value of a number, such as `tabindex="1"`, which would make that element the first element focused when the user presses the Tab key. A tabindexed element with a higher value would be the next, and so on.

The `tabindex` attribute has its uses, but you should use it with care. If you have built your site well, with your navigational elements in a logical order in the source markup, then you are unlikely to even need to apply `tabindex`s to your links, as the browser will be able to work out the logical tab order automatically. You also run the risk of confusing users if your `tabindex` structure is radically different from what they're used to, and you may encounter maintenance problems if you find you need to insert a new link in the middle of your list, leading to a renumbering of all the subsequent `tabindex`s.

The `accesskey` attribute comes with a similar "use with caution" warning. This attribute allows you to specify a character to be used as a keyboard shortcut, such as `accesskey="s"`, which would result in that element gaining focus when the user activates that shortcut. Depending on the browser, this shortcut could be with `Ctrl+S`, `Alt+S`, or even `Ctrl+Alt+S` (plus the Mac equivalents).

It sounds nice enough, but there are strong arguments for avoiding `accesskeys` altogether.¹¹ To start with, there is no standard `accesskey` scheme, which results in many sites having different shortcut keys. Most browsers do not come with any ability to display what the `accesskeys` actually are to the user, which means the user must either find an accessibility statement and refer to it or view the source markup. Also, conflicts between `accesskeys` and hard-wired browser shortcuts can upset user expectations, and with so many different

11. To hear what various web standards notables have said about `accesskeys`, I suggest starting with Dave Shea's blog entry titled "I Do Not Use Accesskeys" (www.mezzoblue.com/archives/2003/12/29/i_do_not_use) and going from there.

browsers available, it was discovered by Web Accessibility Technical Services that / \ and] were the only characters not already used by a browser.¹²

Opera, which provides keyboard shortcuts for everything but the kitchen sink, deals with this problem by including a keyboard shortcut that “enables” accesskeys—pressing Shift+Esc will then allow you to use the accesskeys without needing any modifier key (so to jump to a form field that had an accesskey value of s, you would just need to press S instead of Shift+S or Ctrl+S). However, this does not address the problem of discovering exactly what accesskeys are in use on any given website, and Opera doesn’t even provide a notification to let you know whether you’re in an enabled mode or not—or let you know, without delving into the help files, that such a system exists at all.

A system in use at Accessify attempts to get around this problem by allowing users to set their own accesskeys (<http://accessify.com/preferences/accesskeys>), but this solution is unlikely to become widespread: it’s site-specific; it can’t be shared easily between sites, as cookies cannot be read from across different domains for security reasons; and users would still have problems on sites that did not implement this method.

There is a place for both tabindex and accesskey attributes, such as in web-based applications that need to mimic a desktop application, or in complex forms where the source order of the form elements may differ from the desired layout.

Marking up changes to your document

The `` and `<ins>` tags are both used for marking up changes to a document, most commonly for changes within the text (e.g., correcting details within a blog post after newer, more accurate information has been acquired). They can be used to note changes in document structure as well (e.g., wrapping whole sections of your website). They’re versatile, as they can be either block or inline, depending on the context. Both of the following examples are valid:

```
<p> Lorem ipsum dolor sit amet, ➡  
<del>consectetuer adipiscing elit</del>.</p>  
<del><p> Lorem ipsum dolor sit amet, ➡  
consectetuer adipiscing elit.</p></del>
```

If you have these tags within a block element—that is, a block element that doesn’t allow nested block elements, such as a paragraph or a heading—then they can be considered as inline elements, like `` or ``. However, if you use these tags to *contain* a block element, then they can be considered as block elements themselves (though you may need to apply `display: block` in your CSS to take advantage of block-style margins, paddings, and borders).

These tags do have some limitations, though. For instance, you can’t use `` to mark the removal of an ``, like this:

```
<ul><del><li>list item</li></del></ul>
```

12. Web Accessibility Technical Services, “Using Accesskeys - Is it worth it?”, www.wats.ca/show.php?contentid=32, January 2005.

because the only valid tag you can place directly within a `` is the ``. The same is true for removing table rows (`<tr>`) from tables—it would be invalid markup, even though it makes logical sense.

