CHAPTER 14

The Window Object

Chapter 13 introduced the Window object and the central role it plays in client-side JavaScript: it is the global object for client-side JavaScript programs. This chapter covers the properties and methods of the Window object. These properties define a number of different APIs, only some of which are actually related to the browser windows for which the Window object was named. This chapter covers the following:

- §14.1 shows how to use `setTimeout()` and `setInterval()` to register a function to be invoked at specified times in the future.
- §14.2 explains how to use the `location` property to obtain the URL of the currently displayed document and to load new documents.
- §14.3 covers the `history` property, and shows how to move the browser backward and forward through its history.
- §14.4 shows how to use the `navigator` property to obtain browser vendor and version information and how to use the `screen` property to query the size of the desktop.
- §14.5 shows how to display simple text dialogs with the `alert()`, `confirm()`, and `prompt()` methods and how to display HTML dialog boxes with `showModalDialog()`.
- §14.6 explains how you can register an `onerror` handler method to be invoked when uncaught JavaScript exceptions occur.
- §14.7 explains that the IDs and names of HTML elements are used as properties of the Window object.
- §14.8 is a long section that explains how to open and close browser windows and how to write JavaScript code that works with multiple windows and nested frames.

14.1 Timers

`setTimeout()` and `setInterval()` allow you to register a function to be invoked once or repeatedly after a specified amount of time has elapsed. These are important global functions of client-side JavaScript, and are therefore defined as methods of Window,

but they are general-purpose functions and don't really have anything to do with the window.

The `setTimeout()` method of the Window object schedules a function to run after a specified number of milliseconds elapses. `setTimeout()` returns a value that can be passed to `clearTimeout()` to cancel the execution of the scheduled function.

`setInterval()` is like `setTimeout()` except that the specified function is invoked repeatedly at intervals of the specified number of milliseconds:

```
setInterval(updateClock, 60000);  // Call updateClock() every 60 seconds
```

Like `setTimeout()`, `setInterval()` returns a value that can be passed to `clearInterval()` to cancel any future invocations of the scheduled function.

Example 14-1 defines a utility function that waits a specified amount of time, invokes a function repeatedly, and then cancels the invocations after another specified amount of time. It demonstrates `setTimeout()`, `setInterval()`, and `clearInterval()`.

Example 14-1. A timer utility function

```
/*
 * Schedule an invocation or invocations of f() in the future.
 * Wait start milliseconds, then call f() every interval milliseconds,
 * stopping after a total of start+end milliseconds.
 * If interval is specified but end is omitted, then never stop invoking f.
 * If interval and end are omitted, then just invoke f once after start ms.
 * If only f is specified, behave as if start was 0.
 * Note that the call to invoke() does not block: it returns right away.
 */
function invoke(f, start, interval, end) {
    if (!start) start = 0;          // Default to 0 ms
    if (arguments.length <= 2)      // Single-invocation case
        setTimeout(f, start);       // Single invocation after start ms.
    else {                          // Multiple invocation case
        setTimeout(repeat, start);  // Repetitions begin in start ms
        function repeat() {         // Invoked by the timeout above
            var h = setInterval(f, interval); // Invoke f every interval ms.
            // And stop invoking after end ms, if end is defined
            if (end) setTimeout(function() { clearInterval(h); }, end);
        }
    }
}
```

For historical reasons, you can pass a string as the first argument to `setTimeout()` and `setInterval()`. If you do this, the string will be evaluated (as with `eval()`) after the specified timeout or interval. The HTML5 specification (and all browsers except IE) allow additional arguments to `setTimeout()` and `setInterval()` after the first two. Any such arguments are passed to the function that is invoked. If portability with IE is required, however, you shouldn't use this feature.

If you call setTimeout() with a time of 0 ms, the function you specify is not invoked right away. Instead, it is placed on a queue to be invoked "as soon as possible" after any currently pending event handlers finish running.

14.2 Browser Location and Navigation

The location property of the Window object refers to a Location object, which represents the current URL of the document displayed in the window, and which also defines methods for making the window load a new document.

The location property of the Document object also refers to the Location object:

```
window.location === document.location   // always true
```

The Document object also has a URL property, which is a static string that holds the URL of the document when it was first loaded. If you navigate to fragment identifiers (like "#table-of-contents") within the document, the Location object is updated to reflect this, but the document.URL property remains unchanged.

14.2.1 Parsing URLs

The location property of a window is a reference to a Location object; it represents the current URL of the document being displayed in that window. The href property of the Location object is a string that contains the complete text of the URL. The toString() method of the Location object returns the value of the href property, so in contexts that will implicitly invoke toString(), you can just write location rather than location.href.

Other properties of this object—protocol, host, hostname, port, pathname, search, and hash—specify the various individual parts of the URL. They are known as "URL decomposition" properties, and they are also supported by Link objects (created by <a> and <area> elements in HTML documents). See the Location and Link entries in Part IV for further details.

The hash and search properties of the Location object are interesting ones. The hash property returns the "fragment identifier" portion of the URL, if there is one: a hash mark (#) followed by an element ID. The search property is similar. It returns the portion of the URL that starts with a question mark: often some sort of query string. In general, this portion of a URL is used to parameterize the URL and provides a way to embed arguments in it. While these arguments are usually intended for scripts run on a server, there is no reason why they cannot also be used in JavaScript-enabled pages. Example 14-2 shows the definition of a general-purpose urlArgs() function you can use to extract arguments from the search property of a URL. The example uses decodeURIComponent(), which is a global function defined by client-side JavaScript. (See Global in Part III for details.)

Example 14-2. Extracting arguments from the search string of a URL

```
/*
 * This function parses ampersand-separated name=value argument pairs from
 * the query string of the URL. It stores the name=value pairs in
 * properties of an object and returns that object. Use it like this:
 *
 * var args = urlArgs();  // Parse args from URL
 * var q = args.q || "";  // Use argument, if defined, or a default value
 * var n = args.n ? parseInt(args.n) : 10;
 */
function urlArgs() {
    var args = {};                              // Start with an empty object
    var query = location.search.substring(1);   // Get query string, minus '?'
    var pairs = query.split("&");               // Split at ampersands
    for(var i = 0; i < pairs.length; i++) {     // For each fragment
        var pos = pairs[i].indexOf('=');        // Look for "name=value"
        if (pos == -1) continue;                // If not found, skip it
        var name = pairs[i].substring(0,pos);   // Extract the name
        var value = pairs[i].substring(pos+1);  // Extract the value
        value = decodeURIComponent(value);      // Decode the value
        args[name] = value;                     // Store as a property
    }
    return args;                                // Return the parsed arguments
}
```

14.2.2 Loading New Documents

The `assign()` method of the Location object makes the window load and display the document at the URL you specify. The `replace()` method is similar, but it removes the current document from the browsing history before loading the new document. When a script unconditionally loads a new document, the `replace()` method is often a better choice than `assign()`. Otherwise, the Back button would take the browser back to the original document, and the same script would again load the new document. You might use `location.replace()` to load a static HTML version of your web page if you detected that the user's browser did not have the features required to display the full-featured version:

```
// If the browser does not support the XMLHttpRequest object
// redirect to a static page that does not require it.
if (!XMLHttpRequest) location.replace("staticpage.html");
```

Notice that the URL passed to `replace()` is a relative one. Relative URLs are interpreted relative to the page in which they appear, just as they would be if they were used in a hyperlink.

