CHAPTER
5

Dynamically Modifying Web Pages

JavaScript gives you the power to change a Web page before your very eyes. Using JavaScript, you can add pictures and text, remove content, or change the appearance of an element on a page instantly. In fact, dynamically changing a Web page is the hallmark of the newest breed of JavaScript-powered Web sites. For example, Google Maps (*http://maps.google.com/*) provides access to a map of the world; when you zoom into the map or scroll across it, the page gets updated without the need to load a new Web page. Similarly, when you mouse over a movie title at Netflix (*www.netflix.com*) an information bubble appears on top of the page providing more detail about the movie (see Figure 5-1). In both of these examples, JavaScript is changing the HTML that the Web browser originally downloaded.

The first four chapters of this book covered many of the fundamentals of the JavaScript programming language—the keywords, concepts, and syntax of JavaScript. Now that you have a handle on how to write a basic JavaScript program and add it to a Web page, it's time to see what JavaScript programming is all about. This chapter, and the next one on JavaScript events, together show you how to create the great interactive effects you see on the Web these days.

Modifying Web Pages: An Overview

In this chapter, you'll learn how to alter a Web page using JavaScript. You'll add new content, HTML tags and HTML attributes, and also alter content and tags that are already on the page. In other words, you'll use JavaScript to generate new HTML and change the HTML that's already on the page.

Figure 5-1: *JavaScript can make Web pages simpler to scan and read, by only showing content when it's needed. At Netflix.com, movie descriptions are hidden from view, but revealed when the mouse travels over the movie title or thumbnail image.*

Whenever you change the content or HTML of a page—whether you're adding a navigation bar complete with pop-up menus, creating a JavaScript-driven slide show, or simply adding alternating stripes to table rows (like you did in the tutorial in Chapter 1)—you'll perform two main steps.

1. **Identify an element on a page.**

   An element is any existing tag, and before you have to do anything with that element, you need to *identify* it in your JavaScript (which you'll learn how to do in this chapter). For example, to add a color to a table row, you first must identify the row you wish to color; to make a pop-up menu appear when you mouse over a button, you need to identify that button. Even if you simply want to use JavaScript to add text to the bottom of a Web page, you need to identify a tag to insert the text either inside, before, or after that tag.

2. **Do something with the element.**

   OK, "do something" isn't a very specific instruction. That's because there's nearly an endless number of things you can *do* with an element to alter the way your Web page looks or acts. In fact, most of this book is devoted to teaching you different things to do to page elements. Here are a few examples:

   - **Add/remove a class attribute.** In the example on page 30, you used JavaScript to assign a class to every other row of a table. The JavaScript didn't actually "color" the row; it merely applied a class, and the Web browser used the information in the CSS style sheet to change the appearance of the row.
   - **Change a property of the element.** When animating a across a page, for example, you change that element's position on the page.

- **Add new content.** If, while filling out a Web form, a visitor incorrectly fills out a field, it's common to make an error message appear—"Please supply an email address," for example. In this case, you're adding content somewhere in relation to that form field.
- **Remove the element.** In the Netflix example pictured in Figure 5-1, the pop-up bubble disappears when you mouse off the movie title. In this case, JavaScript removes that pop-up bubble from the page.
- **Extract information from the element.** Other times, you'll want to know something about the tag you've identified. For example, to validate a text field, you need to identify that text field, then find out what text was typed into that field—in other words, you need to get the value of that field.

The first step above—identifying an element on a page—is mainly what this chapter is about. To understand how to identify and modify a part of a page using JavaScript you first need to get to know the *Document Object Model*.

Understanding the Document Object Model

When a Web browser loads an HTML file, it displays the contents of that file on the screen (appropriately styled with CSS, of course). But that's not all the Web browser does with the tags, attributes, and contents of the file: it also creates and memorizes a "model" of that page's HTML. In other words, the Web browser remembers the HTML tags, their attributes, and the order in which they appear in the file—this representation of the page is called the *Document Object Model*, or DOM for short.

The DOM provides the information needed for JavaScript to communicate with the elements on the Web page. The DOM also provides the tools necessary to navigate through, change, and add to the HTML on the page. The DOM itself isn't actually JavaScript—it's a standard from the World Wide Web Consortium (W3C) that most browser manufacturers have adopted and added to their browsers. The DOM lets JavaScript communicate with and change a page's HTML.

