



# DOM Scripting

## WHAT YOU WILL LEARN IN THIS CHAPTER:

---

- Finding elements in the page
- Creating and inserting elements into the page dynamically
- Navigating the web page, travelling from one element to another
- Changing elements' style after they are loaded in the page
- Animating elements by manipulating their positioning

## WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at <http://www.wiley.com/go/BeginningJavaScript5E> on the Download Code tab. You can also view all of the examples and related files at <http://beginningjs.com>.

JavaScript's primary role in web development is to interact with the user, to add some kind of behavior to your web page. JavaScript enables you to completely change all aspects of a web page after it's loaded in the browser. What gives JavaScript this power over a web page is the *document object model* (DOM), a tree-like representation of the web page.

The DOM is one of the most misunderstood standards set forth by the World Wide Web Consortium (W3C), a body of developers who recommend standards for browser makers and web developers to follow. The DOM gives developers a way of representing everything on a web page so that it is accessible via a common set of properties and methods in JavaScript. By everything, we mean *everything*. You can literally change anything on the page: the graphics, tables, forms, style, and even text itself by altering a relevant DOM property with JavaScript.

The DOM should not be confused with the browser object model (BOM) that was introduced in Chapter 8. You'll see the differences between the two in detail shortly. For now, though, think of the BOM as a browser-dependent representation of every feature of the browser, from the browser buttons, URL address line, and title bar to the browser window controls, as well as parts of the web page, too. The DOM, however, deals only with the contents of the browser window or web page (in other words, the HTML document). It makes the document available in such a way that any browser can use exactly the same code to access and manipulate the content of the document. To summarize, the BOM gives you access to the browser and some of the document, whereas the DOM gives you access to all of the document, but *only* the document.

The great thing about the DOM is that it is browser- and platform-independent. This means that developers can write JavaScript code that dynamically updates the page, and that will work on any DOM-compliant browser without any tweaking. You should not need to code for different browsers or take excessive care when coding.

The DOM achieves this independence by representing the contents of the page as a generic tree structure. Whereas in the BOM you might expect to access something by looking up a property relevant to that part of the browser and adjusting it, the DOM requires navigation through its representation of the page through nodes and properties that are not specific to the browser. You explore this structure a little later.

However, to use the DOM standard, ultimately developers require browsers that completely implement the standard, something that no browser does 100 percent efficiently, unfortunately. To make matters worse, no one browser implements the exact same DOM features that other browsers support, but don't be scared off yet. All modern browsers support many of the same features outlined by the DOM standard.

To provide a true perspective on how the DOM fits in, we need to take a brief look at its relationship with some of the other currently existing web standards. We should also talk about why there is more than one version of the DOM standard, as well as different sections within the standard itself. After understanding the relationships, you can look at using JavaScript to navigate the DOM and to dynamically change a web page's content in more than one browser. The following items are on your agenda:

- The HTML and ECMAScript standards
- The DOM standards
- Manipulating the DOM
- Writing cross-browser JavaScript

**NOTE** Remember that the examples within this chapter are targeted only at the DOM (with very few exceptions) and will be supported by modern browsers (IE 9+, Chrome, Firefox, Opera, and Safari). Legacy browsers (IE8 and below, earlier versions of Chrome, and similar early browsers) may or may not support them.

## THE WEB STANDARDS

When Tim Berners-Lee created HTML in 1991, he probably had little idea that this technology for marking up scientific papers via a set of tags for his own global hypertext project, known as the World Wide Web, would, within a matter of years, become a battleground between the two giants of the software business of the mid-1990s. HTML was a simple derivation from the meta-language Standard Generalized Markup Language (SGML) that had been kicking around academic institutions for decades. Its purpose was to preserve the structure of the documents created with it. HTML depends on a protocol, HyperText Transfer Protocol (HTTP), to transmit documents back and forth between the resource and the viewer (for example, the server and the client computer). These two technologies formed the foundation of the web, and it quickly became obvious in the early 1990s that there needed to be some sort of policing of both specifications to ensure a common implementation of HTML and HTTP so that communications could be conducted worldwide.

In 1994, Tim founded the World Wide Web Consortium (W3C), a body that set out to oversee the technical evolution of the web. It has three main aims:

- To provide universal access, so that anybody can use the web
- To develop a software environment to allow users to make use of the web
- To guide the development of the web, taking into consideration the legal, social, and commercial issues that arise

Each new version of a specification of a web technology has to be carefully vetted by W3C before it can become a standard. The HTML and HTTP specifications are subject to this process, and each new set of updates to these specifications yields a new version of the standard. Each standard has to go through a working draft, a candidate recommendation, and a proposed recommendation stage before it can be considered a fully operational standard. At each stage of the process, members of the W3C consortium vote on which amendments to make, or even on whether to cancel the standard completely and send it back to square one.

It sounds like a very painful and laborious method of creating a standard format, and not something you'd think of as spearheading the cutting edge of technical revolution. Indeed, the software companies of the mid-1990s found the processes involved too slow, so they set the tone by implementing new innovations themselves and then submitting them to the standards body for approval. Netscape started by introducing new elements in its browser, such as the `<font />` element, to add presentational content to the web pages. This proved popular, so Netscape added a whole raft of elements that enabled users to alter aspects of presentation and style on web pages. Indeed, JavaScript itself was such an innovation from Netscape.

When Microsoft entered the fray, it was playing catch up for the first two iterations of its Internet Explorer browser. However, with Internet Explorer 3 in 1996, Microsoft established a roughly equal set of features to compete with Netscape and so was able to add its own browser-specific elements. Very quickly, the web polarized between these two browsers, and pages viewable on one browser quite often wouldn't work on another. One problem was that Microsoft had used its much stronger position in the market to give away its browser for free, whereas Netscape still needed to sell its own browser because it couldn't afford to freely distribute its flagship product. To maintain

a competitive position, Netscape needed to offer new features to make the user want to purchase its browser rather than use the free Microsoft browser.

Things came to a head with both companies' version 4 browsers, which introduced dynamic page functionality. Unfortunately, Netscape did this by the means of a `<layer />` element, whereas Microsoft chose to implement it via scripting language properties and methods. The W3C needed to take a firm stand here, because one of its three principal aims had been compromised: that of universal access. How could access be universal if users needed a specific vendor's browser to view a particular set of pages? They decided on a solution that used existing standard HTML elements and cascading style sheets, both of which had been adopted as part of the Microsoft solution. As a result, Microsoft gained a dominant position in the browser war, and it held that position for many years. Today, Microsoft's Internet Explorer is still the dominant browser, but it has lost a lot of its market share to Chrome and Firefox.

With a relatively stable version of the HTML standard in place with version 4.01, which boasts a set of features that will take any browser manufacturer a long time to implement completely, attention was turned to other areas of the web. A new set of standards was introduced in the late 1990s to govern the means of presenting HTML (style sheets) and the representation of the HTML document in script (the DOM). Other standards emerged, such as Extensible Markup Language (XML), which offers a common format for representing data in a way that preserves its structure.

The W3C website ([www.w3.org](http://www.w3.org)) has a huge number of standards in varying stages of creation. Not all of these standards concern us, and not all of the ones that concern us can be found at this website. However, the vast majority of standards that do concern us can be found there.

You're going to take a brief look now at the technologies and standards that have an impact on JavaScript and find out a little background information about each. Some of the technologies may be unfamiliar, but you need to be aware of their existence at the very least.

## HTML

The HTML standard is maintained by W3C. This standard might seem fairly straightforward, given that each version should have introduced just a few new elements, but in reality the life of the standards body was vastly complicated by the browser wars. The 1.0 and 2.0 versions of HTML were simple, small documents, but when the W3C came to debate HTML version 3.0, they found that much of the new functionality it was discussing had already been superseded by new additions (such as the `<applet />` and `<style />` elements) to the version 3.0 browser's `appletstyle`. Version 3.0 was discarded, and a new version, 3.2, became the standard.

However, a lot of the features that went into HTML 3.2 had been introduced at the behest of the browser manufacturers and ran contrary to the spirit of HTML, which was intended solely to define structure. The new features, stemming from the `<font />` element, just confused the issue and added unnecessary presentational features to HTML. These features really became redundant with the introduction of style sheets. So suddenly, in the version 3 browsers, there were three distinct ways to define the style of an item of text. Which was the correct way? And if all three ways were used, which style did the text ultimately assume? Version 4.0 of the HTML standard was left with the job of unmuddling this chaotic mess and designated a lot of elements for deprecation (removal) in the next version of the standards. It was the largest version of the standard so far and included

features that linked it to style sheets and the DOM, and also added facilities for the visually impaired and other unfairly neglected minority interest areas.

In 2004, the W3C was focusing on XHTML 2.0, a specification that many, and perhaps most, in the web development community thought to be the wrong direction for the web. So another body, the Web Hypertext Application Technology Working Group (WHATWG) started working on HTML5. In 2009, the W3C officially dropped XHTML 2.0, and today the W3C and the WHATWG work together on developing HTML5.

HTML5 introduces many new features. First are new elements that identify a page's navigation, header, and footer with the `<nav />`, `<header />`, and `<footer />` elements. It also adds `<audio />` and `<video />` elements to replace `<object />`. HTML5 also removes elements like `<font />` and `<center />`, elements that are purely used for presentation purposes. HTML5 also defines native support for drag and drop, geolocation, storage, and much more.