In addition to the `assign()` and `replace()` methods, the Location object also defines `reload()`, which makes the browser reload the document.

A more traditional way to make the browser navigate to a new page is to simply assign the new URL directly to the location property:

```
location = "http://www.oreilly.com"; // Go buy some books!
```

You can also assign relative URLs to `location`. They are resolved against the current URL:

```
location = "page2.html";              // Load the next page
```

A bare fragment identifier is a special kind of relative URL that does not cause the browser to load a new document but simply scroll to display a new section of the document. The identifier `#top` is a special case: if no document element has the ID "top", it makes the browser jump to the start of the document:

```
location = "#top";                    // Jump to the top of the document
```

The URL decomposition properties of the Location object are writable, and setting them changes the location URL and also causes the browser to load a new document (or, in the case of the `hash` property, to navigate within the current document):

```
location.search = "?page=" + (pagenum+1);  // load the next page
```

14.3 Browsing History

The `history` property of the Window object refers to the History object for the window. The History object models the browsing history of a window as a list of documents and document states. The `length` property of the History object specifies the number of elements in the browsing history list, but for security reasons scripts are not allowed to access the stored URLs. (If they could, any scripts could snoop through your browsing history.)

The History object has `back()` and `forward()` methods that behave like the browser's Back and Forward buttons do: they make the browser go backward or forward one step in its browsing history. A third method, `go()`, takes an integer argument and can skip any number of pages forward (for positive arguments) or backward (for negative arguments) in the history list:

```
history.go(-2);   // Go back 2, like clicking the Back button twice
```

If a window contains child windows (such as `<iframe>` elements—see §14.8.2), the browsing histories of the child windows are chronologically interleaved with the history of the main window. This means that calling `history.back()` (for example) on the main window may cause one of the child windows to navigate back to a previously displayed document but leave the main window in its current state.

Modern web applications can dynamically alter their own content without loading a new document (see Chapters 15 and 18, for example). Applications that do this may want to allow the user to use the Back and Forward buttons to navigate between these dynamically created application states. HTML5 standardizes two techniques for doing this, and they are described in §22.2.

History management before HTML5 is a more complex problem. An application that manages its own history must be able to create a new entry in the window browsing history, associate its state information with that history entry, determine when the user

has used the Back button to move to a different history entry, get the state information associated with that entry, and re-create the previous state of the application. One approach uses a hidden `<iframe>` to save state information and create entries in the browser's history. In order to create a new history entry, you dynamically write a new document into this hidden frame using the `open()` and `write()` methods of the Document object (see §15.10.2). The document content should include whatever state information is required to re-create the application state. When the user clicks the Back button, the content of the hidden frame will change. Before HTML5, no events are generated to notify you of this change, however, so in order to detect that the user has clicked Back you might use `setInterval()` (§14.1) to check the hidden frame two or three times a second to see if it has changed.

In practice, developers who need this kind of pre-HTML5 history management usually rely on a prebuilt solution. Many JavaScript frameworks include one. There is a history plug-in for jQuery, for example, and standalone history management libraries are also available. RSH (Really Simple History) is one popular example. You can find it at *http://code.google.com/p/reallysimplehistory/*. §22.2 explains how to do history management with HTML5.

14.4 Browser and Screen Information

Scripts sometimes need to obtain information about the web browser in which they are running or the desktop on which the browser appears. This section describes the `navigator` and `screen` properties of the Window object. Those properties refer to Navigator and Screen objects, respectively, and these objects provide information that allows a script to customize its behavior based on its environment.

14.4.1 The Navigator Object

The `navigator` property of a Window object refers to a Navigator object that contains browser vendor and version number information. The Navigator object is named after the early Navigator browser from Netscape, but it is also supported by all other browsers. (IE also supports `clientInformation` as a vendor-neutral synonym for `navigator`. Unfortunately, other browsers have not adopted this more sensibly named property.)

In the past, the Navigator object was commonly used by scripts to determine if they were running in Internet Explorer or Netscape. This "browser-sniffing" approach is problematic because it requires constant tweaking as new browsers and new versions of existing browsers are introduced. Today, feature testing (see §13.4.3) is preferred: rather than making assumptions about particular browser versions and their features, you simply test for the feature (i.e., the method or property) you need.

Browser sniffing is sometimes still valuable, however, such as when you need to work around a specific bug that exists in a specific version of a specific browser. The Navigator object has four properties that provide information about the browser that is running, and you can use these properties for browser sniffing:

`appName`
: The full name of the web browser. In IE, this is "Microsoft Internet Explorer". In Firefox, this property is "Netscape". For compatibility with existing browser-sniffing code, other browsers often report the name "Netscape" as well.

`appVersion`
: This property typically begins with a number and follows that with a detailed string that contains browser vendor and version information. The number at the start of this string is often 4.0 or 5.0 to indicate generic compatibility with fourth- and fifth-generation browsers. There is no standard format for the `appVersion` string, so parsing it in a browser-independent way isn't possible.

`userAgent`
: The string that the browser sends in its `USER-AGENT` HTTP header. This property typically contains all the information in `appVersion` and may contain additional details as well. Like `appVersion`, there is no standard format. Since this property contains the most information, browser-sniffing code typically uses it.

`platform`
: A string that identifies the operating system (and possibly the hardware) on which the browser is running.

The complexity of the Navigator properties demonstrates the futility of the browser-sniffing approach to client-side compatibility. In the early days of the Web, lots of browser-specific code was written that tested properties like `navigator.appName`. As new browsers were written, vendors discovered that in order to correctly display existing websites, they had to set the `appName` property to "Netscape". A similar process caused the number at the start of the `appVersion` to lose meaning, and today browser-sniffing code must rely on the `navigator.userAgent` string and is more complicated than it once was. Example 14-3 shows how to use regular expressions (from jQuery) to extract the browser name and version number from `navigator.userAgent`.