Visually, the `` tag will usually default to drawing a line through its content, while the `<ins>` tag will draw an underline, as shown in Figure 2-13. I generally remove the underline with CSS (`ins {text-decoration: none;}`), because if something is underlined on a web page, people tend to try and click it, but it may be OK if your actual links are distinctive enough.¹³

Some normal paragraph text.

~~Now I'm using .~~

Now I'm using <ins>.

Figure 2-13. A paragraph with some deleted and inserted content

Two attributes are specific to `` and `<ins>`: `datetime` and `cite`. The former allows you to set a date for when the correction took place, and the latter is for you to provide a link to a page explaining the reason behind the correction:

```
<p> Lorem ipsum dolor sit amet, <del datetime="20060509"
cite="http://example.com/errata.html">consectetuer
adipiscing elit</del>.</p>
```

As is often the case with these kinds of attributes, their information is hidden from most users, so if you want to display it, you will need to use JavaScript, as discussed earlier when we examined how to retrieve the `cite` attribute value from a `<blockquote>`. You can also use JavaScript to control the display of your deletions and insertions. This could be in the form of a simple display/hide toggle link, or you could go a bit further.

Jonathan Snook experimented with using JavaScript to create a `<div>` containing both the `datetime` and `cite` values for each `` or `<ins>`, and then displaying that `<div>` alongside the containing paragraph. See his article “An experiment with INS and DEL” at www.snook.ca/archives/html_and_css/an_experiment_w for more information.

Presentational elements

A **presentational element** says nothing about the content it describes, but instead says everything about how it *appears*. A range of these elements are still in the HTML specification, some of which we can still use and others of which have more appropriate

13. The W3C uses `<ins>` and `` for its working drafts of specifications. For instance, at www.w3.org/Style/css21-updates/WD-CSS21-20050613-20040225-diff/cover.html the W3C has really gone to town with them.

alternatives. The HTML specification doesn't explicitly list elements as being presentational *per se*, so I'm going to go over elements that are commonly held to *be* presentational: `<hr>`, `<pre>`, `<sup>`, `<sub>`, `<i>`, ``, `<strike>` and `<s>`, `<u>`, `<tt>`, `<big>`, and `<small>`.

When considering using any of these elements, the acid test should be this: are you using the element purely for the visual effect? If the answer is no, then you're probably OK; if the answer is yes, a more suitable alternative may be out there. A further test is if you can remove the style sheet from your document and have the document still make sense. If this isn't possible, then you may find a presentational element is exactly what you need.

Font style elements

A **font style element** is defined by the W3C as an element that simply specifies font information, and thus has no semantic meaning or value. Although they have not all been deprecated, their use is generally discouraged in favor of style sheets. The font style elements as defined by the W3C are `<i>`, ``, `<tt>`, `<big>`, `<small>`, `<strike>`, `<s>`, and `<u>`. There are also two **font modifier elements**, `` and `<basefont>`, but both have been deprecated and are no longer of any use to us. Font style elements go against what I've been saying about considering what your content *means*, rather than what it *looks like*—these elements say a lot about appearances and nothing about meaning. Nevertheless, some of them can still be useful to us, particularly when (X)HTML does not provide us with a meaningful alternative.

The `<i>` and `` elements are the two most obvious presentational elements, and in the drive toward increased accessibility and a standards-based design methodology they have often been eschewed in favor of `` and ``, respectively, because the former tags convey purely visual information while the latter tags convey meaning as well—emphasis. However, people can go too far and use `` and `` as standard replacements, using them constantly for any instance where they require italic and bold text. This error has been perpetuated by numerous WYSIWYG tools that keep the common I and B buttons but change their function, adding an `` or `` element to the markup instead. This would be fine if web designers and authors actually meant, every time, to emphasize, but that isn't always the case.

Both of these elements still have their uses, and neither has been deprecated or removed from any (X)HTML specification. For instance, `<i>` can be used when italicizing text in a foreign language (i.e., `<i lang="it">Molte grazie</i>`). Using a span for this purpose would require an additional class name,¹⁴ such as ``, so with `<i>` and `` both as semantically meaningless as each other, I recommend simply using the `<i>` element, because it already does what you want and will take up less space in your markup.

The `<tt>` element renders text in a teletype or monospaced font. Again, this element is semantically meaningless, and you may find other elements such as `<code>`, `<pre>`, or

14. If you're only concerned about CSS2-supporting browsers, then you can use an attribute selector in your CSS to achieve the italic effect: `span[lang] { font-style: italic; }`. This means that you don't need the extra class name, but it also means you lose the effect in Internet Explorer, content aggregators, and other non-CSS user agents.

