№3. WHAT DOES IT ALL MEAN?

show table of contents

❧

DIVING IN

his chapter will take an HTML page that has absolutely nothing wrong with it, and improve it. Parts of it will become shorter. Parts will become longer. All of it will become more semantic. It'll be awesome.

Here is the page in question. Learn it. Live it. Love it. Open it in a new tab and don't come back until you've hit "View Source" at least once.

❧

THE DOCTYPE

From the top:

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

This is called the "doctype." There's a long history — and a black art — behind the doctype. While working on Internet Explorer 5 for Mac, the developers at Microsoft found themselves with a surprising problem. The upcoming version of their browser had improved its standards support so much, older pages no longer rendered properly. Or rather, they rendered properly (according to specifications), but people expected them to render *improperly*. The pages themselves had been authored based on the quirks of the dominant browsers of the day, primarily Netscape 4 and Internet Explorer 4. IE5/Mac was so advanced, it actually broke the web.

Microsoft came up with a novel solution. Before rendering a page, IE5/Mac looked at the "doctype," which is typically the first line of the HTML source (even before the `<html>` element). Older pages (that relied on the rendering quirks of older

browsers) generally didn't have a doctype at all. IE5/Mac rendered these pages like older browsers did. In order to "activate" the new standards support, web page authors had to opt in, by supplying the right doctype before the `<html>` element.

This idea spread like wildfire, and soon all major browsers had two modes: "quirks mode" and "standards mode." Of course, this being the web, things quickly got out of hand. When Mozilla tried to ship version 1.1 of their browser, they discovered that there were pages being rendered in "standards mode" that were actually relying on one specific quirk. Mozilla had just fixed its rendering engine to eliminate this quirk, and thousands of pages broke all at once. Thus was created — and I am not making this up — "almost standards mode."

In his seminal work, Activating Browser Modes with Doctype, Henri Sivonen summarizes the different modes:

> Quirks Mode
>
> > In the Quirks mode, browsers violate contemporary Web format specifications in order to avoid "breaking" pages authored according to practices that were prevalent in the late 1990s.
>
> Standards Mode
>
> > In the Standards mode, browsers try to give conforming documents the specification-wise correct treatment to the extent implemented in a particular browser. HTML5 calls this mode the "no quirks mode."
>
> Almost Standards Mode
>
> > Firefox, Safari, Chrome, Opera (since 7.5) and IE8 also have a mode known as "Almost Standards mode," that implements the vertical sizing of table cells traditionally and not rigorously according to the CSS2 specification. HTML5 calls this mode the "limited quirks mode."

(You should read the rest of Henri's article, because I'm simplifying immensely here. Even in IE5/Mac, there were a few older doctypes that didn't count as far as opting into standards support. Over time, the list of quirks grew, and so did the list of doctypes that triggered "quirks mode." The last time I tried to count, there were 5 doctypes that triggered "almost standards mode," and 73 that triggered "quirks mode." But I probably missed some, and I'm not even going to talk about the crazy things that Internet Explorer 8 does to switch between its four — four! — different rendering modes. Here's a flowchart. Kill it. Kill it with fire.)

Now then. Where were we? Ah yes, the doctype:

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

That happens to be one of the 15 doctypes that trigger "standards mode" in all modern browsers. There is nothing wrong with it. If you like it, you can keep it. Or you can change it to the HTML5 doctype, which is shorter and sweeter and also triggers "standards mode" in all modern browsers.

This is the HTML5 doctype:

```
<!DOCTYPE html>
```

That's it. Just 15 characters. It's so easy, you can type it by hand and not screw it up.

THE ROOT ELEMENT

An HTML page is a series of nested elements. The entire structure of the page is like a tree. Some elements are "siblings," like two branches that extend from the same tree trunk. Some elements can be "children" of other elements, like a smaller branch that extends from a larger branch. (It works the other way too; an element that contains other elements is called the "parent" node of its immediate child elements, and the "ancestor" of its grandchildren.) Elements that have no children are called "leaf" nodes. The outer-most element, which is the ancestor of all other elements on the page, is called the "root element." The root element of an HTML page is always `<html>`.

In this example page, the root element looks like this:

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

There is nothing wrong with this markup. Again, if you like it, you can keep it. It is valid HTML5. But parts of it are no longer necessary in HTML5, so you can save a few bytes by removing them.

The first thing to discuss is the `xmlns` attribute. This is a vestige of XHTML 1.0. It says that elements in this page are in the XHTML namespace, `http://www.w3.org/1999/xhtml`. But elements in HTML5 are always in this namespace, so you no longer need to declare it explicitly. Your HTML5 page will work exactly the same in all browsers, whether this attribute is present or not.

Dropping the `xmlns` attribute leaves us with this root element:

```
<html lang="en" xml:lang="en">
```

The two attributes here, `lang` and `xml:lang`, both define the language of this HTML page. (`en` stands for "English." Not writing in English? Find your language code.) Why two attributes for the same thing? Again, this is a vestige of XHTML. Only the `lang` attribute has any effect in HTML5. You can keep the `xml:lang` attribute if you like, but if you do, you need to ensure that it contains the same value as the `lang` attribute.