To see how the DOM works, take look at this very simple Web page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<title>A web page</title>
</head>
<body class="home">
<h1 id="header">A headline</h1>
<p>Some <strong>important</strong> text</p>
</body>
</html>
```

On this and all other Web sites, some tags wrap around other tags—like the <html> tag, which surrounds all other tags, or the <body> tag, which wraps around the tags and contents that appear in the browser window. You can represent the relationship between tags with a kind of family tree (see Figure 5-2). The <html> tag is the "root" of the tree—like the great-great-great granddaddy of all of the other tags on the page—while other tags represent different "branches" of the family tree; for example, the <head> and <body> tags, which each contain their own set of tags.

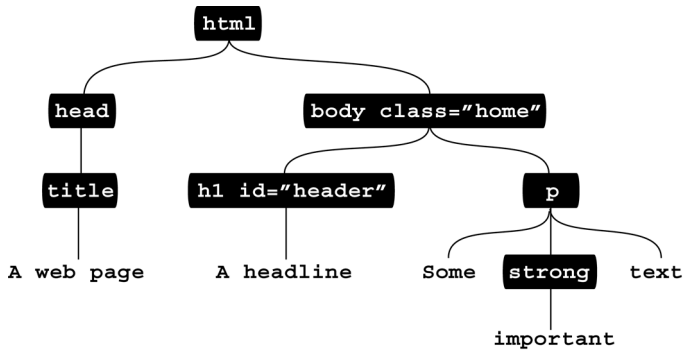

Figure 5-2:
The basic nested structure of an HTML page, where tags wrap around other tags, is often represented in the form of a family tree, where tags that wrap around other tags are called ancestors, and tags inside other tags are called descendents.

In addition to HTML tags, Web browsers also keep track of the text that appears inside a tag (for example, "A headline" inside the <h1> tag in Figure 5-2), as well as the *attributes* that are assigned to each tag (the class attribute applied to the <body> and <h1> tags in Figure 5-2). In fact, the DOM treats each of these—tags (also called *elements*), attributes, and text—as individual units called *nodes*.

Selecting a Page Element

A Web browser thinks of a Web page simply as an organized collection of tags, tag attributes, and text, or, in DOM-talk, a bunch of *nodes*. So for JavaScript to manipulate the contents of a page, it needs a way to communicate with a page's nodes. There are two main methods for selecting nodes: *getElementById( )* and *getElementsByTagName( )*.

getElementById()

Getting an element by ID means locating a single node that has a unique ID applied to it. For example, in Figure 5-2, the <h1> tag has an ID attribute with the value of *header*. The following JavaScript selects that node:

```
document.getElementById('header')
```

In plain English, this line means, "Search this page for a tag with an ID of 'header' assigned to it." The *document* part of *document.getElementById('header')* is a keyword that refers to the entire document. It's not optional, so you can't type

getElementById( ) by itself. The command *getElementById( )* is the method name (a command for the document) and the *'header'* part is simply a string (the name of the ID you're looking for) that's sent to the method as an argument. (See page 101 for the definition of an argument.)

Note: The *getElementById( )* method requires a single string–the name of a tag's ID attribute. For example:

```
document.getElementById('header')
```

However, this doesn't mean you have to provide a literal string to the method: you can also pass a variable that contains a string with the sought after ID:

```
var lookFor = 'header';
var foundNode = document.getElementById(lookFor);
```

Frequently, you'll assign the results of this method to a variable to store a reference to the particular tag, so you can later manipulate it in your program. For example, say you want to use JavaScript to change the text of the headline in the HTML pictured on page 157. You can do this:

```
var headLine = document.getElementById('header');
headLine.innerHTML = 'JavaScript was here!';
```

The *getElementById( )* command returns a reference to a single node, which in this example is stored in a variable named *headline*. Storing the results of *getElementById( )* in a variable is very convenient; it lets you refer simply to the variable name each time you wish to manipulate that tag, rather than the much more longwinded *document.getElementById('idName')*. For example, the second line of code uses the variable to access the tag's *innerHTML* property: *headline.innerHTML* (you'll learn what *innerHTML* is on page 163).

getElementsByTagName( )

Sometimes, you'll want more than just the single element that *getElementById( )* provides. For example, maybe you'd like to find every link on a Web page and do something to those links—like force every link that points outside your site to open in a new window. In that case, you need to get a list of elements, not just one element marked with an ID. The command *getElementsByTagName( )* does the trick.

This method works similarly to *getElementById( )* but instead of providing the name of an ID, you supply the name of the tag you're looking for. For example, to find all of the links on a page, you write this:

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

Translated, this means, "Search this document for every <a> tag and store the results in a variable named *pageLinks*." The *getElementsByTagName( )* method returns a list of nodes, instead of just a single node. In that sense, the list acts a lot

like an array: You can access a single node using the same index notation, find the total number of elements using the *length* property, and loop through the list of elements using a *for* loop (see page 94).

For example, the first item in the *pageLinks* variable from the code above is *pageLinks[0]*—the first <a> tag on the page—and *pageLinks.length* is the total number of <a> tags on the page.

Tip: It's easy to make a typo with these two methods. Most commonly, beginners (and pros) will capitalize both letters of *Id*. Only the first letter is capitalized. Likewise, *Elements* is plural in *getElementsByTagName()*–don't forget the *s*:

```
document.getElementById('banner');
document.getElementsByTagName('a');
```

You can also use *getElementById()* and *getElementsByTagName()* together. For example, say you have a Web page containing a tag, and that tag has an ID of 'banner' applied to it. If you want to find out how many links were in just that , you can use *getElementById()* to retrieve the , and then use *getElementsByTagName()* to search the . Here's how it works:

```
var banner = document.getElementById('banner');
var bannerLinks = banner.getElementsByTagName('a');
var totalBannerLinks = bannerLinks.length;
```

While searching for an element with an ID is one method of searching within the document (*document.getElementById()*), you can find tags of a particular type by searching the entire document (*document.getElementsByTagName()*) or by searching the tags within a particular node. For example, in the above code, the variable *banner* contains a reference to a tag, so the code *banner.getElementsByTagName('a')* only searches for <a> tags *inside* that .

Selecting nearby nodes

As mentioned earlier, text is also considered a node, so the text "A headline" inside the <h1> tag on page 157 is a separate node from the <h1> tag that surrounds it. In other words, if you select that <h1> tag using the techniques you've just learned, you've just selected that tag and not the text inside. So, what if you want to get at that text? Unfortunately, the way the DOM provides to do so involves a rather roundabout technique: You have to start at the <h1> node, move to the text node, and then get the value of the text node.

To understand how this process works, you need to understand how tags are related to each other. If you've spent some time working with Cascading Style Sheets, you're probably familiar with *descendent* selectors—one of the most powerful tools in CSS. In a nutshell, a descendent selector lets you format a particular tag based on its relationship to another tag. Thus, using a descendent selector, you can make a paragraph (<p>) tag look one way when it's in the sidebar of a page, and look another way when that same tag is in the footer of the page.

Descendent selectors rely on the kind of relationship pictured in Figure 5-2; if a tag is inside another tag, it's called a descendent. The <h1> tag in the sample HTML on page 157 is a descendent of the <body> tag, and, because it's also inside the <html> tag, it's a descendent of that tag as well. Tags that wrap around other tags are called *ancestors*; so in Figure 5-2, the <p> tag is an ancestor of the tag.

The DOM also thinks of tags that wrap around other tags as being related, but the DOM only provides access to the "immediate family." That is, the DOM can access a "parent" node, "child" node, or "sibling" node. Figure 5-3 demonstrates these relationships: If a node is directly inside another node, like the text "Some" inside the <p> tag, then it's a *child*; a node that directly surrounds another node, like the tag surrounding the text "important", is a *parent*. Nodes that share the same parent, like the two text nodes—"Some" and "text"—and the tag are like brothers and sisters, so they're called *siblings*.

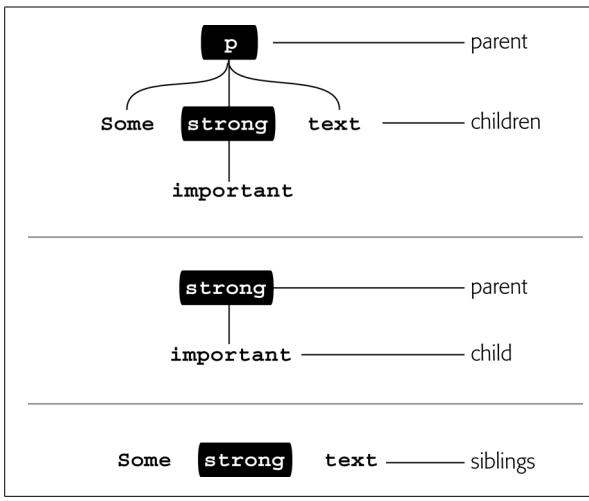

Figure 5-3:
There are no first cousins, great aunts, or even grandparents in the current DOM standard. The DOM only recognizes parents, children, and sibling relationships between tags.

The DOM provides several methods of accessing nearby nodes:

- *.childNodes* is a property of a node. It contains a list of all nodes that are direct children of that node. The list of nodes works just like the list that's returned by the *getElementsByTagName( )* method (see page 159). For example, suppose you add the following JavaScript to the HTML file on page 157:

```
var headline = document.getElementById('header');
var headlineKids = headline.childNodes;
```

  The variable *headlineKids* will contain a list of all nodes that are children of the tag that has the ID of 'headline' (the <h1> tag in this example). In this case, there's only one child, the text node containing the text "A headline." So, if you want to know what the text inside that node is, add an additional line of code, like this:

```
var headlineText = headlineKids[0].nodeValue;
```

The first child in the list is *headlineKids[0]*—since there is only one child for the headline (see Figure 5-2), it's also the only node in the list. To get the text inside a text node, you access the *nodeValue* property. (On the other hand, there's also an easier way to do so, as you'll see on page 181.)

- *.parentNode* is a node property that represents the direct parent of a particular node. For example, if you wanted to know what tag wraps around the <h1> tag in Figure 5-2, you could write this:

```
var headline = document.getElementById('header');
var headlineParent = headline.parentNode;
```

The variable *headlineParent* is a reference to the <body> tag in this case.

- *.nextSibling* and *.previousSibling* are properties that point to the node that comes directly after the current node, or the node that comes before. For example, in Figure 5-2, the <h1> and <p> tags are siblings: the <p> tag comes directly after the ending </h1> tag.

```
var headline = document.getElementById('header');
var headlineSibling = headline.nextSibling;
```

The variable *headlineSibling* is a reference to the <p> tag that follows the <h1> tag. If you try to access a sibling node that doesn't exist, JavaScript returns the value of *null* (see the Tip on page 131). For example, you can check to see if a node has a *previousSibling* like this:

```
var headline = document.getElementById('header');
var headlineSibling = headline.prevSibling;
if (! headlineSibling) {
  alert('This node does not have a previous sibling!');
} else {
  // do something with the sibling node
}
```

As you can see, it takes a fair amount of gymnastics to move around a page's DOM structure. For instance, to get all of the text inside the <p> tag in Figure 5-2, you'd have to get a list of all of the <p> tags children, and then go through each child node and look for text. In the case of the tag pictured in Figure 5-2, you'd have to look at its child nodes to get the text inside it! Fortunately, there's a much easier way to work with the DOM, as you'll see on page 169.

Adding Content to a Page

JavaScript programs frequently need to add, delete, or change content on a page. For example, in the quiz program you wrote in Chapter 3 (page 106), you used the *document.write( )* method to add the test-taker's final score to the page. On the Netflix site (Figure 5-1), a description appears on the page when a visitor mouses over a movie title.

Note: In earlier chapters you used the *document.write( )* command to add JavaScript-generated content to a page (see page 29 for an example). That command is easy to learn and use, but very limited in what it can do–for example, *document.write( )* lets you add new content, but not alter what's already on the page. Furthermore, that command works when the page loads, so you can't use it to add content to a page later (for example, when a visitor clicks a button or types into a form field).

Adding content using the DOM is a big chore. It involves creating each node of the content you require, and then injecting the results into the page. In other words, if you want to add a tag with a couple of other tags and some text, you have to create each node individually and place them in the proper relation to each other. Fortunately, browser manufacturers have provided a much simpler method: the *innerHTML* property.

The *innerHTML* property isn't a standard part of the DOM. It was first implemented in Internet Explorer, but all current, JavaScript-savvy Web browsers support it. Basically, *innerHTML* represents all of the HTML inside of a node. For example, if you look at the HTML code on page 157, the <p> tag wraps around other HTML. So the *innerHTML* for that <p> tag node is *Some important text*. Here's how you use JavaScript to access that HTML:

```
//get a list of all <p> tags on page
var pTags = document.getElementsByTagName('p');
//get the first <p> tag on page
var theP = pTags[0];
alert(theP.innerHTML);
```

In this case, the variable *theP* represents the node for the first paragraph on the page. The last line of code opens an alert box that displays all of the code inside that tag. For example, adding this JavaScript to the HTML on page 157 would make an alert box appear with the text "Some important text".

Note: *innerHTML* is a proposed part of the new HTML 5 standard that is being developed at the W3C (see *www.w3.org/TR/html5*).

Not only can you find out what's inside a node using *innerHTML*, you can also change the contents inside the node by setting the *innerHTML* property:

```
var headLine = document.getElementById('header');
headLine.innerHTML = 'JavaScript was here!';
```

In this example, the contents inside the tag with an ID of 'header' is changed to "JavaScript was here!" You aren't limited to just text either: you can set the *innerHTML* property to complete chunks of HTML, including tags and tag attributes. You'll see an example of this in the next section.