`<samp>` more suitable to describe your content (discussed further in the next section). Nevertheless, it remains part of current (X)HTML specifications, so it is there for you if required.

`<big>` and `<small>` both affect the font size of their contained content, and nested `<big>` or `<small>` elements will cause the contents to be even bigger or smaller. It's purely presentational, and you may find that the CSS `font-size` property is more appropriate for your needs, though some, such as accessibility advocate Joe Clark, have suggested using nested `<big>` and `<small>` elements in tag clouds¹⁵ to achieve the “weighted by importance” visual effect. Others, such as Tantek Çelik, suggest using nested `` elements instead,¹⁶ which has the benefit of being more meaningful but loses the visual effect in non-CSS user agents—you pay your money and you make your choice.

The only font style elements that have been deprecated in both HTML 4 and XHTML 1 are `<strike>`, `<s>`, and `<u>`. `<strike>` and `<s>` have an identical function: they draw a horizontal line through their content, and this function can be visually replicated by using the CSS `text-decoration` property. If you wish to indicate that some content has been deleted, then use the `` element (discussed previously). `<u>` is used for underlining content, and again this function can be replicated with `text-decoration`. It is also advisable to not underline anything that isn't a link—links and underlines are so commonly associated with one another that anything on the Web that's underlined now looks like a link to users.

As far as styling these elements goes, there's not much point beyond specifying further font information (such as specifying a range of specific typewriter-style fonts for `<tt>`, or increasing the size variation between different levels of `<big>` and `<small>`). However, don't be put off using any of these elements just because they're presentational. With the exceptions of `<strike>`, `<s>`, and `<u>`, all of these elements are part of current and future (X)HTML specifications, and if you find yourself writing `...` or `...` a lot, then consider using `<tt>` or `<i>` instead; they're as semantically meaningless as a `span`, but they already do what you want and take up less space in your markup. You can always restyle these elements, just as you would a `span` if required, but if you think you might wish to do this in the future, then bear in mind that you might end up with an `<i>` styled as a ``, and vice versa.

It is also worth considering these elements when designing WYSIWYG interfaces for average users, because having incorrect semantics is worse than having no semantics at all. For example, if you think your users may want to italicize a citation, don't give them the chance to do so with the `` element; the `<i>` element is a safer choice because it isn't going to confuse any tool or browser trying to utilize your semantic information.

The `<hr>`, `<pre>`, `<sup>`, and `<sub>` elements

The `<hr>`, `<pre>`, `<sup>` and `<sub>` elements are all technically presentational elements, but they *also* convey meaning that cannot (or should not) be replicated with CSS. Take the

15. See <http://blog.fawny.org/2005/01/23/weighted>.

16. See <http://tantek.com/log/2005/02.html#d02t0800>.

<hr> element, for instance: it's a straightforward horizontal rule. In many cases, this can (and should) be reproduced by simply adding a CSS border to the top or bottom of a block element, but you should only do this if you're using an <hr> element to apply a border. If you're using an <hr> element specifically as a *separator* of the sort you might find separating sections of a book chapter, then your rule does have a structural purpose and should be used instead of the CSS border.¹⁷

The <hr> tag comes with several attributes—size, width, noshade, and align—but all have been deprecated, so we must use CSS for style, which can be a little quirky, as Internet Explorer essentially treats <hr> as an inline element, whereas Firefox *et al.* treat it as a block element. For instance, if you want to color your horizontal rule red, you must set both the color property *and* the background-color property. If you want to align the rule to the left or right, you must set the text-align property for Internet Explorer and use margin: 0 auto for other browsers.

For an even more interesting rule, you can use the background-image property—but this won't work in Internet Explorer. There are two solutions at the time of this writing. The first is to wrap your <hr> in a <div>, give your <div> the background image, and then use display: none to remove the <hr> from view. Non-CSS user agents will then see the <hr>, but CSS user agents will see your background image instead. The second solution, for the markup purists, is to dynamically include a <div> with JavaScript. You could then style the <div> and hide the <hr> with CSS.