**NOTE** *The HTML5 specification is not completely finalized, but many of its individual features are said to be complete. As a result, you'll find many features implemented in today's modern browsers. You can visit the W3C web-site at <http://www.w3.org/TR/html5/> or the WHATWG living standard at <http://html.spec.whatwg.org/multipage/> if you want to read the actual specifications.*

## ECMAScript

JavaScript itself followed a trajectory similar to that of HTML. It was first used in Netscape Navigator and then added to Internet Explorer. The Internet Explorer version of JavaScript was christened JScript and wasn't far removed from the version of JavaScript found in Netscape Navigator. However, once again, there were differences between the two implementations and a lot of care had to be taken in writing script for both browsers.

Oddly enough, it was left to the European Computer Manufacturers Association (ECMA) to propose a standard specification for JavaScript. This didn't appear until a few versions of JavaScript had already been released. Unlike HTML, which had been developed from the start with the W3C consortium, JavaScript was a proprietary creation. This is the reason that it is governed by a different standards body. Microsoft and Netscape both agreed to use ECMA as the standards vehicle/debating forum, because of its reputation for fast-tracking standards and perhaps also because of its perceived neutrality. The name ECMAScript was chosen so as not to be biased toward either vendor's creation and also because the "Java" part of JavaScript was a trademark of Sun licensed to Netscape. The standard, named ECMA-262, laid down a specification that was roughly equivalent to the JavaScript 1.1 specification.

That said, the ECMAScript standard covers only core JavaScript features, such as the primitive data types of numbers, strings, and booleans, native objects like the `Date`, `Array`, and `Math` objects, and the procedural statements like `for` and `while` loops, and `if` and `else` conditionals. It makes no reference to client-side objects or collections, such as `window`, `document`, `forms`, `links`, and `images`.

So, although the standard helps to make core programming tasks compatible when both JavaScript and JScript comply with it, it is of no use in making the scripting of client-side objects compatible between the main browsers. Some incompatibilities remain.

All current implementations of JavaScript are expected to conform to the current ECMAScript standard, which is ECMAScript edition 5, published in December 2009.

Although there used to be quite a few irregularities between the different dialects of JavaScript, they're now similar enough to be considered the same language. This is a good example of how standards have provided a uniform language across browser implementations, although a debate that is similar to the one that took place over HTML still rages to a lesser degree over JavaScript.

It's now time for you to consider the document object model itself.

## THE DOCUMENT OBJECT MODEL

The document object model (DOM) is, as previously mentioned, a way of representing the document independent of browser type. It allows a developer to access the document via a common set of objects, properties, methods, and events, and to alter the contents of the web page dynamically using scripts.

You should be aware that some small variations are usually added to the DOM by the browser vendor. So, to guarantee that you don't fall afoul of a particular implementation, the W3C has provided a generic set of objects, properties, and methods that should be available in all browsers, in the form of the DOM standard.

## The DOM Standard

We haven't talked about the DOM standard so far, and for a particular reason: It's not the easiest standard to follow. Supporting a generic set of properties and methods has proved to be a very complex task, and the DOM standard has been broken down into separate levels and sections to deal with the different areas. The different levels of the standard are all at differing stages of completion.

### Level 0

Level 0 is a bit of a misnomer, because there wasn't really a level 0 of the standard. This term in fact refers to the "old way" of doing things—the methods implemented by the browser vendors before the DOM standard. Someone mentioning level 0 properties is referring to a more linear notation of accessing properties and methods. For example, typically you'd reference items on a form with the following code:

```
document.forms[0].elements[1].value = "button1";
```

We're not going to cover such properties and methods in this chapter, because they have been superseded by newer methods.

### Level 1

Level 1 is the first version of the standard. It is split into two sections: One is defined as core (objects, properties, and methods that can apply to both XML and HTML) and the other as HTML

(HTML-specific objects, properties, and methods). The first section deals with how to go about navigating and manipulating the structure of the document. The objects, properties, and methods in this section are very abstract. The second section deals with HTML only and offers a set of objects corresponding to all the HTML elements. This chapter mainly deals with the second section—level 1 of the standard.

In 2000, level 1 was revamped and corrected, though it only made it to a working draft and not to a full W3C recommendation.

## Level 2

Level 2 is complete and many of the properties, methods, and events have been implemented by today's browsers. It has sections that add specifications for events and style sheets to the specifications for core and HTML-specific properties and events. (It also provides sections on views and traversal ranges, neither of which is covered in this book; you can find more information at [www.w3.org/TR/2000/PR-DOM-Level-2-Views-20000927/](http://www.w3.org/TR/2000/PR-DOM-Level-2-Views-20000927/) and [www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927/](http://www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927/).)

## Level 3

Level 3 achieved recommendation status in 2004. It is intended to resolve a lot of the complications that still exist in the event model in level 2 of the standard, and adds support for XML features, such as content models and being able to save the DOM as an XML document.

## Level 4

In May 2014, DOM level 4 reached candidate recommendation status. It consolidates DOM level 3 with several independent components. At the time of this writing, no modern browser supports DOM level 4, although that will change in the future.

## Browser Compliance with the Standards

Almost no browser has 100 percent compliance with any standard. Therefore, there is no guarantee that all the objects, properties, and methods of the DOM standard will be available in a given version of a browser. However, all modern browsers do a very good job of supporting the standard DOM. The only browsers you truly have to watch out for are IE8 and below.

Much of the material in the DOM standard has only recently been clarified, and a lot of DOM features and support have been added to only the latest browser versions. For this reason, examples in this chapter will be guaranteed to work on only the latest versions of IE, Chrome, Firefox, Opera, and Safari. Although cross-browser scripting is a realistic goal, backward-compatible support isn't at all.

Although the standards might still not be fully implemented, they do give you an idea as to how a particular property or method should be implemented, and provide a guideline for all browser manufacturers to agree to work toward in later versions of their browsers. The DOM doesn't introduce any new HTML elements or style sheet properties to achieve its ends. The idea of the DOM is to make use of the existing technologies, and quite often the existing properties and methods of one or other of the browsers.



## Differences between the DOM and the BOM

As mentioned earlier, two main differences exist between the document object model and the browser object model. However, complicating the issue is the fact that a BOM is sometimes referred to under the name DOM. Look out for this in any literature on the subject.

- First, the DOM covers only the document of the web page, whereas the BOM offers scripting access to all areas of the browsers, from the buttons to the title bar, including some parts of the page.
- Second, the BOM is unique to a particular browser. This makes sense if you think about it: you can't expect to standardize browsers, because they have to offer competitive features. Therefore, you need a different set of properties and methods and even objects to be able to manipulate them with JavaScript.

## Representing the HTML Document as a Tree Structure

Because HTML is standardized so that web pages can contain only the standard features supported in the language, such as forms, tables, images, and the like, a common method of accessing these features is needed. This is where the DOM comes in. It provides a uniform representation of the HTML document, and it does this by representing the entire HTML document/web page as a *tree structure*.

In fact, it is possible to represent any HTML as a tree structure, and for best results, the HTML document should be well formed. Browsers tolerate, to a greater or lesser extent, quirks such as unclosed tags, or HTML form controls not being enclosed within a `<form/>` element; however, for the structure of the HTML document to be accurately depicted, you need to be able to always predict the structure of the document. The ability to access elements via the DOM depends on the ability to represent the page as a hierarchy.

### What Is a Tree Structure?

If you're not familiar with the concept of trees, don't worry. They're just a diagrammatic means of representing a hierarchical structure.

Let's consider the example of a book with several chapters. If instructed to, you could find the third line on page 543 after a little searching. If an updated edition of the book were printed with extra chapters, more likely than not you'd fail to find the same text if you followed those same instructions. However, if the instructions were changed to, say, "Find the chapter on still-life painting, the section on using watercolors, and the paragraph on positioning light sources," you'd be able to find that even in a reprinted edition with extra pages and chapters, albeit with perhaps a little more effort than the first request required.

Books aren't particularly dynamic examples, but given something like a web page, where the information could be changed daily, or even hourly, can you see why it would be of more use to give the second set of directions than the first? The same principle applies with the DOM. Navigating the DOM in a hierarchical fashion, rather than in a strictly linear way, makes much more sense. When you treat the DOM as a tree, it becomes easy to navigate the page in this fashion. Consider how you locate files on your computer. The file/folder manager (Windows Explorer in Windows, Finder in Mac OS, and so on) creates a tree view of folders through which you can drill down. Instead of looking for a file alphabetically, you locate it by going into a particular folder.



The rules for creating trees are simple. You start at the top of the tree with the document and the element that contains all other elements in the page. The document is the *root node*. A *node* is just a point on the tree representing a particular element or attribute of an element, or even the text that an element contains. The root node contains all other nodes, such as the DTD declaration and the root element (the HTML or XML element that contains all other elements). The root element should always be the `<html/>` element in an HTML document. Underneath the root element are the HTML elements that the root element contains. Typically, an HTML page will have `<head/>` and `<body/>` elements inside the `<html/>` element. These elements are represented as nodes underneath the root element's node, which itself is underneath the root node at the top of the tree (see Figure 9-1).

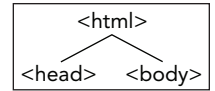


FIGURE 9-1

The two nodes representing the `<head/>` and `<body/>` elements are examples of *child nodes*, and the `<html/>` element's node above them is a *parent node*. Because the `<head/>` and `<body/>` elements are both child nodes of the `<html/>` element, they both go on the same level underneath the parent node `<html/>` element. The `<head/>` and `<body/>` elements in turn contain other child nodes/HTML elements, which will appear at a level underneath their nodes. So child nodes can also be parent nodes. Each time you encounter a set of HTML elements within another element, they each form a separate node at the same level on the tree. The easiest way of explaining this clearly is with an example.