Example 14-3. Browser sniffing using navigator.userAgent

```
// Define browser.name and browser.version for client sniffing, using code
// derived from jQuery 1.4.1. Both the name and number are strings, and both
// may differ from the public browser name and version. Detected names are:
//
//   "webkit": Safari or Chrome; version is WebKit build number
//   "opera": the Opera browser; version is the public version number
//   "mozilla": Firefox or other gecko-based browsers; version is Gecko version
//   "msie": IE; version is public version number
//
// Firefox 3.6, for example, returns: { name: "mozilla", version: "1.9.2" }.
var browser = (function() {
    var s = navigator.userAgent.toLowerCase();
    var match = /(webkit)[ \/]([\w.]+)/.exec(s) ||
    /(opera)(?:.*version)?[ \/]([\w.]+)/.exec(s) ||
    /(msie) ([\w.]+)/.exec(s) ||
    !/compatible/.test(s) && /(mozilla)(?:.*? rv:([\w.]+))?/.exec(s) ||
    [];
```

```
    return { name: match[1] || "", version: match[2] || "0" };
}());
```

In addition to its browser vendor and version information properties, the Navigator object has some miscellaneous properties and methods. The standardized and widely implemented nonstandard properties include:

`onLine`
: The `navigator.onLine` property (if it exists) specifies whether the browser is currently connected to the network. Applications may want to save state locally (using techniques from Chapter 20) while they are offline.

`geolocation`
: A Geolocation object that defines an API for determining the user's geographical location. See §22.1 for details.

`javaEnabled()`
: A nonstandard method that should return `true` if the browser can run Java applets.

`cookiesEnabled()`
: A nonstandard method that should return `true` if the browser can store persistent cookies. May not return the correct value if cookies are configured on a site-by-site basis.

14.4.2 The Screen Object

The `screen` property of a Window object refers to a Screen object that provides information about the size of the user's display and the number of colors available on it. The `width` and `height` properties specify the size of the display in pixels. The `availWidth` and `availHeight` properties specify the display size that is actually available; they exclude the space required by features such as a desktop taskbar. The `colorDepth` property specifies the bits-per-pixel value of the screen. Typical values are 16, 24, and 32.

The `window.screen` property and the Screen object to which it refers are both nonstandard but widely implemented. You might use the Screen object to determine whether your web app is running in a small form factor device such as a netbook computer. If screen space is limited, you might choose to use smaller fonts and images, for example.

14.5 Dialog Boxes

The Window object provides three methods for displaying simple dialog boxes to the user. `alert()` displays a message to the user and waits for the user to dismiss the dialog. `confirm()` displays a message, waits for the user to click an OK or Cancel button and returns a boolean value. And `prompt()` displays a message, waits for the user to enter a string, and returns that string. The following code uses all three methods:

```
do {
    var name = prompt("What is your name?");                    // Get a string
    var correct = confirm("You entered '" + name + "'.\n" +     // Get a boolean
```

```
                        "Click Okay to proceed or Cancel to re-enter.");
} while(!correct)
alert("Hello, " + name);  // Display a plain message
```

Although the `alert()`, `confirm()`, and `prompt()` methods are very easy to use, good design dictates that you use them sparingly, if at all. Dialog boxes like these are not a common feature on the Web, and most users will find the dialog boxes produced by these methods disruptive to their browsing experience. The only common use for these methods today is debugging: JavaScript programmers sometimes insert `alert()` methods in code that is not working in an attempt to diagnose the problem.

Note that the messages displayed by `alert()`, `confirm()`, and `prompt()` are plain text, not HTML-formatted text. You can format these dialog boxes only with spaces, newlines, and punctuation characters.

The `confirm()` and `prompt()` methods *block*—that is, these methods do not return until the user dismisses the dialog boxes they display. This means that when you pop up one of these boxes, your code stops running, and the currently loading document, if any, stops loading until the user responds with the requested input. In most browsers, the `alert()` method also blocks and waits for the user to dismiss the dialog box, but this is not required. For complete details on these methods, see `Window.alert`, `Window.confirm`, and `Window.prompt` in Part IV.

In addition to the Window methods `alert()`, `confirm()`, and `prompt()`, a more complicated method, `showModalDialog()`, displays a modal dialog box containing HTML-formatted content and allows arguments to be passed to, and a value returned from, the dialog. `showModalDialog()` displays a modal dialog in a browser window of its own. The first argument is the URL that specifies the HTML content of the dialog box. The second argument is an arbitrary value (arrays and objects are allowed) that will be made available to scripts in the dialog as the value of the `window.dialogArguments` property. The third argument is a nonstandard list of semicolon-separated name=value pairs that, if supported, may configure the size or other attributes of the dialog. Use "dialogwidth" and "dialogheight" to set the size of the dialog window, and use "resizable=yes" to allow the user to resize the window.

The window displayed by this method is modal, and the call to `showModalDialog()` does not return until the window is closed. When the window closes, the value of the `window.returnValue` property becomes the return value of the method call. The HTML content of the dialog must typically include an OK button that sets `returnValue`, if desired, and calls `window.close()` (see §14.8.1.1).

Example 14-4 is an HTML file suitable for use with `showModalDialog()`. The comment at the top of the code includes a sample invocation of `showModalDialog()`, and Figure 14-1 shows the dialog created by the sample call. Note that most of the text that appears in the dialog comes from the second argument to `showModalDialog()`, rather than being hard-coded in the HTML.

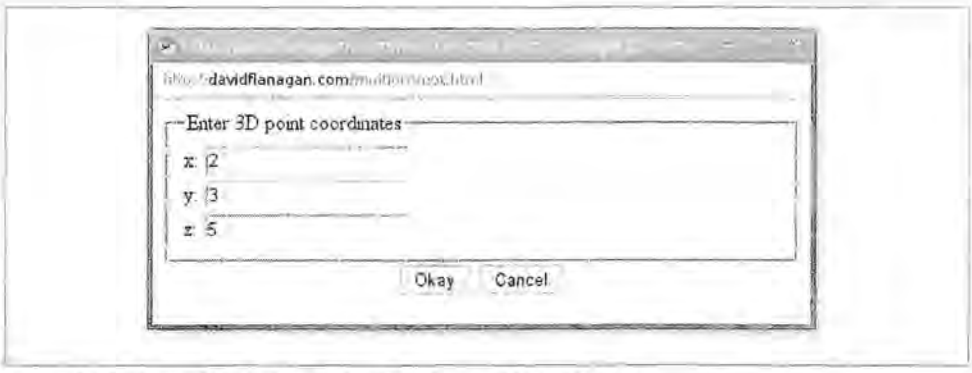


Figure 14-1. An HTML dialog displayed with showModalDialog()

Example 14-4. An HTML file for use with showModalDialog()

```
<!--
  This is not a stand-alone HTML file. It must be invoked by showModalDialog().
  It expects window.dialogArguments to be an array of strings.
  The first element of the array is displayed at the top of the dialog.
  Each remaining element is a label for a single-line text input field.
  Returns an array of input field values when the user clicks Okay.
  Use this file with code like this:

  var p = showModalDialog("multiprompt.html",
                          ["Enter 3D point coordinates", "x", "y", "z"],
                          "dialogwidth:400; dialogheight:300; resizable:yes");
  -->
<form>
<fieldset id="fields"></fieldset> <!-- Dialog body filled in by script below -->
<div style="text-align:center">   <!-- Buttons to dismiss the dialog -->
<button onclick="okay()">Okay</button>     <!-- Set return value and close -->
<button onclick="cancel()">Cancel</button> <!-- Close with no return value -->
</div>
<script>
// Create the HTML for the dialog body and display it in the fieldset
var args = dialogArguments;
var text = "<legend>" + args[0] + "</legend>";
for(var i = 1; i < args.length; i++)
    text += "<label>" + args[i] + ": <input id='f" + i + "'></label><br>";
document.getElementById("fields").innerHTML = text;

// Close the dialog without setting a return value
function cancel() { window.close(); }

// Read the input field values and set a return value, then close
function okay() {
    window.returnValue = [];              // Return an array
    for(var i = 1; i < args.length; i++) // Set elements from input fields
        window.returnValue[i-1] = document.getElementById("f" + i).value;
    window.close();  // Close the dialog. This makes showModalDialog() return.
}
</script>
</form>
```