For example, the following script notes where all of the <hr> elements are, and then creates a <div> with a class name of rule and inserts it just before each <hr>. Your markup remains free of extraneous <div> tags, but you can still style those that are being dynamically inserted.

```
function lovelyrules() {
    if (!document.getElementsByTagName || !
!document.createElement || !document.insertBefore) return;
    var rules = document.getElementsByTagName("hr");
    for (var i=0; i<rules.length; i++) {
        var div = document.createElement("div");
        div.className = "rule";
        rules[i].parentNode.insertBefore(div, rules[i]);
    }
}
window.onload = lovelyrules;
```

Now, what about <pre>? The visual effect caused by the <pre> tag is to preserve the white-space (i.e., the tabs, spaces, and line breaks) in your markup, so if that whitespace is important in understanding the content, such as code samples, then use <pre> (see Figure 2-14). The effect can be replicated with the CSS white-space property, but using this property in place of <pre> means you'll lose the effect in non-CSS user agents.

17. This is exactly the sort of thinking that has led the W3C to suggest a <separator> element in XHTML 2.0 (see Appendix A).

What follows is some pre-formatted text:

```
h1 {
    background-color: white;
    color: blue;
    font-size: 3em;
    font-style: italic;
}
```

2

Figure 2-14. A comparison of `<pre>` in both source code and rendered in a browser. The whitespace (tabs and carriage returns) that I inserted in the source code have been retained.

Similarly, the `<sup>` and `<sub>` (superscript and subscript) elements can convey important meaning via presentation. Consider the two following equations:

- $e=mc^2$
- $e=mc2$

Although they look alike, only one of the preceding equations is Einstein's; spelled out, the former equation is "e equals m times c squared," while the latter is "e equals m times c times 2." Or how about this:

- H_2O
- H2O

The first is chemical equation for water—two hydrogen atoms and one oxygen atom—and the second is simply the letter "H" followed by the number 2, then the letter "O", and is meaningless. So, the placing and styling of the "2" is therefore important, and if you removed its styling and positioning and placed it in a style sheet, some browsers could lose the meaning.

The W3C also notes that some languages (other than English) require the use of subscripted and superscripted text. Here's an example in French:

- M^{lle} Dupont

Stylistically, you can also use superscripts and subscripts in English. You'll most likely have seen them in dates, or to indicate the presence of footnotes/endnotes:

- The 14th of September
- The committee report stated that the minister had acted in good faith.ⁱⁱ

Also usually italicized by default is the `<dfn>` element, which indicates the first usage of a term that will be used repeatedly throughout a document (i.e., its defining instance), for example:

```
<p>You can keep your CSS rules in a file separate from your HTML.
This file is known as a <dfn>style sheet</dfn>. You can also have a
separate style sheet that determines how your web page looks when it is
being printed...</p>
```

The preceding code would render as follows:

“You can keep your CSS rules in a file separate from your HTML. This file is known as a *style sheet*. You can also have a separate style sheet that determines how your web page looks when it is being printed . . .”

It’s worth considering adding unique id attributes to your definitions, as this enables you to link directly to them from a glossary if you have one (which could be constructed using a definition list).

Coding

Four phrase elements are particularly useful for describing programming or other computing-related tasks, such as describing user input: `<code>`, `<var>`, `<samp>`, and `<kbd>`. The former two are used to display raw computer code, (X)HTML markup, CSS, and so on. `<code>` will usually display in user agents in a monospaced font, while `<var>` will display in an italic font (see Figure 2-15).

```
<code>
  #!/usr/local/bin/perl<br />
  print "<var>Content-type: text/html\n</var>";<br />
  print "<var>Hello World</var>\n";<br />
</code>
```

```
#!/usr/local/bin/perl
print "Content-type: text/html\n";
print "Hello World\n";
```

Figure 2-15. `<code>` and `<var>` in use. The content within `<var>` tags is italicized by default.

As noted earlier, you may wish to lose the line breaks in such an example and wrap the code in `<pre>` tags, to preserve any breaks and tabs and other whitespace formatting. If you’re displaying long lines of code, though, eschewing `<pre>` and using breaks instead will better prevent those long lines from breaking out of your layout.

While `<code>` and `<var>` are used for displaying code, `<samp>` (as in “sample”) describes the output of that code. It is used simply, and it will also usually display in a monospaced font:

Phrase elements

A **phrase element** adds meaning to a fragment of text. It's likely that you've already encountered phrase elements without even knowing it; `` and `` are two such elements, but there are several other, underused phrase elements that can help to make your document more structurally and semantically meaningful while still maintaining your desired visual style. The full list of phrase elements is as follows: ``, ``, `<cite>`, `<dfn>`, `<code>`, `<samp>`, `<kbd>`, `<var>`, `<abbr>`, and `<acronym>`.