## An Example HTML Page

Let's consider a basic HTML page such as this:

```

<!DOCTYPE html>

<html lang="en">
<head>
</head>
<body>
  <h1>My Heading</h1>
  <p>This is some text in a paragraph.</p>
</body>
</html>
  
```

The `<html/>` element contains `<head/>` and `<body/>` elements. The `<body/>` element contains an `<h1/>` element and a `<p/>` element. The `<h1/>` element contains the text `My Heading`. When you reach an item, such as text, an image, or an element, that contains no others, the tree structure will terminate at that node. Such a node is termed a *leaf node*. You then continue to the `<p/>` node, which contains some text, which is also a node in the document. You can depict this with the tree structure shown in Figure 9-2.

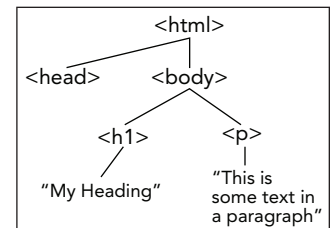


FIGURE 9-2

Simple, eh? This example is almost too straightforward, so let's move on to a slightly more complex one that involves a table as well:

```

<!DOCTYPE html>

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

```
<title>This is a test page</title>
</head>
<body>
  <span>Below is a table</span>
  <table>
    <tr>
      <td>Row 1 Cell 1</td>
      <td>Row 1 Cell 2</td>
    </tr>
  </table>
</body>
</html>
```

There is nothing out of the ordinary here; the document contains a table with two rows with two cells in each row. You can once again represent the hierarchical structure of your page (for example, the fact that the `<html/>` element contains a `<head/>` and a `<body/>` element, and that the `<head/>` element contains a `<title/>` element, and so on) using your tree structure, as shown in Figure 9-3.

The top level of the tree is simple enough; the `<html/>` element contains `<head/>` and `<body/>` elements. The `<head/>` element in turn contains a `<title/>` element and the `<title/>` element contains some text. This text node is a child node that terminates the branch (a leaf node). You can then go back to the next node, the `<body/>` element node, and go down that branch. Here you have two elements contained within the `<body/>` element: the `<span/>` and `<table/>` elements. Although the `<span/>` element contains only text and terminates there, the `<table/>` element contains one row (`<tr/>`), and it contains two table cell (`<td/>`) elements. Only then do you get to the bottom of the tree with the text contained in each table cell. Your tree is now a complete representation of your HTML code.

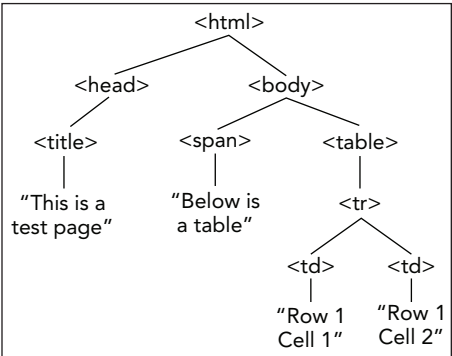


FIGURE 9-3

## The Core DOM Objects

What you have seen so far has been highly theoretical, so let’s get a little more practical now.

The DOM provides you with a concrete set of objects, properties, and methods that you can access through JavaScript to navigate the tree structure of the DOM. Let’s start with the set of objects, within the DOM, that is used to represent the nodes (elements, attributes, or text) on your tree.

### Base DOM Objects

Three objects, shown in the following table, are known as the base DOM objects.

OBJECT	DESCRIPTION
Node	Each node in the document has its own <code>Node</code> object.
NodeList	This is a list of <code>Node</code> objects.
NamedNodeMap	This provides access by name rather than by index to all the <code>Node</code> objects.

This is where the DOM differs from the BOM quite extensively. The BOM objects have names that relate to a specific part of the browser, such as the `window` object, or the `forms` and `images` collections. As mentioned earlier, to be able to navigate in the web page as though it were a tree, you have to do it abstractly. You can have no prior knowledge of the structure of the page; everything ultimately is just a node. To move around from HTML element to HTML element, or element to attribute, you have to go from node to node. This also means you can add, replace, or remove parts of your web page without affecting the structure as a whole, because you're just changing nodes. This is why you have three rather obscure-sounding objects that represent your tree structure.

I've already mentioned that the top of your tree structure is the root node, and that the root node contains the DTD and root element. Therefore, you need more than just these three objects to represent your document. In fact, there are different objects to represent the different types of nodes on the tree.

## High-Level DOM Objects

Because everything in the DOM is a node, it's no wonder that nodes come in a variety of types. Is the node an element, an attribute, or just plaintext? The `Node` object has different objects to represent each possible type of node. The following is a complete list of all the different node type objects that can be accessed via the DOM. A lot of them won't concern you in this book, because they're better suited for XML documents and not HTML documents, but you should notice that your three main types of nodes, namely element, attribute, and text, are all covered.

OBJECT	DESCRIPTION
<code>Document</code>	The root node of the document
<code>DocumentType</code>	The DTD or schema type of the XML document
<code>DocumentFragment</code>	A temporary storage space for parts of the document
<code>EntityReference</code>	A reference to an entity in the XML document
<code>Element</code>	An element in the document
<code>Attr</code>	An attribute of an element in the document
<code>ProcessingInstruction</code>	A processing instruction
<code>Comment</code>	A comment in an XML document or HTML document
<code>Text</code>	Text that must form a child node of an element
<code>CDATASection</code>	A CDATA section within the XML document
<code>Entity</code>	An unparsed entity in the DTD
<code>Notation</code>	A notation declared within a DTD

We won't go over most of these objects in this chapter.

Each of these objects inherits all the properties and methods of the `Node` object, but also has some properties and methods of its own. You look at some examples in the next section.

## DOM Objects and Their Properties and Methods

If you tried to look at the properties and methods of all the objects in the DOM, it would take up half the book. Instead you're going to actively consider only three of the objects, namely the `Node` object, the `Element` object, and the `Document` object. This is all you'll need to be able to create, amend, and navigate your tree structure. Also, you're not going to spend ages trawling through each of the properties and methods of these objects, but rather look only at some of the most useful properties and methods and use them to achieve specific ends.

**NOTE** *Appendix C contains a relatively complete reference to the DOM, its objects, and their properties.*

### The Document Object and its Methods

The `Document` reference type exposes various properties and methods that are very helpful to someone scripting the DOM. Its methods enable you to find individual or groups of elements and create new elements, attributes, and text nodes. Any DOM scripter should know these methods and properties, because they're used quite frequently.

The `Document` object's methods are probably the most important methods you'll learn. Although many tools are at your disposal, the `Document` object's methods let you find, create, and delete elements in your page.

#### Finding Elements or an Element

Let's say you have an HTML web page—how do you go about getting back a particular element on the page in script? The `Document` reference type exposes the following methods to perform this task.

METHODS OF THE DOCUMENT OBJECT	DESCRIPTION
<code>getElementById(idValue)</code>	Returns a reference (a node) of an element, when supplied with the value of the <code>id</code> attribute of that element
<code>getElementsByTagName(tagName)</code>	Returns a reference (a node list) to a set of elements that have the same tag as the one supplied in the argument
<code>querySelector(cssSelector)</code>	Returns a reference (a node) of the first element that matches the given CSS selector
<code>querySelectorAll(cssSelector)</code>	Returns a reference (a node list) to a set of elements that match the given CSS selector

The first of these methods, `getElementById()`, requires you to ensure that every element you want to quickly access in the page uses an `id` attribute, otherwise a `null` value (a word indicating a missing or unknown value) will be returned by your method. Let's go back to the first example and add some `id` attributes to the elements.

```
<!DOCTYPE html>

<html lang="en">
<head>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph.</p>
</body>
</html>
```

Now you can use the `getElementById()` method to return a reference to any of the HTML elements with `id` attributes on your page. For example, if you add the following code in the highlighted section, you can find and reference the `<h1/>` element:

```
<!DOCTYPE html>

<html lang="en">
<head>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph.</p>
  <script>
    alert(document.getElementById("heading1"));
  </script>
</body>
</html>
```

Figure 9-4 shows the result of this code in Firefox.

**NOTE** `HTMLHeadingElement` is an object of the HTML DOM. All HTML elements have a corresponding reference type in the DOM. See Appendix C for more objects of the HTML DOM.

You might have been expecting it to return something along the lines of `<h1/>` or `<h1 id="heading1">`, but all it's actually returning is a reference to the `<h1/>` element. This reference to the `<h1/>` element is more useful though, because you can use it to alter attributes of the element, such as by changing the color or size. You can do this via the `style` object:

```
<!DOCTYPE html>

<html lang="en">
```

```
<head>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph.</p>

  <script>
    var h1Element = document.getElementById("heading1");
    h1Element.style.color = "red";
  </script>
</body>
</html>
```

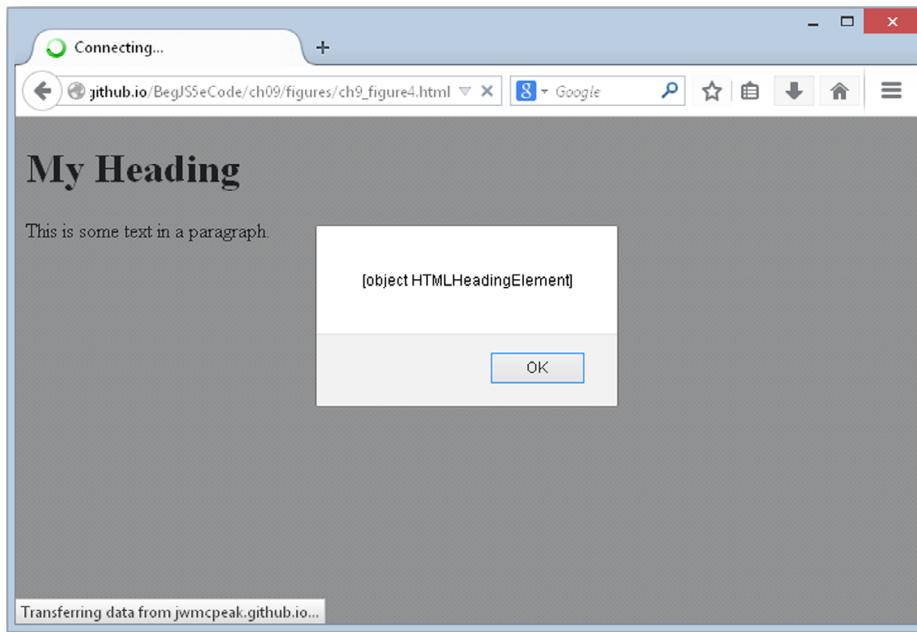


FIGURE 9-4

If you display this in the browser, you see that you can directly influence the attributes of the `<h1/>` element in script, as you have done here by changing its text color to red.

**NOTE** The `style` object points to the style attribute of an element; it enables you to change the CSS style assigned to an element. The `style` object is covered later in the chapter.

The second method, `getElementsByName()`, works in the same way, but, as its name implies, it can return more than one element. If you were to go back to the example HTML document with the table and use this method to return the table cells (`<td/>`) in your code, you would get a node list containing a total of four tables. You'd still have only one object returned, but this object would be a collection of elements. Remember that collections are array-like structures, so specify the index number for the specific element you want from the collection. You can use the square brackets if you want; another alternative is to use the `item()` method of the `NodeList` object, like this:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>This is a test page</title>
</head>
<body>
  <span>Below is a table</span>
  <table>
    <tr>
      <td>Row 1 Cell 1</td>
      <td>Row 1 Cell 2</td>
    </tr>
  </table>
  <script>
    var tdElement = document.getElementsByTagName("td").item(0);
    tdElement.style.color = "red";
  </script>
</body>
</html>
```

If you ran this example, once again using the `style` object, it would alter the style of the contents of the first cell in the table. If you wanted to change the color of all the cells in this way, you could loop through the node list, like this:

```
<script>
  var tdElements = document.getElementsByTagName("td");
  var length = tdElements.length;

  for (var i = 0; i < length; i++) {
    tdElements[i].style.color = "red";
  }
</script>
```

One thing to note about the `getElementsByName()` method is that it takes the element names within quotation marks and without the angle brackets (`<>`) that normally surround tags.

The third method, `querySelector()`, retrieves the first element that matches the provided CSS selector. This is a convenient way of retrieving an element that does not have an `id` attribute (if it does have an `id` attribute, you want to use `getElementById()`).



For example, consider the following HTML:

```
<p class="sub-title">This is a <span>special</span> paragraph element  
  that contains <span>some text</span></p>.
```

Using the `querySelector()` method, you can retrieve the first `<span/>` element in this HTML with the following code:

```
var firstSpan = document.querySelector(".sub-title span");
```

The provided CSS selector matches all `<span/>` elements contained within a parent element with a CSS class of `sub-title`. This HTML contains two such `<span/>` elements, but `querySelector()` only returns the first: the `<span/>` element containing the text “special.” Just as with the previous examples, you can modify the element’s text color with its `style` property:

```
<script>  
  var firstSpan = document.querySelector(".sub-title span");  
  firstSpan.style.color = "red";  
</script>
```

If you wanted to retrieve all of the `<span/>` elements in this HTML, you want to use the fourth method, `querySelectorAll()`, like this:

```
var spans = document.querySelectorAll(".sub-title span");
```

And just as with the `getElementsByTagName()` example, you can use a loop and modify all the elements contained within the `spans` `NodeList`:

```
<script>  
  var spans = document.querySelectorAll(".sub-title span");  
  var length = spans.length;  
  
  for (var i = 0; i < length; i++) {  
    spans[i].style.color = "red";  
  }  
</script>
```

**NOTE** The `querySelector()` and `querySelectorAll()` methods aren’t actually part of the DOM standard. They’re defined within the W3C’s *Selectors API*, which is one of the components to be consolidated with DOM level 3 into DOM level 4. You can use these methods on `Element` objects, too.

## Creating Elements and Text

The `Document` object also boasts some methods for creating elements and text, shown in the following table.

METHODS OF THE DOCUMENT OBJECT	DESCRIPTION
<code>createElement (elementName)</code>	Creates an element node with the specified tag name. Returns the created element
<code>createTextNode (text)</code>	Creates and returns a text node with the supplied text

The following code demonstrates the use of these methods:

```
var pElement = document.createElement("p");
var text = document.createTextNode("This is some text.");
```

This code creates a `<p/>` element and stores its reference in the `pElement` variable. It then creates a text node containing the text `This is some text.` and stores its reference in the `text` variable.

It's not enough to create nodes, however; you have to add them to the document. We'll discuss how to do this in just a bit.

## Property of the Document Object: Getting the Document's Root Element

You now have a reference to individual elements on the page, but what about the tree structure mentioned earlier? The tree structure encompasses all the elements and nodes on the page and gives them a hierarchical structure. If you want to reference that structure, you need a particular property of the document object that returns the outermost element of your document. In HTML, this should always be the `<html/>` element. The property that returns this element is `documentElement`, as shown in the following table.

PROPERTY OF THE DOCUMENT OBJECT	DESCRIPTION
<code>documentElement</code>	Returns a reference to the outermost element of the document (the root element; for example, <code>&lt;html/&gt;</code> )

You can use `documentElement` as follows. If you go back to the simple HTML page, you can transfer your entire DOM into one variable like this:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title></title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph</p>
  <script>
    var container = document.documentElement;
  </script>
</body>
</html>
```

The variable `container` now contains the root element, which is `<html/>`. The `documentElement` property returned a reference to this element in the form of an object, an `Element` object to be precise. The `Element` object has its own set of properties and methods. If you want to use them, you can refer to them by using the variable name, followed by the method or property name:

```
container.elementObjectProperty
```

Fortunately, the `Element` object has only one property.

## The Element Object

The `Element` object is quite simple, especially compared to the `Node` object (which you are introduced to later). It exposes only a handful of *members* (properties and methods).

MEMBER NAME	DESCRIPTION
<code>tagName</code>	Gets the element's tag name
<code>getAttribute()</code>	Gets the value of an attribute
<code>setAttribute()</code>	Sets an attribute with a specified value
<code>removeAttribute()</code>	Removes a specific attribute and its value from the element

### Getting the Element's Tag Name: The `tagName` Property

The sole property of the `Element` object is a reference to the tag name of the element: the `tagName` property.

In the previous example, the variable `container` contained the `<html/>` element. Add the following highlighted line, which makes use of the `tagName` property:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title></title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph</p>
  <script>
    var container = document.documentElement;
    alert(container.tagName);
  </script>
</body>
</html>
```

This code will now return proof that your variable `container` holds the outermost element, and by implication all other elements within it (see Figure 9-5).



FIGURE 9-5

## Methods of the Element Object: Getting and Setting Attributes

If you want to set any element attributes, other than the `style` attribute, you should use the DOM-specific methods of the `Element` object.

The three methods you can use to return and alter the contents of an HTML element's attributes are `getAttribute()`, `setAttribute()`, and `removeAttribute()`, as shown in the following table.

METHODS OF THE ELEMENT OBJECT	DESCRIPTION
<code>getAttribute(attributeName)</code>	Returns the value of the supplied attribute Returns <code>null</code> or an empty string if the attribute does not exist
<code>setAttribute(attributeName, value)</code>	Sets the value of an attribute
<code>removeAttribute(attributeName)</code>	Removes the value of an attribute and replaces it with the default value

Let's take a quick look at how these methods work now.

**TRY IT OUT**    **Playing with Attributes**

Open your text editor and type the following code:

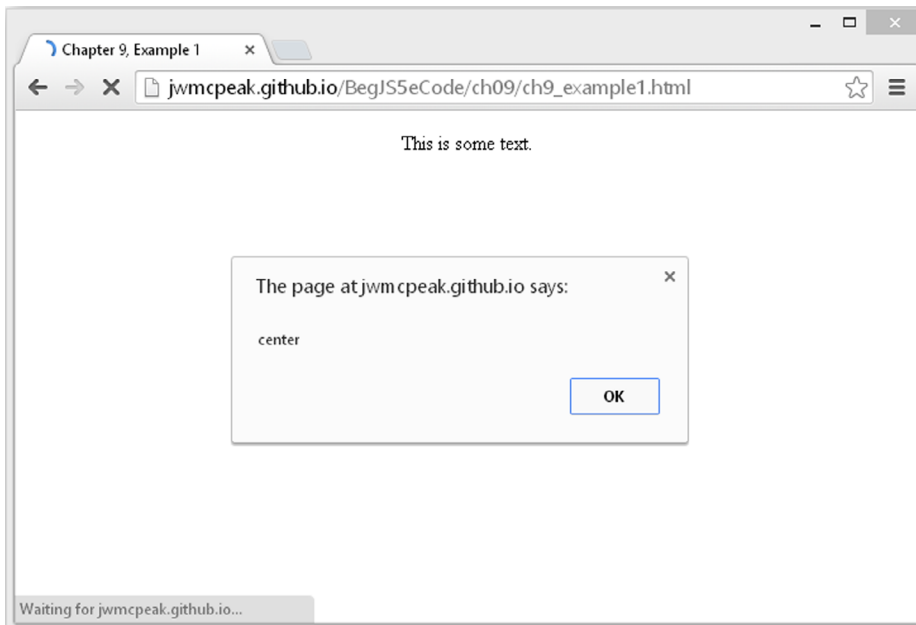
```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Chapter 9, Example 1</title>
</head>
<body>
  <p id="paragraph1">This is some text.</p>
  <script>
    var pElement = document.getElementById("paragraph1");
    pElement.setAttribute("align", "center");

    alert(pElement.getAttribute("align"));

    pElement.removeAttribute("align");
  </script>
</body>
</html>
```

Save this as `ch9_example1.html` and open it in a browser. You'll see the text of the `<p/>` element in the center of the screen and an alert box displaying the text center (Figure 9-6).



