Covers Remote Scripting/AJAX

sitepoint

DHTML Utopia: Modern Web Design Using JavaScript & DOM

By Stuart Langridge

A Practical Step-by-Step Guide

DHTML Utopia
Modern Web Design Using JavaScript & DOM
(First 4 Chapters)

Summary of Contents of this Excerpt

Summary of Additional Book Contents

DHTML Utopia
Modern Web Design Using JavaScript & DOM

by Stuart Langridge

DHTML Utopia: Modern Web Design Using JavaScript & DOM

by Stuart Langridge

Managing Editor: Simon Mackie
Technical Director: Kevin Yank
Technical Editor: Simon Willison
Technical Editor: Nigel McFarlane
Editor: Georgina Laidlaw
Printing History:
First Edition: May 2005

Index Editor: Bill Johncocks
Cover Designer: Julian Carroll
Cover Illustrator: Lucas Licata

Latest Update: May 2005

Notice of Rights

Notice of Liability

Trademark Notice

sitepoint

Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood
VIC Australia 3066.
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 0–9579218–9–6
Printed and bound in the United States of America

About the Author

Stuart Langridge has been playing with the Web since 1994, and is quite possibly the only person in the world to have a BSc in Computer Science and Philosophy. He invented the term "unobtrusive DHTML," and has been a leader in the quest to popularize this new approach to scripting. When not working on the Web, he's a keen Linux user and part of the team at open-source radio show LUGRadio. He likes drinking decent beers, studying stone circles and other ancient phenomena, and trying to learn the piano. Stuart contributes to Stylish Scripting: SitePoint's DHTML and CSS Blog.

About The Technical Editors

Simon Willison is a seasoned Web developer from the UK, with a reputation for pioneering in the fields of CSS and DHTML. He specializes in both client- and server-side development, and recently became a member of the Web Standards project. Visit him at http://simon.incutio.com/, and at Stylish Scripting: SitePoint's DHTML and CSS Blog, to which he contributes.

Nigel McFarlane is the Mozilla community's regular and irregular technical commentator. He is the author of *Firefox Hacks* (O'Reilly Media) and *Rapid Application Development with Mozilla* (Prentice Hall PTR). When not working for SitePoint, Nigel writes for a number of trade publications and for the print media. He also consults to industry and government. Nigel's background is in science and technology, and in Web-enabled telecommunications software. He resides in Melbourne, Australia.

About The Technical Director

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters and blogs. He has written over 50 articles for SitePoint on technologies including PHP, XML, ASP.NET, Java, JavaScript and CSS, but is perhaps best known for his book, *Build Your Own Database Driven Website Using PHP & MySQL*, also from SitePoint. Kevin now lives in Melbourne, Australia. In his spare time he enjoys flying light aircraft and learning the fine art of improvised acting. Go you big red fire engine!

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for Web professionals. Visit http://www.sitepoint.com/ to access our books, newsletters, articles and community forums.

For Sam, who doesn't know what all this is about, but listens anyway.

Table of Contents

Introduction

In a single decade, the Web has evolved from a simple method of delivering technical documents to an essential part of daily life, making and breaking relationships and fortunes along the way. "Looking something up on the Internet," by which is almost always meant the Web, is now within reach of almost anyone living in a first-world country, and the idea of conducting conversations and business (and probably orchestras) in your Web browser is no longer foreign, but part of life.

As Joe Average grows more used to the technology, he demands more: more information, more ease-of-use, more functionality, more interactivity. And here we are, ready to provide, because he (and we) wants it, and because it's fun. (One of those fortunes mentioned earlier wouldn't go amiss, either.) As the Web becomes a major (if not *the* major) application development platform, there's a greater need to give Websites the flexibility and power that client-side applications can provide. More importantly, even the simplest Website can benefit from a little interactivity here and there—making it better, more responsive, or easier to use. HTML, the workhorse, manages some of this; CSS adds a few more tricks and a breadth of possibility for the designer. For true flexibility and interactivity, though, we need scripting.

Browser scripting has a long, albeit rather undistinguished, history. From the earliest popup boxes, through rollover images, and into scrolling status bars, it has provided the means to add that touch of the dynamic—even if it wasn't used for anything very exciting. But, all the while, a quiet movement was building. The JavaScript language was refined and made more powerful; the very building blocks of the Website were made available for manipulation; the real communicative strengths of the Web were given form and the potential for use. Modern scripting—DOM scripting—is a quantum leap away from the way things were.

In this book, I'll be explaining how you can get your hands dirty with all this juicy scripting goodness, and make your sites truly come alive. From the first moment in which you use JavaScript to examine the structure of the page that contains that JavaScript, a huge vista of potential really does open up before you. The techniques described in this book will help you make your sites more dynamic and more usable. They'll assist you to overcome browser limitations and add new functions, and occasionally, to do one or two cool things.

Who Should Read This Book?

This book is aimed at people who have built Websites before. Although I'll briefly cover HTML and CSS, you should already have experience working with these technologies. Some experience with JavaScript might also be useful, but it is by no means critical: modern scripting techniques are sometimes quite different than "old-style" JavaScript.

By the time you've read the whole book, you'll have a clear understanding of how to build your sites so that you can easily hook DHTML scripts into them; you'll know how to work in a cross-browser and cross-platform way; lastly, you will understand the power and flexibility that can be brought to your sites through DOM enhancements.

What's In This Book?

The book comprises ten chapters. The chapters do build on one another, so if this is your first time working with DOM techniques, you might want to read them in order. Once you have some experience with the DOM, hopping around to refresh your memory on various points may suit you best.

Chapter 1: *DHTML Technologies*
To successfully write DOM scripts, a few essential basics—which most readers of this book will already know—are required. In this first chapter, I'll quickly run through the essentials of HTML, CSS, and JavaScript. This chapter is worth reading, because it's critical for good scripting that your HTML and CSS are valid and well-structured; this chapter tells you what that means.

Chapter 2: *The Document Object Model*
DOM scripting requires a deep understanding of the DOM—the Document Object Model—itself. Everything else builds on this knowledge. In this chapter, I'll explain what the DOM is, how it can be manipulated, and what such manipulations make possible.

Chapter 3: *Handling DOM Events*
Events occur when the user does something with your HTML document: clicks a link, loads a page, or moves the mouse. In order to make your sites interactive—to react to user input—you will need to work with such events. Here, I explain what events are, show how to attach your code to them, and reveal some of the complexities inherent in DOM events.

Chapter 4: ***Detecting Browser Features***

Not every Web browser supports the features required to use DOM code effectively; those that do offer various levels of DOM support. Feature sniffing is the name given to a set of techniques that have been designed to ensure that your DOM code operates only in browsers that understand it; this eradicates situations in which your sites work—but not as you expected!—and avoids the dreaded JavaScript error box.

Chapter 5: ***Animation***

Animation can be a key to improving a site's usability; letting the user know when something's happening, or that something has changed, can enhance the user experience, and be of great value to your site's success. In this chapter, I describe how to add animation to your pages using DOM scripting techniques—and how to ensure that animation works across different browsers.

Chapter 6: ***Forms and Validation***

Any reasonably-sized Website will contain at least a few forms to collect user input. Scripting can provide some serious improvements to these forms: the validation of user input, ease-of-use for users, the collection of better feedback, and so on. Forms are built from HTML, like everything else, but the DOM can be said to apply to them more than it does to other elements, because forms have such a wide range of actions that you can manipulate in your scripts.

Chapter 7: ***Advanced Concepts and Menus***

In this chapter, we look at a more complex script: a multilevel animated dropdown menu. The chapter describes the code required to build such a script, pulling the techniques described in previous chapters together into a single, real-world example that demonstrates how much power the DOM provides, and how much easier it can be to work with than previous DHTML methods for achieving the same tasks.

Chapter 8: ***Remote Scripting***

While DOM scripting alone is an extremely useful tool, it can be made more powerful still with a little assistance from the server. In this chapter, we explore how your scripts can retrieve dynamic content from the server, and integrate that content with the site, eliminating the need for constant page refreshes.

Chapter 9: ***Communicating With The Server***

Communication with the server doesn't mean simply that the server hands out data. Your scripts can also pass data back, and engage in a real dialogue:

sending back a "something interesting has happened!" message can make your Websites work much more like real dynamic applications. This chapter enlarges on the previous one, describing the full power that server communication can create.

Chapter 10: ***DOM Alternatives: XPath***
JavaScript offers opportunities for more advanced work through its integration with other technologies. In this final chapter, I describe two of those integrations: using XPath to work with XML, and integrating your DOM scripts with Flash.

Whither XHTML?

Some people may wonder why all the examples in this book are HTML 4.01 Strict. "Why are you using HTML?" they ask. "Why not XHTML? It's all, y'know, XML and stuff! It *must* be better."

There is a reason: using XHTML can cause a lot of upgrade issues, particularly with the DHTML that we use in this book.

If you choose XHTML, then you're placed in a "complete upgrade or do nothing" position. When XHTML is served to an ordinary browser, that browser will treat your lovely XML-compliant XHTML as perfectly ordinary HTML, unless you make a special effort to do things differently. XHTML treated as ordinary HTML removes all the supposed benefits of XHTML; it's not checked for well-formedness by the browser, for example.

The special effort that you need to make is to change the MIME type with which your Web server serves your XHTML document. By default, Web servers will serve it as `text/html`, which means that it will be treated as "tag soup" HTML, without enjoying any of the XHTML benefits, as mentioned above. Moreover, Ian "Hixie" Hickson, who's part of both the Mozilla and Opera teams as well as the CSS working group, has laid out a set of objections[1] which states that XHTML should not be served as `text/html` at all.

In order to have a browser treat your XHTML as XHTML (and thence as XML), rather than as tag soup, it must be served with MIME type `application/xhtml+xml`. Unfortunately, Internet Explorer (for one, and it's not alone) does not support XHTML documents served as `application/xhtml+xml`; it will

[1] http://www.hixie.ch/advocacy/xhtml

give you a “download this document” dialog rather than displaying it in the browser. That’s a disaster for most Web pages.

It’s possible to have the Web server detect whether the user’s browser can cope with `application/xhtml+xml` and serve with an appropriate MIME type: `text/html` for those browsers that do not support `application/xhtml+xml`. (Remember that serving XHTML as `text/html` is wrong, according to Hixie’s objections above.) But, even in those browsers that do support `application/xhtml+xml`, and therefore parse your XHTML document as it should be parsed, there are still other problems that take some getting around.

Here are a few examples. CSS in properly-parsed XHTML documents works differently: selectors are case-sensitive, and setting backgrounds and the like on the `body` doesn’t propagate those styles up to the document as it does in HTML (the styles must be set on `html` instead).

Most importantly for *this* book, XHTML makes using DOM scripting pretty awkward. The HTML collections `document.images`, `document.forms`, `document.links`, and so on, do not exist in many browsers’ implementations of the XHTML DOM. Arguably, one should avoid using these anyway in preparation for XHTML later. Instead, you must use `document.getElementsByTagName` appropriately. The element names in the DOM are also case-sensitive (and always lowercase, since XML element names are lowercase and XHTML is XML). That can be a bit of coding style trap. You also can’t use `document.write` at all, although you probably should avoid it anyway, for reasons I’ll explain in this book.

These are not major problems, and if you’re into standards then most of these issues won’t affect your code anyway, but a final issue remains: you can’t use `document.createElement` to create new elements with the DOM. Instead, because XHTML is XML, and therefore supports namespaces, you must create each element specifically within the XHTML namespace. So, instead of using `document.createElement('a')`, to create a new `a` element, you must use `document.createElementNS('http://www.w3.org/1999/xhtml', 'a')`.

Of course, you must only use `document.createElementNS` when your document is being parsed as XHTML—not when it’s being parsed as HTML (as in Internet Explorer)—so you’ll need to detect which case you’re dealing with, and change what the script does appropriately.

In short, using XHTML right now provides very little in the way of benefits, but brings with it a fair few extra complications. HTML 4.01 Strict is just as “valid” as XHTML—XHTML did not replace HTML but sits alongside it. It’s just as easy to validate an HTML 4.01 page as it is to validate an XHTML page. I’ve used HTML 4.01 Strict for all the examples in this book, and I recommend that you use it, too.

Mark Pilgrim has written in more detail about using XHTML[2] and the problems that lie therein. For this book, we're sticking with tried-and-true HTML 4.01.

The Book's Website

Located at http://www.sitepoint.com/books/dhtml1/, the Website supporting this book will give you access to the following facilities:

The Code Archive

As you progress through the text, you'll note that most of the code listings are labelled with filenames, and a number of references are made to the code archive. This is a downloadable ZIP archive that contains complete code for all the examples presented in this book.

Updates and Errata

The Errata page on the book's Website will always have the latest information about known typographical and code errors, and necessary updates for changes to technologies.

The SitePoint Forums

While I've made every attempt to anticipate any questions you may have, and answer them in this book, there is no way that *any* book could cover everything there is to know about DHTML. If you have a question about anything in this book, the best place to go for a quick answer is http://www.sitepoint.com/forums/—SitePoint's vibrant and knowledgeable community.

The SitePoint Newsletters

In addition to books like this one, SitePoint offers free email newsletters.

[2] http://www.xml.com/pub/a/2003/03/19/dive-into-xml.html

The SitePoint Tech Times covers the latest news, product releases, trends, tips, and techniques for all technical aspects of Web development. The long-running *SitePoint Tribune* is a biweekly digest of the business and moneymaking aspects of the Web. Whether you're a freelance developer looking for tips to score that dream contract, or a marketing major striving to keep abreast of changes to the major search engines, this is the newsletter for you. *The SitePoint Design View* is a monthly compilation of the best in Web design. From new CSS layout methods to subtle PhotoShop techniques, SitePoint's chief designer shares his years of experience in its pages.

Browse the archives or sign up to any of SitePoint's free newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can't find your answer through the forums, or you wish to contact me for any other reason, the best place to write is `books@sitepoint.com`. We have a well-manned email support system set up to track your inquiries, and if our support staff are unable to answer your question, they send it straight to me. Suggestions for improvement as well as notices of any mistakes you may find are especially welcome.

Acknowledgements

The two Simons, Simon Mackie, my editor, and Simon Willison, my expert reviewer, deserve quite an enormous vote of thanks. This book would not be anywhere near as good as it is without them.

I'd also like to raise a hand to the Web development community: there are people everywhere diving into these new technologies with gusto, establishing guidelines, making discoveries, and revealing hitherto unsuspected truths about how cool all this stuff is. Keep it up. We're fixing the world, and I'm proud to be a part of it.

1 DHTML Technologies

The White Rabbit put on his spectacles. 'Where shall I begin, please your Majesty?' he asked. 'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'
—Lewis Carroll, *Alice's Adventures in Wonderland*

Dynamic HTML, called DHTML for short, is the name given to a set of Web development techniques that are mostly used in Web pages that have non-trivial user-input features. DHTML means manipulating the Document Object Model of an HTML document, fiddling with CSS directives in style information, and using client-side JavaScript scripting to tie everything together.

In this introductory chapter, I'll provide a brief overview of some of the things you'll need to know about: the building blocks that make up DHTML Websites. You'll find it useful reading if you need to refresh your memory. If you already know all these details, you might want to flick through the chapter anyway; you may even be a little surprised by some of it. In the coming pages, we'll come to understand that DHTML is actually a combination of proper HTML for your content, Cascading Style Sheets for your design, and JavaScript for interactivity. Mixing these technologies together can result in a humble stew or a grandiose buffet. It's all in the art of cooking, so let's start rattling those pots and pans!

HTML Starting Points

Websites are written in HTML. If you're reading this book, you'll almost certainly know what HTML is and will probably be at least somewhat experienced with it. For a successful DHTML-enhanced Website, it's critical that your HTML is two things: valid and semantic. These needs may necessitate a shift away from your previous experiences writing HTML. They may also require a different approach than having your preferred tools write HTML for you.

Step up to Valid HTML

A specific set of rules, set out in the HTML recommendation[1], dictate how HTML should be written. HTML that complies with these rules is said to be "valid." Your HTML needs to be valid so that it can be used as a foundation on which you can build DHTML enhancements. While the set of rules is pretty complex, you can ensure that your HTML is valid by following a few simple guidelines.

Correctly Nest Tags

Don't let tags "cross over" one another. For example, don't have HTML that looks like the snippet shown below:

```
Here is some <strong>bold and <em>italic</strong> text</em>.
```

Here, the `<strong>` and `<em>` tags cross over one another; they're incorrectly nested. Nesting is extremely important for the proper use of DHTML. In later chapters of this book, we'll study the DOM tree, and the reasons why incorrect nesting causes problems will become clear. For now, simply remember that if you cross your tags, each browser will interpret your code in a different way, according to different rules (rather than according to the standard). Any hope of your being able to control the appearance and functionality of your pages across browsers goes right out the window unless you do this right.

Close Container Tags

Tags such as `<strong>` or `<p>`, which contain other items, should always be closed with `</strong>` or `</p>`, or the appropriate closing tag. It's important to know which tags contain things (e.g. text or other tags) and to make sure you close

[1] http://www.w3.org/TR/html4/

them. `<p>`, for example, doesn't mean "put a paragraph break here," but "a paragraph begins here," and should be paired with `</p>`, "this paragraph ends here."[1] The same logic applies to `<li>` tags as well.

Always Use a Document Type

A document type (or DOCTYPE) describes the dialect of HTML that's been used; there are several different options. In this book, we'll use the dialect called HTML 4.01 Strict.[2] Your DOCTYPE, which should appear at the very top of every HTML page, should look like this:

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

That information can be typed on a single line, or with a line break after `EN"`. Don't worry, for the moment, about what this means: just be sure to place it at the top of every page. The article *Fix Your Site With the Right DOCTYPE!*[2], published on A List Apart[3], lists all the DOCTYPEs you might want to use, and why you'd need to use them at all. I visit that article all the time to cut and paste the one I need!

Validate your Page

The most important page creation step is to check that your HTML is valid. There are numerous tools that you can download and run on your own computer to test your code's validity—some HTML editors even have such tools built in—or you can use one of the many online validators, the most common of which is the W3C's own validator[4]. A validator will tell you how you need to adjust your HTML in order to make it compatible with DHTML techniques. The ultimate reference for what constitutes valid HTML is the HTML recommendation[5]. It's complex and detailed, but if you have any questions about how HTML should be written, or whether a tag really exists, you'll find the answers there. As mentioned above, browsers rely on a standard that describes how validated HTML

[1]Those who know what they're doing with container tags will be aware that HTML 4.01 does not actually require that all container tags are closed (though XHTML still does). However, it's never invalid to close a container tag, though it is sometimes invalid to not do so. It's considerably easier to just close everything than it is to remember which tags you're allowed to leave open.
[2]If you're thinking, "but I want to use XHTML!" then I bet you already know enough about DOCTYPEs to use them properly.
[2] http://www.alistapart.com/articles/doctype/
[3] http://www.alistapart.com/
[4] http://validator.w3.org/
[5] http://ww.w3.org/TR/html4/

should be interpreted. However, there are no standards to describe how invalid HTML should be interpreted; each browser maker has established their own rules to fill that gap. Trying to understand each of these rules would be difficult and laborious, and you have better things to do with your time. Sticking to valid HTML means that any problems you find are deemed to be bugs in that browser—bugs that you may be able to work around. Thus, using valid HTML gives you more time to spend with your family, play snooker, etc. which, if you ask me, is a good reason to do it.