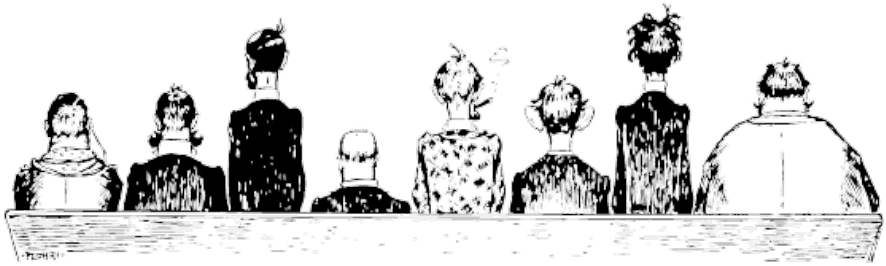
> To ease migration to and from XHTML, authors may specify an attribute in no namespace with no prefix and with the literal localname "xml:lang" on HTML elements in HTML documents, but such attributes must only be specified if a `lang` attribute in no namespace is also specified, and both attributes must have the same value when compared in an ASCII case-insensitive manner. The attribute in no namespace with no prefix and with the literal localname "xml:lang" has no effect on language processing.

Are you ready to drop it? It's OK, just let it go. Going, going… gone! That leaves us with this root element:

```
<html lang="en">
```

And that's all I have to say about that.

THE <HEAD> ELEMENT



The first child of the root element is usually the `<head>` element. The `<head>` element contains metadata — information *about* the page, rather than the body of the page itself. (The body of the page is, unsurprisingly, contained in the `<body>` element.) The `<head>` element itself is rather boring, and it hasn't changed in any interesting way in HTML5. The good stuff is what's *inside* the `<head>` element. And for that, we turn once again to our example page:

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>My Weblog</title>
  <link rel="stylesheet" type="text/css" href="style-original.css" />
  <link rel="alternate" type="application/atom+xml"
                       title="My Weblog feed"
                       href="/feed/" />
  <link rel="search" type="application/opensearchdescription+xml"
                       title="My Weblog search"
                       href="opensearch.xml"  />
  <link rel="shortcut icon" href="/favicon.ico" />
</head>
```

First up: the `<meta>` element.

CHARACTER ENCODING

When you think of "text," you probably think of "characters and symbols I see on my computer screen." But computers don't deal in characters and symbols; they deal in bits and bytes. Every piece of text you've ever seen on a computer screen is actually stored in a particular *character encoding*. There are hundreds of different character encodings, some optimized for particular languages like Russian or Chinese or English, and others that can be used for multiple languages. Roughly speaking, the character encoding provides a mapping between the stuff you see on your screen and the stuff your computer actually stores in memory and on disk.

In reality, it's more complicated than that. The same character might appear in more than one encoding, but each encoding

might use a different sequence of bytes to actually store the character in memory or on disk. So, you can think of the character encoding as a kind of decryption key for the text. Whenever someone gives you a sequence of bytes and claims it's "text," you need to know what character encoding they used so you can decode the bytes into characters and display them (or process them, or whatever).

So, how does your browser actually determine the character encoding of the stream of bytes that a web server sends? I'm glad you asked. If you're familiar with HTTP headers, you may have seen a header like this:

```
Content-Type: text/html; charset="utf-8"
```

Briefly, this says that the web server thinks it's sending you an HTML document, and that it thinks the document uses the `UTF-8` character encoding. Unfortunately, in the whole magnificent soup of the World Wide Web, few authors actually have control over their HTTP server. Think Blogger: the content is provided by individuals, but the servers are run by Google. So HTML 4 provided a way to specify the character encoding in the HTML document itself. You've probably seen this too:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Briefly, this says that the web author thinks they have authored an HTML document using the `UTF-8` character encoding.

Both of these techniques still work in HTML5. The HTTP header is the preferred method, and it overrides the `<meta>` tag if present. But not everyone can set HTTP headers, so the `<meta>` tag is still around. In fact, it got a little easier in HTML5. Now it looks like this:

```
<meta charset="utf-8" />
```

This works in all browsers. How did this shortened syntax come about? Here is the best explanation I could find:

> The rationale for the `<meta charset>` attribute combination is that UAs already implement it, because people tend to leave things unquoted, like:
>
> ```
> <META HTTP-EQUIV="Content-Type" CONTENT="text/html;" charset="ISO-8859-1">
> ```

There are even a few `<meta charset>` test cases if you don't believe that browsers already do this.

ASK PROFESSOR MARKUP

Q: I never use funny characters. Do I still need to declare my character encoding?

A: Yes! You should *always* specify a character encoding on every HTML page you serve. Not specifying an encoding can lead to security vulnerabilities.

To sum up: character encoding is complicated, and it has not been made any easier by decades of poorly written software used by copy-and-paste–educated authors. You should **always** specify a character encoding on **every** HTML document, or bad things will happen. You can do it with the HTTP `Content-Type` header, the `<meta http-equiv>` declaration, or the shorter `<meta charset>` declaration, but please do it. The web thanks you.

❧

FRIENDS & (LINK) RELATIONS

Regular links (`<a href>`) simply point to another page. Link relations are a way to explain *why* you're pointing to another page. They finish the sentence "I'm pointing to this other page because..."

- ...it's a stylesheet containing CSS rules that your browser should apply to this document.
- ...it's a feed that contains the same content as this page, but in a standard subscribable format.
- ...it's a translation of this page into another language.
- ...it's the same content as this page, but in PDF format.
- ...it's the next chapter of an online book of which this page is also a part.

And so on. HTML5 breaks link relations into two categories:

> Two categories of links can be created using the link element. **Links to external resources** are links to resources that are to be used to augment the current document, and **hyperlink links** are links to other documents. ...
>
> The exact behavior for links to external resources depends on the exact relationship, as defined for the relevant link type.

Of the examples I just gave, only the first (`rel="stylesheet"`) is a link to an external resource. The rest are hyperlinks to other documents. You may wish to follow those links, or you may not, but they're not required in order to view the current page.

Most often, link relations are seen on `<link>` elements within the `<head>` of a page. Some link relations can also be used on `<a>` elements, but this is uncommon even when allowed. HTML5 also allows some relations on `<area>` elements, but this is even *less* common. (HTML 4 did not allow a `rel` attribute on `<area>` elements.) See the full chart of link relations to check where you can use specific `rel` values.

ASK PROFESSOR MARKUP

☞ Q: Can I make up my own link relations?

A: There seems to be an infinite supply of ideas for new link relations. In an attempt to prevent people from just making stuff up, the microformats community maintains a registry of proposed `rel` values and the HTML specification defines the process for getting them accepted.

REL = STYLESHEET

Let's look at the first link relation in our example page:

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

This is the most frequently used link relation in the world (literally). `<link rel="stylesheet">` is for pointing to CSS rules that are stored in a separate file. One small optimization you can make in HTML5 is to drop the `type` attribute. There's only one stylesheet language for the web, CSS, so that's the default value for the `type` attribute. This works in all browsers. (I suppose someone could invent a new stylesheet language someday, but if that happens, just add the `type` attribute back.)