Emphasis

As mentioned previously, `` and `` should be used not to italicize or bold text; rather, they indicate emphasis, with `` being more emphatic than ``, and the combination of the two more emphatic still. Where visual browsers will usually display an `` and `` with italic and bold text, respectively, screen readers may change volume, pitch, and rate when encountering these elements.

Here's an example usage:

```
<p>I was a <em>little</em> bit angry, then I was ➡
<strong>very</strong> angry, then I was ➡
<em><strong>extremely</strong></em> angry!</p>
```

The preceding code would render in most user agents as follows:

I was a *little* bit angry, then I was **very** angry, then I was ***extremely*** angry!

You may not want your emphasis displayed in such a way. Remember that you can always restyle `` and `` elements to display however you like, while still retaining their semantic meaning. For instance, if the text of your document was in Japanese ideographic text, then you would be unlikely to need an italic version for emphasis, and a change in background color may be more appropriate.

This preceding issue of internationalization is discussed in more detail in Molly E. Holzschlag's article "World Grows Small: Open Standards for the Global Web" (www.alistapart.com/articles/worldgrowsmall).

Citations and definitions

We've already encountered the `cite` attribute, used within `<blockquote>` tags to attribute a source to the quote, and within `` and `<ins>` tags to refer to an explanatory document, but there also exists a `<cite>` tag to contain stand-alone references not associated with any particular element, or citations of other material. You would use this tag for referring to other sources, such as book or movie titles. Most user agents will display a citation in italic font, a typographic convention you'll often see in the print world as well.

```
<p><samp> [paul@localhost ~]$ perl hello.cgi</samp></p>
<p><samp>Content-type: text/html<br />
&lt;H1&gt;Hello World&lt;/H1&gt; </samp></p>
```

With all these monospaced fonts now in your document, it's probably worth considering using CSS to mix things up a little. For instance, you could display `<samp>` as the output of a command prompt window, as shown in Figure 2-16.



Figure 2-16. `<samp>` displays in a monospaced font by default, and as such it can look a bit too much like `<code>`. Using CSS to style it differently can help distinguish the two.

The CSS for this is straightforward enough:

```
samp {
  background: #000;
  border: 3px groove #ccc;
  color: #ccc;
  display: block;
  padding: 5px;
  width: 300px;
}
```

Finally, there is `<kbd>`, which is used to indicate keyboard input by the user, like so:

```
<p>Press <kbd>A</kbd>, then <kbd>B</kbd>, then ➡
<kbd>C</kbd>. Finally, press the <kbd>Enter</kbd> key to continue.</p>
```

I think that the obvious thing to do with a `<kbd>` element is to style it to look vaguely keyboardlike:

```
kbd {
  background-color: #F1E7DD;
  border: 1px outset #333;
  color: #333;
  padding: 2px 5px;
}
```

The preceding CSS yields a result like that shown in Figure 2-17.

Press **A**, then **B**, then **C**. Finally, press the **Enter** key to continue.

Figure 2-17. Several `<kbd>` elements styled to look like actual keys

Abbreviations

For displaying abbreviated text, you have two options available: `<abbr>` and `<acronym>`. The `<abbr>` element indicates abbreviated text, with the full, unabbreviated text often contained within a `title` attribute, while `<acronym>` is used for acronyms (and possibly initialisms as well) instead of abbreviations. What's the difference? Well, the W3C is a little hazy on the issue, but the *Oxford English Dictionary* defines the three terms as follows:

- **Abbreviation:** A shortened form of a word or phrase.
- **Acronym:** A word formed from the initial letters of other words (e.g., *laser*, which is an acronym of Light Amplification by Stimulated Emission of Radiation).
- **Initialism:** An abbreviation consisting of initial letters pronounced separately (e.g., *BBC*).