**FIGURE 9-6**

When you click the OK button, you'll see the text become left-aligned (Figure 9-7).

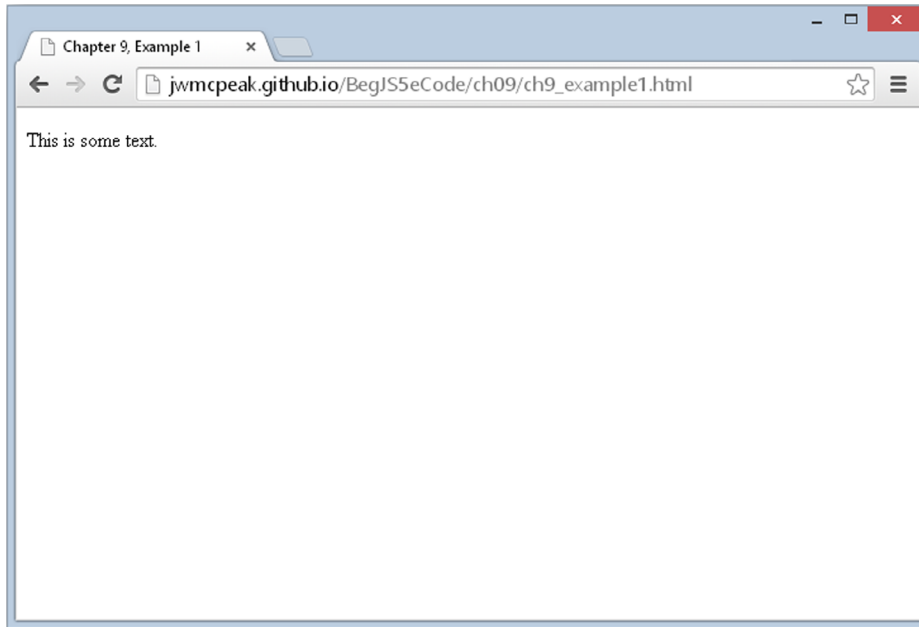


FIGURE 9-7

This HTML page contains one `<p/>` element with an `id` value of `paragraph1`. You use this value in the JavaScript code to find the element node and store its reference in the `pElement` variable with the `getElementById()` method:

```
var pElement = document.getElementById("paragraph1");
```

Now that you have a reference to the element, you use the `setAttribute()` method to set the `align` attribute to center:

```
pElement.setAttribute("align", "center");
```

The result of this code moves the text to the center of the browser's window.

You then use the `getAttribute()` method to get the `align` attribute's value and display it in an alert box:

```
alert(pElement.getAttribute("align"));
```

This code displays the value `"center"` in the alert box.

Finally, you remove the `align` attribute with the `removeAttribute()` method, effectively making the text left-aligned.

**NOTE** *Strictly speaking, the align attribute is deprecated, but you used it because it works and because it has one of the most easily demonstrable visual effects on a web page.*

## The Node Object

You now have your element or elements from the web page, but what happens if you want to move through your page systematically, from element to element or from attribute to attribute? This is where you need to step back to a lower level. To move among elements, attributes, and text, you have to move among nodes in your tree structure. It doesn't matter what is contained within the node, or rather, what sort of node it is. This is why you need to go back to one of the objects of the core DOM specification. Your whole tree structure is made up of these base-level `Node` objects.

### The Node Object: Navigating the DOM

The following table lists some common properties of the `Node` object that provide information about the node, whether it is an element, attribute, or text, and enable you to move from one node to another.

PROPERTIES OF THE NODE OBJECT	DESCRIPTION OF PROPERTY
<code>firstChild</code>	Returns the first child node of an element
<code>lastChild</code>	Returns the last child node of an element
<code>previousSibling</code>	Returns the previous child node of an element at the same level as the current child node
<code>nextSibling</code>	Returns the next child node of an element at the same level as the current child node
<code>ownerDocument</code>	Returns the root node of the document that contains the node (note this is not available in IE 5 or 5.5)
<code>parentNode</code>	Returns the element that contains the current node in the tree structure
<code>nodeName</code>	Returns the name of the node
<code>nodeType</code>	Returns the type of the node as a number
<code>nodeValue</code>	Gets or sets the value of the node in plaintext format

Let's take a quick look at how some of these properties work. Consider this familiar example:

```
<!DOCTYPE html>

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



```

    <title></title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph</p>

  <script>
    var h1Element = document.getElementById("heading1");
    h1Element.style.color = "red";
  </script>
</body>
</html>

```

You can now use `h1Element` to navigate your tree structure and make whatever changes you desire. The following code uses `h1Element` as a starting point to find the `<p/>` element and change its text color:

```

<!DOCTYPE html>

<html lang="en">
<head>
  <title></title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph</p>
  <script>
    var h1Element = document.getElementById("heading1");
    h1Element.style.color = "red";

    var pElement;

    if (h1Element.nextSibling.nodeType == 1) {
      pElement = h1Element.nextSibling;
    } else {
      pElement = h1Element.nextSibling.nextSibling;
    }
    pElement.style.color = "red";
  </script>
</body>
</html>

```

This code demonstrates a fundamental difference between the DOM present in modern browsers and that in older versions of IE. The DOM in modern browsers treats everything as a node in the DOM tree, including the whitespace between elements. On the other hand, older versions of IE strip out this whitespace. So to locate the `<p/>` element in the previous example, a sibling to the `<h1/>` element, you must check the next sibling's `nodeType` property. An element's node type is 1 (text nodes are 3). If the `nextSibling`'s `nodeType` is 1, you assign that sibling's reference to `pElement`. If not, you get the next sibling (the `<p/>` element) of `h1Element`'s sibling (the whitespace text node).

In effect, you are navigating through the tree structure as shown in Figure 9-8.

The same principles also work in reverse. You can go back and change the code to navigate from the `<p/>` element to the `<h1/>` element:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title></title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
  <p id="paragraph1">This is some text in a paragraph</p>
  <script>
    var pElement = document.getElementById("paragraph1");
    pElement.style.color = "red";

    var h1Element;

    if (pElement.previousSibling.nodeType == 1) {
      h1Element = pElement.previousSibling;
    } else {
      h1Element = pElement.previousSibling.previousSibling;
    }
    h1Element.style.color = "red";
  </script>
</body>
</html>
```

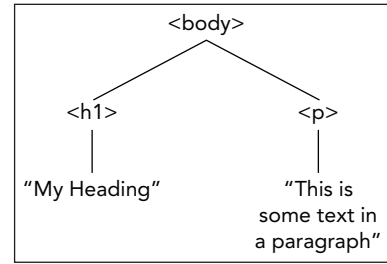


FIGURE 9-8

What you're doing here is the exact opposite; you find the `<p/>` by passing the value of its `id` attribute to the `getElementById()` method and storing the returned element reference to the `pElement` variable. You then find the correct previous sibling so that your code works in all browsers, and you change its text color to red.

## TRY IT OUT Navigating Your HTML Document Using the DOM

Up until now, you've been cheating, because you haven't truly navigated your HTML document. You've just used `document.getElementById()` to return an element and navigated to different nodes from there. Now let's use the `documentElement` property of the `document` object and do this properly. You'll start at the top of your tree and move down through the child nodes to get at those elements; then you'll navigate through your child nodes and change the properties in the same way as before.