14.6 Error Handling

The onerror property of a Window object is an event handler that is invoked when an uncaught exception propagates all the way up the call stack and an error message is about to be displayed in the browser's JavaScript console. If you assign a function to this property, the function is invoked whenever a JavaScript error occurs in that window: the function you assign becomes an error handler for the window.

For historical reasons, the onerror event handler of the Window object is invoked with three string arguments rather than with the one event object that is normally passed. (Other client-side objects have onerror handlers to handle different error conditions, but these are all regular event handlers that are passed a single event object.) The first argument to window.onerror is a message describing the error. The second argument is a string that contains the URL of the JavaScript code that caused the error. The third argument is the line number within the document where the error occurred.

In addition to those three arguments, the return value of the onerror handler is significant. If the onerror handler returns false, it tells the browser that the handler has handled the error and that no further action is necessary—in other words, the browser should not display its own error message. Unfortunately, for historical reasons, an error handler in Firefox must return true to indicate that it has handled the error.

The onerror handler is a holdover from the early days of JavaScript, when the core language did not include the try/catch exception handling statement. It is rarely used in modern code. During development, however, you might define an error handler like this to explicitly notify you when an error occurs:

```
// Display error messages in a dialog box, but never more than 3
window.onerror = function(msg, url, line) {
    if (onerror.num++ < onerror.max) {
        alert("ERROR: " + msg + "\n" + url + ":" + line);
        return true;
    }
}
onerror.max = 3;
onerror.num = 0;
```

14.7 Document Elements As Window Properties

If you name an element in your HTML document using the id attribute, and if the Window object does not already have a property by that name, the Window object is given a nonenumerable property whose name is the value of the id attribute and whose value is the HTMLElement object that represents that document element.

As we've already noted, the Window object serves as the global object at the top of the scope chain in client-side JavaScript, so this means that the id attributes you use in your HTML documents become global variables accessible to your scripts. If your document

includes the element <button id="okay"/>, you can refer to that element using the global variable okay.

There is an important caveat, however: this doesn't happen if the Window object already has a property by that name. Elements with the ids "history", "location," or "navigator", for example, won't appear as global variables, because those IDs are already in use. Similarly, if your HTML document includes an element whose id is "x" and you also declare and assign a value to the global variable x in your code, the explicitly declared variable will hide the implicit element variable. If the variable is declared in a script that appears before the named element, its existence will prevent the element from getting a window property of its own. And if the variable is declared in a script that appears after the named element, your explicit assignment to the variable overwrites the implicit value of the property.

In §15.2, you'll learn that you can look up document elements by the value of their HTML id attribute using the document.getElementById() method. Consider this example:

```
var ui = ["input","prompt","heading"];     // An array of element ids
ui.forEach(function(id) {                  // For each id look up the element
    ui[id] = document.getElementById(id); // and store it in a property
});
```

After running this code, ui.input, ui.prompt, and ui.heading refer to document elements. A script could use the global variables input and heading instead of ui.input and ui.heading. But recall from §14.5 that the Window object has a method named prompt(), so a script cannot use the global variable prompt instead of ui.prompt.

The implicit use of element IDs as global variables is a historical quirk of web browser evolution. It is required for backward compatibility with existing web pages, but its use is not recommended—any time a browser vendor defines a new property of the Window object it breaks any code that uses an implicit definition of that property name. Instead, use document.getElementById() to look up elements explicitly. The use of this method seems less onerous if we give it a simpler name:

```
var $ = function(id) { return document.getElementById(id); };
ui.prompt = $("prompt");
```

Many client-side libraries define a $ function that looks up elements by ID like this. (We'll see in Chapter 19 that jQuery's $ function is a general-purpose element selection method that returns one or more elements based on their ID, tag name, class attribute, or other criteria.)

Any HTML element with an id attribute will become the value of a global variable, assuming the ID is not already used by the Window object. The following HTML elements also behave this way when given a name attribute:

```
<a> <applet> <area> <embed> <form> <frame> <frameset> <iframe> <img> <object>
```

The id element is required to be unique within a document: two elements cannot have the same id. This is not true for the name attribute, however. If more than one of the

elements above has the same name attribute (or if one element has a name attribute, and another element has an id with the same value), the implicit global variable with that name will refer to an array-like object that holds each of the named elements.

There is a special case for <iframe> elements with a name or id attribute. The implicitly created variable for these elements refers not to the Element object that represents the element itself, but to the Window object that represents the nested browser frame created by the <iframe> element. We'll talk about this again in §14.8.2.

14.8 Multiple Windows and Frames

A single web browser window on your desktop may contain several tabs. Each tab is an independent *browsing context*. Each has its own Window object, and each is isolated from all the others. The scripts running in one tab usually have no way of even knowing that the other tabs exist, much less of interacting with their Window objects or manipulating their document content. If you use a web browser that does not support tabs, or if you have tabs turned off, you may have many web browser windows open on your desktop at one time. As with tabs, each desktop window has its own Window object, and each is usually independent of and isolated from all of the others.

But windows are not always isolated from one another. A script in one window or tab can open new windows or tabs, and when a script does this, the windows can interact with one another and with one another's documents (subject to the constraints of the same-origin policy of §13.6.2). §14.8.1 has more about opening and closing windows.

HTML documents may contain nested documents using an <iframe> element. An <iframe> creates a nested browsing context represented by a Window object of its own. The deprecated <frameset> and <frame> elements also create nested browsing contexts, and each <frame> is represented by a Window. Client-side JavaScript makes very little distinction between windows, tabs, iframes, and frames: they are all browsing contexts, and to JavaScript, they are all Window objects. Nested browsing contexts are not isolated from one another the way independent tabs usually are. A script running in one frame can always see its ancestor and descendant frames, though the same-origin policy may prevent the script from inspecting the documents in those frames. Nested frames are the topic of §14.8.2.

Since the Window is the global object of client-side JavaScript, each window or frame has a separate JavaScript execution context. Nevertheless, JavaScript code in one window can, subject to same-origin constraints, use the objects, properties, and methods defined in other windows. This is discussed in more detail in §14.8.3. When the same-origin policy prevents the scripts in two distinct windows from interacting directly, HTML5 provides an event-based message passing API for indirect communication. You can read about it in §22.3.