Step up to Semantic HTML

In addition to its validity, your HTML should be semantic, not presentational. What this means is that you should use HTML tags to describe the nature of an element in your document, rather than the appearance of that element. So don't use a `<p>` tag if you mean, "put a blank line here." Use it to mean, "a paragraph begins here" (and place a `</p>` at the end of that paragraph). Don't use `<blockquote>` to mean, "indent this next bit of text." Use it to mean, "this block is a quotation." If you mark up your HTML in this way, you'll find it much easier to apply DHTML techniques to it further down the line. This approach is called **semantic markup**—a fancy way of saying, "uses tags to describe meaning."

Let's look at a few example snippets. First, imagine your Website has a list of links to different sections. That list should be marked up on the basis of what it is: a list. Don't make it a set of `<a>` tags separated by `<br>` tags; it's a list, so it should be marked up as such, using `<ul>` and `<li>` tags. It might look something like this:

```
<ul>
  <li><a href="index.html">Home</a></li>
  <li><a href="about.html">About this Website</a></li>
  <li><a href="email.html">Contact details</a></li>
</ul>
```

You'll find yourself using the `<ul>` tag a lot. Many of the items within a Website are really lists: a breadcrumb trail is a list of links, a menu structure is a list of lists of links, and a photo gallery is a list of images.

Similarly, if your list contains items with which comments are associated, maybe it should be marked up as a definition list:

```
<dl>
  <dt><a href="index.html">Home</a></dt>
    <dd>Back to the home page</dd>
  <dt><a href="about.html">About this Website</a></dt>
    <dd>Why this site exists, how it was set up, and who did it
    </dd>
  <dt><a href="email.html">Contact details</a></dt>
    <dd>Getting in contact with the Webmaster: email addresses
      and phone numbers</dd>
</dl>
```

Remember: the way your page looks isn't really relevant. The important part is that the information in the page is marked up in a way that describes what it is. There are lots of tags in HTML; don't think of them as a way to lay out information on your page, but as a means to define what that information means.

If you don't use HTML to control the presentation of your pages, how can you make them look the way you want them to? That's where Cascading Style Sheets come in.

Adding CSS

Cascading Style Sheets (CSS) is a technique that allows you to describe the presentation of your HTML. In essence, it allows you to state how you want each **element** on your page to look. An element is a piece of HTML that represents one thing: one paragraph, one heading, one image, one list. Elements usually correspond to a particular tag and its content. When CSS styles are used, DHTML pages can work on the appearance and the content of the page independently. That's a handy and clean separation. If you want to look good, you need to learn how to dress up *and* go to the gym regularly!

A Simple CSS Example

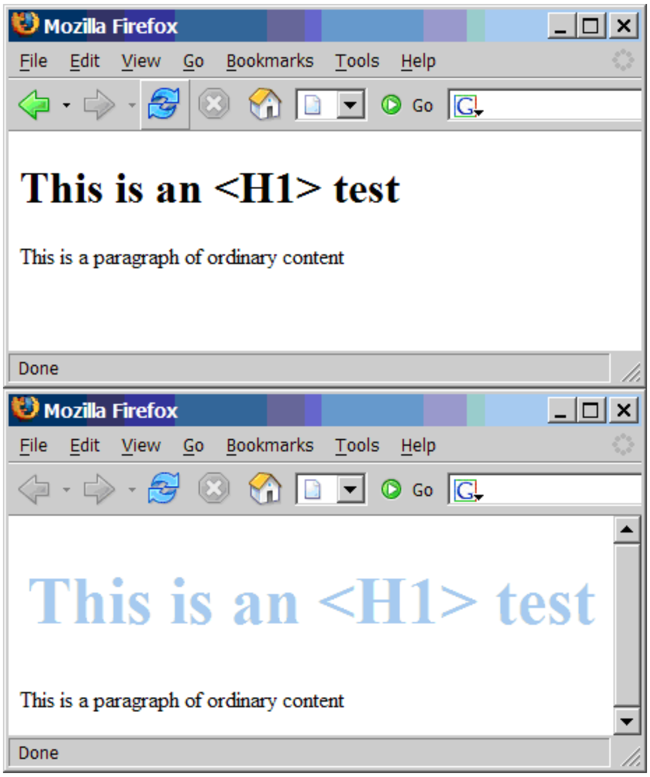
Imagine you want your main page heading (an `<h1>` tag) to be displayed in big, red, centered text. You should specify that in your style sheet as follows:

```
h1 {
  font-size: 300%;
  color: #FF0000;
  text-align: center;
}
```

See the section called "Further Reading" at the end of this chapter for some links to introductory tutorials on CSS, which should help if the above lines don't make a lot of sense to you.

Here's a simple HTML page before and after these styles have been applied:

Figure 1.1. That HTML's stylin'!


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Done
Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Done

This is an <H1> test

This is a paragraph of ordinary content

This is an <H1> test

This is a paragraph of ordinary content

The key point here is to remove the presentation aspects from your HTML and put them into your style sheet. If , for example, you made your page heading bigger by putting `<font>` tags in your HTML, then you'd need to paste those tags into every page on which a header was used. By making your HTML semantic and moving the page's presentation into CSS, you can control the look of headings across the whole site through a single style sheet. This makes your job as Website developer much easier.

Of course, it's not quite as easy as that. Although the full definition of CSS allows you to do some fairly amazing things, and to control the presentation of your pages to a high degree, not every browser supports everything that CSS has to offer.

In order to know about the differences in browser support for CSS, you need to know what CSS can do. There are two sorts of browser incompatibilities: things that a given browser doesn't implement, and things that it implements incorrectly. Occasionally, browsers add their own "special features" as well, but we won't be worried about those in this book.

Missing implementations are relatively easy to deal with: don't rely on such rules if you want your CSS to work in browsers that have failed to implement them. This can be a pain, especially since the most commonly used browser in the world, Internet Explorer for Windows, has some serious holes in its CSS support; however, this "solution" is often a necessary compromise. Learning which rules you can and can't use is one of the steps on the path to CSS guru-hood.

Badly implemented standards are a bigger problem. In such cases, the browser gets it wrong. Another step to CSS guru-hood is understanding exactly what each browser does wrong, and how you can work around those failings. You don't need that knowledge to start with, though: you'll pick it up as you go along. Workarounds for CSS bugs in different browsers are usually achieved using CSS **hacks**. These hacks take advantage of the bugs in a browser's CSS parser to deliver it specific style sheet directives that work around its poor implementation of the standards. A huge variety of these CSS hacks is documented for each browser in various places around the Web; see the section called "Further Reading" for more.

Learning to understand and adapt to the vagaries of CSS handling in various browsers is part of the work that's required to use CSS effectively. While it can be a lot of work, many CSS bugs only become apparent with the complex use of this technology; most CSS is handled perfectly across platforms and browsers without the need for hacks or complex tests.

While CSS is powerful, it doesn't quite give us true flexibility in presentation. The capabilities of CSS increase all the time, and more "interactive" features are constantly being added to the CSS specification. However, it's not designed for building truly interactive Websites. For that, we need the final building block of DHTML: JavaScript.

Adding JavaScript

JavaScript is a simple but powerful programming language. It's used to add dynamic behavior to your Website—the D in DHTML. HTML defines the page's structure, and CSS defines how it looks, but actions, the things that happen when you interact with the page—by clicking a button, dragging an image, or moving the mouse—are defined in JavaScript. JavaScript works with the Document Object Model, described in the next chapter, to attach actions to different events (mouseovers, drags, and clicks). We're not going to describe all the gory JavaScript syntax in detail here—the section called "Further Reading" has some links to a few JavaScript tutorials if you need them.

A Simple JavaScript Example

Here's a simple piece of JavaScript that converts a text field's value to uppercase when the user tabs out of the field. First let's see the old, bad way of doing it:

File: **oldlisteners.html** **(excerpt)**

```
<input id="street" type="text"
    onchange="this.value = this.value.toUpperCase();">
```

In this book, we'll recommend a more modern technique. First, the HTML:

File: **newlisteners.html** **(excerpt)**

```
<input id="street" type="text">
```

Second, the JavaScript, which is usually located in the `<head>` part of the page:

File: **newlisteners.html** **(excerpt)**

```
<script type="text/javascript">
function uppercaseListener() {
  this.value = this.value.toUpperCase();
}

function installListeners() {
  var element = document.getElementById('street');
  element.addEventListener('change', uppercaseListener, false);
}

window.addEventListener('load', installListeners, false);
</script>
```

The first function does the work of converting the text. The second function makes sure that the first is connected to the right HTML tag. The final line performs this connection once the page has loaded in full. Although this means more code, notice how it keeps the HTML content clean and simple. In future chapters, we'll explore this kind of approach a lot. Don't worry about the mechanics too much for now—there's plenty of time for that!

Get Some Tools!

A good JavaScript development environment makes working with JavaScript far easier than it would otherwise be. Testing pages in Internet Explorer (IE) can leave something to be desired; if your page generates JavaScript errors (as it will do all the time while you're building it!), IE isn't likely to be very helpful at diagnosing where, or what, they are. The most useful, yet simple, tool for JavaScript debugging is the JavaScript Console in Mozilla or Mozilla Firefox. This console will clearly display where any JavaScript error occurs on your page, and what that error is. It's an invaluable tool when building JavaScript scripts. Mozilla Firefox works on virtually all platforms, and it's not a big download; it also offers better support for CSS than Internet Explorer, and should be part of your development toolkit. Beyond this, there's also the JavaScript debugger in Mozilla, which is named Venkman; if you're the sort of coder who has worked on large projects in other languages and are used to a debugger, Venkman can be useful, but be aware that it takes a bit of setting up. In practice, though, when you're enhancing your site with DHTML, you don't need anything as complex as a debugger; the JavaScript Console and judicious use of alert statements to identify what's going on will help you through almost every situation.

Another tool that's definitely useful is a good code editor in which to write your Website. Syntax highlighting for JavaScript is a really handy feature; it makes your code easier to read while you're writing it, and quickly alerts you when you leave out a bracket or a quote. Editors are a very personal tool, and you might have to kiss a fair few frogs before you find your prince in this regard, but a good editor will seriously speed and simplify your coding work. Plenty of powerful, customizable editors are available for free, if you don't already have a preferred program. But, if you're currently writing code in Windows Notepad, have a look at what else is available to see if any other product offers an environment that's more to your liking. You'll want syntax highlighting, as already mentioned; a way to tie in the external validation of your pages is also useful. Textpad[6] and Crimson Editor[7] are free, Windows-based editors that cover the basics if you're developing on a Windows platform; Mac users tend to swear by BBEdit[8]; Linux users have gedit or Kate or vim to do the basics, and there's always Emacs.

JavaScript is the engine on which DHTML runs. DHTML focuses on manipulating your HTML and CSS to make your page do what the user wants, and it's Java-

[6] http://www.textpad.com/
[7] http://www.crimsoneditor.com/
[8] http://www.barebones.com/

Script that effects that manipulation. Through the rest of this book, we'll explore that manipulation in more and more detail.

Further Reading

Try these links if you're hungry for more on CSS itself.

http://www.sitepoint.com/article/css-is-easy
SitePoint's easy introduction to the world of CSS is a great place to start.

http://www.w3schools.com/css/
W3Schools' CSS tutorials are helpful whether you're learning, or simply brushing up on your knowledge of CSS.

http://www.csszengarden.com/
The CSS Zen Garden is a marvelous demonstration of the power of Cascading Style Sheets alone. It has a real wow factor!

http://centricle.com/ref/css/filters/
This comprehensive list of CSS hacks shows you which browsers will be affected by a given hack, if you need to hide certain CSS directives (or deliver certain directives) to a particular browser.

http://www.positioniseverything.net/
This site demonstrates CSS issues in various browsers and explains how to work around them.

http://www.css-discuss.org/
The CSS-Discuss mailing list is "devoted to talking about CSS and ways to use it in the real world; in other words, practical uses and applications." The associated wiki[15] is a repository of useful tips and tricks.

http://www.sitepoint.com/books/
If you're after something more definitive, SitePoint's book, *HTML Utopia: Designing Without Tables Using CSS[17]* is a complete guide and reference for the CSS beginner. *The CSS Anthology: 101 Tips, Tricks & Hacks[18]* is a perfect choice if you prefer to learn by doing.

[15] http://css-discuss.incutio.com/
[17] http://www.sitepoint.com/books/css1/
[18] http://www.sitepoint.com/books/cssant1/

A lot of tutorials on the Web cover JavaScript. Some explore both DHTML and the DOM, while others do not; you should try to find the former.

http://www.sitepoint.com/article/javascript-101-1
This tutorial provides an introduction to the basics of JavaScript for the total non-programmer. Some of the techniques presented in this article aren't as modern as the alternatives presented in this book, but you'll get a good feel for the language itself.

http://www.quirksmode.org/
Peter-Paul Koch's list of JS techniques and scripts covers a considerable amount of ground in this area.

Summary

In this chapter, we've outlined the very basic building-blocks of DHTML: what HTML really is, how to arrange and display it in your documents using CSS, and how to add interactivity using JavaScript. Throughout the rest of this book, we'll look at the basic techniques you can use to start making your Websites dynamic, then move on to discuss certain advanced scripting techniques that cover specific areas. On with the show!

2 The Document Object Model

One day someone came in and observed, on the paper sticking out of one of the Teletypes, displayed in magnificent isolation, this ominous phrase:

```
values of β will give rise to dom!
```

…the phrase itself was just so striking! Utterly meaningless, but it looks like what… a warning? What is "dom?"
—Dennis M. Richie[1]

A Web page is a document. To see that document, you can either display it in the browser window, or you can look at the HTML source. It's the same document in both cases. The World Wide Web Consortium's Document Object Model (DOM) provides another way to look at that same document. It describes the document content as a set of objects that a JavaScript program can see. Naturally, this is very useful for DHTML pages on which a lot of scripting occurs. (The quote above is a pure coincidence—it's from the days before the Web!)

According to the World Wide Web Consortium[2], "the Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of doc-

[1] http://cm.bell-labs.com/cm/cs/who/dmr/odd.html
[2] http://www.w3.org/DOM/#what

uments. The document can be further processed and the results of that processing can be incorporated back into the presented page." This statement basically says that the DOM is not just a novelty—it is useful for doing things. In the coming pages, we'll take a brief look at the history of the DOM before investigating more deeply what it is and how we can use it. We'll finish up with some example scripts that demonstrate the power of this critical aspect of DHTML.

The Origins of the DOM

In Netscape Navigator 2, Netscape Communications introduced JavaScript (briefly called LiveScript), which gave Web developers scripting access to elements in their Web pages—first to forms, then, later, to images, links, and other features. Microsoft implemented JavaScript in Internet Explorer 3 (although they called it JScript) in order to keep up with Netscape.

By version 4, the two browsers had diverged significantly in terms of their respective feature sets and the access they provided to page content. Each browser manufacturer implemented its own proprietary means of providing scripting access to layers. Scripts that wanted to work in both browsers needed to contain code for each method. The ill-fated "browser wars" were all about these proprietary extensions to the Web, as each manufacturer strove to attract more developers to its platform through the lure of new features. There was little regard for cross-browser compatibility, although Microsoft copied and supported most of the early innovations made by Netscape.

While all this was taking place, the W3C developed a specification for the Document Object Model Level 1, which outlined a generic and standard method to access the various parts of an XML document using script. Since HTML can be thought of as a dialect of XML, the DOM Level 1 spec applied to HTML as well.

Both major browser manufacturers implemented the DOM Level 1 specification: in Internet Explorer 5 and in Netscape 6. The previously existing proprietary specifications were retrospectively titled; since the new standard was DOM Level 1, those old and now deprecated methods were called DOM Level 0. (Since then, the W3C has also released the DOM Level 2 and DOM Level 3 specifications, which add more features and are broken into separate modules.) There's no formal DOM Level 0 standard, though.

What is the DOM?

So, you know what the DOM *used* to be. Now let's discuss what it *is*.

Essentially, the DOM provides access to the structure of an HTML page by mapping the elements in that page to a tree of nodes. Each element becomes an element node, and each bit of text becomes a text node. Take this HTML snippet, for example:

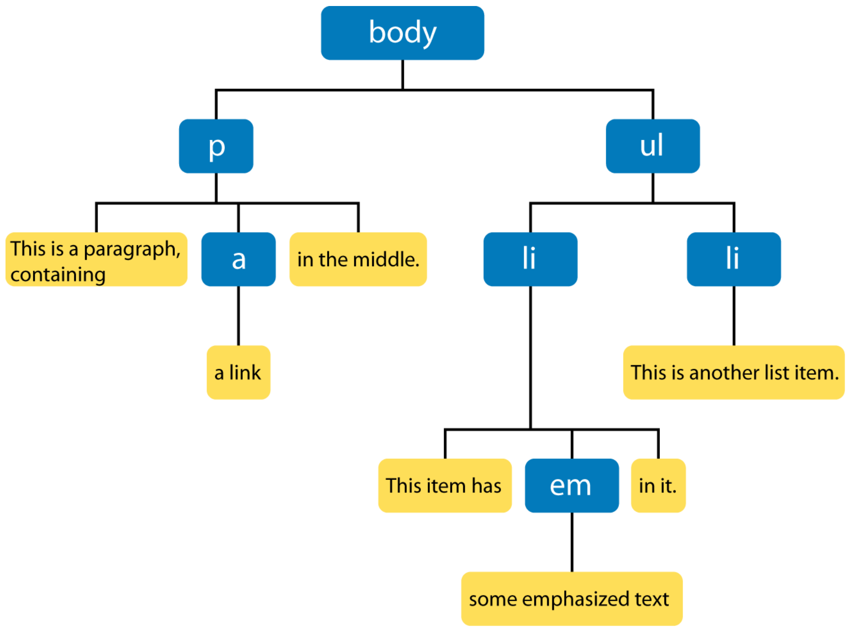
```
<body>
  <p>
    This is a paragraph, containing
    <a href="#">
      a link
    </a>
    in the middle.
  </p>
  <ul>
    <li>
      This item has
      <em>
        some emphasized text
      </em>
      in it.
    </li>
    <li>
      This is another list item.
    </li>
  </ul>
</body>
```

I added lots of extra indenting so that you can compare this snippet with the matching DOM tree. Don't do that in real life—I'm just trying to make things clearer in this case. The matching DOM tree is shown in Figure 2.1.

As you can see, the `a` element, which is located inside the `p` element in the HTML, becomes a **child node**, or just **child**, of the `p` node in the DOM tree. (Symmetrically, the `p` node is the **parent** of the `a` node. The two `li` nodes, children of the same parent, are called **sibling nodes** or just **siblings**.)