```
<link rel="stylesheet" href="style-original.css" />
```

REL = ALTERNATE

Continuing with our example page:

```
<link rel="alternate"
      type="application/atom+xml"
      title="My Weblog feed"
      href="/feed/" />
```

This link relation is also quite common. `<link rel="alternate">`, combined with either the RSS or Atom media type in the `type` attribute, enables something called "feed autodiscovery." It allows syndicated feed readers (like Google Reader) to discover that a site has a news feed of the latest articles. Some browsers also support feed autodiscovery by displaying a special icon next to the URL. (Unlike with `rel="stylesheet"`, the `type` attribute matters here. Don't drop it!)

The `rel="alternate"` link relation has always been a strange hybrid of use cases, even in HTML 4. In HTML5, its definition has been clarified and extended to more accurately describe existing web content. As you just saw, using `rel="alternate"` in conjunction with `type=application/atom+xml` indicates an Atom feed for the current page. But you can also use `rel="alternate"` in conjunction with other `type` attributes to indicate the same content in another format, like PDF.

HTML5 also puts to rest a long-standing confusion about how to link to translations of documents. HTML 4 says to use the `lang` attribute in conjunction with `rel="alternate"` to specify the language of the linked document, but this is incorrect. The HTML 4 Errata document lists four outright errors in the HTML 4 specification. One of these outright errors is how to specify the language of a document linked with `rel="alternate"` The correct way, described in the HTML 4 Errata and now in HTML5, is to use the `hreflang` attribute. Unfortunately, these errata were never re-integrated into the HTML 4 spec, because no one in the W3C HTML Working Group was working on HTML anymore.

OTHER LINK RELATIONS IN HTML5

`rel="author"` is used to link to information about the author of the page. This can be a `mailto:` address, though it doesn't

have to be. It could simply link to a contact form or “about the author” page.

HTML 4 defined `rel="start"`, `rel="prev"`, and `rel="next"` to define relations between pages that are part of a series (like chapters of a book, or even posts on a blog). The only one that was ever used correctly was `rel="next"`. People used `rel="previous"` instead of `rel="prev"`; they used `rel="begin"` and `rel="first"` instead of `rel="start"`; they used `rel="end"` instead of `rel="last"`.

Oh, and — all by themselves — they made up `rel="up"` to point to a “parent” page. The best way to think of `rel="up"` is to look at your breadcrumb navigation (or at least imagine it). Your home page is probably the first page in your breadcrumbs, and the current page is at the tail end. `rel="up"` points to the next-to-last page in the breadcrumbs.

HTML5 includes `rel="next"` and `rel="prev"`, just like HTML 4, and supports `rel="previous"` for backward compatibility. The specification once had `rel="first"`, `rel="last"`, and `rel="up"`, too. However, “based on the lack of interest from implementors and users” the HTML Working Group decided to drop these values from the specification.

rel="icon" is the second most popular link relation, after `rel="stylesheet"`. It is usually found together with `shortcut`, like so:

```
<link rel="shortcut icon" href="/favicon.ico">
```

All major browsers support this usage to associate a small icon with the page. Usually it’s displayed in the browser’s location bar next to the URL, or in the browser tab, or both.

Also new in HTML5: the `sizes` attribute can be used in conjunction with the `icon` relationship to indicate the size of the referenced icon.

rel="license" was invented by the microformats community. It “indicates that the referenced document provides the copyright license terms under which the current document is provided.”

rel="nofollow" “indicates that the link is not endorsed by the original author or publisher of the page, or that the link to the referenced document was included primarily because of a commercial relationship between people affiliated with the two pages.” It was invented by Google and standardized within the microformats community. WordPress adds `rel="nofollow"` to links added by commenters. The thinking was that if “nofollow” links did not pass on PageRank, spammers would give up trying to post spam comments on weblogs. That didn’t happen, but `rel="nofollow"` persists.

rel="noreferrer" “indicates that no referrer information is to be leaked when following the link.” WebKit supports rel="noreferrer" so it works on Google Chome and Safari (and hopefully on other WebKit-based browsers). [rel="noreferrer" test case]

rel="prefetch" “indicates that preemptively fetching and caching the specified resource is likely to be beneficial, as it is highly likely that the user will require this resource.” Search engines sometimes add `<link rel="prefetch" href="URL of top search result">` to the search results page if they feel that the top result is wildly more popular than any other. For example: using Firefox, search Google for CNN, view the page source, and search for the keyword `prefetch`. Mozilla Firefox

is the only current browser that supports `rel="prefetch"`.

rel="search" "indicates that the referenced document provides an interface specifically for searching the document and its related resources." Specifically, if you want `rel="search"` to do anything useful, it should point to an OpenSearch document that describes how a browser could construct a URL to search the current site for a given keyword. OpenSearch (and `rel="search"` links that point to OpenSearch description documents) has been supported in Internet Explorer since version 7, Mozilla Firefox since version 2, and Google Chrome.

rel="tag" "indicates that the tag that the referenced document represents applies to the current document." Marking up "tags" (category keywords) with the `rel` attribute was invented by Technorati to help them categorize blog posts. Early blogs and tutorials thus referred to them as "Technorati tags." (You read that right: a commercial company convinced the entire world to add metadata that made the company's job easier. Nice work if you can get it!) The syntax was later standardized within the microformats community, where it was simply called `rel="tag"`. Most blogging systems that allow associating categories, keywords, or tags with individual posts will mark them up with `rel="tag"` links. Browsers do not do anything special with them; they're really designed for search engines to use as a signal of what the page is about.

❧

NEW SEMANTIC ELEMENTS IN HTML5

HTML5 is not just about making existing markup shorter (although it does a fair amount of that). It also defines new semantic elements.

`<section>` The `section` element represents a generic document or application section. A section, in this context, is a thematic grouping of content, typically with a heading. Examples of sections would be chapters, the tabbed pages in a tabbed dialog box, or the numbered sections of a thesis. A Web site's home page could be split into sections for an introduction, news items, contact information.

`<nav>` The `nav` element represents a section of a page that links to other pages or to parts within the page: a section with navigation links. Not all groups of links on a page need to be in a `nav` element — only sections that consist of major navigation blocks are appropriate for the `nav` element. In particular, it is common for footers to have a short list of links to common pages of a site, such as the terms of service, the home page, and a copyright page. The `footer` element alone is sufficient for such cases, without a `nav` element.

`<article>` The `article` element represents a component of a page that consists of a self-contained composition in a document, page, application, or site and that is intended to be independently distributable or reusable, e.g. in syndication. This could be a forum post, a magazine or newspaper article, a Web log entry, a user-submitted comment, an interactive widget or gadget, or any other independent item of content.

`<aside>` The `aside` element represents a section of a page that consists of content that is tangentially related to the content around the `aside` element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography. The element can be used for typographical effects

like pull quotes or sidebars, for advertising, for groups of `nav` elements, and for other content that is considered separate from the main content of the page.

`<hgroup>` The `hgroup` element represents the heading of a section. The element is used to group a set of `h1–h6` elements when the heading has multiple levels, such as subheadings, alternative titles, or taglines.

`<header>` The `header` element represents a group of introductory or navigational aids. A `header` element is intended to usually contain the section's heading (an `h1–h6` element or an `hgroup` element), but this is not required. The `header` element can also be used to wrap a section's table of contents, a search form, or any relevant logos.

`<footer>` The `footer` element represents a footer for its nearest ancestor sectioning content or sectioning root element. A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like. Footers don't necessarily have to appear at the end of a section, though they usually do. When the `footer` element contains entire sections, they represent appendices, indexes, long colophons, verbose license agreements, and other such content.

`<time>` The `time` element represents either a time on a 24 hour clock, or a precise date in the proleptic Gregorian calendar, optionally with a time and a time-zone offset.

`<mark>` The `mark` element represents a run of text in one document marked or highlighted for reference purposes.

I know you're anxious to start using these new elements, otherwise you wouldn't be reading this chapter. But first we need to take a little detour.

A LONG DIGRESSION INTO HOW BROWSERS HANDLE UNKNOWN ELEMENTS

Every browser has a master list of HTML elements that it supports. For example, Mozilla Firefox's list is stored in nsElementTable.cpp. Elements not in this list are treated as "unknown elements." There are two fundamental problems with unknown elements:

1. **How should the element be styled?** By default, `<p>` has spacing on the top and bottom, `<blockquote>` is indented with a left margin, and `<h1>` is displayed in a larger font. But what default styles should be applied to unknown elements?
2. **What should the element's DOM look like?** Mozilla's `nsElementTable.cpp` includes information about what kinds of other elements each element can contain. If you include markup like `<p><p>`, the second paragraph element implicitly closes the first one, so the elements end up as siblings, not parent-and-child. But if you write `<p><span>`, the `span` does not close the paragraph, because Firefox knows that `<p>` is a block element that can contain the inline element `<span>`. So, the `<span>` ends up as a child of the `<p>` in the DOM.

Different browsers answer these questions in different ways. (Shocking, I know.) Of the major browsers, Microsoft Internet Explorer's answer to both questions is the most problematic, but every browser needs a little bit of help here.

The first question should be relatively simple to answer: don't give any special styling to unknown elements. Just let them inherit whatever CSS properties are in effect wherever they appear on the page, and let the page author specify all styling with CSS. And that works, mostly, but there's one little gotcha you need to be aware of.

PROFESSOR MARKUP SAYS

All browsers render unknown elements inline, *i.e.* as if they had a `display:inline` CSS rule.

There are several new elements defined in HTML5 which are block-level elements. That is, they can contain other block-level elements, and HTML5-compliant browsers will style them as `display:block` by default. If you want to use these elements in older browsers, you will need to define the display style manually:

```
article,aside,details,figcaption,figure,
footer,header,hgroup,menu,nav,section {
    display:block;
}
```

(This code is lifted from Rich Clark's HTML5 Reset Stylesheet, which does many other things that are beyond the scope of this chapter.)

But wait, it gets worse! Prior to version 9, Internet Explorer did not apply *any* styling on unknown elements. For example, if you had this markup:

```
<style type="text/css">
  article { display: block; border: 1px solid red }
</style>
...
<article>
<h1>Welcome to Initech</h1>
<p>This is your <span>first day</span>.</p>
</article>
```

Internet Explorer (up to and including IE 8) will not treat the `<article>` element as a block-level element, nor will it put a red border around the article. All the style rules are simply ignored. Internet Explorer 9 fixes this problem.

The second problem is the DOM that browsers create when they encounter unknown elements. Again, the most problematic

browser is older versions of Internet Explorer (before version 9, which fixes this problem too). If IE 8 doesn't explicitly recognize the element name, it will insert the element into the DOM *as an empty node with no children*. All the elements that you would expect to be direct children of the unknown element will actually be inserted as siblings instead.

Here is some righteous ASCII art to illustrate the difference. This is the DOM that HTML5 dictates:

```
article
|
+--h1 (child of article)
|  |
|  +--text node "Welcome to Initech"
|
+--p (child of article, sibling of h1)
   |
   +--text node "This is your "
   |
   +--span
   |  |
   |  +--text node "first day"
   |
   +--text node "."
```

But this is the DOM that Internet Explorer actually creates:

```
article (no children)
h1 (sibling of article)
|
+--text node "Welcome to Initech"
p (sibling of h1)
|
+--text node "This is your "
|
+--span
|  |
|  +--text node "first day"
|
+--text node "."
```

There is a wonderous workaround for this problem. If you create a dummy `<article>` element with JavaScript before you use it in your page, Internet Explorer will magically recognize the `<article>` element and let you style it with CSS. There is no need to ever insert the dummy element into the DOM. Simply creating the element once (per page) is enough to teach IE to style the element it doesn't recognize.

```
<html>
<head>
<style>
  article { display: block; border: 1px solid red }
</style>
<script>document.createElement("article");</script>
```

```
</head>
<body>
<article>
<h1>Welcome to Initech</h1>
<p>This is your <span>first day</span>.</p>
</article>
</body>
</html>
```

This works in all versions of Internet Explorer, all the way back to IE 6! We can extend this technique to create dummy copies of all the new HTML5 elements at once — again, they're never inserted into the DOM, so you'll never see these dummy elements — and then just start using them without having to worry too much about non-HTML5-capable browsers.

Remy Sharp has done just that, with his aptly named HTML5 enabling script. The script has gone through more than a dozen revisions since I started writing this book, but this is the basic idea:

```
<!--[if lt IE 9]>
<script>
  var e = ("abbr,article,aside,audio,canvas,datalist,details," +
    "figure,footer,header,hgroup,mark,menu,meter,nav,output," +
    "progress,section,time,video").split(',');
  for (var i = 0; i < e.length; i++) {
    document.createElement(e[i]);
  }
</script>
<![endif]-->
```

The `<!--[if lt IE 9]>` and `<![endif]-->` bits are conditional comments. Internet Explorer interprets them like an `if` statement: "if the current browser is a version of Internet Explorer less than version 9, then execute this block." Every other browser will treat the entire block as an HTML comment. The net result is that Internet Explorer (up to and including version 8) will execute this script, but other browsers will ignore the script altogether. This makes your page load faster in browsers that don't need this hack.

The JavaScript code itself is relatively straightforward. The variable `e` ends up as an array of strings like `"abbr"`, `"article"`, `"aside"`, and so on. Then we loop through this array and create each of the named elements by calling `document.createElement()`. But since we ignore the return value, the elements are never inserted into the DOM. But this is enough to get Internet Explorer to treat these elements the way we want them to be treated, once we actually use them later in the page.

That "later" bit is important. This script needs to be at the top of your page, preferably in your `<head>` element, not at the bottom. That way, Internet Explorer will execute the script *before* it parses your tags and attributes. If you put this script at the bottom of your page, it will be too late. Internet Explorer will have already misinterpreted your markup and constructed the wrong DOM, and it won't go back and adjust it just because of this script.

Remy Sharp has "minified" this script and hosted it on Google Project Hosting. (In case you were wondering, the script itself is open source and MIT-licensed, so you can use it in any project.) If you like, you can even "hotlink" the script by pointing directly to the hosted version, like this:

```
<head>
```

```
  <meta charset="utf-8" />
  <title>My Weblog</title>
  <!--[if lt IE 9]>
  <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>
  <![endif]-->
</head>
```

Now we're ready to start using the new semantic elements in HTML5.

❧

HEADERS

Let's go back to our example page. Specifically, let's look at just the headers:

```
<div id="header">
  <h1>My Weblog</h1>
  <p class="tagline">A lot of effort went into making this
effortless.</p>
</div>

…

<div class="entry">
  <h2>Travel day</h2>
</div>

…

<div class="entry">
  <h2>I'm going to Prague!</h2>
</div>
```

There is nothing wrong with this markup. If you like it, you can keep it. It is valid HTML5. But HTML5 provides some additional semantic elements for headers and sections.

First off, let's get rid of that `<div id="header">`. This is a common pattern, but it doesn't mean anything. The `div` element has no defined semantics, and the `id` attribute has no defined semantics. (User agents are not allowed to infer any meaning from the value of the `id` attribute.) You could change this to `<div id="shazbot">` and it would have the same semantic value, *i.e.*, nothing.

HTML5 defines a `<header>` element for this purpose. The HTML5 specification has real-world examples of using the `<header>` element. Here is what it would look like on our example page:

```
<header>
  <h1>My Weblog</h1>
```

```
  <p class="tagline">A lot of effort went into making this effortless.</p>
  …
</header>
```

That's good. It tells anyone who wants to know that this is a header. But what about that tagline? Another common pattern, which up until now had no standard markup. It's a difficult thing to mark up. A tagline is like a subheading, but it's "attached" to the primary heading. That is, it's a subheading that doesn't create its own section.

Header elements like `<h1>` and `<h2>` give your page structure. Taken together, they create an outline that you can use to visualize (or navigate) your page. Screenreaders use document outlines to help blind users navigate through your page. There are online tools and browser extensions that can help you visualize your document's outline.

In HTML 4, `<h1>–<h6>` elements were the *only* way to create a document outline. The outline on the example page looks like this:

```
My Weblog (h1)
   |
   +--Travel day (h2)
   |
   +--I'm going to Prague! (h2)
```

That's fine, but it means that there's no way to mark up the tagline "A lot of effort went into making this effortless." If we tried to mark it up as an `<h2>`, it would add a phantom node to the document outline:

```
My Weblog (h1)
   |
   +--A lot of effort went into making this effortless. (h2)
   |
   +--Travel day (h2)
   |
   +--I'm going to Prague! (h2)
```

But that's not the structure of the document. The tagline does not represent a section; it's just a subheading.

Perhaps we could mark up the tagline as an `<h2>` and mark up each article title as an `<h3>`? No, that's even worse:

```
My Weblog (h1)
   |
   +--A lot of effort went into making this effortless. (h2)
         |
         +--Travel day (h3)
         |
         +--I'm going to Prague! (h3)
```

Now we still have a phantom node in our document outline, but it has "stolen" the children that rightfully belong to the root node. And herein lies the problem: HTML 4 does not provide a way to mark up a subheading without adding it to the document outline. No matter how we try to shift things around, "A lot of effort went into making this effortless" is going to end up in that graph. And that's why we ended up with semantically meaningless markup like `<p class="tagline">`.

HTML5 provides a solution for this: the `<hgroup>` element. The `<hgroup>` element acts as a wrapper for two or more *related* heading elements. What does "related" mean? It means that, taken together, they only create a single node in the document outline.

Given this markup:

```
<header>
  <hgroup>
    <h1>My Weblog</h1>
    <h2>A lot of effort went into making this effortless.</h2>
  </hgroup>
  …
</header>

…

<div class="entry">
  <h2>Travel day</h2>
</div>

…

<div class="entry">
  <h2>I'm going to Prague!</h2>
</div>
```

This is the document outline that is created:

```
My Weblog (h1 of its hgroup)
   |
   +--Travel day (h2)
   |
   +--I'm going to Prague! (h2)
```

You can test your own pages in the HTML5 Outliner to ensure that you're using the heading elements properly.

ARTICLES

Continuing with our example page, let's see what we can do about this markup:

```
<div class="entry">
  <p class="post-date">October 22, 2009</p>
  <h2>
    <a href="#"
```

```
        rel="bookmark"
        title="link to this post">
        Travel day
      </a>
    </h2>
    …
  </div>
```

Again, this is valid HTML5. But HTML5 provides a more specific element for the common case of marking up an article on a page — the aptly named `<article>` element.

```
  <article>
    <p class="post-date">October 22, 2009</p>
    <h2>
      <a href="#"
        rel="bookmark"
        title="link to this post">
        Travel day
      </a>
    </h2>
    …
  </article>
```

Ah, but it's not quite that simple. There is one more change you should make. I'll show it to you first, then explain it:

```
  <article>
    <header>
      <p class="post-date">October 22, 2009</p>
      <h1>
        <a href="#"
          rel="bookmark"
          title="link to this post">
          Travel day
        </a>
      </h1>
    </header>
    …
  </article>
```

Did you catch that? I changed the `<h2>` element to an `<h1>`, and wrapped it inside a `<header>` element. You've already seen the `<header>` element in action. Its purpose is to wrap all the elements that form the article's header (in this case, the article's publication date and title). But…but…but… shouldn't you only have one `<h1>` per document? Won't this screw up the document outline? No, but to understand why not, we need to back up a step.

In HTML 4, the *only* way to create a document outline was with the `<h1>`–`<h6>` elements. If you only wanted one root node in your outline, you had to limit yourself to one `<h1>` in your markup. But the HTML5 specification defines an algorithm for generating a document outline that incorporates the new semantic elements in HTML5. The HTML5 algorithm says that an `<article>` element creates a new section, that is, a new node in the document outline. And in HTML5, each section can

have its own <h1> element.

This is a drastic change from HTML 4, and here's why it's a good thing. Many web pages are really generated by templates. A bit of content is taken from one source and inserted into the page up here; a bit of content is taken from another source and inserted into the page down there. Many tutorials are structured the same way. "Here's some HTML markup. Just copy it and paste it into your page." That's fine for small bits of content, but what if the markup you're pasting is an entire section? In that case, the tutorial will read something like this: "Here's some HTML markup. Just copy it, paste it into a text editor, and fix the heading tags so they match the nesting level of the corresponding heading tags in the page you're pasting it into."

Let me put it another way. HTML 4 has no *generic* heading element. It has six strictly numbered heading elements, <h1>–<h6>, which must be nested in exactly that order. That kind of sucks, especially if your page is "assembled" instead of "authored." And this is the problem that HTML5 solves with the new sectioning elements and the new rules for the existing heading elements. If you're using the new sectioning elements, I can give you this markup:

```
<article>
  <header>
    <h1>A syndicated post</h1>
  </header>
  <p>Lorem ipsum blah blah…</p>
</article>
```

and you can copy it and paste it *anywhere in your page* without modification. The fact that it contains an <h1> element is not a problem, because the entire thing is contained within an <article>. The <article> element defines a self-contained node in the document outline, the <h1> element provides the title for that outline node, and all the other sectioning elements on the page will remain at whatever nesting level they were at before.

PROFESSOR MARKUP SAYS

As with all things on the web, reality is a little more complicated than I'm letting on. The new "explicit" sectioning elements (like <h1> wrapped in <article>) may interact in unexpected ways with the old "implicit" sectioning elements (<h1>–<h6> by themselves). Your life will be simpler if you use one or the other, but not both. If you must use both on the same page, be sure to check the result in the HTML5 Outliner and verify that your document outline makes sense.

❧

DATES AND TIMES

This is exciting, right? I mean, it's not "skiing down Mount Everest naked while reciting the Star Spangled Banner backwards" exciting, but it's pretty exciting as far as semantic markup goes. Let's continue with our example page. The next

line I want to highlight is this one:

```
<div class="entry">
  <p class="post-date">October 22, 2009</p>
  <h2>Travel day</h2>
</div>
```

Same old story, right? A common pattern — designating the publication date of an article — that has no semantic markup to back it up, so authors resort to generic markup with custom `class` attributes. Again, this is valid HTML5. You're not *required* to change it. But HTML5 doe provide a specific solution for this case: the `<time>` element.