Or in other words, *all acronyms and initialisms are abbreviations, but not all abbreviations are acronyms or initialisms*. So when in doubt, using `<abbr>` to describe your content will be correct. Unfortunately (did you see this coming?), Internet Explorer 6 and below do not support the `<abbr>` element, which has led to many people using `<acronym>` instead (which Internet Explorer does support), even though not all abbreviations are acronyms. Typical. The question of whether you use `<acronym>` incorrectly or use `<abbr>` correctly and forget about Internet Explorer is one you will have to answer yourself, but my preference is for the former—I'd rather use the right semantics for the situation than use the wrong ones.

Dean Edwards has come up with a way of tricking Internet Explorer into sort of supporting `<abbr>`, detailed in his article "abbr-cadabra" (<http://dean.edwards.name/my/abbr-cadabra.html>). Internet Explorer 7 includes native support for `<abbr>`.

Another factor to take into consideration is how screen readers and other aural devices treat abbreviations. An abbreviation that is an acronym or a truncation should be pronounced as if it's a regular word; an abbreviation that is an initialism should be spelled out rather than pronounced. We can control this with a specific aural style sheet and, due to the lack of a specific initialism tag, a couple of extra class names:

```
<abbr>Mr</abbr>
<acronym>NATO</acronym>
<abbr class="initialism">BBC</abbr>
```

The aural styling of these elements using CSS is as follows:

```
@media aural {
  abbr , acronym {speak : normal;}
  abbr.initialism {speak : spell-out;}
}
```

Alternatively, you can include an aural style sheet separately in the head of your document like this:

```
<link rel="stylesheet=" media="aural" href="aural.css" />
```

Images and other media

Without images, the Web would be far less interesting to look at, though on the other hand we'd also never have to run across any animated construction-worker GIFs on an unfinished page. You can always find a silver lining if you look hard enough.

Using images within your website is something you are likely very familiar with, but this topic is still worth discussing, due to the different ways images can be included: inline, via CSS background images, via image maps, and through the `<object>` element. Each method has pros and cons, as described in the sections that follow.

Inline images

The simplest method of including an image is directly within the markup, using the self-closing `` element:

```

```

There are some things to consider here. First, let's look at `alt` attributes. In both XHTML and HTML, all images should have `alt` attributes regardless of whether the attribute contains a value. An `alt` attribute's one and only purpose in life is to provide an *alternative* to the image, which means that the value of the attribute must duplicate any meaningful content within the image. If your image has no meaningful content, then leave the `alt` attribute in, but leave it empty: `alt=""`. (Also consider whether you can remove the image entirely and place it within your CSS as a decorative background image, as discussed in the next section.)

The `alt` attribute is not there to provide any additional information or a descriptive caption—this is the purpose of the `title` attribute. The content of an `alt` attribute should only be displayed if the image is unavailable for whatever reason; it is an alternative (either display the image or the alternative text, not both). Unfortunately, due to Internet Explorer's (incorrect) behavior of displaying the content of an `alt` attribute as a tooltip,¹⁸ it's been frequently misused for that very purpose when, in fact, a `title` attribute might be more appropriate.

Also available is the `longdesc` attribute, which provides a link to a page containing a more elaborate supplement to the `alt` attribute value. Browser support for this attribute is historically very poor, though, so you should use with caution, if at all.

18. If a `title` attribute is present as well, then Internet Explorer will display that value in the tooltip rather than the `alt` attribute value. If both attributes are present but the `title` attribute is empty, then no tooltip will display.

The `img` element allows you to specify an image width and height value in pixels, in the form of `width="150"`. These attributes are optional, but an advantage of using them is that a web browser can accurately display textual content in the correct space before the images have loaded, avoiding the problem of text jumping around to make way for images as they arrive. A disadvantage is that they are presentational attributes, and ideally these sorts of values would be set in the CSS to make it easier to change if the image dimensions ever changed. Furthermore, the presence of size attributes will cause the `alt` attribute value to display in a box sized to the same dimensions, which may not be desirable if the text is larger than the space allowed.

CSS background images

Images can also be included via the CSS `background-image` property. You should use this property to place decorative images within your pages to help keep your markup as clean as possible, allowing you to make significant changes to the look and feel of your website by changing only your style sheet (and also allowing you to specify different images for printed, mobile, and projected versions of your site). A disadvantage is that when images are not available but CSS is, alternative content will not be displayed—so using a CSS background image for navigation button text or similar is not advised.