Notice that the nesting level of each tag in the HTML markup matches the number of lines it takes to reach the same item in the DOM tree. For example, the `<a>` tag is nested twice inside other tags (the `<p>` and `<body>` tags), so the `a` node in the tree is located two lines from the top.

Figure 2.1. An example of a DOM tree.



The Importance of Valid HTML

From this last example, we can see more clearly why valid HTML, including properly nested elements, is important. If elements are improperly nested, problems arise. Take the following line:

```
<strong>These <em>elements are</strong> badly nested</em>.
```

The DOM tree that results from this incorrectly nested code won't be a tree at all: it would need to be malformed in order to express the invalid element layout that this HTML requests. Each browser fixes malformed content in a different way, which can generate such horrors as an element that is its own parent node. Keeping your HTML valid avoids all these problems.

Walking DOM Trees

Trees of nodes turn up a lot in computing, because, among other things, they have a very useful property: it's easy to "walk the tree" (that is, to iterate through every one of the tree's nodes in order) with very little code. Walking a tree is easy because any element node can be considered as the top of its own little tree. Therefore, to walk through a tree, you can use a series of steps, for example:

1. Do something with the node we're looking at
2. Does this node have children? If so:
3. For each of the child nodes, go to step 1

This process is known as **recursion**, and is defined as the use of a function that calls itself. Each child is the same type of thing as the parent and can therefore be handled in the same way. We don't do much with recursion ourselves, but we rely quite heavily on the browser recursing through the page's tree. It's especially useful when it comes time to work with events, as we'll see in Chapter 3.

Finding the Top of the Tree

In order to walk the DOM tree, you need a reference to the node at its top: the root node. That "reference" will be a variable that points to the root node. The root node should be available to JavaScript as `document.documentElement`. Not all browsers support this approach, but fortunately it doesn't matter, because you'll rarely need to walk through an entire document's DOM tree starting from the root. Instead, the approach taken is to use one of the `getElementsBy`*`Whatever`* methods to grab a particular part of the tree directly. Those methods start from the `window.document` object—or `document` for short.

Getting an Element from the Tree

There are two principal methods that can be used to get a particular element or set of elements. The first method, which is used all the time in DHTML programming, is `getElementById`. The second is `getElementsByTagName`. Another method, `getElementsByName`, is rarely used, so we'll look at the first two only for now.

`getElementById`

In HTML, any element can have a unique ID. The ID must be specified with the HTML `id` attribute:

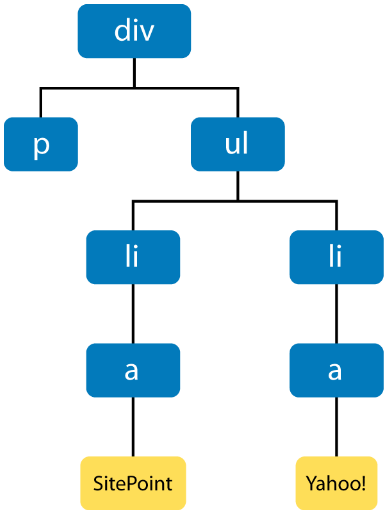
```
<div id="codesection">
  <p id="codepara">
  </p>
  <ul>
    <li><a href="http://www.sitepoint.com/" id="splink"
        >SitePoint</a></li>
    <li><a href="http://www.yahoo.com/" id="yalink"
        >Yahoo!</a></li>
  </ul>
</div>
```

Each non-list element in that snippet has been given an ID. You should be able to spot four of them. IDs must be unique within your document—each element must have a different ID (or no ID at all)—so you can know that a specific ID identifies a given element alone. To get a reference to that element in JavaScript code, use `document.getElementById(`*`elementId`*`)`:

```
var sitepoint_link = document.getElementById('splink')
```

Now the variable `sitepoint_link` contains a reference to the first `<a>` tag in the above HTML snippet. We'll see a little later what you can do with that element reference. The DOM tree for this snippet of HTML is depicted in Figure 2.2.

Figure 2.2. The snippet's DOM tree.



getElementsByTagName

The `document.getElementsByTagName` method is used to retrieve all elements of a particular type. The method returns an array[1] that contains all matching elements:

```
var all_links = document.getElementsByTagName('a');
var sitepoint_link = all_links[0];
```

The `all_links` variable contains an array, which contains two elements: a reference to the SitePoint link, and a reference to the Yahoo! link. The elements are returned in the order in which they are found in the HTML, so `all_links[0]` is the SitePoint link and `all_links[1]` is the Yahoo! link.

Note that `document.getElementsByTagName` always returns an array, even if only one matching element was found. Imagine we use the method as follows:

```
var body_list = document.getElementsByTagName('body');
```

To get a reference to the sole `body` element in this case, we would need to use the following:

```
var body = body_list[0];
```

We would be very surprised if `body_list.length` (the array's size) was anything other than `1`, since there should be only one `<body>` tag! We could also shorten the process slightly by replacing the previous two lines with this one:

```
var body = document.getElementsByTagName('body')[0];
```

JavaScript allows you to collapse expressions together like this. It can make your code a lot more compact, and save you from declaring a lot of variables which aren't really used for anything.

There is another useful feature; `getElementsByTagName` is defined on any node at all, not just the document. So, to find all `<a>` tags in the body of the document, we could use the method like this:

```
var links_in_body = body.getElementsByTagName('a');
```

[1]Technically, it returns a node collection, but this works just like an array.

Note that "Element" is plural in this method's name, but singular for `getElementById`. This is a reminder that the former returns an array of elements, while the latter returns only a single element.

Walking from Parents to Children

Each node has one parent (except the root element) and may have multiple children. You can obtain a reference to a node's parent from its `parentNode` property; a node's children are found in the node's `childNodes` property, which is an array. The `childNodes` array may contain nothing if the node has no children (such nodes are called **leaf nodes**).

Suppose the variable node points to the `ul` element of the DOM tree. We can get the node's parent (the `div` element) like this:

```
parent = node.parentNode;
```

We can check if the unordered list has any list items (children) by looking at the `length` property of the `childNodes` array:

```
if (node.childNodes.length == 0) {
  alert('no list items found!');
}
```

If there are any children, their numbering starts at zero. We can obtain the second child in our example HTML (an `li` element) as follows:

```
list_item = node.childNodes[1];
```

For the special case of the first child, located here:

```
list_item = node.childNodes[0];
```

we can also use this shorthand:

```
child = node.firstChild;
```

Similarly, the last child (in this case, the second `li`) has its own special property:

```
child = node.lastChild;
```

We'll see all these properties used routinely through the rest of this book.

What to do with Elements

Now you know how to get references to elements—the nodes in your HTML page. The core of DHTML—the D-for-dynamic bit—lies in our ability to change those elements, to remove them, and to add new ones. Throughout the rest of this chapter, we'll work with the following code snippet, which we saw earlier:

```
<div id="codesection">
  <p id="codepara">
  </p>
  <ul>
    <li><a href="http://www.sitepoint.com/" id="splink"
        >SitePoint</a></li>
    <li><a href="http://www.yahoo.com/" id="yalink"
        >Yahoo!</a></li>
  </ul>
</div>
```

Changing Element Attributes

Every property of an element, and every CSS style that can be applied to it, can be set from JavaScript. The attributes that can be applied to an element in HTML—for example, the `href` attribute of an `<a>` tag—can also be set and read from your scripts, as follows:

```
// using our sitepoint_link variable from above
sitepoint_link.href = "http://www.google.com/";
```

Click on that link after the script has run, and you'll be taken to Google rather than SitePoint. The new HTML content, as it exists in the browser's imagination (the HTML file itself hasn't changed), looks like this:

```
<div id="codesection">
  <p id="codepara">
  </p>
  <ul>
    <li><a href="http://www.google.com/" id="splink"
        >SitePoint</a></li>
    <li><a href="http://www.yahoo.com/" id="yalink"
        >Yahoo!</a></li>
  </ul>
</div>
```

Each element has a different set of attributes that can be changed: `a` elements have the `href` attribute, `<img>` elements have the `src` attribute, and so on. In general, an attribute that can be applied to a tag in your HTML is also gettable and settable as a property on a node from JavaScript. So, if our code contains a reference to an `img` element, we can change the image that's displayed by altering the `img_element.src` property.[2]

The two most useful references that document elements and their supported attributes are those provided by the two major browser makers: the Microsoft DOM reference[3], and the Mozilla Foundation's DOM reference[4].

Importantly, though, when we altered our link's `href` above, all we changed was the destination for the link. The text of the link, which read "SitePoint" before, has not changed; if we need to alter that, we have to do so separately. Changing the text in a page is slightly more complex than changing an attribute; to alter text, you need to understand the concept of text nodes.

Changing Text Nodes

In Figure 2.1 above, you can see how the HTML in a document can be represented as a DOM tree. One of the important things the figure illustrates is that the text inside an element is not part of that element. In fact, the text is in a different node: a child of the element node. If you have a reference to that text node, you can change the text therein using the node's `nodeValue` property:

```
myTextNode.nodeValue = "Some text to go in the text node";
```

How can we get a reference to that text node? We need to walk the DOM tree—after all, we have to know where the text node is before we can alter it. If we consider the `sitepoint_link` node above, we can see that its `childNodes` array should contain one node: a text node with a `nodeValue` of `"SitePoint"`. We can change the value of that text node as follows:

```
sitepoint_link.childNodes[0].nodeValue = 'Google';
```

[2]One notable divergence from this rule is that an element's `class` attribute in HTML is available in JavaScript as `node.className`, not `node.class`. This is because "class" is a JavaScript reserved word.
[3] http://msdn.microsoft.com/workshop/author/dhtml/reference/dhtml_reference_entry.asp
[4] http://www.mozilla.org/docs/dom/domref/

Now, the text displayed on-screen for that link will read Google, which matches the link destination that we changed earlier. We can shorten the code slightly to the following:

```
sitepoint_link.firstChild.nodeValue = 'Google';
```

You may recall that a node's `firstChild` property, and `childNodes[0]`, both refer to the same node; in this case, you can substitute `childNodes[0]` with success. After this change, the browser will see the following document code:

```
<div id="codesection">
  <p id="codepara">
  </p>
  <ul>
    <li><a href="http://www.google.com/" id="splink"
        >Google</a></li>
    <li><a href="http://www.yahoo.com/" id="yalink"
        >Yahoo!</a></li>
  </ul>
</div>
```

Changing Style Properties

As we have seen, the attributes that are set on an HTML tag are available as properties of the corresponding DOM node. CSS style properties can also be applied to that node through the DOM, using the node's style property. Each CSS property is a property of that style property, with its name slightly transformed: a CSS property in words-and-dashes style becomes a property of style with dashes removed and all words but the first taking an initial capital letter. This is called **InterCaps format**. Here's an example. A CSS property that was named:

```
some-css-property
```

would appear to a script as the following JavaScript property:

```
someCssProperty
```

So, to set the CSS property `font-family` for our `sitepoint_link` element node, we'd use the following code:

```
sitepoint_link.style.fontFamily = 'sans-serif';
```

CSS values in JavaScript are almost always set as strings; some values, such as `font-size`, are strings because they must contain a dimension[3], such as “px” or “%”. Only entirely numeric properties, such as `z-index` (which is set as *`node`*`.style.zIndex`, as per the above rule) may be set as a number:

```
sitepoint_link.style.zIndex = 2;
```

Many designers alter style properties to make an element appear or disappear. In CSS, the `display` property is used for this: if it’s set to `none`, the element doesn’t display in the browser. So, to hide an element from display, we can set its `display` property to `none`:

```
sitepoint_link.style.display = 'none';
```

To show it again, we give it another valid value:

```
sitepoint_link.style.display = 'inline';
```

For a complete reference to the available CSS style properties and what each does, see SitePoint’s *HTML Utopia: Designing Without Tables Using CSS[5]*.

Bigger DOM Tree Changes

The next level of DOM manipulation, above and beyond changing the properties of elements that are already there, is to add and remove elements dynamically. Being able to change the display properties of existing elements, and to read and alter the attributes of those elements, puts a lot of power at your disposal, but the ability to dynamically create or remove parts of a page requires us to leverage a whole new set of techniques.

Moving Elements

To add an element, we must use the `appendChild` method of the node that will become the added node’s parent. In other words, to add your new element as a child of an existing node in the document, we use that node’s `appendChild` method:

[3]Internet Explorer will let you get away without using a dimension, as it assumes that a dimensionless number is actually a pixel measurement. However, do not try to take advantage of this assumption; it will break your code in other browsers, and it’s in violation of the specification.
[5] http://www.sitepoint.com/books/css1/

```
// We'll add the link to the end of the paragraph
var para = document.getElementById('codepara');
para.appendChild(sitepoint_link);
```

After this, our page will look a little odd. Here's the updated HTML code:

```
<div id="codesection">
  <p id="codepara">
    <a href="http://www.google.com/" id="splink">Google</a>
  </p>
  <ul>
    <li></li>
    <li><a href="http://www.yahoo.com/" id="yalink"
        >Yahoo!</a></li>
  </ul>
</div>
```

Another useful thing to know is that, in order to move the node to its new place in the document, we don't have to *remove* it first. If you use `appendChild` to insert a node into the document, and that node already exists elsewhere in the document, the node will not be duplicated; instead, it will move from its previous location to the new location at which you've inserted it. We can do the same thing with the Yahoo! link:

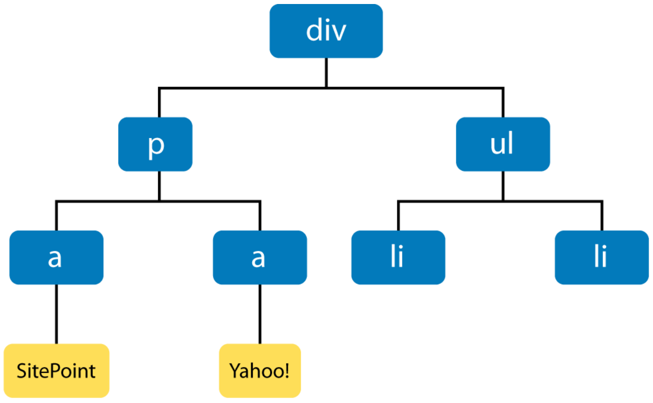
```
para.appendChild(document.getElementById('yalink'));
```

After this, the page will again be rearranged to match the HTML:

```
<div id="codesection">
  <p id="codepara">
    <a href="http://www.google.com/" id="splink">Google</a>
    <a href="http://www.yahoo.com/" id="yalink">Yahoo!</a>
  </p>
  <ul>
    <li></li>
    <li></li>
  </ul>
</div>
```

Figure 2.3 shows the new DOM tree so far.

Figure 2.3. The DOM tree after changes.



What if you didn't want to add your new (or moved) element to the end of that paragraph? In addition to `appendChild`, each node has an `insertBefore` method, which is called with two arguments: the node to insert, and the node before which it will be inserted. To move the Yahoo! link to the beginning of the paragraph, we want to insert it as a child of the paragraph that appears *before* the Google link. So, to insert the Yahoo! link (the first argument) as a child of the paragraph right before the Google link (`sitepoint_link`, the second argument), we'd use the following:

```
para.insertBefore(document.getElementById('yalink'),
    sitepoint_link);
```

Be sure that the second argument (`sitepoint_link`) really is an existing child node of `para`, or this method will fail.

Throwing Away Elements

Removing an element is very similar to the process of adding one: again, we use the `removeChild` method on the element's parent node. Remembering from earlier that we can access a given node's parent as *`node`*`.parentNode`, we can remove our `sitepoint_link` from the document entirely:

```
// never hurts to be paranoid: check that our node *has* a parent
if (sitepoint_link.parentNode) {
  sitepoint_link.parentNode.removeChild(sitepoint_link);
}
```

That action will change the HTML code to that shown below:

```
<div id="codesection">
  <p id="codepara">
    <a href="http://www.yahoo.com/" id="yalink">Yahoo!</a>
  </p>
  <ul>
    <li></li>
    <li></li>
  </ul>
</div>
```

note

Even after the node's removal, `sitepoint_link` still constitutes a reference to that link. It still exists, it's just not in the document any more: it's floating in limbo. We can add it back to the document somewhere else if we want to. Set the variable to `null` to make the deleted element disappear forever.

Creating Elements

Moving existing elements around within the page is a powerful and useful technique (with which you're well on the way to implementing Space Invaders or Pac Man!). But, above and beyond that, we have the ability to create brand new elements and add them to the page, providing the capacity for truly dynamic content. The point to remember is that, as before, a page's text resides in text nodes, so if we need to create an element that contains text, we must create both the new element node and a text node to contain its text. To achieve this, we need two new methods: `document.createElement` and `document.createTextNode`.

First, we create the element itself:

```
var linux_link = document.createElement('a');
```

Even though we've created the element, it's not yet part of the document. Next, we set some of its properties in the same way that we'd set properties on an existing link:

```
linux_link.href = 'http://www.linux.org/';
```

We then create the text node for the text that will appear inside the link. We pass the text for the text node as a parameter:

```
var linux_tn =
    document.createTextNode('The Linux operating system');
```

The text node is also floating around, separate from the document. We add the text node to the element's list of children, as above:

```
linux_link.appendChild(linux_tn);
```

The element and text node now form a mini-tree of two nodes (officially a **document fragment**), but they remain separate from the DOM. Finally, we insert the element into the page, which is the same as putting it into the DOM tree:

```
para.appendChild(linux_link);
```

Here's the resulting HTML:

```
<div id="codesection">
  <p id="codepara">
    <a href="http://www.yahoo.com/" id="yalink">Yahoo!</a>
    <a href="http://www.linux.org/">The Linux operating system</a>
  </p>
  <ul>
    <li></li>
    <li></li>
  </ul>
</div>
```

As you can see, to create elements, we use the same techniques and knowledge—text nodes are children of the element node, we append a child with `*node*.appendChild`—we use to work with nodes that are already part of the document. To the DOM, a node is a node whether it's part of the document or not: it's just a node object.

Copying Elements

Creating one element is simple, as we've seen. But what if you want to add a lot of dynamic content to a page? Having to create a whole batch of new elements and text nodes—appending the text nodes to their elements, the elements to each other, and the top element to the page—is something of a laborious process. Fortunately, if you're adding to the page a copy of something that's already there, a shortcut is available: the `cloneNode` method. This returns a *copy* of the node, including all its attributes and all its children.[4] If you have a moderately complex piece of HTML that contains many elements, `cloneNode` is a very quick way to return a copy of that block of HTML ready for insertion into the document:

[4]You can elect to clone the node only—not its children—by passing `false` to the `cloneNode` method.

```
var newpara = para.cloneNode(true);
document.getElementById('codesection').appendChild(newpara);
```

You can't rush ahead and just do this, though: it pays to be careful with `cloneNode`. This method clones all attributes of the node and all its child nodes, *including* IDs, and IDs must be unique within your document. So, if you have elements with IDs in your cloned HTML block, you need to fix those IDs before you append the cloned block to the document.