Type the following into your text editor:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Chapter 9, Example 2</title>
</head>
<body>
  <h1 id="heading1">My Heading</h1>
```

```

<p id="paragraph1">This is some text in a paragraph</p>

<script>
    var htmlElement; // htmlElement stores reference to <html>
    var headElement; // headElement stores reference to <head>
    var bodyElement; // bodyElement stores reference to <body>
    var h1Element; // h1Element stores reference to <h1>
    var pElement; // pElement stores reference to <p>

    htmlElement = document.documentElement;
    headElement = htmlElement.firstChild;

    alert(headElement.tagName);

    if (headElement.nextSibling.nodeType == 3) {
        bodyElement = headElement.nextSibling.nextSibling;
    } else {
        bodyElement = headElement.nextSibling;
    }

    alert(bodyElement.tagName);

    if (bodyElement.firstChild.nodeType == 3) {
        h1Element = bodyElement.firstChild.nextSibling;
    } else {
        h1Element = bodyElement.firstChild;
    }

    alert(h1Element.tagName);
    h1Element.style.fontFamily = "Arial";

    if (h1Element.nextSibling.nodeType == 3) {
        pElement = h1Element.nextSibling.nextSibling;
    } else {
        pElement = h1Element.nextSibling;
    }

    alert(pElement.tagName);
    pElement.style.fontFamily = "Arial";

    if (pElement.previousSibling.nodeType == 3) {
        h1Element = pElement.previousSibling.previousSibling;
    } else {
        h1Element = pElement.previousSibling;
    }

    h1Element.style.fontFamily = "Courier";
</script>
</body>
</html>

```

Save this as `ch9_example2.html` and open it in your browser.

Click OK in each of the message boxes until you see the page shown in Figure 9-9 (unfortunately, IE does not render the style changes until all alert boxes have been opened and closed).



FIGURE 9-9

You’ve hopefully made this example very transparent by adding several alerts to demonstrate where you are along each section of the tree. You’ve also named the variables with their various elements, to give a clearer idea of what is stored in each variable. (You could just as easily have named them *a*, *b*, *c*, *d*, and *e*, so don’t think you need to be bound by this naming convention.)

You start at the top of the script block by retrieving the whole document using the `documentElement` property:

```
var htmlElement = document.documentElement;
```

The root element is the `<html/>` element, hence the name of your first variable. Now if you refer to your tree, you’ll see that the HTML element must have two child nodes: one containing the `<head/>` element and the other containing the `<body/>` element. You start by moving to the `<head/>` element. You get there using the `firstChild` property of the `Node` object, which contains your `<html/>` element. You use your first alert to demonstrate that this is true:

```
alert(headingElement.tagName);
```

Your `<body/>` element is your next sibling across from the `<head/>` element, so you navigate across by creating a variable that is the next sibling from the `<head/>` element:

```
if (headingElement.nextSibling.nodeType == 3) {  
    bodyElement = headingElement.nextSibling.nextSibling;  
} else {
```

```

        bodyElement = headingElement.nextSibling;
    }

    alert (bodyElement.tagName);

```

Here you check to see what the `nodeType` of the `nextSibling` of `headingElement` is. If it returns 3, (`nodeType` 3 is a text node), you set `bodyElement` to be the `nextSibling` of the `nextSibling` of `headingElement`; otherwise, you just set it to be the `nextSibling` of `headingElement`.

You use an alert to prove that you are now at the `<body/>` element:

```

    alert (bodyElement.tagName);

```

The `<body/>` element in this page also has two children, the `<h1/>` and `<p/>` elements. Using the `firstChild` property, you move down to the `<h1/>` element. Again you check whether the child node is whitespace for standard-compliant browsers. You use an alert again to show that you have arrived at `<h1/>`:

```

    if (bodyElement.firstChild.nodeType == 3) {
        h1Element = bodyElement.firstChild.nextSibling;
    } else {
        h1Element = bodyElement.firstChild;
    }

    alert (h1Element.tagName);

```

After the third alert, the style will be altered on your first element, changing the font to Arial:

```

    h1Element.style.fontFamily = "Arial";

```

You then navigate across to the `<p/>` element using the `nextSibling` property, again checking for whitespace:

```

    if (h1Element.nextSibling.nodeType == 3) {
        pElement = h1Element.nextSibling.nextSibling;
    } else {
        pElement = h1Element.nextSibling;
    }

    alert (pElement.tagName);

```

You change the `<p/>` element's font to Arial also:

```

    pElement.style.fontFamily = "Arial";

```

Finally, you use the `previousSibling` property to move back in your tree to the `<h1/>` element and this time change the font to Courier:

```

    if (pElement.previousSibling.nodeType == 3) {
        h1Element = pElement.previousSibling.previousSibling;
    } else {
        h1Element = pElement.previousSibling;
    }

    h1Element.style.fontFamily = "Courier";

```

This is a fairly easy example to follow because you’re using the same tree structure you created with diagrams, but it does show how the DOM effectively creates this hierarchy and that you can move around within it using script.

### Methods of the Node Object

Whereas the `Node` object’s properties enable you to navigate the DOM, its methods provide the completely different ability to add and remove nodes from the DOM, thus fundamentally altering the structure of the HTML document. The following table lists these methods.

METHODS OF NODE OBJECTS	DESCRIPTION
<code>appendChild(newNode)</code>	Adds a new <code>Node</code> object to the end of the list of child nodes. This method returns the appended node.
<code>cloneNode(cloneChildren)</code>	Returns a duplicate of the current node. It accepts a boolean value. If the value is <code>true</code> , the method clones the current node and all child nodes. If the value is <code>false</code> , only the current node is cloned and child nodes are left out of the clone.
<code>hasChildNodes()</code>	Returns <code>true</code> if a node has any child nodes and <code>false</code> if not
<code>insertBefore(newNode, referenceNode)</code>	Inserts a new <code>Node</code> object into the list of child nodes before the node stipulated by <code>referenceNode</code> . Returns the inserted node
<code>removeChild(childNode)</code>	Removes a child node from a list of child nodes of the <code>Node</code> object. Returns the removed node

TRY IT OUT

### Creating HTML Elements and Text with DOM Methods

In this Try It Out you create a web page with just paragraph `<p/>` and heading `<h1/>` elements, but instead of HTML you’ll use the DOM properties and methods to place these elements on the web page. Start up your preferred text editor and type the following:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Chapter 9, Example 3</title>
</head>
<body>
  <script>
    var newText = document.createTextNode("My Heading");
```

```
var newElem = document.createElement("h1");

newElem.appendChild(newText);
document.body.appendChild(newElem);

newText = document.createTextNode("This is some text in a paragraph");
newElem = document.createElement("p");

newElem.appendChild(newText);
document.body.appendChild(newElem);
</script>
</body>
</html>
```

Save this page as `ch9 _ example3.html` and open it in a browser (Figure 9-10).

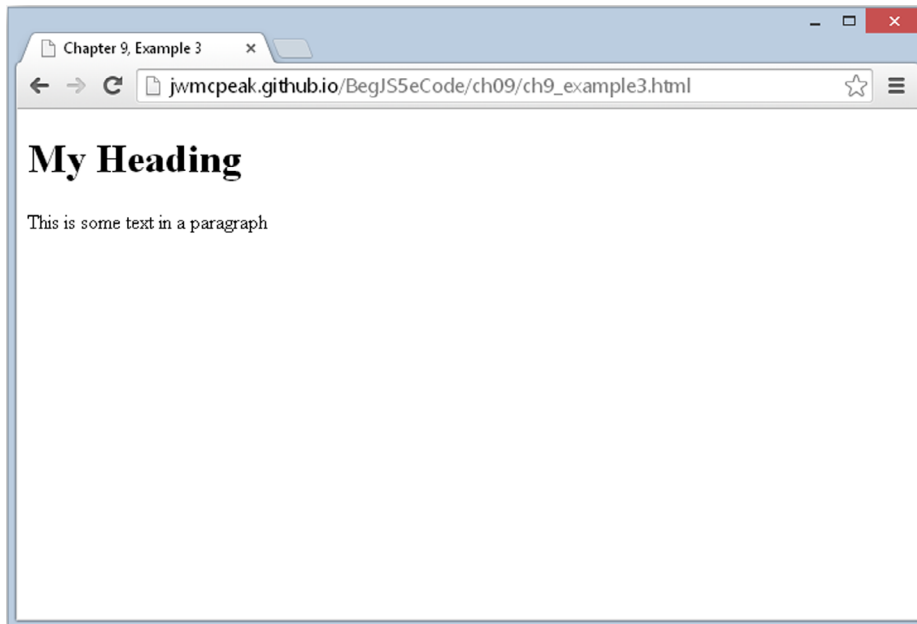


FIGURE 9-10

It all looks a bit dull and tedious, doesn't it? And yes, you could have done this much more simply with HTML. That isn't the point, though. The idea is that you use DOM properties and methods, accessed with JavaScript, to insert these elements. The first two lines of the script block are used to define the variables in your script, which are initialized to hold the text you want to insert into the page and the HTML element you want to insert:

```
var newText = document.createTextNode("My Heading");
var newElem = document.createElement("h1");
```



You start at the bottom of your tree first, by creating a text node with the `createTextNode()` method. Then use the `createElement()` method to create an HTML heading.

At this point, the two variables are entirely separate from each other. You have a text node, and you have an `<h1/>` element, but they're not connected. The next line enables you to attach the text node to your HTML element. You reference the HTML element you have created with the variable name `newElem`, use the `appendChild()` method of your node, and supply the contents of the `newText` variable you created earlier as a parameter:

```
newElem.appendChild(newText);
```

Let's recap. You created a text node and stored it in the `newText` variable. You created an `<h1/>` element and stored it in the `newElem` variable. Then you appended the text node as a child node to the `<h1/>` element. That still leaves you with a problem: You've created an element with a value, but the element isn't part of your document. You need to attach the entirety of what you've created so far to the document body. Again, you can do this with the `appendChild()` method, but this time call it on the `document.body` object (which, too, is a `Node`):

```
document.body.appendChild(newElem);
```

This completes the first part of your code. Now all you have to do is repeat the process for the `<p/>` element:

```
newText = document.createTextNode("This is some text in a paragraph");
newElem = document.createElement("p");

newElem.appendChild(newText);
document.body.appendChild(newElem);
```

You create a text node first; then you create an element. You attach the text to the element, and finally you attach the element and text to the body of the document.

---

It's important to note that the order in which you create nodes does not matter. This example had you create the text nodes before the element nodes; if you wanted, you could have created the elements first and the text nodes second.