So long as you're not reproducing textual content, background images can be very useful to help us achieve certain visual effects. In his article "Faux Columns" (www.alistapart.com/articles/fauxcolumns), Dan Cederholm documented the concept of **faux columns**, where using a single tiled background image on the `<body>` of your web page creates the illusion of two equal-height columns, irrespective of which column contains the most content. Or see Doug Bowman's article "Sliding Doors of CSS" (www.alistapart.com/articles/slidingdoors), where he details a technique in which using an oversized background image in a navigation menu allows the text to be resized and the background image scaled with it. Also worth a look is Tim Murtaugh's article "CSS Design: Mo' Betta Rollovers" (www.alistapart.com/stories/rollovers), where he describes how judicious use of the `background-image` property and the `:hover` pseudo-class allows you to create CSS-based image rollovers without recourse to JavaScript.

Image maps

Image maps come in two varieties: client side and server side. A client-side image map consists of an image with a series of predetermined hotspot areas of varying shapes and sizes that represent links. A server-side image map is a similar construct, but the pixel coordinates of the mouse click are sent to the server, which calculates the subsequent action. Client-side image maps are preferable as they can be made accessible to people browsing with images disabled or unavailable, and they offer immediate feedback as to whether users are clicking an active region.

The muddy markup required for an image map is anathema to the mantra of separating presentation from content. There are two distinct parts: the map element (`<map>`) and the image element (``), neither of which is nested within the other. The map element is a container tag with a `name` attribute, and the tag contains any number of self-closing `<area>` tags. These `<area>` tags use the `shape` attribute to determine the shape of the area (`circle`,

While it is not possible to re-create circular or polygonal areas with a pure CSS solution, you can create rectangular areas by styling a simple unordered or ordered list of links. The steps to do this are as follows:

1. Create a list of links with standard (X)HTML.
2. Give each link a unique id attribute.
3. In the CSS, give the list a background image, width, and height, and add `position: relative;` to make sure your links (that you will position absolutely) are placed in relation to the top left of the list and not the browser window.
4. Also in the CSS, give each link the required width, height, and background image or color.
5. Using `position: absolute;` and the `top`, `left`, `right`, and `bottom` CSS properties, position each link within the list as appropriate.
6. Hide the link text with `text-indent: -9999px;`¹⁹

Let's go through that process again, this time with examples. The (X)HTML is simple enough:

```
<ul>
<li><a href="/" id="homelink">Home</a></li>
<li><a href="/about/" id="aboutlink">About</a></li>
<li><a href="/contact/" id="contactlink">Contact</a></li>
</ul>
```

The CSS is slightly less simple, but it shouldn't give you much of a headache. You'll most likely need a graphical editor of some kind to help you work out the coordinates, though. First of all, you style the list:

```
ul {
background: url(imagemap.gif) top left no-repeat;
height: 272px;
position: relative;
width: 300px;
}
```

Next, reset the margins and padding of both the list and the list items to 0. This may not always be necessary, but it will help when trying to position elements, and it also aids in cross-browser consistency.

```
ul, li {
margin: 0; padding: 0;
}
```

¹⁹ The negative `text-indent` value will shunt the text content of the link way off to the left side of the screen, making it effectively invisible. When doing this on a link, however, Firefox (and possibly some other modern browsers) will draw its "active" link outline around the entire space, leading to a large outline box stretching off to the left. To overcome this, just add `overflow: hidden;` and the outline box will only surround the visible, clickable area instead.

HTML MASTERY: SEMANTICS, STANDARDS, AND STYLING

rect, poly, or default), a coords attribute to stake out the dimensions of the shape, and either an href attribute to determine where users should be taken to after they've clicked or a nohref attribute if no link is in use.

In the meantime, the element gains a usemap attribute, the value of which should be the same as the value of the map's name attribute. It also needs an ismap attribute to say that the image is a map. Phew.

Here's an example (following XHTML rules) that will—hopefully—clarify the preceding explanation:

```
<map name="Map">
  <area shape="rect" coords="118,192,203,249" ↪
href="http://joeblade.com" alt="Home" />
  <area shape="circle" coords="52,76,39" ↪
href="http://joeblade.com" alt="About" />
  <area shape="poly" ↪
coords="159,145,115,96,115,39,160,21,205,40,206,103" ↪
href="http://joeblade.com" alt="Contact" />
</map>

```

This image map contains three clickable areas: a rectangle, a circle, and a six-sided polygon. As you can imagine, hand-coding image maps this way is fairly laborious, but most WYSIWYG software comes with the ability to create areas just by pointing, clicking, and dragging. Figure 2-18 shows an example of an image map created in Dreamweaver.