It would be nice to be able to grab the Yahoo! link in our cloned block using the following code:

```
var new_yahoo_link = newpara.getElementById('yalink');
```

But, unfortunately, we can't. The `getElementById` method is defined only on a document, not on any arbitrary node. The easiest way around this is to refrain from defining IDs on elements in a block that you wish to clone. Here's a line of code that will remove the Yahoo! link's `id`:

```
newpara.firstChild.removeAttribute('id');
```

We still have the ID on the paragraph itself, though, which means that when we append the new paragraph to the document, we'll have two paragraphs with the ID `codepara`. This is bad—it's not supposed to happen. We must fix it before we append the new paragraph, revising the above code as follows:

```
var newpara = para.cloneNode(true);
newpara.id = 'codepara2';
newpara.firstChild.removeAttribute('id');
document.getElementById('codesection').appendChild(newpara);
```

This code returns the following results:

```
<div id="codesection">
  <p id="codepara">
    <a href="http://www.yahoo.com/">Yahoo!</a>
    <a href="http://www.linux.org/">The Linux operating system</a>
  </p>
  <p id="codepara2">
    <a href="http://www.yahoo.com/">Yahoo!</a>
    <a href="http://www.linux.org/">The Linux operating system</a>
  </p>
  <ul>
    <li></li>
    <li></li>
  </ul>
</div>
```

As you can see, there's a little bit of surgery involved if you choose to copy big chunks of the document. This demonstration concludes our experimentation with this particular bit of code.

Making an Expanding Form

As our first full example, we'll use the DOM's element creation methods to build a form that can grow as the user fills it. This allows users to add to the form as many entries as they like.

Let's imagine an online system through which people can sign up themselves, and any number of their friends, for free beer.[5] The users add their own names, then the names of all of the friends they wish to invite. Without the DOM, we'd require the form either to contain a large number of slots for friends' names (more than anyone would use), or to submit regularly back to the server to get a fresh (empty) list of name entry areas.

In our brave new world, we can add the extra name entry fields dynamically. We'll place a button on the form that says, Add another friend. Clicking that button will add a new field to the list, ready for submission to the server. Each newly-created field will need a different `name` attribute, so that it can be distinguished when the server eventually receives the submitted form.[6]

Our form will provide a text entry box for the user's name, a `fieldset` containing one text entry box for a friend's name, and a button to add more friends. When the button is clicked, we'll add a new text entry box for another friend's name.

File: **`expandingForm.html`**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Free beer signup form</title>

    <script type="text/javascript">
```

[5]Maybe there's a mad millionaire philanthropist on the loose. No, I can't give you a URL at which this system is running for real!
[6]Depending on the server-side language used to process the form, this isn't strictly necessary. Since our example form won't actually submit to anything, we'll implement it as a useful exercise.

```
      var fieldCount = 1;
      function addFriend() {
        fieldCount++;
        var newFriend = document.createElement('input');
        newFriend.type = 'text';
        newFriend.name = 'friend' + fieldCount;
        newFriend.id = 'friend' + fieldCount;
        document.getElementById('fs').appendChild(newFriend);
      }
    </script>

    <style type="text/css">
      input {
        display: block;
        margin-bottom: 2px;
      }
      button {
        float: right;
      }
      fieldset {
        border: 1px solid black;
      }
    </style>

  </head>
  <body>
    <h1>Free beer signup form</h1>
    <form>
      <label for="you">Your name</label>
      <input type="text" name="you" id="you">
      <fieldset id="fs">
        <legend>Friends you wish to invite</legend>
        <button onclick="addFriend(); return false;">
          Add another friend
        </button>
        <input type="text" name="friend1" id="friend1">
      </fieldset>
      <input type="submit" value="Save details">
    </form>
  </body>
</html>
```

Notice our `fieldCount` variable; this keeps track of how many friend fields there are.

File: **expandingForm.html** **(excerpt)**

```
    var fieldCount = 1;
```

When the button is clicked, we run the `addFriend` function (we'll discuss handling clicks—and various other kinds of events—more in the next chapter):

```
    <button onclick="addFriend(); return false;">
```

The `addFriend` function completes a number of tasks each time it's run:

1. Increments the `fieldCount`:

   File: **expandingForm.html** **(excerpt)**

   ```
       fieldCount++;
   ```

2. Creates a new `input` element:

   File: **expandingForm.html** **(excerpt)**

   ```
       var newFriend = document.createElement('input');
   ```

3. Sets its `type` to `text`—we want a text entry box, an element specified by `<input type="text">`:

   File: **expandingForm.html** **(excerpt)**

   ```
       newFriend.type = 'text';
   ```

4. Sets a unique `id` and `name` (because the ID must be unique, and all the entry boxes must have different names so they can be distinguished when the form's submitted):

   File: **expandingForm.html** **(excerpt)**

   ```
       newFriend.name = 'friend' + fieldCount;
       newFriend.id = 'friend' + fieldCount;
   ```

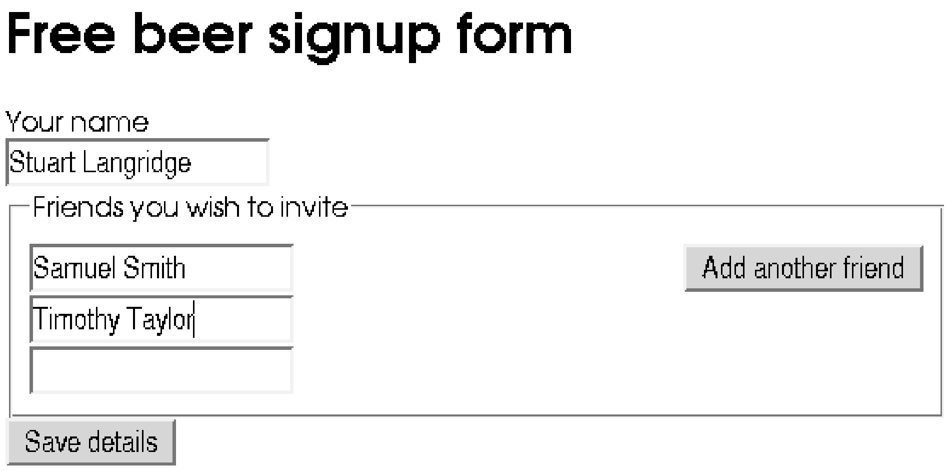
5. Adds this newly-created element to the document:

   File: **expandingForm.html** **(excerpt)**

   ```
       document.getElementById('fs').appendChild(newFriend);
   ```

Here’s what the page looks like after the “add another friend” button has been clicked twice, and two friends’ names have been added:

Figure 2.4. Signing up for free beer.



Free beer signup form

Your name: Stuart Langridge

Friends you wish to invite:

Samuel Smith

Timothy Taylor

Free beer, thanks to the power of the DOM. We can’t complain about that!

Making Modular Image Rollovers

Image rollover scripts, in which an image is used as a link, and that image changes when the user mouses over it, are a mainstay of JavaScript programming on the Web. Traditionally, they’ve required a lot of script, and a lot of customization, on the part of the developer. The introspective capability of the DOM—the ability of script to inspect the structure of the page in which it’s running—gives us the power to detect rollover images automatically and set them up without any customization. This represents a more systematic approach than the old-fashioned use of `onmouseover` and `onmouseout` attributes, and keeps rollover code separate from other content.

We’ll build our page so that the links on which we want to display rollover effects have a class of `rollover`. They’ll contain one `img` element—nothing else. We’ll also provide specially named rollover images: if an image within the page is called `foo.gif`, then the matching rollover image will be named `foo_over.gif`. When the page loads, we’ll walk the DOM tree, identify all the appropriate links (by checking their class and whether they contain an `img` element), and set up the rollover on each. This specially-named rollover image allows us to deduce the name of any rollover image without saving that name anywhere. It reduces the amount of data we have to manage.

An alternative technique involves use of a non-HTML attribute in the image tag:

```
<img src="basic_image.gif" oversrc="roll_image.gif">
```

However, since `oversrc` isn't a standard attribute, this approach would cause your HTML to be invalid.

Some of the following script may seem a little opaque: we will be attaching listeners to DOM events to ensure that scripts are run at the appropriate times. If this is confusing, then feel free to revisit this example after you've read the discussion of DOM events in the next chapter.

A Sample HTML Page

First, the HTML: here we have our links, with class `rollover`, containing the images.

File: **rollovers.html**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Modular rollovers</title>
    <script type="text/javascript" src="rollovers.js"></script>
    <style type="text/css">
      /* Remove the blue border on the rollover images */
      a.rollover img {
        border-width: 0;
      }
    </style>
  </head>
  <body>
    <h1>Modular rollovers</h1>
    <p>Below we have two links, containing images that we want
      to change on mouseover.</p>
        <ul>
      <li>
        <a href="" class="rollover" alt="Roll"
            ><img src="basic_image.gif" /></a>
      </li>
      <li>
        <a href="" class="rollover" alt="Roll"
            ><img src="basic_image2.gif"></a>
      </li>
    </ul>
  </body>
</html>
```

The page also includes the JavaScript file that does all the work:

File: **rollovers.js**

```
function setupRollovers() {
  if (!document.getElementsByTagName)
    return;
  var all_links = document.getElementsByTagName('a');
  for (var i = 0; i < all_links.length; i++) {
    var link = all_links[i];
    if (link.className &&
        (' ' + link.className + ' ').indexOf(' rollover ') != -1)
    {
      if (link.childNodes &&
          link.childNodes.length == 1 &&
          link.childNodes[0].nodeName.toLowerCase() == 'img') {
        link.onmouseover = mouseover;
        link.onmouseout = mouseout;
      }
    }
  }
}

function findTarget(e)
{
  /* Begin the DOM events part, which you */
  /* can ignore for now if it's confusing */
  var target;

  if (window.event && window.event.srcElement)
    target = window.event.srcElement;
  else if (e && e.target)
    target = e.target;
  if (!target)
    return null;

  while (target != document.body &&
      target.nodeName.toLowerCase() != 'a')
    target = target.parentNode;
  if (target.nodeName.toLowerCase() != 'a')
    return null;

  return target;
}

function mouseover(e) {
  var target = findTarget(e);
  if (!target) return;

  // the only child node of the a-tag in target will be an img-tag
  var img_tag = target.childNodes[0];

  // Take the "src", which names an image called "something.ext",
  // Make it point to "something_over.ext"
  // This is done with a regular expression
  img_tag.src = img_tag.src.replace(/(\.[^.]+)$/, '_over$1');
}

function mouseout(e) {
  var target = findTarget(e);
  if (!target) return;

  // the only child node of the a-tag in target will be an img-tag
  var img_tag = target.childNodes[0];

  // Take the "src", which names an image as "something_over.ext",
  // Make it point to "something.ext"
  // This is done with a regular expression
  img_tag.src = img_tag.src.replace(/_over(\.[^.]+)$/, '$1');
}

// When the page loads, set up the rollovers
window.onload = setupRollovers;
```

The DOM-walking parts of this code are found in `setupRollovers` and in `findTarget`, which is called from the two `mouseover`/`mouseout` functions. Let's look at each of these in turn.

The `setupRollovers` Function

The code for the `setupRollovers` function starts like this:

File: **rollovers.js (excerpt)**

```
if (!document.getElementsByTagName)
  return;
```

This code confirms that we're in a DOM-supporting browser. If we're not (i.e. if `document.getElementsByTagName`, the method, doesn't exist), we exit here and progress no further. If the method does exist, we continue:

File: **rollovers.js (excerpt)**

```
var all_links = document.getElementsByTagName('a');
```

Here, we make `all_links` a reference to a list of all the `<a>` tags in the document.

File: **rollovers.js (excerpt)**

```
for (var i = 0; i < all_links.length; i++) {
  var link = all_links[i];
```

The above code iterates through the retrieved list of tags in standard JavaScript fashion. We assign the `link` variable to each link, as a way to simplify the following code.

File: **rollovers.js (excerpt)**

```
  if (link.className &&
      (' ' + link.className + ' ').indexOf(' rollover ') != -1)
  {
```

We need to know whether each link is of class `rollover`. However, an element may have more than one class; if this tag had two classes, `rollover` and `hotlink`, for example, it would have `className="rollover hotlink"`. This would mean that we could not check for an element having a specific class using the following:

```
if (element.className == "myclass")
```

If the element has multiple classes, the above condition will always evaluate to `false`. A useful approach here is to look for the string `' myclass '` (the class name with a space before and after it) in the string `' ' + element.className + ' '` (the element's `class` attribute with a space before and after it). This will always find your class, as you're expecting. It also avoids a problem with a similar technique, which uses `className.indexOf` to look for `'myclass'`. If the element in question is of class `myclassroom`, this technique will give a false positive.[7]

[7]Another option is to use a regular expression to spot the class name. In the interests of simplicity, however, we'll stick with the method already presented.

File: **rollovers.js (excerpt)**

```
      if (link.childNodes &&
          link.childNodes.length == 1 &&
          link.childNodes[0].nodeName.toLowerCase() == 'img') {
```

We want to confirm that this link contains nothing but an `img` element, so we make use of a very handy property of JavaScript, called **short-circuit evaluation**. In an `if` statement of the form `if (a && b && c)`, if `a` is false, then `b` and `c` are not evaluated at all. This means that `b` and `c` can be things that depend on `a`'s trueness: if `a` is not true, then they are not evaluated, so it's safe to put them into the `if` statement.

Looking at the above code may make this clearer. We need to test if the `nodeName` of the link's first child node is `img`. We might use the following code:

```
if (link.childNodes[0].nodeName.toLowerCase == 'img')
```

However, if the current link doesn't have any child nodes, this code will cause an error because there is no `link.childNodes[0]`. So, we must first check that child nodes exist; second, we confirm that there is one and only one child; third, we check whether that one-and-only first child is an image. We can safely assume in the image check that `link.childNodes[0]` exists, because we've already confirmed that that's the case: if it didn't exist, we wouldn't have got this far.

File: **rollovers.js (excerpt)**

```
        link.onmouseover = mouseover;
```

This code attaches an event handler to the `mouseover` event on a node.

File: **rollovers.js (excerpt)**

```
        link.onmouseout = mouseout;
```

And this line attaches an event handler to the `mouseout` event on that node. That's all!

The `findTarget` Function

This little function is called by the `mouseover` and `mouseout` functions. As we'll see, they pass event objects to `findTarget`, which, in return, passes back the link tag surrounding the image that generated the event, if any such tag is to be found.

`findTarget` starts like this:

File: **rollovers.js (excerpt)**

```
  var target;

  if (window.event && window.event.srcElement)
    target = window.event.srcElement;
  else if (e && e.target)
    target = e.target;
  if (!target)
    return null;
```

This first part is related to DOM event handling, which is explained in the next chapter. We'll ignore its workings for now, except to say that it caters for the differences between Internet Explorer and fully DOM-supporting browsers. Once this code has run, however, we should have in our variable target the element that the browser deems to be responsible for the `mouseover` or `mouseout` event—ideally the `<a>` tag.

File: **rollovers.js (excerpt)**

```
  while (target != document.body &&
      target.nodeName.toLowerCase() != 'a')
    target = target.parentNode;

  if (target.nodeName.toLowerCase() != 'a')
    return null;
```

The variable target should be a reference to the `<a>` tag on which the user clicked, but it may be something inside the `<a>` tag (as some browsers handle events this way). In such cases, the above code keeps getting the parent node of that tag until it gets to an `<a>` tag (which will be the one we want). If we find the document body—a `<body>` tag—instead, we've gone too far. We'll give up, returning `null` (nothing) from the function, and going no further.

If we did find an `<a>` tag, however, we return that:

File: **rollovers.js (excerpt)**

```
  return target;
}
```

The `mouseover` / `mouseout` Functions

These functions work in similar ways and do very similar things: `mouseover` is called when we move the mouse over one of our rollover links, while `mouseout` is called when we move the mouse out again.

The code for `mouseover` starts like this:

File: **`rollovers.js`** **(excerpt)**

```
var target = findTarget(e);
if (!target) return;
```

We call the `findTarget` function, described above, to get a reference to the link over which the mouse is located. If no element is returned, we give up, degrading gracefully. Otherwise, we have the moused-over `<a>` tag in target. Next, we dig out the image.

File: **`rollovers.js`** **(excerpt)**

```
var img_tag = target.childNodes[0];
```

We also know that the `<a>` tag has one, and only one, child node, and that's an `<img>` tag. We know this because we checked that this was the case when we set up the event handler in `setupRollovers`.

File: **`rollovers.js`** **(excerpt)**

```
img_tag.src = img_tag.src.replace(/(\.[^.]+)$/, '_over$1');
```

Images have a `src` attribute, which you can access through the DOM with the element's `src` property. In the code snippet above, we apply a regular expression substitution to that string.[8] Changing the value of an `<img>` tag's `src` attribute causes it to reload itself with the new image; thus, making this substitution (replacing `something.gif` with `something_over.gif`) causes the original image to change to the rollover image. The `mouseout` function does the exact opposite: it changes the reference to `something_over.gif` in the image's `src` attribute to `something.gif`, causing the original image to reappear.

Something for Nothing (Almost)

If you look at the code for this modular rollover, you'll see that it's divided into parts. The `setupRollovers` function does nothing but install listeners. The `findTarget` function does nothing but find the link tag for a given event. The `mouseover` and `mouseout` functions do little other than the actual image swapping work. The tasks are neatly divided.

[8]Although the full details of regular expressions are beyond the scope of this book, we'll look at the basics in Chapter 6. A more detailed resource is Kevin Yank's article on sitepoint.com, *Regular Expressions in JavaScript [http://www.sitepoint.com/article/expressions-javascript]*.

That means that this code is good for other applications. We can change the `mouseover` and `mouseout` functions to do something else—for example, to make popup help content appear—without needing to start from scratch to get it working. We get to reuse (or at least rip off with minimal change) the other functions in the script. This is not only convenient; it's also neat and clean. We're on the way to a better kind of scripting!

Summary

In the introduction, we referred to the DOM as a critical part of DHTML. Exploring the DOM—being able to find, change, add, and remove elements from your document—is a powerful technique all by itself, and is a fundamental aspect of modern DHTML. Once you've mastered the techniques described in this chapter, everything else will fall into place. Through the rest of the book, we'll be describing techniques and tricks with which you can do wondrous things on your sites, and in your Web applications, using DHTML. They all build upon this fundamental approach of manipulating the Document Object Model.

3 Handling DOM Events

When I can't handle events, I let them handle themselves.
—Henry Ford

An event is something that happens, be it in real life, or in DHTML programming. But to those working with DHTML, events have a very specific meaning. An event is generated, or fired, when something happens to an element: a mouse clicks on a button, for example, or a change is made to a form. DHTML programming is all about event handling; your code will run in response to the firing of this or that event.

Learning which events are available, how to hook your code up to them, and how to make best use of them is a critical part of building dynamic Web applications.[1] That's what we cover in this chapter, along with a couple of real-world examples.