14.8.1 Opening and Closing Windows

You can open a new web browser window (or tab; this is usually a browser configuration option) with the `open()` method of the Window object. `Window.open()` loads a specified URL into a new or existing window and returns the Window object that represents that window. It takes four optional arguments:

The first argument to `open()` is the URL of the document to display in the new window. If this argument is omitted (or is the empty string), the special blank-page URL `about:blank` is used.

The second argument to `open()` is a string that specifies a window name. If a window by that name already exists (and if the script is allowed to navigate that window), that existing window is used. Otherwise a new window is created and is assigned the specified name. If this argument is omitted, the special name "_blank" is used: it opens a new, unnamed window.

Note that scripts cannot simply guess window names and take over the windows in use by other web applications: they can only name existing windows that they are "allowed to navigate" (the term is from the HTML5 specification). Loosely, a script can specify an existing window by name only if that window contains a document from the same origin or if the script opened that window (or recursively opened a window that opened that window). Also, if one window is a frame nested within the other, a script in either one can navigate the other. In this case, the reserved names "_top" (the top-level ancestor window) and "_parent" (the immediate parent window) can be useful.

Window Names

The name of a window is important because it allows the `open()` method to refer to existing windows, and also because it can be used as the value of the HTML `target` attribute on `<a>` and `<form>` elements to indicate that the linked document (or the result of submitting the form) should be displayed in the named window. The `target` attribute on these elements can also be set to "_blank", "_parent", or "_top" to direct the linked document into a new blank window, the parent window or frame, or the top-level window.

The `name` property of a Window object holds its name, if it has one. This property is writable, and scripts can set it as desired. If a name (other than "_blank") is passed to `Window.open()`, the window created by that call will have the specified name as the initial value of its `name` property. If an `<iframe>` element has a `name` attribute, the Window object that represents that frame will use that `name` attribute as the initial value of the `name` property.

The third optional argument to `open()` is a comma-separated list of size and features attributes for the new window to be opened. If you omit this argument, the new window is given a default size and has a full set of UI components: a menu bar, status line, toolbar, and so on. In tabbed browsers, this usually results in the creation of a new tab.

On the other hand, if you specify this argument, you can explicitly specify the size of the window and the set of features it includes. (Explicitly specifying a size is likely to result in the creation of a new window rather than a tab.) For example, to open a small but resizable browser window with a status bar but no menu bar, toolbar, or location bar, you might write:

```
var w = window.open("smallwin.html", "smallwin",
                    "width=400,height=350,status=yes,resizable=yes");
```

This third argument is nonstandard and the HTML5 specification insists that browsers be able to ignore it. See `Window.open()` in the reference section for more details on what you can specify in this argument. Note that when you specify this third argument, any features you do not explicitly specify are omitted. For various security reasons, browsers include restrictions on the features you can specify. You are typically not allowed to specify a window that is too small or is positioned offscreen, for example, and some browsers will not allow you to create a window without a status line.

The fourth argument to `open()` is useful only when the second argument names an existing window. This fourth argument is a boolean value that indicates whether the URL specified as the first argument should replace the current entry in the window's browsing history (`true`) or create a new entry in the window's browsing history (`false`). Omitting this argument is the same as passing `false`.

The return value of the `open()` method is the Window object that represents the named or newly created window. You can use this Window object in your JavaScript code to refer to the new window, just as you use the implicit Window object `window` to refer to the window within which your code is running:

```
var w = window.open();                          // Open a new, blank window.
w.alert("About to visit http://example.com");   // Call its alert() method
w.location = "http://example.com";              // Set its location property
```

In windows created with the `window.open()` method, the `opener` property refers back to the Window object of the script that opened it. In other windows, `opener` is `null`:

```
w.opener !== null;      // True for any window w created by open()
w.open().opener === w;  // True for any window w
```

`Window.open()` is the method by which advertisements are made to "pop up" or "pop under" while you browse the Web. Because of this flood of annoying pop ups, most web browsers have now instituted some kind of pop up–blocking system. Typically, calls to the `open()` method are successful only if they occur in response to a user action such as clicking on a button or a link. JavaScript code that tries to open a pop-up window when the browser first loads (or unloads) a page will usually fail. Testing the lines of code shown above by pasting them into the JavaScript console of your browser may also fail for the same reason.

14.8.1.1 Closing windows

Just as the open() method opens a new window, the close() method closes one. If you create a Window object w, you can close it with:

```
w.close();
```

JavaScript code running within that window itself can close it with:

```
window.close();
```

Note the explicit use of the window identifier to distinguish the close() method of the Window object from the close() method of the Document object—this is important if you're calling close() from an event handler.

Most browsers allow you to automatically close only those windows that your own JavaScript code has created. If you attempt to close any other window, the request either fails or the user is presented with a dialog box that asks him to allow (or cancel) that request to close the window. The close() method of a Window object that represents a frame rather than a top-level window or tab does nothing: it is not possible to close a frame (instead you'd delete the <iframe> from its containing document).

A Window object continues to exist after the window it represents has been closed. A window that has been closed will have a closed property of true, its document will be null, and its methods will typically no longer work.

14.8.2 Relationships Between Frames

As we saw above, the open() method of a Window object returns a new Window object that has an opener property that refers back to the original window. In this way, the two windows can refer to each other, and each can read properties and invoke methods of the other. A similar thing is possible with frames. Code running in a window or frame can refer to the containing window or frame and to nested child frames using the properties described below.

You already know that the JavaScript code in any window or frame can refer to its own Window object as window or as self. A frame can refer to the Window object of the window or frame that contains it using the parent property:

```
parent.history.back();
```

A Window object that represents a top-level window or tab has no container, and its parent property simply refers to the window itself:

```
parent == self;  // For any top-level window
```

If a frame is contained within another frame that is contained within a top-level window, that frame can refer to the top-level window as parent.parent. The top property is a general-case shortcut, however: no matter how deeply a frame is nested, its top property refers to the top-level containing window. If a Window object represents a top-level window, top simply refers to that window itself. For frames that are direct children of a top-level window, the top property is the same as the parent property.

The parent and top properties allow a script to refer to its frame's ancestors. There is more than one way to refer to the descendant frames of a window or frame. Frames are created with <iframe> elements. You can obtain an Element object that represents an <iframe> just as you would do for any other element. Suppose your document contains <iframe id="f1">. Then, the Element object that represents this iframe is:

```
var iframeElement = document.getElementById("f1");
```

<iframe> elements have a contentWindow property that refers to the Window object of the frame, so the Window object for this frame is:

```
var childFrame = document.getElementById("f1").contentWindow;
```

You can go in the reverse direction—from the Window that represents a frame to the <iframe> Element that contains the frame—with the frameElement property of the Window. Window objects that represent top-level windows rather than frames have a null frameElement property:

```
var elt = document.getElementById("f1");
var win = elt.contentWindow;
win.frameElement === elt     // Always true for frames
window.frameElement === null // For toplevel windows
```

It is not usually necessary to use the getElementById() method and the contentWindow property to obtain references to the child frames of a window, however. Every Window object has a frames property that refers to the child frames contained within the window or frame. The frames property refers to an array-like object that can be indexed numerically or by frame name. To refer to the first child frame of a window, you can use frames[0]. To refer to the third child frame of the second child, you can use frames[1].frames[2]. Code running in a frame might refer to a sibling frame as parent.frames[1]. Note that the elements of the frames[] array are Window objects, not <iframe> elements.