Figure 2-18. An image map created in Dreamweaver. Creating clickable areas this way is a simple matter.

HTML MASTERY: SEMANTICS, STANDARDS, AND STYLING

Then you deal with the links. Each one needs to be positioned separately, by referencing the `id` attribute set in the markup, but shared values can be dealt with in one try. I'm going to give these links background colors of green to demonstrate where the clickable areas will lie—in the real world, you would blend them in better with their backgrounds.

```
a {
  background: green;
  display: block;
  overflow: hidden;
  position: absolute;
  text-indent: -9999px;
}
```

Finally, position each link and give them all a width and height:

```
#homelink {
  top: 15px;
  left: 50px;
  width: 75px;
  height: 75px;
}

#aboutlink {
  top: 20px;
  left: 110px;
  width: 100px;
  height: 125px;
}

#contactlink {
  top: 190px;
  left: 120px;
  width: 75px;
  height: 50px;
}
```

If all has gone well, your CSS-based image map should look a little like Figure 2-19.

The advantage of creating a maplike structure in this way is that the (X)HTML is just a plain old list of links, making life easier for users of text browsers and the like. The disadvantage is as mentioned earlier: background images set via CSS will not provide any alternative text if images are disabled but CSS is enabled, so the map becomes inaccessible in this scenario.



Figure 2-19. A CSS-based image map, as seen in the browser

Being objective

`<object>` is designed to include *objects* such as images, videos, and Java applets in a web page. It was intended to replace the more specific `` and `<applet>` tags, as well as the proprietary `<embed>` and `<bgsound>` tags. It comes with a fallback mechanism, whereby you can nest `<object>`s, allowing the user agent to display alternative content if it cannot render the preferred choice. For instance, you can nest a video, an image, and finally some text like so:

```
<object data="myVideo.mpg" type="application/mpeg">
  <object data="myPicture.gif" type="image/gif">
    Some descriptive text, and <a href="link.htm">a link</a>.
  </object>
</object>
```

The user agent should first try to display the video, but if it can't, it should then try to display the image, and if it can't do that, it displays the text—no need for `alt` attributes here. Unfortunately, poor browser support has made `<object>` very hard to use as it was intended, and the tag itself is overloaded, with 17 element-specific attributes.²⁰ Internet Explorer treats any `<object>` content as if it were an ActiveX control, prompting the user

20. A trimmed-down `<object>` is likely to appear in XHTML 2.0, as detailed in Appendix A.

with a security warning if the user's security settings are set to do this, even if the tag is just displaying a static image. There can also be problems with scrollbars appearing around the image, as if you were including a web page within an `<iframe>`²¹—this behavior is clearly undesirable.

But `<object>` isn't just about including images: it is also used to embed rich media players such as Windows Media Player, RealPlayer, QuickTime,²² and Flash Player,²³ along with the `<param>` element, which passes various parameters to the media player in question. It's unlikely that you'll ever be hand-coding these embedded objects by hand, unless you're comfortable writing markup like `classid='CLSID:22d6f312-b0f6-11d0-94ab-0080c74c7e95'`, so I don't cover this sort of usage in great detail here. Simply be aware that when embedding media, the `<object>` tag is what you usually need to use.

Summary

This chapter covered a substantial number of (X)HTML tags available to you, with the exception of table- and form-related markup, which will follow in their own dedicated chapters. As you've seen, you have a wide range of options when it comes to structuring, describing, and displaying your content, and although you may only ever use a fraction of the available tags, knowing the tags you are able to use in the first place and how to correctly use them are important parts of mastering HTML.

-
21. An `<iframe>` is frame that can contain other content, including other web pages. It can be treated the same as any other frame, except it can also be positioned anywhere on a page and given fixed dimensions.
 22. Apple recommends that you use JavaScript to embed QuickTime movies, due to Internet Explorer now requiring any embedded ActiveX control to be manually activated by the user due to a patent dispute between Microsoft and Eolas. You can find more details at www.apple.com/quicktime/tutorials/embed.html.
 23. Anyone interested in embedding Flash without invalidating their document should read the article "Flash Satay: Embedding Flash While Supporting Standards" by Drew McLellan (www.alistapart.com/articles/flashesatay).