About Elements and Events

We're using a modern approach to DHTML, so all our DHTML code will be set to run in response to the firing of an event. If you've done any JavaScript Web programming before, you may already be using this technique without knowing it. Let's look at the procedure by which code has traditionally been hooked up

[1]It does seem that there are quite a few "critical" bits, I know!

to events, learn how to do it under the DOM (and why the DOM method is better), and find out exactly what these techniques make possible.

Common Events

Every page element fires a given selection of events. Some events are common to all elements; others are more specific. For example, all visible elements will fire a `mouseover` event when the mouse is moved over them. A `change` event, however, will only be fired by elements whose contents can be changed: text boxes, text areas, and drop-down lists.

You might have noticed above that I used `mouseover`, rather than `onmouseover`, for the event name. Even though the HTML attribute for handling this event is `onmouseover`, the modern way to describe the event itself is simply `mouseover`. This allows us to talk about the *event* (`mouseover`) and the *event handler* (`onmouseover`) separately. The **event handler** is the location at which an event handler is placed. In the bad old browser days, these concepts were all mixed up, but now we can safely think of them as separate entities.

The documents that describe the events fired by a given element are the W3C DOM specifications and HTML recommendations, which were mentioned in the last chapter, as well as the W3C DOM 2 Events specification[1]. There's also some extra information on key events in the DOM 3 Events specification[2].

A summary of the events that you're likely to find useful, and that have cross-browser support, is given in Table 3.1. Note that this isn't an exhaustive survey: it's a listing of events that you're likely to use often, rather than everything under the sun.

[1] http://www.w3.org/TR/DOM-Level-2-Events/Overview.html
[2] http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/events.html

Table 3.1. Useful Events.

Event(s)	Fired by Element(s)	Fired when
`load`	`window`	The page finishes loading.
`unload`	`window`	The page is unloaded (i.e. the user closes the browser, or clicks a link, and a new page loads).
`change`	`input`, `select`, `textarea`	The element loses focus (the user clicks outside it or tabs away from it), and the content has been changed (note: the event does not fire immediately when the change is made!).
`focus`	`label`, `input`, `select`, `textarea`, `button`	The element gets the focus (it is tabbed to, or clicked upon).
`blur`	`label`, `input`, `select`, `textarea`, `button`	The element loses the focus.
`resize`	`window`	The user resizes the window.
`scroll`	`window`	The user scrolls the window.
`submit`	`form`	The user submits the form by clicking the submit button or hitting **Enter** in a text field.
`mouseover`	any visible	The user moves the mouse onto an element.
`mouseout`	any visible	The user moves the mouse off an element.
`mousedown`	any visible	The user presses any mouse button while on the element.
`mouseup`	any visible	The user releases the mouse button while on the element.
`mousemove`	any visible	The user moves the mouse anywhere on the element.
`click`	any	The user clicks any mouse button while on the element (this is the same as a `mousedown` followed by a `mouseup`).
`keypress`	any element that can be focused	A key is pressed while the element has focus.

Hooking Code to Events

So, now you know some common events, and when they fire. But how do you make your code run in response to those events?

Hooking up the Old Way

If you've done any JavaScript coding before, you'll probably have written something like this:

```
<a href="somewhere.html"
    onclick="myJavaScriptFunction(); return false;"
    >click me!</a>
```

That `onclick` attribute connects some JavaScript code to that link's `click` event. When the link is clicked, it will fire a click event, and that code will run. No problem! Notice, though, that the code never actually mentions "click," which is the actual name of the event.

What if we wanted to detect a keypress? Here's the equivalent script:

```
function aKeyWasPressed() {
  // put event handler code here ...
}
```

And here's the matching snippet of HTML:

```
<textarea id="myta" onkeypress="aKeyWasPressed()"></textarea>
```

In this case, how does our `aKeyWasPressed` function know which key was pressed? Well, it doesn't. That's a major limitation of the old-fashioned approach. But we can improve on that!

Hooking up the DOM Way

The DOM specifications enlarge the idea of event handlers by providing **event targets** and **event listeners**. An event target is the thing at which an event is aimed—an element, essentially. An event listener is the thing that grabs the event when it appears, and responds to it. Where do events come from in the first place? They come from the user. The browser software captures the user action and sends the event to the right event target.

A given event source can be relevant to more than one event listener. Using the old-fashioned method above, only one piece of code could be run in response to any event. For example, an element could have only one `onclick` attribute.[2] Using the modern method, you can run as many pieces of code as you want upon the firing of an event or events. Listeners get to share events, and events get to share listeners. To facilitate this, we must move our "hookup" code from the HTML to a separate script section: as noted above, no element can have more than one `onclick` attribute.

Event handling works in different ways, depending on the browser. We'll examine the W3C-approved way first, before we look at event handling in Internet Explorer. Here's the W3C approach.

File: **`keycodedetect.html`** **(excerpt)**

```
function aKeyWasPressed(e) {
  // put event listener code here...
}

var textarea = document.getElementById('myta');
textarea.addEventListener('keyup', aKeyWasPressed, false);
```

And here's the matching bit of HTML:

File: **`keycodedetect.html`** **(excerpt)**

```
<textarea id="myta"></textarea>
```

IMPORTANT

HTML Before Script... for Now

If you're working through this example in your HTML editor of choice, be sure to place the JavaScript code after the HTML in this and the next few examples in this chapter. The `textarea` must exist before the JavaScript code can assign an event listener to it.

If you're used to placing JavaScript at the top of your HTML files, don't fret. We'll discuss an elegant way around this restriction at the end of the section.

Those few lines of code contain a number of complex concepts. Consider this snippet:

[2]Actually, you could have as many as you liked, but each one would overwrite the one before it, so, effectively, you have only one. Alternatively, you could string JavaScript statements together, using semicolons in the attribute, but this makes the HTML code even more cluttered.

File: **keycodedetect.html (excerpt)**

```
var textarea = document.getElementById('myta');
```

Here, we see a familiar reference to the `<textarea>`. Next, there's something new:

File: **keycodedetect.html (excerpt)**

```
textarea.addEventListener('keyup', aKeyWasPressed, false);
```

This is the crucial line that sets everything up. Each element has an `addEventListener` method, which allows you to hook a function to any event[3] that the element receives. The method takes three arguments: the event, the function that should be called, and a true-or-false value for *`useCapture`*. This last item relates to a rarely-used feature of DOM events called **event capture**. For the moment, we'll just set it to `false`, to indicate that we don't want to use event capture. If you'd like to get the full story, see the DOM Level 3 Events specification[3] (not for the faint of heart!).

The event is specified as a string, which is the (modern) name of the event (i.e. without the "on" prefix). The function is specified using only the name of the function; do not place brackets after it, as in `aKeyWasPressed()`, as this would call the function. We don't want to call it now; we want to call it later, when the event is fired.[4]

Now, when a key is pressed in our `<textarea>`, our `aKeyWasPressed` function will be called. Note that JavaScript no longer clutters up our HTML; much like the separation of design and content facilitated by CSS, *we've separated our page content (HTML) from our page behavior (JavaScript)*. This is an important benefit of the new technique: we can switch new event listeners in and out without altering the HTML in our page. It's the modern way!

We still haven't addressed the question we posed earlier, though: how does the `aKeyWasPressed` function know which key was pressed?

[3]We've used the `keyup` event here, rather than the more commonly expected `keypress`, because, at the time of writing, Safari on Macintosh does not support the assigning of `keypress` events using `addEventListener`. Perhaps more importantly, the DOM3 recommendation does not mention a `keypress` event.

[3] http://www.w3.org/TR/DOM-Level-3-Events/events.html#Events-flow

[4]If you have worked in other languages, you may recognize that this means that functions are first-class objects in JavaScript; we can pass around references to a function using its name, but without calling it. This procedure doesn't work in all languages, but it's a very useful feature of JavaScript.

Getting Event Information

A subtle change that we made in the above code was to give the `aKeyWasPressed` function an argument, `e`.

File: **`keycodedetect.html`** **(excerpt)**

```
function aKeyWasPressed(e) {
  ...
```

When a function is called as an event listener, it is passed, in the case of a W3C events-compliant browser, to an **event object**, which holds details of the event. This object has a number of properties containing useful information, such as target, and a reference to the element that fired the event. The precise properties that are available will depend on the type of event in question, but the most useful properties are listed in Table 3.2.

Table 3.2. Useful Properties.

Event object property	**Meaning**
`target`	The element that fired the event.
`type`	The event that was fired (e.g. `keyup`).
`button`	The mouse button that was pressed (if this is a mouse event): `0` for the left button, `1` for middle, `2` for right.
`keyCode`	The character code of the key that was pressed[5]
`shiftKey`	Whether the **Shift** key was pressed (`true` or `false`).

[5]Don't use `charCode` here, even though some Websites tell you to. `keyCode` has good cross-browser support, and `charCode` does not. Key codes in the DOM are a standards mess! There are three ways to get the code: `keyCode` (IE), `charCode` (Mozilla/Netscape) and `data` (the official DOM 3 Events way). Fortunately, all major browsers support the nonstandard `keyCode`. So always use this, at least until the `data` property is widespread (in about 2010!).

Code that identifies which key was pressed would look like this:

File: **`keycodedetect.html`** **(excerpt)**

```
function aKeyWasPressed(e) {
  var key = e.keyCode;
  alert('You pressed the key: ' + String.fromCharCode(key));
}
```

```
var textarea = document.getElementById('myta');
textarea.addEventListener('keyup', aKeyWasPressed, false);
```

When a key is pressed, our function will pop up a dialog box to tell us so.[6]

Re-using Listeners Across Targets

The target attribute might not seem very useful; after all, we know that it will be a reference to the `<textarea>`. But we can hook up the same function as an event listener on more than one element. We can, for example, attach one single function as an event listener for click events to every link in our page. When any link is clicked, our function will be called; we can then tell which link was clicked by examining the function's `e.target`. We'll come back to this in later examples in this chapter.

For now, all we need to know is that we don't have to write a separate event listener for every single tag in which we're interested.

What Happens After an Event Fires?

Events have two further important properties: **bubbling** and **default actions**. Think about an HTML document. It's hierarchical: elements are contained by other elements. Consider this HTML snippet:

```
<div>
  <p>
    <a href="">a link</a>
  </p>
</div>
```

Clicking on the link will cause that link to fire a click event. But the link is contained within the paragraph, and the paragraph is contained within the `<div>`. So clicking the link will also cause both the paragraph and the `<div>` to see the click event. This is called **event bubbling**; an event "bubbles" up through the DOM tree, starting with the target element, until it reaches the top. Not all events bubble; for example, `focus` and `blur` events do not. Bubbling can often be ignored,[7] but there are times when you'll want to prevent a specific event from bubbling.

[6]Note that we use the `String.fromCharCode` method to convert the keyboard code provided by `keyCode` to a human-readable string.
[7]There are a lot of complex rules about event bubbling and event capturing, the phase of event propagation that occurs before event bubbling. In practice, we don't need to know much beyond how to stop it happening, but a complete write-up is available at

Once you've got an event, the DOM Events specification says that you can stop any further bubbling like this:

```
function aKeyWasPressed(e) {
  var key = e.keyCode;
  e.stopPropagation();
  ...
}
```

Once the call to `stopPropagation` is in place, the event will occur on the `<a>` tag only: any listeners on the `<p>` or `<div>` tags will miss out. If there are no listeners on those other tags, there's no need to stop bubbling. In this case, the event silently passes through the parent tags, having no extra effect.

Some events have a default action. The most obvious example is clicking a link: the default action for this event is to navigate the current window or frame to the link's destination. If we wanted to handle clicks on a link entirely within our JavaScript code, we might want to prevent that default action from being taken.

In our examples so far, we have handled the `keyup` event, which is fired when a key is released. As it turns out, this event has no default action. A closely-related event that does have a default action is `keypress`, which occurs whenever a character is typed using the combination of `keydown` and `keyup`. The `keypress` event is nonstandard (i.e. it is not described by the W3C DOM standard), which is why I have avoided mentioning it until now, but it is well supported by the major browsers.

Let's say we want to prevent `keypress` events from inputting text into our `textarea`. We could do this by setting up an event listener that cancelled the default action of that type of event. The DOM standard specifies a method, named `preventDefault`, that achieves this, but again, Internet Explorer implements its own proprietary technique. Here's the DOM approach:

```
function aKeyWasPressed(e) {
  e.preventDefault();
}
var textarea = document.getElementById('myta');
textarea.addEventListener('keypress', aKeyWasPressed, false);
```

http://www.quirksmode.org/js/events_order.html for those who would like to know more of the theory underlying this aspect of the DOM.

Assigning Event Listeners on Page Load

In all of the examples we've seen so far in this chapter, the JavaScript code has had to follow the HTML code to which it assigns event listeners. If the JavaScript code were to come first, it would be unable to find the HTML elements in question, as they would not yet exist.

A solution to this problem is to assign event listeners for specific document elements in a listener assigned to the window's `load` event. As a result, event listeners will only be assigned once the document has finished loading, and all elements are available.

Here's the complete listing for our keystroke detection example, restructured in this way:

File: **`keycodedetect.html`**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Detect keystrokes</title>
    <script type="text/javascript">
      function aKeyWasPressed(e) {
        var key = e.keyCode;
        alert('You pressed the key: ' + String.fromCharCode(key));
      }

      function addListeners(e) {
        var textarea = document.getElementById('myta');
        textarea.addEventListener('keyup', aKeyWasPressed, false);
      }

      window.addEventListener('load', addListeners, false);
    </script>
  </head>
  <body>
    <form>
      <textarea id="myta"></textarea>
    </form>
  </body>
</html>
```

Our main event listener, `aKeyWasPressed`, has not been changed. What has changed is the way in which this listener is assigned. The code that assigns it has been placed inside a new function, `addListeners`:

File: **`keycodedetect.html`** **(excerpt)**

```
      function addListeners(e) {
        var textarea = document.getElementById('myta');
        textarea.addEventListener('keyup', aKeyWasPressed, false);
      }
```

This function is itself an event listener, which we assign to the `window` object's `load` event:

File: **`keycodedetect.html`** **(excerpt)**

```
      window.addEventListener('load', addListeners, false);
```

This event is fired once the document has finished loading, to signal that all HTML elements are now available. The `addListeners` function takes this opportunity to assign listeners to elements as required.

We'll continue to use this structure as we move forward through this chapter, and the rest of the book.

Making Events Work Cross-Browser

Naturally, making events work cross-browser is not as easy as just following the DOM standard. Internet Explorer doesn't implement the DOM Events model very well. Instead, it offers a proprietary and different way to hook up event listeners and gain access to event data.

Adding Event Listeners Portably

Instead of using an `addEventListener` method on an element, IE has an `attachEvent` method, and instead of passing an event object to each event listener, it has a global event object in `window.event`. This is inconvenient but not catastrophic; it just means that you have to take different actions for different browsers. In practice, what this means is that you have a small number of standard functions and techniques that you use to carry out event handling actions. One of these is the `addEvent` function, created by Scott Andrew:

File: **portabledetect.php (excerpt)**

```
function addEvent(elm, evType, fn, useCapture)
// cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
// By Scott Andrew
{
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  } else if (elm.attachEvent) {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  } else {
    elm['on' + evType] = fn;
  }
}
```

IE's `attachEvent` method is called, with an event name and a function to be the listener, but the event name should have "on" at the beginning. The `addEvent` function above takes care of the cross-browser differences;[8] simply include it in your code, then use it to attach events. As such, the code above becomes:

```
function aKeyWasPressed(e) {
  var key = e.keyCode;
  alert('You pressed the key: ' + String.fromCharCode(key));
}

function addListeners(e) {
  var textarea = document.getElementById('myta');
  addEvent(textarea, 'keyup', aKeyWasPressed, false);
}

addEvent(window, 'load', addListeners, false);

function addEvent(elm, evType, fn, useCapture)
// cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
// By Scott Andrew
{
  if (elm.addEventListener) {
```

[8]Note that if the browser doesn't support either `addEventListener` or `attachEvent`, which is the case for IE5 for Macintosh, the code assigns the event listener directly to the element as an event handler using its *`onevent`* property. This will overwrite any previous event handler that was attached to that event, which isn't good, but it's an interim solution (and better than it not working at all). There is a way around this issue, which, though it makes the code significantly more complex, does avoid this problem; details can be found in Simon Willison's Stylish Scripting blog post at http://www.sitepoint.com/blog-post-view.php?id=171578.

```
    elm.addEventListener(evType, fn, useCapture);
    return true;
  } else if (elm.attachEvent) {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  } else {
    elm['on' + evType] = fn;
  }
}
```

We're now using the `addEvent` function to make `aKeyWasPressed` listen for `keyup` events on the `textarea`.

Inspecting Event Objects Portably

This is not the only change that's required; we also have to take into account the fact that IE doesn't pass an event object to our event listener, but instead stores the event object in the `window` object. Just to make our lives as DHTML developers a little more complex, it also uses slightly different properties on the event object that it creates. These are shown in Table 3.3.

Table 3.3. W3C Event Object Properties.

W3C Event Object Property	IE `window.event` Property
`target`	`srcElement`
`type`	`type`
`button`[9]	`button`[10]
`data`[11]	`keyCode`
`shiftKey`	`shiftKey`

[9]0 = left button; 2 = right button; 1 = middle button.
[10]1 = left button; 2 = right button; 4 = middle button. For combinations, add numbers: 7 means all three buttons pressed.
[11]As previously noted, the standard `data` property is not well supported.

Taking all this into consideration, our portable code becomes:

File: **`portabledetect.html`** **(excerpt)**

```
function aKeyWasPressed(e) {
  if (window.event) {
    var key = window.event.keyCode;
  } else {
    var key = e.keyCode;
  }
  alert('You pressed the key: ' + String.fromCharCode(key));
}

function addListeners(e) {
  var textarea = document.getElementById('myta');
  addEvent(textarea, 'keyup', aKeyWasPressed, false);
}

addEvent(window, 'load', addListeners, false);

function addEvent(elm, evType, fn, useCapture)
// cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
// By Scott Andrew
{
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  } else if (elm.attachEvent) {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  } else {
    elm['on' + evType] = fn;
  }
}
```

This updated version of `aKeyWasPressed` first checks whether a `window.event` object exists:

File: **`portabledetect.html`** **(excerpt)**

```
  if (window.event) {
```

If it does, then it and its corresponding `window.event.keyCode`[12] property, are used to obtain the code of the pressed key. If not, the event object passed to the function (as `e`), which also has a `keyCode` property, is used.

Stopping Propagation and Default Actions Portably

Halting bubbling can be done in two ways, as is the case with much event handling: via the DOM approach and the Internet Explorer approach. In DOM-com-

[12]This technique for checking that something exists is called **feature sniffing**, and will be explained in more detail in the next chapter.

pliant browsers, we can prevent an event from bubbling by calling the event object's `stopPropagation` method inside the event listener.

In Internet Explorer (where there is a global `window.event` object), we set `window.event.cancelBubble` to `true` inside the event listener. In practice, the usual technique is to use feature sniffing to Do The Right Thing:

```
if (window.event && window.event.cancelBubble) {
  window.event.cancelBubble = true;
}
if (e && e.stopPropagation) {
  // e is the event object passed to this listener
  e.stopPropagation();
}
```

Unfortunately, even this doesn't cover all the major browsers. Arguably a worse offender even than Internet Explorer, Apple's Safari browser provides the `stopPropagation` method, but doesn't actually do anything when it is called. There is no easy way around this, but since event bubbling will not significantly affect any of the examples in this book, we'll just ignore this problem for now.

We also need to feature-sniff to stop default actions. With the DOM, we use the passed event object's `preventDefault` method; with Internet Explorer, we set the global `event` object's `returnValue` property to `false`.