However, the order in which you append nodes is very important for performance reasons. Updating the DOM can be an expensive process, and performance can suffer if you make many changes to the DOM. For example, this example updated the DOM only two times by appending the completed elements to the document's body. It would require four updates if you appended the element to the document's body and then appended the text node to the element. As a rule of thumb, only append completed element nodes (that is, the element, its attributes, and any text) to the document whenever you can.

Now that you can navigate and make changes to the DOM, let's look further into manipulating DOM nodes.

## MANIPULATING THE DOM

DOM scripting is the manipulation of an HTML page after it's loaded into the browser. Up to this point, you've examined the properties and methods of the basic DOM objects and learned how to traverse the DOM through JavaScript.

Throughout the previous section, you saw some examples of manipulating the DOM; more specifically, you saw that you can change the color and font family of text contained within an element. In this section, you expand on that knowledge.

### Accessing Elements

As you saw in the previous section, the DOM holds the tools you need to find and access HTML elements; you used the `getElementById()` method quite frequently, and through examples you saw how easy it was to find specific elements in the page.

When scripting the DOM, chances are you have a pretty good idea of what elements you want to manipulate. The easiest ways to find those elements are with the `getElementById()`, `querySelector()`, and `querySelectorAll()` methods. If an element has an `id` attribute, use `getElementById()` because it is the fastest way to find an element in the page. Otherwise, you'll need to use the `querySelector()` and `querySelectorAll()` methods.

### Changing Appearances

Probably the most common DOM manipulation is to change the way an element looks. Such a change can create an interactive experience for visitors to your website and can even be used to alert them to important information or that an action is required by them. Changing the way an element looks consists almost exclusively of changing CSS properties for an HTML element. You can do this two ways through JavaScript:

- Change each CSS property with the `style` property.
- Change the value of the element's `class` attribute.

### Using the style Property

To change specific CSS properties, you must look to the `style` property. All browsers implement this object, which maps directly to the element's `style` attribute. This object contains CSS properties, and by using it you can change any CSS property that the browser supports. You've already seen the `style` property in use, but here's a quick refresher:

```
element.style.cssProperty = value;
```

The CSS property names generally match those used in a CSS style sheet; therefore, changing the text color of an element requires the use of the `color` property, like this:

```
var divAdvert = document.getElementById("divAdvert");
divAdvert.style.color = "blue";
```

In some cases, however, the property name is a little different from the one seen in a CSS file. CSS properties that contain a hyphen (-) are a perfect example of this exception. In the case of these properties, you remove the hyphen and capitalize the first letter of the word that follows the hyphen. The following code shows the incorrect and correct ways to do this:

```
divAdvert.style.background-color = "gray"; // wrong

divAdvert.style.backgroundColor = "gray"; // correct
```

You can also use the `style` object to retrieve styles that have previously been declared. However, if the `style` property you try to retrieve has not been set with the `style` attribute (inline styles) or with the `style` object, you will not retrieve the property's value. Consider the following HTML containing a style sheet and `<div/>` element:

```
<style>
#divAdvert {
    background-color: gray;
}
</style>

<div id="divAdvert" style="color: green">I am an advertisement.</div>
```

When the browser renders this element, it will have green text on a gray background. If you had used the `style` object to retrieve the value of both the `background-color` and `color` properties, you'd get the following mixed results:

```
var divAdvert = document.getElementById("divAdvert");
alert(divAdvert.style.backgroundColor); // alerts an empty string
alert(divAdvert.style.color); // alerts green
```

You get these results because the `style` object maps directly to the `style` attribute of the element. If the style declaration is set in a `<style/>` element, or in an external style sheet, you cannot retrieve that property's value with the `style` object.

## TRY IT OUT Using the style Object

Let's look at a simple example of changing the appearance of some text by using the `style` object. Type the following code:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 9, Example 4</title>
    <style>
        #divAdvert {
            font: 12pt arial;
        }
    </style>
</head>
```

```
<body>
  <div id="divAdvert">
    Here is an advertisement.
  </div>

  <script>
    var divAdvert = document.getElementById("divAdvert");
    divAdvert.style.fontStyle = "italic";
    divAdvert.style.textDecoration = "underline";
  </script>

</body>
</html>
```

Save this as `ch9_example4.html`. When you run this in your browser, you should see a single line of text that is italicized and underlined, as shown in Figure 9-11.

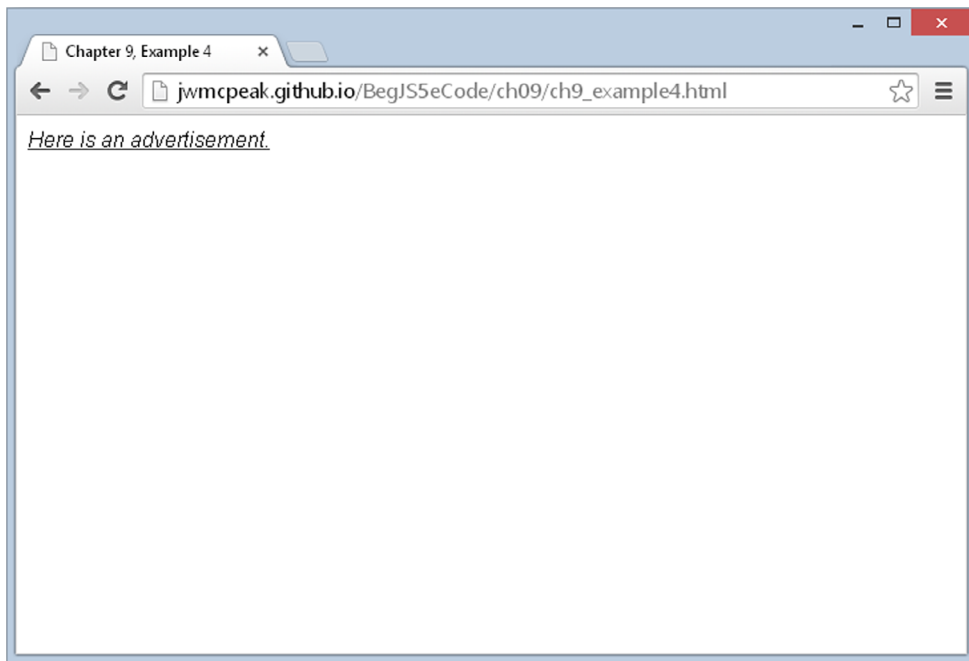


FIGURE 9-11

In the page's body, a `<div/>` element is defined with an `id` of `divAdvert`. The `<script/>` element appears after the `<div/>`, and it contains the following JavaScript code:

```
var divAdvert = document.getElementById("divAdvert");
divAdvert.style.fontStyle = "italic";
divAdvert.style.textDecoration = "underline";
```

Before you can do anything to the `<div/>` element, you must first retrieve it. You do this simply by using the `getElementById()` method. Now that you have the element, you manipulate its style by first italicizing the text with the `fontStyle` property. Next, you underline the text by using the `textDecoration` property and assigning its value to underline.

It's very important that the `<div id="divAdvert"/>` element is loaded into the browser before you retrieve it with `getElementById()`. This is why the `<script/>` element appears after `<div id="divAdvert"/>`. By the time the browser loads and executes the JavaScript code, the `<div/>` element is loaded into the DOM.

---

## Changing the class Attribute

You can assign a CSS class to elements by using the element's `class` attribute. This attribute is exposed in the DOM by the `className` property and can be changed through JavaScript to associate a different style rule with the element:

```
element.className = sNewClassName;
```

Using the `className` property to change an element's style is advantageous in two ways:

- It reduces the amount of JavaScript you have to write, which no one is likely to complain about.
- It keeps style information out of the JavaScript file and puts it into the CSS file where it belongs. Making any type of changes to the style rules is easier because you do not have to have several files open in order to change them.

### TRY IT OUT Using the className Property

Let's revisit the code from `ch9_example4.html` from the previous section and make some revisions. Type the following code:

```
-transitional.dtd">
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Chapter 9, Example 5</title>
  <style>
    #divAdvert {
      font: 12pt arial;
    }

    .new-style {
      font-style: italic;
      text-decoration: underline;
    }
  </style>
</head>
<body>
  <div id="divAdvert">
```

```

        Here is an advertisement.
    </div>

    <script>
        var divAdvert = document.getElementById("divAdvert");
        divAdvert.className = "new-style";
    </script>

</body>
</html>

```

Save this as `ch9_example5.html`.

Two key differences exist between `ch9_example4.html` and `ch9_example5.html`. The first is the addition of a CSS class called `new-style`:

```

.newStyle {
    font-style: italic;
    text-decoration: underline;
}

```

This class contains style declarations to specify italicized and underlined text.

The second change is in the JavaScript itself:

```

    var divAdvert = document.getElementById("divAdvert");
    divAdvert.className = "new-style";
}

```

The first statement retrieves the `<div/>` element by using the `getElementById()` method. The second statement changes the `className` property to the value `new-style`. With this line, the `divAdvert` element takes on a new style rule and the browser changes the way it looks.

**NOTE** Although it wasn't demonstrated here, the HTML `class` attribute, and thus the `className` property, can contain multiple CSS class names. You see more about multiple class names in Chapter 16.

## Positioning and Moving Content

Changing the appearance of an element is an important pattern in DOM scripting, and it finds its place in many scripts. However, there is more to DOM scripting than just changing the way content appears on the page; you can also change the position of an element with JavaScript.

Moving content with JavaScript is just as easy as using the `style` object. You use the `position` property to change the type of position desired, and by using the `left` and `top` properties, you can position the element:

```

var divAdvert = document.getElementById("divAdvert");

divAdvert.style.position = "absolute";

```

```
divAdvert.style.left = "100px"; // set the left position
divAdvert.style.top = "100px"; // set the right position
```

This code first retrieves the `divAdvert` element. Then it sets the element's position to absolute and moves the element 100 pixels from the left and top edges. Notice the addition of `px` to the value assigned to the positions. You must specify a unit when assigning a positional value; otherwise, the browser will not position the element.

**NOTE** Positioning elements requires the position of absolute or relative.

## Example: Animated Advertisement

Perhaps the most creative use of DOM scripting is in animating content on the page. You can perform a variety of animations: you can fade text elements or images in and out, give them a swipe animation (making it look like as if they are wiped onto the page), and animate them to move around on the page.

Animation can give important information the flair it needs to be easily recognized by your reader, as well as adding a “that’s cool” factor. Performing animation with JavaScript follows the same principles of any other type of animation: You make seemingly insignificant changes one at a time in a sequential order until you reach the end of the animation. Essentially, with any animation, you have the following requisites:

1. The starting state
2. The movement toward the final goal
3. The end state; stopping the animation

Animating an absolutely positioned element, as you’re going to do in this section, is no different. First, with CSS, position the element at the start location. Then perform the animation up until you reach the end point, which signals the end of the animation.

In this section, you learn how to animate content to bounce back and forth between two points. To do this, you need one important piece of information: the content’s current location.

### Are We There Yet?

The DOM in modern browsers exposes the `offsetTop` and `offsetLeft` properties of an HTML element object. These two properties return the calculated position relative to the element’s parent element: `offsetTop` tells you the top location, and `offsetLeft` tells you the left position. The values returned by these properties are numerical values, so you can easily check to see where your element currently is in the animation. For example:

```
var endPointX = 394;

if (element.offsetLeft < endPointX) {
    // continue animation
}
```



The preceding code specifies the end point (in this case, 394) and assigns it to the `endPointX` variable. You can then check to see if the element's `offsetLeft` value is currently less than that of the end point. If it is, you can continue the animation. This example brings us to the next topic in content movement: performing the animation.

## Performing the Animation

To perform an animation, you need to modify the `top` and `left` properties of the `style` object incrementally and quickly. You do this with periodic function execution until it's time to end the animation. To do this, use one of two methods of the `window` object: `setTimeout()` or `setInterval()`. This example uses the `setInterval()` method to periodically move an element.

### TRY IT OUT Animating Content

This page moves an element across the page from right to left. Open your text editor and type the following:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Chapter 9, Example 6</title>
  <style>
    #divAdvert {
      position: absolute;
      font: 12px Arial;
      top: 4px;
      left: 0px;
    }
  </style>
</head>
<body>
  <div id="divAdvert">
    Here is an advertisement.
  </div>

  <script>
    var switchDirection = false;

    function doAnimation() {
      var divAdvert = document.getElementById("divAdvert");
      var currentLeft = divAdvert.offsetLeft;
      var newLocation;

      if (!switchDirection) {
        newLocation = currentLeft + 2;

        if (currentLeft >= 400) {
          switchDirection = true;
        }
      }
    }
  </script>
</body>
</html>
```

```
        } else {
            newLocation = currentLeft - 2;

            if (currentLeft <= 0) {
                switchDirection = false;
            }
        }

        divAdvert.style.left = newLocation + "px";
    }

    setInterval(doAnimation, 10);
</script>

</body>
</html>
```

Save this page as `ch9_example6.html` and load it into your browser. When you load the page into the browser, the content should start moving from left to right, starting at the left edge of the viewport. When the content reaches a left position of 400 pixels, the content switches directions and begins to move back toward the left edge. This animation is continuous, so it should bounce between the two points (0 and 400) perpetually.

Inside the body of the page is a `<div/>` element. This element has an `id` of `divAdvert` so that you can retrieve it with the `getElementById()` method, because this is the element you want to animate:

```
<div id="divAdvert">
    Here is an advertisement.
</div>
```

This element has no `style` attributes because all the style information is inside the style sheet located in the head of the page. In the style sheet, you define a starting point for this `<div/>`. You want the animation to go first from left to right, and you want it to start at the left edge of the browser:

```
#divAdvert {
    position: absolute;
    font: 12pt arial;
    top: 4px;
    left: 0px;
}
```

The first style declaration positions the element absolutely, and the second specifies the font as 12-point Arial. The next declaration positions the element four pixels from the top of the browser's viewport. Setting the top position away from the topmost edge makes the text a little easier to read. Finally, the last line positions the `divAdvert` element along the left edge of the viewport with the `left` property.

Within the script block is a global variable called `switchDirection`:

```
var switchDirection = false;
```

This variable keeps track of the direction in which the content is currently going. If `switchDirection` is `false`, the content is moving from left to right, which is the default. If `switchDirection` is `true`, the content is moving from right to left.

Next in the script block is the `doAnimation()` function, which performs the animation:

```
function doAnimation() {
    var divAdvert = document.getElementById("divAdvert");
    var currentLeft = divAdvert.offsetLeft;
    var newLocation;
```

First, you retrieve the `divAdvert` element with the `getElementById()` method; you also retrieve the `offsetLeft` property and assign its value to the `currentLeft` variable. You use this variable to check the content's current position. Next, create a variable called `newLocation` that will contain the new `left` position, but before you assign its value you need to know the direction in which the content is moving:

```
    if (!switchDirection) {
        newLocation = currentLeft + 2;

        if (currentLeft >= 400) {
            switchDirection = true;
        }
    }
```

First, check the direction by checking the `switchDirection` variable. Remember, if it is `false`, the animation is moving from left to right; so assign `newLocation` to contain the content's current position and add 2, thus moving the content two pixels to the right.

You then need to check if the content has reached the `left` position of 400 pixels. If it has, you need to switch the direction of the animation, and you do this by changing `switchDirection` to `true`. So the next time `doAnimation()` runs, it will begin to move the content from right to left.

The code to move the element in this new direction is similar to the previous code, except for a few key differences:

```
    else {
        newLocation = currentLeft - 2;

        if (currentLeft <= 0) {
            switchDirection = false;
        }
    }
```

The first difference is the value assigned to `newLocation`; instead of adding 2 to the current location, you subtract 2, thus moving the content two pixels to the left. Next, check if `currentLeft` is less than or equal to 0. If it is, you know you've reached the ending point of the right-to-left movement and need to switch directions again by assigning `switchDirection` to be `false`.

Finally, set the new position of the content:

```
    divAdvert.style.left = newLocation + "px";
}
```

This final line of the function sets the element's `left` property to the value stored in the `newLocation` variable plus the string `"px"`.

To run the animation, use `setInterval()` to continuously execute `doAnimation()`. The following code runs `doAnimation()` every 10 milliseconds:

```
setInterval(doAnimation, 10);
```

At this speed, the content moves at a pace that is easily seen by those viewing the page. If you want to speed up or slow down the animation, simply change how often the `setInterval()` function calls `doAnimation()` by changing the second parameter.

---

What have you seen so far? Well, you've seen the DOM hierarchy and how it represents the HTML document as a tree-like structure. You navigated through the different parts of it via DOM objects (the `Node` objects) and their properties, and you changed the properties of objects, thus altering the content of the web page. This leaves just one area of the DOM to cover: the event model. You learn about events in the next chapter.

## SUMMARY

This chapter has featured quite a few diversions and digressions, but these were necessary to demonstrate the position and importance of the document object model in JavaScript.

This chapter covered the following points:

- It started by outlining two of the main standards—HTML and ECMAScript—and examined the relationships between them. You saw that a common aim emerging from these standards was to provide guidelines for coding HTML web pages. Those guidelines in turn benefited the document object model, making it possible to access and manipulate any item on the web page using script if web pages were coded according to those guidelines.
- You examined the document object model and saw that it offered a browser-independent means of accessing the items on a web page, and that it resolved some of the problems that dogged older browsers. You saw how the DOM represents the HTML document as a tree structure and how it is possible for you to navigate through the tree to different elements and use the properties and methods it exposes to access the different parts of the web page.
- The DOM lets you change a page after it is loaded into the browser, and you can perform a variety of user interface tricks to add some flair to your page.
- You learned how to change a tag's style by using the `style` and `className` properties.
- You also learned the basics of animation and made text bounce back and forth between two points.

---

**EXERCISES**

You can find suggested solutions to these questions in Appendix A.

1. Here's some HTML code that creates a table. Re-create this table using only JavaScript and the core DOM objects to generate the HTML. Test your code in all browsers available to you to make sure it works in them. Hint: Comment each line as you write it to keep track of where you are in the tree structure, and create a new variable for every element on the page (for example, not just one for each of the TD cells but nine variables).

```
<table>
  <tr>
    <td>Car</td>
    <td>Top Speed</td>
    <td>Price</td>
  </tr>
  <tr>
    <td>Chevrolet</td>
    <td>120mph</td>
    <td>$10,000</td>
  </tr>
  <tr>
    <td>Pontiac</td>
    <td>140mph</td>
    <td>$20,000</td>
  </tr>
</table>
```

2. Modify `ch9_example6.html` from the "Animating Content" Try It Out so that the amount of pixels moved in either direction is controlled by a global variable. Call it `direction`. Remove the `switchDirection` variable, and change the code to use the new `direction` variable to determine when the animation should change directions.
-