If you specify the name or id attribute of an <iframe> element, that frame can be indexed by name as well as by number. A frame named "f1" would be frames["f1"] or frames.f1, for example.

Recall from §14.7 that the names or IDs of <iframe> and other elements are automatically used as properties of the Window object, and that <iframe> elements are treated differently than other elements: for frames, the value of these automatically created properties refer to a Window object rather than an Element object. What this means is that we can refer to a frame named "f1" as f1 instead of as frames.f1. In fact, HTML5 specifies that the frames property is a self-referential property, just like window and self, and that it is the Window object itself that acts like an array of frames. This means that we can refer to the first child frame as window[0], and we can query the number of frames with window.length or just length. It is usually clearer, and still traditional, to use frames instead of window here, however. Note that current browsers do not all make frame==window, but those that do not make them equal do allow child frames to be indexed by number or by name through either object.

You can use the name or id attribute of an <iframe> element to give the frame a name that can be used in JavaScript code. If you use the name attribute, however, the name you specify also becomes the value of the name property of the Window that represents the frame. A name specified in this way can be used as the target attribute of a link, and it can be used as the second argument to window.open().

14.8.3 JavaScript in Interacting Windows

Each window or frame is its own JavaScript execution context with a Window as its global object. But if code in one window or frame can refer to another window or frame (and if the same-origin policy does not prevent it), the scripts in one window or frame can interact with the scripts in the other.

Imagine a web page with two <iframe> elements named "A" and "B", and suppose that those frames contain documents from the same server and that those documents contain interacting scripts. The script in frame A might define a variable i:

```
var i = 3;
```

That variable is nothing more than a property of the global object—a property of the Window object. Code in frame A can refer to the variable with the identifier i, or it can explicitly reference it through the window object:

```
window.i
```

Since the script in frame B can refer to the Window object for frame A, it can also refer to the properties of that window object:

```
parent.A.i = 4;     // Change the value of a variable in frame A
```

Recall that the function keyword that defines functions creates a variable just like the var keyword does. If a script in frame B declares a (non-nested) function f, that function is a global variable in frame B, and code in frame B can invoke f as f(). Code in frame A, however, must refer to f as a property of the Window object of frame B:

```
parent.B.f();       // Invoke a function defined in frame B
```

If the code in frame A needs to use this function frequently, it might assign the function to a variable of frame A so that it can more conveniently refer to the function:

```
var f = parent.B.f;
```

Now code in frame A can invoke the function as f(), just as code in frame B does.

When you share functions between frames or windows like this, it is important to keep the rules of lexical scoping in mind. A function is executed in the scope in which it was defined, not in the scope from which it is invoked. Thus, if the function f above refers to global variables, these variables are looked up as properties of frame B, even when the function is invoked from frame A.

Remember that constructors are also functions, so when you define a class (see Chapter 9) with a constructor function and an associated prototype object, that class is

defined only within a single window. Suppose that the window that contains frames A and B includes the Set class from Example 9-6.

Scripts within that top-level window can create new Set objects like this:

```
var s = new Set();
```

But scripts in either of the frames must explicitly refer to the `Set()` constructor as a property of the parent window:

```
var s = new parent.Set();
```

Alternatively, code in either frame can define its own variable to refer more conveniently to the constructor function:

```
var Set = top.Set();
var s = new Set();
```

Unlike user-defined classes, the built-in classes like String, Date, and RegExp are automatically predefined in all windows. This means, however, that each window has an independent copy of the constructor and an independent copy of the prototype object. For example, each window has its own copy of the `String()` constructor and the `String.prototype` object. So if you write a new method for manipulating JavaScript strings and then make it a method of the String class by assigning it to the `String.prototype` object in the current window, all strings created by code in that window can use the new method. However, the new method is not accessible to strings created in other windows.

The fact that each Window has its own prototype objects means that the `instanceof` operator does not work across windows. `instanceof` will evaluate to `false`, for example, when used to compare a string from frame B to the `String()` constructor from frame A. §7.10 describes the related difficulty of determining the type of arrays across windows.

The WindowProxy Object

We've noted repeatedly that the Window object is the global object of client-side JavaScript. Technically, however, this is not true. Each time a web browser loads new content into a window or a frame, it must start with a fresh JavaScript execution context, including a newly created global object. But when multiple windows or frames are in use, it is critical that the Window object that refers to a frame or window continue to be a valid reference even if that frame or window loads a new document.

So client-side JavaScript has two important objects. The client-side global object is the top of the scope chain and is where global variables and functions are defined. This global object is, in fact, replaced whenever the window or frame loads new content. The object we have been calling the Window object is not actually the global object, but a proxy for it. Whenever you query or set a property of the Window object, that object queries or sets the same property on the *current* global object of the window or frame. The HTML5 specification calls this proxy object WindowProxy, but we will continue to use the term *Window object* in this book.

Because of its proxying behavior, the proxy object behaves just like the true global object, except that it has a longer lifetime. If you could compare the two objects, it would be difficult to distinguish them. In fact, however, there is no way to refer to the true client-side global object. The global object is at the top of the scope chain, but the `window`, `self`, `top`, `parent`, and `frames` properties all return proxy objects. The `window.open()` method returns a proxy object. Even the value of the `this` keyword within a top-level function is a proxy object rather than the true global object.[1]

1. This last point is a minor violation of the ES3 and ES5 specifications, but it is necessary to support the multiple execution contexts of client-side JavaScript.

CHAPTER 15

Scripting Documents

Client-side JavaScript exists to turn static HTML documents into interactive web applications. Scripting the content of web pages is the central purpose of JavaScript. This chapter—one of the most important in the book—explains how to do this.

Chapters 13 and 14 explained that every web browser window, tab, and frame is represented by a Window object. Every Window object has a `document` property that refers to a Document object. The Document object represents the content of the window, and it is the subject of this chapter. The Document object does not stand alone, however. It is the central object in a larger API, known as the *Document Object Model*, or DOM, for representing and manipulating document content.

This chapter begins by explaining the basic architecture of the DOM. It then moves on to explain:

- How to query or *select* individual elements from a document.
- How to *traverse* a document as a tree of nodes, and how to find the ancestors, siblings, and descendants of any document element.
- How to query and set the attributes of document elements.
- How to query, set, and modify the content of a document.
- How to modify the structure of a document by creating, inserting, and deleting nodes.
- How to work with HTML forms.