```
if (window.event && window.event.returnValue) {
  window.event.returnValue = false;
}
if (e && e.preventDefault) {
  e.preventDefault();
}
```

Again, Safari appears to support `preventDefault`, but doesn't actually do anything when it is called. Unfortunately, preventing the default action associated with an event is a rather vital feature for many of the examples we'll look at in this book. The only way to do it in Safari (at least until Apple fixes its DOM standard event support) is to use an old-style event handler that returns `false`.

For example, to prevent the `click` event of a link from navigating to the target of the link, we would normally just use an event listener that prevented the default action of the link:

```
function cancelClick(e) {
  if (window.event && window.event.returnValue) {
    window.event.returnValue = false;
  }
  if (e && e.preventDefault) {
    e.preventDefault();
  }
}
addEvent(myLink, 'click', cancelClick, false);
```

To make this work in Safari, we need a second function, which will return `false` to cancel the event, and which we will assign as the `onclick` event handler of the link:

```
function cancelClick(e) {
  if (window.event && window.event.returnValue) {
    window.event.returnValue = false;
  }
  if (e && e.preventDefault) {
    e.preventDefault();
  }
}
function cancelClickSafari() {
  return false;
}
addEvent(myLink, 'click', cancelClick, false);
myLink.onclick = cancelClickSafari;
```

This is actually quite an ugly solution, as it will overwrite any `onclick` event handler that another script may have installed. This kind of inter-script conflict is what modern event listeners are designed to avoid. Unfortunately, there is simply no better way around the problem in Safari. We'll see an example of this solution in practice later in this chapter.

This sort of cross-browser coding is obviated to a large extent by browser manufacturers coming together to implement the W3C DOM, but for event handling it's still required.

Smart Uses of Events

That's enough about how events work. Let's see a couple of practical examples. You should also know enough now to fully understand the image rollover code we saw in Chapter 2.

Creating Smarter Links

Some Websites open all clicked links in a new window. Often, they do this with the intention that the user will return to their site more readily if it's still open in another browser window. Some users find this useful; others find it heartily annoying. It would be possible, given our event-handling techniques above, to give them the choice.

Imagine we placed a checkbox on the page, which, initially unchecked, was accompanied by the label Open links in new window. Clicking any link will open that link in a new window if the box is checked.

We could implement this functionality using a combination of event listeners: we attach to each link on the page a click listener, which investigates the checkbox and opens the corresponding link in a new window if the box is checked. We also need a listener to run upon page load, to actually attach the listener to each link.

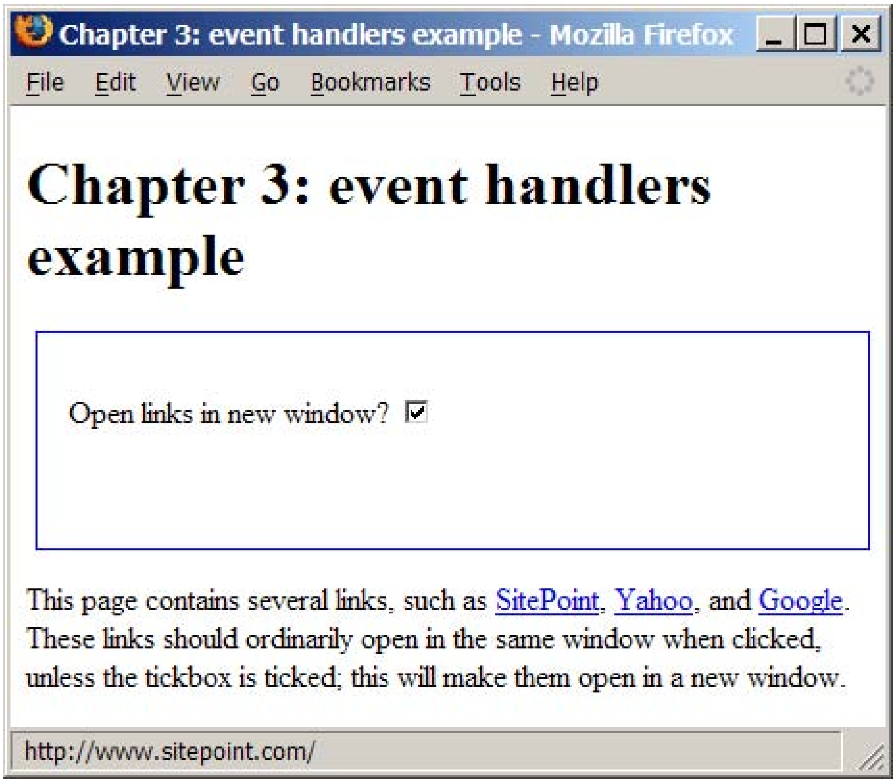
First, here's the HTML page we'll work on:

File: **smartlinks.html**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Smart Links</title>
    <script type="text/javascript" src="smartlink.js"></script>
    <style type="text/css">
      form {
        float: right;
        width: 25em;
        height: 5em;
        border: 1px solid blue;
        padding: 1em;
      }
    </style>
  </head>
  <body>
    <h1>Smart Links</h1>
    <form action=""><p>
      <label for="newwin">Open links in new window?
        <input type="checkbox" id="newwin">
      </label>
    </p></form>
    <p>This page contains several links, such as
      <a href="http://www.sitepoint.com/">SitePoint</a>,
      <a href="http://www.yahoo.com/">Yahoo!</a>, and
      <a href="http://www.google.com/">Google</a>.
      These links should ordinarily open in the same window when
      clicked, unless the checkbox is checked; this will make them
      open in a new window.
    </p>
  </body>
</html>
```

As you can see, this page is quite simple, and contains no JavaScript except for the file that the `<script>` tag brings in. Figure 3.1 shows how the code displays:

Figure 3.1. The example "smart links" Web page.


Chapter 3: event handlers example - Mozilla Firefox
File
Edit
View
Go
Bookmarks
Tools
Help
http://www.sitepoint.com/

Chapter 3: event handlers example

Open links in new window? ☑

This page contains several links, such as SitePoint, Yahoo, and Google. These links should ordinarily open in the same window when clicked, unless the tickbox is ticked; this will make them open in a new window.

Next, let's look at the content of `smartlink.js`. This code has been assembled from our earlier discussions, although it contains some extra code for this particular page. First, here's an outline of what the script holds:

File: **`smartlink.js`** **(excerpt)**

```
function addEvent(elm, evType, fn, useCapture) { ... }
function handleLink(e) { ... }
function cancelClick() { ... }
function addListeners(e) { ... }

addEvent(window, 'load', addListeners, false);
```

And here are those four items in detail:

File: **`smartlink.js`**

```
function addEvent(elm, evType, fn, useCapture) {
  // cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
  // By Scott Andrew
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  } else if (elm.attachEvent) {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  } else {
    elm['on' + evType] = fn;
  }
}

function handleLink(e) {
  var el;
  if (window.event && window.event.srcElement)
    el = window.event.srcElement;
  if (e && e.target)
    el = e.target;
  if (!el)
    return;

  while (el.nodeName.toLowerCase() != 'a' &&
      el.nodeName.toLowerCase() != 'body')
    el = el.parentNode;

  if (document.getElementById('newwin') &&
      document.getElementById('newwin').checked) {
    window.open(el.href);
    if (window.event) {
      window.event.cancelBubble = true;
      window.event.returnValue = false;
    }
    if (e && e.stopPropagation && e.preventDefault) {
      e.stopPropagation();
      e.preventDefault();
    }
  }
}

function cancelClick() {
  if (document.getElementById('newwin') &&
      document.getElementById('newwin').checked) {
    return false;
  }
  return true;
}

function addListeners() {
  if (!document.getElementById)
    return;

  var all_links = document.getElementsByTagName('a');
  for (var i = 0; i < all_links.length; i++) {
    addEvent(all_links[i], 'click', handleLink, false);
    all_links[i].onclick = cancelClick;
  }
}

addEvent(window, 'load', addListeners, false);
```

Our code includes the now-familiar `addEvent` function to carry out cross-browser event hookups. We use it to call the `addListeners` function once the page has loaded.

The `addListeners` function uses another familiar technique; it iterates through all the links on the page and does something to them. In this case, it attaches the `handleLink` function as a `click` event listener for each link, so that when a link is clicked, that function will be called. It also attaches the `cancelClick` function as the old-style `click` event listener for each link—this will permit us to cancel the default action of each link in Safari.

When we click a link, that link fires a `click` event, and `handleLink` is run. The function does the following:

File: **smartlink.js (excerpt)**

```
if (window.event && window.event.srcElement)
  el = window.event.srcElement;
if (e && e.target)
  el = e.target;
if (!el)
  return;
```

This is the cross-browser approach to identifying which link was clicked; we check for a `window.event` object and, if it exists, use it to get `window.event.srcElement`, the clicked link. Alternatively, if `e`, the passed-in parameter, exists, and `e.target` exists, then we use that as the clicked link. If we've checked for both `e` and `e.target`, but neither exists, we give up and exit the function (with `return`).

Next up, we want to make sure that we have a reference to our link element:

File: **smartlink.js (excerpt)**

```
while (el.nodeName.toLowerCase() != 'a' &&
    el.nodeName.toLowerCase() != 'body')
  el = el.parentNode;
if (el.nodeName.toLowerCase() == 'body')
  return;
```

Some browsers may pass the text node inside a link as the clicked-on node, instead of the link itself. If the clicked element is not an `<a>` tag, we ascend the DOM tree, getting its parent (and that node's parent, and so on) until we get to the `a` element. (We also check for `body`, to prevent an infinite loop; if we get as far up the tree as the document `body`, we give up.)

Note that we also use `toLowerCase` on the `nodeName` of the element. This is the easiest way to ensure that a browser that returns a `nodeName` of `A`, and one that returns a `nodeName` of `a`, will both be handled correctly by the function.

Next, we check our checkbox:

File: **smartlink.js (excerpt)**

```
if (document.getElementById('newwin') &&
    document.getElementById('newwin').checked) {
```

We first confirm (for paranoia's sake) that there *is* an element with `id newwin` (which is the checkbox). Then, if that checkbox is checked, we open the link in a new window:

File: **smartlink.js (excerpt)**

```
    window.open(el.href);
```

We know that `el`, the clicked link, is a link object, and that link objects have an `href` property. The `window.open` method creates a new window and navigates it to the specified URL.

Finally, we take care of what happens afterward:

File: **smartlink.js (excerpt)**

```
    if (window.event) {
      window.event.cancelBubble = true;
      window.event.returnValue = false;
    }
    if (e && e.stopPropagation && e.preventDefault) {
      e.stopPropagation();
      e.preventDefault();
    }
  }
```

We don't want the link to have its normal effect of navigating the current window to the link's destination. So, in a cross-browser fashion, we stop the link's normal action from taking place.

As previously mentioned, Safari doesn't support the standard method of cancelling the link's default action, so we have an old-style event listener, `cancelClick`, that will cancel the event in that browser:

File: **smartlink.js (excerpt)**

```
function cancelClick() {
  if (document.getElementById('newwin') &&
      document.getElementById('newwin').checked) {
    return false;
  }
  return true;
}
```

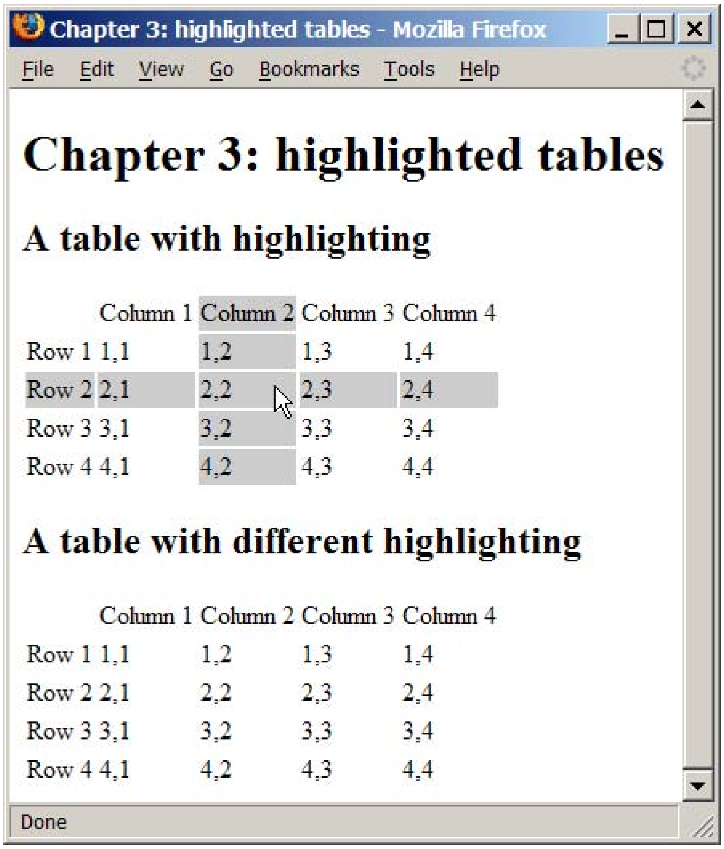
You can see that some of this code is likely to appear in every project we attempt, particularly those parts that have to do with listener installation.

Making Tables More Readable

A handy trick that many applications use to display tables of data is to highlight the individual row and column that the viewer is looking at; paper-based tables often shade table rows and columns alternately to provide a similar (although non-dynamic[13]) effect.

Here's a screenshot of this effect in action. Note the location of the cursor. If we had another cursor, you could see that the second table is highlighted differently. But we don't, so you'll just have to try the example code for yourself…

Figure 3.2. Example of table highlighting in a Web page.


Chapter 3: highlighted tables - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Done

Chapter 3: highlighted tables

A table with highlighting

	Column 1	Column 2	Column 3	Column 4
Row 1	1,1	1,2	1,3	1,4
Row 2	2,1	2,2	2,3	2,4
Row 3	3,1	3,2	3,3	3,4
Row 4	4,1	4,2	4,3	4,4

A table with different highlighting

	Column 1	Column 2	Column 3	Column 4
Row 1	1,1	1,2	1,3	1,4
Row 2	2,1	2,2	2,3	2,4
Row 3	3,1	3,2	3,3	3,4
Row 4	4,1	4,2	4,3	4,4

We can apply this effect to tables in an HTML document using event listeners. We'll attach a `mouseover` listener to each cell in a table, and have that listener highlight all the other cells located in that cell's row and column. We'll also attach a `mouseout` listener that turns the highlight off again.

[13]…until paper technology gets a lot cooler than it is now, at any rate!

The techniques we have explored in this chapter are at their most powerful when we combine the dynamic capabilities of DHTML with the page styling of CSS. Instead of specifically applying a highlight to each cell we wish to illuminate, we'll just apply a new class, `hi`, to those cells; our CSS will define exactly how table cells with class `hi` should be displayed. To change the highlight, simply change the CSS. For a more powerful effect still, use CSS's selectors to apply different styles to highlighted cells depending on the table in which they appear.

Here's an example page that contains tables:

File: **tableHighlight.html**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Highlighted Tables</title>
    <script type="text/javascript" src="tableHighlight.js">
    </script>
    <style type="text/css">
      tr.hi td, td.hi {
        background-color: #ccc;
      }
      table.extra tr.hi td, table.extra td.hi {
        color: red;
        text-decoration: underline overline;
        background-color: transparent;
      }
    </style>
  </head>
  <body>
    <h1>Highlighted Tables</h1>

    <h2>A table with highlighting</h2>
    <table>
      <tr>
        <td></td>
        <td>Column 1</td>
        <td>Column 2</td>
        <td>Column 3</td>
        <td>Column 4</td>
      </tr>
      <tr>
        <td>Row 1</td>
        <td>1,1</td><td>1,2</td><td>1,3</td><td>1,4</td>
      </tr>
      <tr>
        <td>Row 2</td>
        <td>2,1</td><td>2,2</td><td>2,3</td><td>2,4</td>
      </tr>
      <tr>
       <td>Row 3</td>
       <td>3,1</td><td>3,2</td><td>3,3</td><td>3,4</td>
      </tr>
      <tr>
        <td>Row 4</td>
        <td>4,1</td><td>4,2</td><td>4,3</td><td>4,4</td>
      </tr>
    </table>

    <h2>A table with different highlighting</h2>
    <table class="extra">
      <tr>
        <td></td>
        <td>Column 1</td>
        <td>Column 2</td>
        <td>Column 3</td>
        <td>Column 4</td>
      </tr>
      <tr>
        <td>Row 1</td>
        <td>1,1</td><td>1,2</td><td>1,3</td><td>1,4</td>
      </tr>
      <tr>
        <td>Row 2</td>
        <td>2,1</td><td>2,2</td><td>2,3</td><td>2,4</td>
      </tr>
      <tr>
        <td>Row 3</td>
        <td>3,1</td><td>3,2</td><td>3,3</td><td>3,4</td>
      </tr>
      <tr>
        <td>Row 4</td>
        <td>4,1</td><td>4,2</td><td>4,3</td><td>4,4</td>
      </tr>
    </table>
  </body>
</html>
```

That code creates two four-by-four tables, each with column and row headings (so each table contains five rows and five columns in total). Notice that none of the styles have any effect because, as yet, there are no elements with `class="hi"`.

Let's look at the matching `tableHighlight.js` script. Its structure reflects our earlier discussions, but it contains some additional code for this particular technique. Here's an outline of the script:

File: **`tableHighlight.js`** **(excerpt)**

```
function addEvent(elm, evType, fn, useCapture) { ... }
function ascendDOM(e, target) { ... }
function hi_cell(e) { ... }
function lo_cell(e) { ... }
function addListeners() { ... }

addEvent(window, 'load', addListeners, false);
```

Notice how similar the function outline is to the smart links example. Here are the six items in all their detail.

File: **`tableHighlight.js`**

```
function addEvent(elm, evType, fn, useCapture)
// cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
// By Scott Andrew
{
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  } else if (elm.attachEvent) {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  } else {
    elm['on' + evType] = fn;
  }
}

// climb up the tree to the supplied tag.
function ascendDOM(e, target) {
  while (e.nodeName.toLowerCase() != target &&
      e.nodeName.toLowerCase() != 'html')
    e = e.parentNode;

  return (e.nodeName.toLowerCase() == 'html') ? null : e;
}

// turn on highlighting
function hi_cell(e) {
  var el;
  if (window.event && window.event.srcElement)
```