```
<time datetime="2009-10-22" pubdate>October 22, 2009</time>
```

There are three parts to a `<time>` element:

1. A machine-readable timestamp
2. Human-readable text content
3. An optional `pubdate` flag

In this example, the `datetime` attribute only specifies a date, not a time. The format is a four-digit year, two-digit month, and two-digit day, separated by dashes:

```
<time datetime="2009-10-22" pubdate>October 22, 2009</time>
```

If you want to include a time too, add the letter `T` after the date, then the time in 24-hour format, then a timezone offset.

```
<time datetime="2009-10-22T13:59:47-04:00" pubdate>
  October 22, 2009 1:59pm EDT
</time>
```

(The date/time format is pretty flexible. The HTML5 specification contains examples of valid date/time strings.)

Notice I changed the text content — the stuff between `<time>` and `</time>` — to match the machine-readable timestamp. This is not actually required. The text content can be anything you like, as long as you provide a machine-readable date/timestamp in the `datetime` attribute. So this is valid HTML5:

```
<time datetime="2009-10-22">last Thursday</time>
```

And this is also valid HTML5:

```
<time datetime="2009-10-22"></time>
```

The final piece of the puzzle here is the `pubdate` attribute. It's a Boolean attribute, so just add it if you need it, like this:

```
<time datetime="2009-10-22" pubdate>October 22, 2009</time>
```

If you dislike "naked" attributes, this is also equivalent:

```
<time datetime="2009-10-22" pubdate="pubdate">October 22, 2009</time>
```

What does the `pubdate` attribute mean? It means one of two things. If the `<time>` element is in an `<article>` element, it means that this timestamp is the publication date of the article. If the `<time>` element is not in an `<article>` element, it means that this timestamp is the publication date of the entire document.

Here's the entire article, reformulated to take full advantage of HTML5:

```
<article>
  <header>
    <time datetime="2009-10-22" pubdate>
      October 22, 2009
    </time>
    <h1>
      <a href="#"
         rel="bookmark"
         title="link to this post">
         Travel day
      </a>
    </h1>
  </header>
  <p>Lorem ipsum dolor sit amet…</p>
</article>
```

❧

NAVIGATION

One of the most important parts of any web site is the navigation bar. CNN.com has "tabs" along the top of each page that link to the different news sections — "Tech," "Health," "Sports," &c. Google search results pages have a similar strip at the top of the page to try your search in different Google services — "Images," "Video," "Maps," &c. And our example page has a navigation bar in the header that includes links to different sections of our hypothetical site — "home," "blog," "gallery," and "about."

This is how the navigation bar was originally marked up:

```
<div id="nav">
  <ul>
    <li><a href="#">home</a></li>
    <li><a href="#">blog</a></li>
    <li><a href="#">gallery</a></li>
    <li><a href="#">about</a></li>
  </ul>
</div>
```

Again, this is valid HTML5. But while it's marked up as a list of four items, there is nothing about the list that tells you that it's part of the site navigation. Visually, you could guess that by the fact that it's part of the page header, and by reading the text of the links. But semantically, there is nothing to distinguish this list of links from any other.

Who cares about the semantics of site navigation? For one, people with disabilities. Why is that? Consider this scenario: your motion is limited, and using a mouse is difficult or impossible. To compensate, you might use a browser add-on that allows you to jump to (or jump past) major navigation links. Or consider this: if your sight is limited, you might use a dedicated program called a "screenreader" that uses text-to-speech to speak and summarize web pages. Once you get past the page title, the next important pieces of information about a page are the major navigation links. If you want to navigate quickly, you'll tell your screenreader to jump to the navigation bar and start reading. If you want to browse quickly, you might tell your screenreader to jump *over* the navigation bar and start reading the main content. Either way, being able to determine navigation links programmatically is important.

So, while there's nothing wrong with using `<div id="nav">` to mark up your site navigation, there's nothing particularly right about it either. It's suboptimal in ways that affect real people. HTML5 provides a semantic way to mark up navigation sections: the `<nav>` element.

```
<nav>
  <ul>
    <li><a href="#">home</a></li>
    <li><a href="#">blog</a></li>
    <li><a href="#">gallery</a></li>
    <li><a href="#">about</a></li>
  </ul>
</nav>
```

ASK PROFESSOR MARKUP

Q: Are skip links compatible with the `<nav>` element? Do I still need skip links in HTML5?

A: Skip links allow readers to skip over navigation sections. They are helpful for disabled users who use third-party software to read a web page aloud and navigate it without a mouse. (Learn how and why to provide skip links.)

Once screenreaders are updated to recognize the `<nav>` element, skip links will become obsolete, since the screenreader software will be able to automatically offer to skip over a navigation section marked up with the `<nav>` element. However, it will be a while before all the disabled users on the web upgrade to HTML5-savvy screenreader software, so you should continue to provide your own skip links to jump over `<nav>` sections.

FOOTERS

At long last, we have arrived at the end of our example page. The last thing I want to talk about is the last thing on the page: the footer. The footer was originally marked up like this:

```
<div id="footer">
  <p>&#167;</p>
  <p>&#169; 2001&#8211;9 <a href="#">Mark Pilgrim</a></p>
</div>
```

This is valid HTML5. If you like it, you can keep it. But HTML5 provides a more specific element for this: the `<footer>` element.