```
    el = window.event.srcElement;
  if (e && e.target)
    el = e.target;
  if (!el) return;

  el = ascendDOM(el, 'td');
  if (el == null) return;

  var parent_row = ascendDOM(el, 'tr');
  if (parent_row == null) return;

  var parent_table = ascendDOM(parent_row, 'table');
  if (parent_table == null) return;

  // row styling
  parent_row.className += ' hi';

  // column styling
  var ci = -1;
  for (var i = 0; i < parent_row.cells.length; i++) {
    if (el === parent_row.cells[i]) {
      ci = i;
    }
  }
  if (ci == -1) return; // this should never happen

  for (var i = 0; i < parent_table.rows.length; i++) {
    var cell = parent_table.rows[i].cells[ci];
    cell.className += ' hi';
  }
}

// turn off highlighting
function lo_cell(e) {
  var el;
  if (window.event && window.event.srcElement)
    el = window.event.srcElement;
  if (e && e.target)
    el = e.target;
  if (!el) return;

  el = ascendDOM(el, 'td');
  if (el == null) return;

  var parent_row = ascendDOM(el, 'tr');
  if (parent_row == null) return;
  var parent_table = ascendDOM(parent_row, 'table');
  if (parent_table == null) return;

  // row de-styling
  parent_row.className =
      parent_row.className.replace(/\b ?hi\b/, '');

  // column de-styling
  var ci = el.cellIndex;
  for (var i = 0; i < parent_table.rows.length; i++) {
    var cell = parent_table.rows[i].cells[ci];
    cell.className = cell.className.replace(/\b ?hi\b/, '');
  }
}

function addListeners() {
  if (!document.getElementsByTagName) return;

  var all_cells = document.getElementsByTagName('td');
  for (var i = 0; i < all_cells.length; i++) {
    addEvent(all_cells[i], 'mouseover', hi_cell, false);
    addEvent(all_cells[i], 'mouseout', lo_cell, false);
  }
}

addEvent(window, 'load', addListeners, false);
```

We add our `mouseover` and `mouseout` event listeners using the standard approach. The `addListeners` function sets up our `hi_cell` and `lo_cell` functions as `mouseover` and `mouseout` event listeners, respectively.

To minimize duplicate code, we've added a handy little utility function called `ascendDOM`. This marches up the tree from the element supplied in the first argument to find the first enclosing tag whose name matches the second argument.

Processing happens as follows. Mousing over a table cell triggers the `hi_cell` function. This finds the moused-over cell, then calculates the row and the table in which that cell appears. The `ascendDOM` function is called quite often in the code, so you can see the benefit of putting that code into a function. In `hi_cell`, the lines that actually do the styling work are these:

File: **`tableHighlight.js`** **(excerpt)**

```
  parent_row.className += ' hi';
```

File: **`tableHighlight.js`** **(excerpt)**

```
    cell.className += ' hi';
```

The rest of the code is simply concerned with picking out the right elements for these lines to work on.

Our intention here is to apply the class `hi` to the other cells in the row that contains the moused-over cell, and its column. The first line above executes the first task. The second line applies the class to a given cell, but our script needs to find the appropriate cells first.

This is where things get a little complicated. The row is a simple `<tr>` tag, whereas the column is a list of cells scattered across all the rows in the table. According to the DOM Level 2 specification, table cell elements have a `cellIndex` property, which indicates the cell's index in the row. To find the other cells in this column, we could iterate through all the rows in the table and find within each row the cell that has the same `cellIndex`.

Sadly, Safari doesn't properly support `cellIndex`—it is always set to `0`, no matter what the actual index should be. If Safari supported `cellIndex`, the process could have been simple:

```
  var ci = el.cellIndex;
```

In fact, this concise snippet must be replaced with the much longer section below:

File: **`tableHighlight.js`** **(excerpt)**

```
  var ci = -1;
  for (var i = 0; i < parent_row.cells.length; i++) {
    if (el === parent_row.cells[i]) {
      ci = i;
    }
  }
  if (ci == -1) return; // this should never happen
```

`ci` is the `cellIndex`, and can be used to highlight other cells with the same `cellIndex` in the other rows in the table:

File: **`tableHighlight.js`** **(excerpt)**

```
  for (var i = 0; i < parent_table.rows.length; i++) {
    var cell = parent_table.rows[i].cells[ci];
    cell.className += ' hi';
  }
```

All the table's rows are held in the table's `rows` array. We walk through that array, applying the `hi` class to the cell in each row that has the same index as the moused-over cell.

The upshot of this exercise is that all the cells in the same column as the moused-over cell will have class `hi`; the table row containing the cell will also have class `hi`.

Our CSS code takes care of the appearance of these cells:

File: **`tableHighlight.html`** **(excerpt)**

```
      tr.hi td, td.hi {
        background-color: #ccc;
      }
```

We've applied a background color of class `hi` to both `td`s, and `td`s in a `tr` of class `hi`; thus, these cells will be highlighted. The `lo_cell` function works similarly, except that it removes the class `hi` from the row and column rather than applying it. The removal is done with the following lines:

File: **`tableHighlight.js`** **(excerpt)**

```
  parent_row.className =
      parent_row.className.replace(/\b ?hi\b/, '');
```

File: **`tableHighlight.js`** **(excerpt)**

```
    cell.className = cell.className.replace(/\b ?hi\b/, '');
```

Since a `className` is a string, it has all the methods of a string, one of which is `replace`; we can call the `replace` method with a regular expression (first parameter) and a substitute string (second parameter). If a match for the regular expression is found in the string, it is replaced by the substitute string. In our example, we look for matches to the expression `\b ?hi\b` (note that regular expressions are delimited by slashes, not quotes)—that is, a word boundary followed by an optional space, the word 'hi', and another word boundary—and replace it with a blank string, thus removing it from the `className`.

An added bonus of using CSS to provide the style information is that we can apply different highlighting to different tables on the page without changing the script. For example, the HTML of the page contains two tables, one with a class of `extra`. We apply some CSS specifically to tables with class `extra`:

File: **`tableHighlight.html`** **(excerpt)**

```
table.extra tr.hi td, table.extra td.hi {
  color: red;
  text-decoration: underline overline;
  background-color: transparent;
}
```

As a result, the highlighted cells in that particular table will be highlighted differently. CSS makes achieving this kind of effect very easy.

Summary

Understanding the processes by which events are fired, and by which code is hooked to those events, is vital to DHTML programming. Almost everything you do in DHTML will involve attaching code to events, as described in this chapter. We've examined some common events and the two browser models for listening to them. We have also covered what happens when an event fires, and how you can interrupt or alter that process. Finally, we looked at a few events in detail, and saw some simple examples of how code can attach to those events and improve the user experience on sites that employ these techniques.

4 Detecting Browser Features

You just listed all my best features.
—The Cat, *Red Dwarf*, Series 3, Episode *DNA*

An important design constraint when adding DHTML to your Websites is that it should be unobtrusive. By "unobtrusive," I mean that if a given Web browser doesn't support the DHTML features you're using, that absence should affect the user experience as little as possible. Errors should not be shown to the user: the site should be perfectly usable without the DHTML enhancements. The browsers that render your site will fall into the following broad categories:

1. Offer no JavaScript support at all, or have JavaScript turned off.
2. Provide some JavaScript support, but modern features are missing.
3. Have full JavaScript support, but offer no W3C DOM support at all.
4. Provide incomplete DOM support, but some DOM features are missing or buggy.
5. Offer complete DOM support without bugs.

The first and the last categories hold no concerns for you as a DHTML developer. A browser that does not run JavaScript at all will simply work without calling any of your DHTML code, so you can ignore it for the purposes of this discussion.

You just need to make sure that your page displays correctly when JavaScript is turned off.[1] Similarly, a browser that implements the DOM completely and without bugs would make life very easy. It's a shame that such browsers do not exist.

The three categories in the middle of the list are of concern to us in this chapter. Here, we'll explore how to identify which DHTML features are supported by a given browser before we try to utilize those features in running our code.

There are basically two ways[2] to working out whether the browser that's being used supports a given feature. The first approach is to work out which browser is being used, then have a list within your code that states which browser supports which features. The second way is to test for the existence of a required feature directly. In the following discussion, we'll see that classifying browsers by type isn't as good as detecting features on a case-by-case basis.

Old-Fashioned Browser Sniffing

In the bad old days, before browser manufacturers standardized on the DOM, JavaScript developers relied on detection of the browser's brand and version via a process known as browser sniffing. Each browser provides a `window.navigator` object, containing details about the browser, which can be checked from JavaScript. We can, for example, find the name of the browser (the "user agent string") as follows:

```
var browserName = navigator.userAgent;
var isIE = browserName.match(/MSIE/); // find IE and look-alikes
```

Don't do this any more! This technique, like many other relics from the Dark Ages of JavaScript coding (before the W3C DOM specifications appeared), *should not be used*. Browser sniffing is flaky and prone to error, and should be avoided like the black plague. *Really*: I'm not kidding here.

Why am I so unenthusiastic about browser sniffing? There are lots of reasons. Some browsers lie about, or attempt to disguise, their true details; some, such as Opera, can be configured to deliver a user agent string of the user's choice. It's pretty much impossible to stay up-to-date with every version of every browser,

[1] For example, if your DHTML shows and hides some areas of the page, those areas should show initially, then be hidden with DHTML, so that they are available to non-DHTML browsers.
[2]Actually, there's a third way to identify browser support. The DOM standards specify a `document.implementation.hasFeature` method that you can use to detect DOM support. It's rarely used, though.

and it's definitely impossible to know which features each version supported upon its release. Moreover, if your site is required to last for any reasonable period of time, new browser versions will be released after your site, and your browser-sniffing code will be unable to account for them. Browser sniffing—what little of it remains—should be confined to the dustbin of history. Put it in the "we didn't know any better" category. There is a significantly better method available: feature sniffing.

Modern DOM Feature Sniffing

Instead of detecting the user's browser, then working out for yourself whether it supports a given feature, simply ask the browser directly whether it supports the feature. For example, a high proportion of DHTML scripts use the DOM method `getElementById`. To work out whether a particular visitor's browser supports this method, you can use:

```
if (document.getElementById) {
  // and here you know it is supported
}
```

If the `if` statement test passes, we know that the browser supports the feature in question. It is important to note that `getElementById` is not followed by brackets! We do not say:

```
if (document.getElementById())
```

If we include the brackets, we call the method `getElementById`. If we do not include the brackets, we're referring to the JavaScript `Function` object that underlies the method. This is a very important distinction. Including the brackets would mean that we were testing the return value of the method call, which we do not want to do. For a start, this would cause an error in a non-DOM browser, because we can't call the `getElementById` method there at all—it doesn't exist! When we test the `Function` object instead, we're assessing it for existence. Browsers that don't support the method will fail the test. Therefore, they will not run the code enclosed by the `if` statement; nor will they display an error.

This feature of JavaScript—the ability to test whether a method exists—has been part of the language since its inception; thus, it is safe to use it on even the oldest JavaScript-supporting browsers. You may recall from the previous chapter the technique of referring to a `Function` object without calling it. In Chapter 3, we used it to assign a function as an event listener without actually calling it. In

JavaScript, everything can be treated as an object if you try hard enough; methods are no exception!

Which DOM Features Should We Test?

The easiest approach is to test for every DOM method you intend to use. If your code uses `getElementById` and `createElement`, test for the existence of both methods. This will cover browsers in the fourth category above: the ones that implement some—but not all—of the DOM.

It is not reasonable to assume that a browser that supports `getElementById` also supports `getElementsByTagName`. You must explicitly test for each feature.

Where Should We Test for DOM Features?

An easy way to handle these tests is to execute them before your DHTML sets up any event listeners. A large subset of DHTML scripts work by setting on page load some event listeners that will be called as various elements in the browser fire events. If, before setting up the event listeners, you check that the browser supplies all the DOM features required by the code, event listeners will not be set up for browsers that do not support those features. You can therefore reasonably assume in setting up your event listeners that all the features you require are available; this assumption can simplify your code immensely. Here's an example:

```
function myScriptInit() {
  if (!document.getElementById ||
      !document.getElementsByTagName ||
      !document.createElement) {
    return;
  }
  // set up the event listeners here
}

function myScriptEventListener() {
  var foo = document.getElementById('foo');  // safe to use
}

addEvent(window, 'load', myScriptInit, false);
```

This script contains a `myScriptInit` function, which sets up `myScriptEventListener` as an event listener. But, before we set up that listener, we check for the existence of the DOM methods `getElementById`, `getElementsByTagName`, and `createElement`.

The `if` statement says: "if the JavaScript `Function` object `document.getElementById` does not exist, or if the `Function` object `document.getElementsByTagName` does not exist, or if the `Function` object `document.createElement` does not exist, exit the `myScriptInit` function." This means that, should any of those objects not be supported, the `myScriptInit` function will exit at that point: it will not even get as far as setting up the event listeners. Our code will set up listeners only on browsers that do support those methods. Therefore, as above, the listener function `myScriptEventListener` can feel safe in using `document.getElementById` without first checking to ensure that it is supported. If it wasn't supported, the listener function would not have been set up.

All this sniffing relies on JavaScript's runtime behavior. Even though the scripts are read by the browser at load time, no checks are done on the objects stated in the scripts until the code is run. This allows us to put browser objects in all scripts, and use them only when our detection code gets around to it: an arrangement called **late binding**.

Testing Non-DOM Features

Feature sniffing can be used on any JavaScript object: not just methods, and not just those methods that are part of the DOM. Commonly used examples are the offset properties (`offsetWidth`, `offsetHeight`, `offsetLeft` and `offsetTop`) of an element. These JavaScript properties are an extension to the DOM provided by all the major browsers. They return information on the size and position of an element in pixels. We can test whether those properties are defined on a given element's object as follows:

```
var foo = document.getElementById('foo');

if (typeof foo.offsetHeight != 'undefined') {
  var fooHeight = foo.offsetHeight;
}
```

Here, we set `fooHeight` if, and only if, `offsetHeight` is supported on `foo`. This is a different type of check from the method we used before, though: isn't it possible simply to say, `if (foo.offsetHeight)`? This isn't a good approach to use. If `foo.offsetHeight` is not defined, `if (foo.offsetHeight)` will not be true, just as we expect. However, the `if` statement will also fail if `foo.offsetHeight` does exist, but is equal to `0` (zero). This is possible because JavaScript treats zero as meaning `false`. Testing whether a given item is defined just got a little more complex (but only a little!).

If you are testing for the existence of function `functionName`, or method `methodName` (on an object `obj`), use the function/method name without the brackets to do so:

```
if (functionName) { ... }
if (obj.methodName) { ... }
```

Likewise, if you're testing for a variable `v`, or for a DOM property `prop` of an object, you can often use the variable or the DOM attribute's property name directly:

```
if (v) { ... }
if (obj.prop) { ... }
```

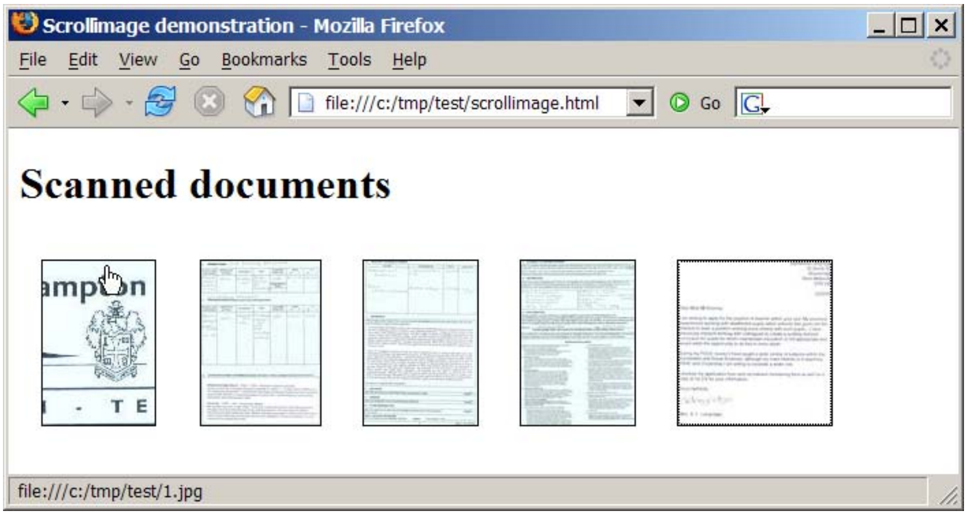
But, watch out! If the variable or property contains numbers or strings (as does `offsetHeight`, for example) then use `typeof`, because a number might be `0` (zero), and a string might be the empty string `""`, both which also evaluate to `false`:

```
if (typeof v != 'undefined') { ... }
if (typeof obj.prop != 'undefined') { ... }
```

Sniffing at Work: `scrollImage`

Lots of Websites contain photo galleries: pages listing thumbnails of photographs that, when clicked on, display the photos at full size. An interesting enhancement to such a site might be to let the user see the full-size photo without having to click to load it. When the user mouses over the thumbnail, that thumbnail could become a "viewing area" in which a snippet of the full-sized image is shown. This technique is useful if your thumbnails aren't detailed enough to enable users to tell the difference between superficially similar images. It's especially handy if your thumbnails display something like a document, rather than a photo. Figure 4.1 shows the final effect:

Figure 4.1. The thumbnail display implemented by the `scrollImage` example.


ScrollImage demonstration - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///c:/tmp/test/scrollimage.html
Go

Scanned documents

file:///c:/tmp/test/1.jpg

We'll describe what's going on here in a moment. We'll review the code first, then see a demonstration before we get to the explanation.

Setting Up the Page

The HTML file for this technique is straightforward:

File: **`scrollImage.html`**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>ScrollImage demonstration</title>
    <script src="scrollImage.js" type="text/javascript"></script>
    <style type="text/css">
      .scrollimage {
        display: block;
        float: left;
        border: 1px solid black;
        margin: 1em;
        padding: 0;
      }
      .scrollimage:hover {
        position: relative;
      }

      .scrollimage img {
        border: none;
      }

      .scrollimage:hover img {
        display: none;
      }
    </style>
  </head>
  <body>

    <h1>Scanned documents</h1>

    <p>
      <a href="1.jpg" class="scrollimage"
         mainx="563" mainy="823" thumbx="82" thumby="120"
         style="background: url(1.jpg); width: 82px;
         height: 120px;"
      ><img src="1-thumb.jpg"></a>

      <a href="2.jpg" class="scrollimage"
         mainx="563" mainy="777" thumbx="87" thumby="120"
         style="background: url(2.jpg); width: 87px;
         height: 120px;"
      ><img src="2-thumb.jpg"></a>

      <a href="3.jpg" class="scrollimage"
         mainx="567" mainy="823" thumbx="83" thumby="120"
         style="background: url(3.jpg); width: 83px;
         height: 120px;"
      ><img src="3-thumb.jpg"></a>

      <a href="4.jpg" class="scrollimage"
         mainx="558" mainy="806" thumbx="83" thumby="120"
         style="background: url(4.jpg); width: 83px;
         height: 120px;"
      ><img src="4-thumb.jpg"></a>

      <a href="5.jpg" class="scrollimage"
         mainx="434" mainy="467" thumbx="112" thumby="120"
         style="background: url(5.jpg); width: 112px;
         height: 120px;"
      ><img src="5-thumb.jpg"></a>
    </p>

  </body>
</html>
```