```
<footer>
  <p>&#167;</p>
  <p>&#169; 2001&#8211;9 <a href="#">Mark Pilgrim</a></p>
</footer>
```

What's appropriate to put in a `<footer>` element? Probably whatever you're putting in a `<div id="footer">` now. OK, that's a circular answer. But really, that's it. The HTML5 specification says, "A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like." That's what's in this example page: a short copyright statement and a link to an about-the-author page. Looking around at some popular sites, I see lots of footer potential.

- CNN has a footer that contains a copyright statement, links to translations, and links to terms of service, privacy, "about us," "contact us," and "help" pages. All totally appropriate `<footer>` material.
- Google has a famously sparse home page, but at the bottom of it are links to "Advertising Programs," "Business Solutions," and "About Google"; a copyright statement; and a link to Google's privacy policy. All of that could be wrapped in a `<footer>`.
- My weblog has a footer with links to my other sites, plus a copyright statement. Definitely appropriate for a `<footer>` element. (Note that the links themselves should *not* be wrapped in a `<nav>` element, because they are not site navigation links; they are just a collection of links to my other projects on other sites.)

"Fat footers" are all the rage these days. Take a look at the footer on the W3C site. It contains three columns, labeled "Navigation," "Contact W3C," and "W3C Updates." The markup looks like this, more or less:

```
<div id="w3c_footer">
  <div class="w3c_footer-nav">
    <h3>Navigation</h3>
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/standards/">Standards</a></li>
      <li><a href="/participate/">Participate</a></li>
      <li><a href="/Consortium/membership">Membership</a></li>
      <li><a href="/Consortium/">About W3C</a></li>
    </ul>
  </div>
```

```
    <div class="w3c_footer-nav">
      <h3>Contact W3C</h3>
      <ul>
        <li><a href="/Consortium/contact">Contact</a></li>
        <li><a href="/Help/">Help and FAQ</a></li>
        <li><a href="/Consortium/sup">Donate</a></li>
        <li><a href="/Consortium/siteindex">Site Map</a></li>
      </ul>
    </div>
    <div class="w3c_footer-nav">
      <h3>W3C Updates</h3>
      <ul>
        <li><a href="http://twitter.com/W3C">Twitter</a></li>
        <li><a href="http://identi.ca/w3c">Identi.ca</a></li>
      </ul>
    </div>
    <p class="copyright">Copyright © 2009 W3C</p>
  </div>
```

To convert this to semantic HTML5, I would make the following changes:

- Convert the outer `<div id="w3c_footer">` to a `<footer>` element.
- Convert the first two instances of `<div class="w3c_footer-nav">` to `<nav>` elements, and the third instance to a `<section>` element.
- Convert the `<h3>` headers to `<h1>`, since they'll now each be inside a sectioning element. The `<nav>` element creates a section in the document outline, just like the `<article>` element.

The final markup might look something like this:

```
<footer>
  <nav>
    <h1>Navigation</h1>
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/standards/">Standards</a></li>
      <li><a href="/participate/">Participate</a></li>
      <li><a href="/Consortium/membership">Membership</a></li>
      <li><a href="/Consortium/">About W3C</a></li>
    </ul>
  </nav>
  <nav>
    <h1>Contact W3C</h1>
    <ul>
      <li><a href="/Consortium/contact">Contact</a></li>
      <li><a href="/Help/">Help and FAQ</a></li>
      <li><a href="/Consortium/sup">Donate</a></li>
      <li><a href="/Consortium/siteindex">Site Map</a></li>
    </ul>
```

```
    </nav>
    <section>
      <h1>W3C Updates</h1>
      <ul>
        <li><a href="http://twitter.com/W3C">Twitter</a></li>
        <li><a href="http://identi.ca/w3c">Identi.ca</a></li>
      </ul>
    </section>
    <p class="copyright">Copyright © 2009 W3C</p>
  </footer>
```

❧

FURTHER READING

Example pages used throughout this chapter:

- Original (HTML 4)
- Modified (HTML5)

On character encoding:

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) by Joel Spolsky
- On the Goodness of Unicode, On Character Strings, and Characters vs. Bytes by Tim Bray

On enabling new HTML5 in Internet Explorer:

- How to style unknown elements in IE by Sjoerd Visscher
- HTML5 shiv by John Resig
- HTML5 enabling script by Remy Sharp
- The Story of the HTML5 Shiv by Paul Irish

On standards modes and doctype sniffing:

- Activating Browser Modes with Doctype by Henri Sivonen. This is the only article you should read on the subject. Any article on doctypes that doesn't reference Henri's work is guaranteed to be out of date, incomplete, or wrong.

HTML5-aware validator:

- html5.validator.nu

❧

This has been “What Does It All Mean?” The full table of contents has more if you’d like to keep reading.

DID YOU KNOW?

In association with Google Press, O’Reilly is distributing this book in a variety of formats, including paper, ePub, Mobi, and DRM-free PDF. The paid edition is called “HTML5: Up & Running,” and it is available now. This chapter is included in the paid edition.

If you liked this chapter and want to show your appreciation, you can buy “HTML5: Up & Running” with this affiliate link or buy an electronic edition directly from O’Reilly. You’ll get a book, and I’ll get a buck. I do not currently accept direct donations.