The content of this page is fairly obvious. Notice how the image elements are hidden by CSS styles when the mouse moves over them. This page also includes—with the `<script src="scrollImage.js" type="text/javascript"></script>` line—this JavaScript file:

File: **`scrollImage.js`**

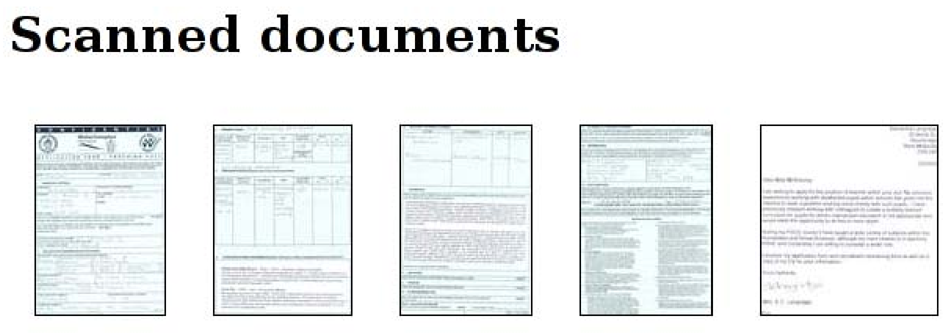
```
// Based on findPos*, by ppk
// (http://www.quirksmode.org/js/findpos.html)
function findPosX(obj) {
  var curLeft = 0;
  if (obj.offsetParent) {
    do {
      curLeft += obj.offsetLeft;
    } while (obj = obj.offsetParent);
  }
  else if (obj.x) {
    curLeft += obj.x;
  }
  return curLeft;
}

function findPosY(obj) {
  var curTop = 0;
  if (obj.offsetParent) {
    do {
      curTop += obj.offsetTop;
    } while (obj = obj.offsetParent);
  }
  else if (obj.y) {
    curTop += obj.y;
  }
  return curTop;
}

// cross-browser event handling for IE5+, NS6+ and Mozilla/Gecko
// By Scott Andrew
function addEvent(obj, evType, fn, useCapture) {
  if (obj.addEventListener) {
    obj.addEventListener(evType, fn, useCapture);
    return true;
  } else if (obj.attachEvent) {
    var r = obj.attachEvent('on' + evType, fn);
    return r;
  } else {
    obj['on' + evType] = fn;
  }
}

addEvent(window, 'load', scrollInit, false);

function scrollInit() {
  if (!document.getElementsByTagName)
    return;
  var allLinks = document.getElementsByTagName('a');
  for (var i = 0; i < allLinks.length; i++) {
    var link = allLinks[i];
    if ((' ' + link . className + ' ').indexOf(' scrollimage ') !=
        -1) {
      addEvent(link, 'mousemove', moveListener, false);
    }
  }
}

function attVal(element, attName) {
  return parseInt(element.getAttribute(attName));
}

function moveListener(ev) {
  var e = window.event ? window.event : ev;
  var t = e.target ? e.target : e.srcElement;

  var xPos = e.clientX - findPosX(t);
  var yPos = e.clientY - findPosY(t);

  if (t.nodeName.toLowerCase() == 'img')
    t = t.parentNode;
  if (t.nodeName.toLowerCase() == 'a') {

    // scaleFactorY = (width(big) - width(small)) / width(small)
    var scaleFactorY =
        (attVal(t, 'mainy') - attVal(t, 'thumby')) / attVal(t,
        'thumby');
    var scaleFactorX =
        (attVal(t, 'mainx') - attVal(t, 'thumbx')) / attVal(t,
        'thumbx');
    t.style.backgroundPosition =
        (-parseInt(xPos * scaleFactorX)) + 'px ' +
        (-parseInt(yPos * scaleFactorY)) + 'px';
  }
}
```

We'll explore (and fix!) this code shortly. Finally, the page also contains images: five at full-size, and five thumbnails. You can find them in the code archive for this book.

Demonstrating the DHTML Effect

Let's see how the page works. The HTML document shows five images as thumbnails; in this example, they're thumbnails of individual pages of a scanned-in document. Figure 4.2 shows the page content under normal circumstances.

Figure 4.2. Thumbnails of a document.

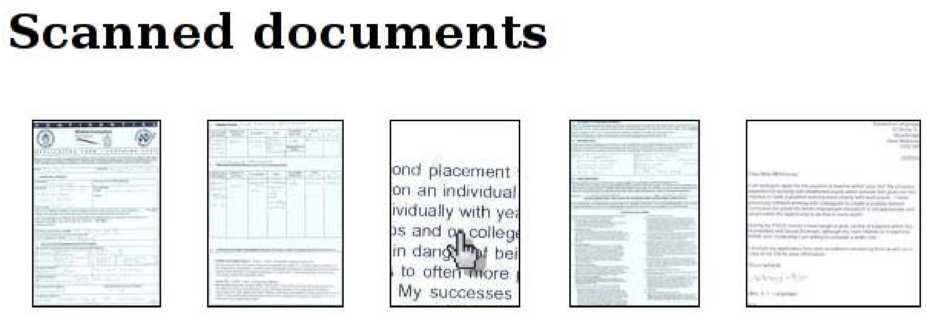


Scanned documents

When we mouse-over a thumbnail image, though, the display of that thumbnail changes to show the actual image to which it's linked, as shown in Figure 4.3.

The thumbnail becomes a viewing area in which we can see a snippet of the full-size image. As the cursor moves over the third image, we see the content of the third image at full size through the viewing area. For a document thumbnail such as this, we can use the cursor to move around the document within the viewing area, so that we can read the content and see if it's the document we want. This technique can also be useful, as mentioned, in photo galleries containing images that look similar when displayed at thumbnail size.

Figure 4.3. Mousing over a thumbnail.



How the Code Works

Conceptually, the code works as follows: we set up the page so that every "scrollable" image is made up of an `<a>` tag of class `scrollimage`, which contains an `<img>` tag displaying the thumbnail. We apply the full-size image as the CSS background image of the `<a>` tag. Then, when the user mouses over the `a` element, we hide the `img` element entirely, allowing the `a` element's background image to show through. We then manipulate the position of that background image so that it moves in accordance with the cursor.[3]

This is all fairly advanced stuff, so we need to confirm that the running browser supports all the features we need in order to make it work. We start by making the script initialize on page load with the line:

File: **`scrollImage.js`** **(excerpt)**

```
addEvent(window, 'load', scrollInit, false);
```

We saw the `addEvent` method in Chapter 3, but, with what we've learned about feature detection, its workings should now be much clearer to you. First, we check for the existence of an `addEventListener` method on the passed object, to see if the user's browser supports the DOM Events model correctly:

[3]We're storing the dimensions of the larger image in custom attributes on the `a` element: `mainx`, `mainy`, `thumbx`, and `thumby`. This is a slightly suspect technique: it will prevent the HTML from validating, and should therefore be approached with caution. In this case, however, it is the easiest way to tie the required values to each of the `a` elements.

File: **scrollImage.js (excerpt)**

```
function addEvent(obj, evType, fn, useCapture) {
  if (obj.addEventListener) {
    obj.addEventListener(evType, fn, useCapture);
    return true;
```

Failing that, we look for Internet Explorer's proprietary `attachEvent` method on the object.

File: **scrollImage.js (excerpt)**

```
  } else if (obj.attachEvent) {
    var r = obj.attachEvent('on' + evType, fn);
    return r;
```

Failing *that*, we attach the event listener directly to the element, as an event handler; this is required for IE5 on Macintosh.

File: **scrollImage.js (excerpt)**

```
  } else {
    obj['on' + evType] = fn;
  }
```

This procedure caters for all the ways by which we might attach an event listener, using feature sniffing to see which option is available.

The initialization function that sets up the scrolling effect, `scrollInit`, uses `document.getElementsByTagName` to find all the `a` elements in the document. Therefore, `scrollInit` checks for this method's existence before proceeding:

File: **scrollImage.js (excerpt)**

```
function scrollInit() {
  if (!document.getElementsByTagName)
    return;
```

If the user's browser doesn't support `document.getElementsByTagName`, then we return from the `scrollInit` function and don't progress any further.

One extra trick in the feature sniffing code, as described in Chapter 3, addresses the way in which we find the event object when we're inside the `moveListener` event listener. As we know, the DOM Events specification mandates that an event object is passed to the event listener as an argument, whereas Internet Explorer makes the event object available as the global `window.event`. So, our code checks for the existence of `window.event`, and uses it as the event object if it exists; the code falls back to the passed-in argument if `window.event` is not present:

File: **`scrollImage.js`** **(excerpt)**

```
function moveListener(ev) {
  var e = window.event ? window.event : ev;
```

Next, we need to get the event's target from that event object; the DOM specifies `e.target`, and Internet Explorer provides `e.srcElement`. Another feature-sniff gives us the appropriate value:

File: **`scrollImage.js`** **(excerpt)**

```
  var t = e.target ? e.target : e.srcElement;
```

This is a compressed, shorthand version of the code we saw in Chapter 3.

The next step is for the code to get the position of the mouse inside the thumbnail image area. This is the code from the full listing above that is supposed to do this:

```
  var xPos = e.clientX - findPosX(t);
  var yPos = e.clientY - findPosY(t);
```

In theory, `e.clientX` and `e.clientY` give the x- and y-coordinates of the mouse within the browser window, respectively. By subtracting from these the x- and y-coordinates of the target element, we obtain the mouse's position within that element.

Depending on your browser of choice, this might seem to work just fine at first glance. Peter-Paul Koch's `findPosX` and `findPosY` functions make short work of getting the target element's position.[4] Unfortunately, the `clientX` and `clientY` properties of the event object are nowhere near as reliable.

`clientX` and `clientY` Problems

The code above is flawed: the event listener uses `e.clientX` and `e.clientY` to ascertain the position of the mouse.

But that's not a flaw, is it? After all, it's in the DOM specifications!

[4]For a complete description of how `findPosX` and `findPosY` work, visit Peter-Paul Koch's page on the subject at http://www.quirksmode.org/js/findpos.html.

Well, it's *sort of* a flaw—a flaw in the way browser manufacturers interpret the specification. Peter-Paul Koch studies this problem in great detail in his comprehensive article, *Mission Impossible—Mouse Position*[2]. The problem occurs only when the page is scrolled (which was not the case with the above page). When a page is scrolled, the specification is rather vague on whether `clientX` and `clientY` are returned relative to the whole document, or to the window (the part of the document that is visible). Internet Explorer returns them relative to the window, as does Mozilla, but all of Opera, Konqueror, and iCab return them relative to the document. Netscape also provides `pageX` and `pageY`, which are mouse coordinates relative to the document. (Ironically enough, Internet Explorer may be the only browser which is fully compliant with the standard; the best reading of the specification is that `clientX` and `clientY` should be relative to the window.)

So, we need to use `pageX` and `pageY` if they exist, and `clientX` and `clientY` if they do not; if we're in Internet Explorer, however, we have to add to `clientX` and `clientY` the amounts by which the page has been scrolled. But how do we know if we're in Internet Explorer? We use browser detection.

Browser Detection You Can't Avoid

That spluttering noise you can hear in the background is the crowd rightly pointing out that we consigned browser detection to the dustbin of history only a few pages back, and they're not wrong. However, there are occasions when different browsers implement the same properties (in this case, `clientX` and `clientY`) in different ways *and* when there are no other objects available for sniffing that can us tell which of the different implementations is in use.

On such occasions, there is no alternative but to use the dreaded browser sniffing to work out what to do. The mouse position issue described here is almost the only such situation. The very thought that it might be necessary to use browser detection should make all right-thinking DHTML developers shudder with guilt, but, sadly, there's nothing for it! We add the browser detection script to the code just before we call `addEvent` to set up our window load listener:

File: **`scrollImage.js`** **(excerpt)**

```
var isIE = !window.opera && navigator.userAgent.indexOf('MSIE') !=
    -1;
```

Note that, first, we check that `window.opera` is `false` or non-existent; Opera sets this variable to make it easy for scripts to detect that it is the browser in use

[2] http://evolt.org/article/Mission_Impossible_mouse_position/17/23335/

(Opera also implements user-agent switching, so that, from a `navigator.userAgent` perspective, it can appear to be Internet Explorer). Once we've established that we're *not* using Opera, we go on to look for "MSIE" in the user agent string; if this is present, Internet Explorer is the browser in use.

Our updated `moveListener` event listener now looks like this:

File: **`scrollImage.js`** **(excerpt)**

```
function moveListener(ev) {
  var e = window.event ? window.event : ev;
  var t = e.target ? e.target : e.srcElement;

  var mX, mY;
  if (e.pageX && e.pageY) {
    mX = e.pageX;
    my = e.pageY;
  } else if (e.clientX && e.clientY) {
    mX = e.clientX;
    mY = e.clientY;
    if (isIE) {
      mX += document.body.scrollLeft;
      mY += document.body.scrollTop;
    }
  }

  var xPos = mX - findPosX(t);
  var yPos = mY - findPosY(t);

// ... the rest as before ...
```

Note that we check first for `pageX` and `pageY` (for Mozilla), then fall through to `clientX` and `clientY`. We handle Internet Explorer by checking the `isIE` variable; if it's `true`, we add the document's scroll amounts as required. We're using the browser detect as little as possible; specifically, Netscape/Mozilla provide the `pageX` and `pageY` properties, and we look for them through feature sniffing, *not* by performing browser detection for Mozilla.

Calculating Screen Positions

The last section of our code has little to do with browser detects, but, having spent all this time to get the right X and Y coordinates, it makes sense to understand how to use them.

The last part of the `moveListener` function starts with a couple of `if`s, which ensure that we have in hand a reference to the `<a>` tag surrounding the thumbnail `<img>` of interest. No surprises there, so we grab the required DOM element:

File: **`scrollImage.js`** **(excerpt)**

```
  if (t.nodeName.toLowerCase() == 'img')
    t = t.parentNode;
  if (t.nodeName.toLowerCase() == 'a') {
```

Next, we have the first of two sets of calculations:

File: **`scrollImage.js`** **(excerpt)**

```
    // scaleFactorY = (width(big) - width(small)) / width(small)
    var scaleFactorY =
        (attVal(t, 'mainy') - attVal(t, 'thumby')) / attVal(t,
        'thumby');
    var scaleFactorX =
        (attVal(t, 'mainx') - attVal(t, 'thumbx')) / attVal(t,
        'thumbx');
```

Code like this is liable to be specific to each DHTML effect you undertake, but the mind-bending you have to do to come up with the code is similar in all cases. Take a deep breath: here we go!

With the large background image showing through the viewing area, what should appear when the cursor is in the top-left corner of that viewing area? The top-left corner of the big image should be in the top-left corner of the viewing area: that's straightforward. Now, what should appear when the cursor is located at the bottom-right corner of the viewing area? Should the bottom-right corner of the full-sized image be in the top-left corner of the viewing area? That's what would happen if the big image were moved by its full size across the viewing area as the cursor was moved the full distance across the viewing area. Think about it carefully; you might like to try experimenting with two pieces of paper, one of which has a rectangular hole in it. The big image would eventually disappear off the top-left corner of the viewing area! If the background image were tiled (the default), additional copies of the image would be visible at this bottom-right corner—a very odd result.

We don't want the image to move that far. If we move the cursor to the extreme bottom-right of the viewing area, we want the big image to move by almost its entire size—but not quite! We want the bottom-right corner of the big image to move only as far as the bottom-right corner of the viewing area, and not move any further towards the top-left.

Now, to make the big image move, we have to calculate a distance by which to move it. Take some example figures: suppose the big image is ten times the size of the thumbnail. Let's suppose the image is 500 pixels on each side, and the thumbnail's 50 pixels on each side. For every pixel by which the cursor moves, the big image should move 500/50: ten times as fast. So the "scale factor" is ten. But, wait a minute! If the cursor moves 50 pixels left, the big image will move 500 pixels left: right off the left edge of the viewing area. That's too far. We want it to move at most 500 *minus* 50 pixels, so that it's always "inside" the viewing area. Therefore, the real scale factor is (500 – 50) / 50 = 9. The full-sized image should move nine times as fast as the cursor. That's what the first set of calculations does, except that it calculates scale factors in both dimensions, since most images are rectangles, not squares.

Next, we want to move the big image. Here's the second set of calculations:

File: **scrollImage.js (excerpt)**

```
    t.style.backgroundPosition =
        (-parseInt(xPos * scaleFactorX)) + 'px ' +
        (-parseInt(yPos * scaleFactorY)) + 'px';
```

Now, if (for example) we move the mouse from the top-left towards the bottom-right, we're scanning diagonally across the viewing area. As we move, we want new areas of the big image to come into view. So the big image had better slide in the opposite direction to the mouse: up towards, and beyond, the top left. It's like using a negative margin to bleed text to the left and top of a page. And that's what we do by calculating negative pixel amounts.

This idea may seem back-to-front initially. Think of it as though you were shooting a scene for a movie. The camera (the thumbnail viewing area) is fixed into place, so it must be the scene at which the camera points that moves if there's to be any panning effect. Alternately, imagine yourself looking out of the window of a moving train without turning your head. It's the same effect again, provided the train goes backwards!

Summary

In this chapter, we've learned that browsers don't always support all the DOM features we'd like, and discussed how feature sniffing helps us as DHTML developers to code defensively around this issue. Browser sniffing allows us to deliver dynamic features to browsers that can handle them and, at the same time, to avoid crashing or throwing errors in browsers that can't. We looked at the old method, browser sniffing, and explained why it shouldn't be used if at all possible.

We then explored one occasion on which feature sniffing can't provide everything we need, leaving us the old method as a last resort.

What’s Next?

If you’ve enjoyed these chapters from *DHTML Utopia: Modern Web Design Using JavaScript & DOM*, why not order yourself a copy?

You’ll learn how to enhance your sites’ interactivity and usability in browsers that can handle it, without breaking the functionality on those that can’t. You’ll discover the modern, standards-compliant DHTML techniques behind some of today’s most impressive web applications, like Flickr, Google Suggest, Google Maps, and GMail. You’ll also gain access to the code archive download, so you can try out all the examples without retyping!

In the remaining chapters, you’ll:

- Discover how to validate form data using regular expressions, and give dynamic feedback to your users
- Build dropdown lists that improve the usability of dropdown menus, just like Google Suggest
- Develop a fully standards-compliant, cross-browser, hierarchical navigation menu
- Get the lowdown on Remote Scripting techniques and AJAX
- Create form fields with name resolution that will automatically resolve an email address from a nickname
- Develop a super-cool drag and drop file manager application
- And much more!

On top of that, order direct from sitepoint.com and you’ll receive a free and indispensable 17” x 24” *DOM & JavaScript Quick Reference Guide* poster.

Order now and get it delivered to your doorstep!

Index