The final section of the chapter covers miscellaneous document features, including the `referrer` property, the `write()` method, and techniques for querying the currently selected document text.

15.1 Overview of the DOM

The Document Object Model, or DOM, is the fundamental API for representing and manipulating the content of HTML and XML documents. The API is not particularly complicated, but there are a number of architectural details you need to understand.

First, you should understand that the nested elements of an HTML or XML document are represented in the DOM as a tree of objects. The tree representation of an HTML document contains nodes representing HTML tags or elements, such as <body> and <p>, and nodes representing strings of text. An HTML document may also contain nodes representing HTML comments. Consider the following simple HTML document:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
</html>
```

The DOM representation of this document is the tree pictured in Figure 15-1.

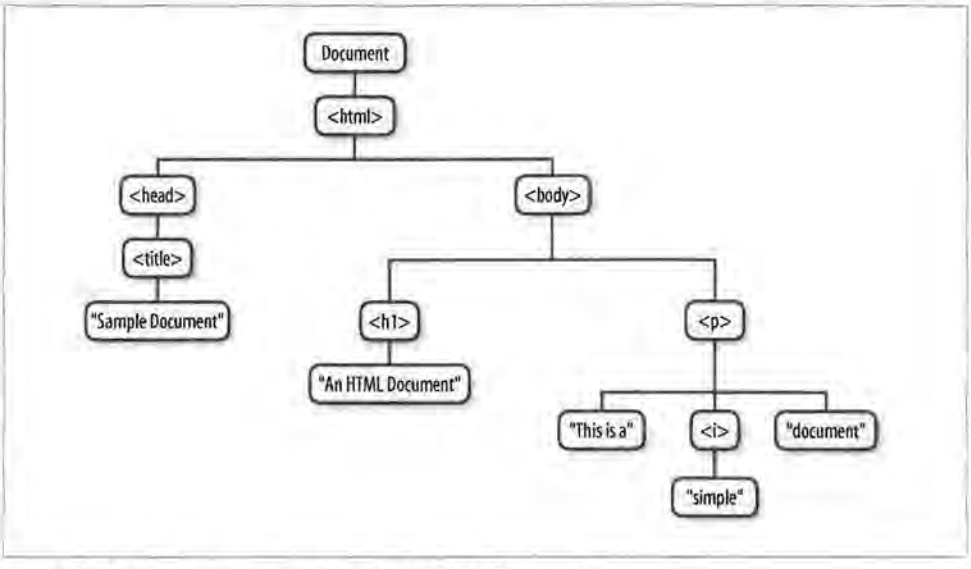


Figure 15-1. The tree representation of an HTML document

If you are not already familiar with tree structures in computer programming, it is helpful to know that they borrow terminology from family trees. The node directly above a node is the *parent* of that node. The nodes one level directly below another node are the *children* of that node. Nodes at the same level, and with the same parent, are *siblings*. The set of nodes any number of levels below another node are the *descendants* of that node. And the parent, grandparent, and all other nodes above a node are the *ancestors* of that node.

Each box in Figure 15-1 is a node of the document and is represented by a Node object. We'll talk about the properties and methods of Node in some of the sections that follow,

and you can look up those properties and methods under Node in Part IV. Note that the figure contains three different types of nodes. At the root of the tree is the Document node that represents the entire document. The nodes that represent HTML elements are Element nodes, and the nodes that represent text are Text nodes. Document, Element, and Text are subclasses of Node and have their own entries in the reference section. Document and Element are the two most important DOM classes, and much of this chapter is devoted to their properties and methods.

Node and its subtypes form the type hierarchy illustrated in Figure 15-2. Notice that there is a formal distinction between the generic Document and Element types, and the HTMLDocument and HTMLElement types. The Document type represents either an HTML or an XML document, and the Element class represents an element of such a document. The HTMLDocument and HTMLElement subclasses are specific to HTML documents and elements. In this book, we often use the generic class names Document and Element, even when referring to HTML documents. This is true in the reference section as well: the properties and methods of the HTMLDocument and the HTMLElement types are documented in the Document and Element reference pages.

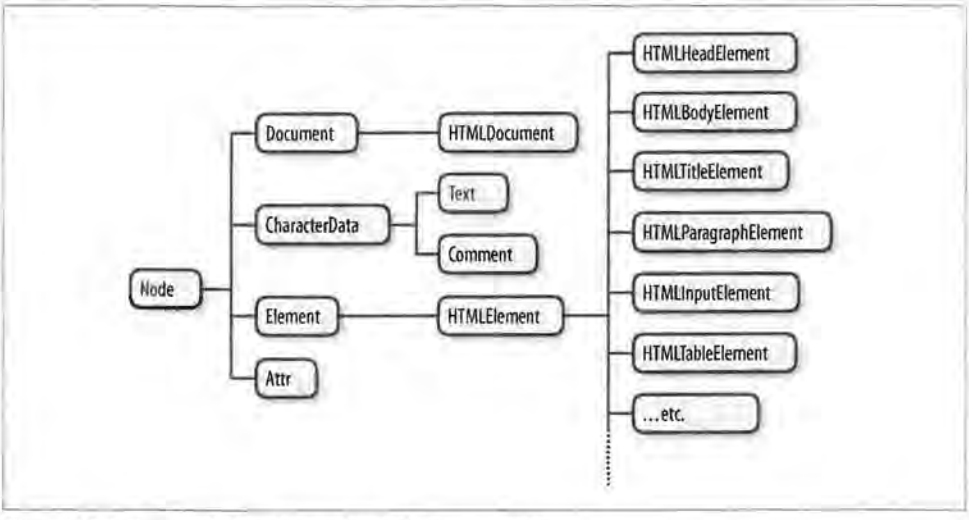


Figure 15-2. A partial class hierarchy of document nodes

It is also worth noting in Figure 15-2 that there are many subtypes of HTMLElement that represent specific types of HTML elements. Each defines JavaScript properties to mirror the HTML attributes of a specific element or group of elements (see §15.4.1). Some of these element-specific classes define additional properties or methods that go beyond simple mirroring of HTML syntax. These classes and their additional features are covered in the reference section.

Finally, note that Figure 15-2 shows some node types that haven't been mentioned so far. Comment nodes represent HTML or XML comments. Because comments are basically strings of text, these nodes are much like the Text nodes that represent the

displayed text of a document. CharacterData, the common ancestor of both Text and Comment, defines methods shared by both nodes. The Attr node type represents an XML or HTML attribute, but it is almost never used because the Element class defines methods for treating attributes as name/value pairs rather than document nodes. The DocumentFragment class (not pictured) is a kind of Node that never exists in an actual document: it represents a sequence of Nodes that do not have a common parent. DocumentFragments are useful for some document manipulations and are covered in §15.6.4. The DOM also defines infrequently used types to represent things like doctype declarations and XML processing instructions.

15.2 Selecting Document Elements

Most client-side JavaScript programs work by somehow manipulating one or more document elements. When these programs start, they can use the global variable `document` to refer to the Document object. In order to manipulate elements of the document, however, they must somehow obtain or *select* the Element objects that refer to those document elements. The DOM defines a number of ways to select elements; you can query a document for an element or elements:

- with a specified `id` attribute;
- with a specified `name` attribute;
- with the specified tag name;
- with the specified CSS class or classes; or
- matching the specified CSS selector

The subsections that follow explain each of these element selection techniques.

15.2.1 Selecting Elements By ID

Any HTML element can have an `id` attribute. The value of this attribute must be unique within the document—no two elements in the same document can have the same ID. You can select an element based on this unique ID with the `getElementById()` method of the Document object. We've already used this method in both Chapter 13 and Chapter 14:

```
var section1 = document.getElementById("section1");
```

This is the simplest and most commonly used way to select elements. If your script is going to manipulate a certain specific set of document elements, give those elements `id` attributes, and look up the Element objects using that ID. If you need to look up more than one element by ID, you might find the `getElements()` function of Example 15-1 useful.

Example 15-1. Looking up multiple elements by ID

```
/**
 * This function expects any number of string arguments. It treats each
 * argument as an element id and calls document.getElementById() for each.
 * Returns an object that maps ids to the corresponding Element object.
 * Throws an Error object if any of the ids is undefined.
 */
function getElements(/*ids...*/) {
    var elements = {};                          // Start with an empty map
    for(var i = 0; i < arguments.length; i++) { // For each argument
        var id = arguments[i];                  // Argument is an element id
        var elt = document.getElementById(id);  // Look up the Element
        if (elt == null)                        // If not defined,
            throw new Error("No element with id: " + id); // throw an error
        elements[id] = elt;                     // Map id to element
    }
    return elements;                            // Return id to element map
}
```

In versions of Internet Explorer prior to IE8, `getElementById()` does a case-insensitive match on element IDs and also returns elements that have a matching `name` attribute.

15.2.2 Selecting Elements by Name

The HTML `name` attribute was originally intended to assign names to form elements, and the value of this attribute is used when form data is submitted to a server. Like the `id` attribute, `name` assigns a name to an element. Unlike `id`, however, the value of a `name` attribute does not have to be unique: multiple elements may have the same name, and this is common in the case of radio buttons and checkboxes in forms. Also, unlike `id`, the `name` attribute is only valid on a handful of HTML elements, including forms, form elements, `<iframe>`, and `<img>` elements.

To select HTML elements based on the value of their `name` attributes, you can use the `getElementsByName()` method of the Document object:

```
var radiobuttons = document.getElementsByName("favorite_color");
```

`getElementsByName()` is defined by the HTMLDocument class, not the Document class, and so it is only available for HTML documents, not XML documents. It returns a NodeList object that behaves like a read-only array of Element objects. In IE, `getElementsByName()` will also return elements that have an `id` attribute with the specified value. For compatibility, you should be careful not to use the same string as both a name and an ID.

We saw in §14.7 that setting the `name` attribute of certain HTML elements automatically created properties with those names on the Window object. A similar thing is true for the Document object. Setting the `name` attribute of a `<form>`, `<img>`, `<iframe>`, `<applet>`, `<embed>`, or `<object>` element (but only `<object>` elements that do not have fallback objects within them) creates a property of the Document object whose name is the

value of the attribute (assuming, of course, that the document does not already have a property with that name).

If there is only a single element with a given name, the value of the automatically created document property is the element itself. If there is more than one element, then the value of the property is a NodeList object that acts as an array of elements. As we saw in §14.7, the document properties created for named <iframe> elements are special: instead of referring to the Element object, they refer to the frame's Window object.

What this means is that some elements can be selected by name simply by using the name as a Document property:

```
// Get the Element object for the <form name="shipping_address"> element
var form = document.shipping_address;
```

The reasons given in §14.7 for not using the automatically created window properties apply equally to these automatically created document properties. If you need to look up named elements, it is better to look them up explicitly with a call to `getElementsByName()`.

15.2.3 Selecting Elements by Type

You can select all HTML or XML elements of a specified type (or tag name) using the `getElementsByTagName()` method of the Document object. To obtain a read-only array-like object containing the Element objects for all elements in a document, for example, you might write:

```
var spans = document.getElementsByTagName("span");
```

Like `getElementsByName()`, `getElementsByTagName()` returns a NodeList object. (See the sidebar in this section for more on the NodeList class.) The elements of the returned NodeList are in document order, so you can select the first <p> element of a document like this:

```
var firstpara = document.getElementsByTagName("p")[0];
```

HTML tags are case-insensitive, and when `getElementsByTagName()` is used on an HTML document, it performs a case-insensitive tag name comparison. The `spans` variable above, for example, will include any elements that were written as <SPAN>.

You can obtain a NodeList that represents all elements in a document by passing the wildcard argument "*" to `getElementsByTagName()`.

The Element class also defines a `getElementsByTagName()` method. It works in the same way as the Document version, but it only selects elements that are descendants of the element on which it is invoked. So to find all elements inside the first <p> element of a document, you could write:

```
var firstpara = document.getElementsByTagName("p")[0];
var firstParaSpans = firstpara.getElementsByTagName("span");
```

For historical reasons, the HTMLDocument class defines shortcut properties to access certain kinds of nodes. The `images`, `forms`, and `links` properties, for example, refer to objects that behave like read-only arrays of `<img>`, `<form>`, and `<a>` elements (but only `<a>` tags that have an `href` attribute). These properties refer to HTMLCollection objects, which are much like NodeList objects, but they can additionally be indexed by element ID or name. Earlier, we saw how you could refer to a named `<form>` element with an expression like this:

```
document.shipping_address
```

With the `document.forms` property, you can also refer more specifically to the named (or ID'ed) form like this:

```
document.forms.shipping_address;
```

The HTMLDocument object also defines synonymous `embeds` and `plugins` properties that are HTMLCollections of `<embed>` elements. The `anchors` property is nonstandard but refers to `<a>` elements that have a `name` attribute rather than an `href` attribute. The `<scripts>` property is standardized by HTML5 to be an HTMLCollection of `<script>` elements, but it is not, at the time of this writing, universally implemented.

HTMLDocument also defines two properties that refer to special single elements rather than element collections. `document.body` is the `<body>` element of an HTML document, and `document.head` is the `<head>` element. These properties are always defined: if the document source does not explicitly include `<head>` and `<body>` elements, the browser creates them implicitly. The `documentElement` property of the Document class refers to the root element of the document. In HTML documents, this is always an `<html>` element.

NodeLists and HTMLCollections

`getElementsByName()` and `getElementsByTagName()` return NodeList objects, and properties like `document.images` and `document.forms` are HTMLCollection objects.

These objects are read-only array-like objects (see §7.11). They have `length` properties and can be indexed (for reading but not writing) like true arrays. You can iterate the contents of a NodeList or HTMLCollection with a standard loop like this:

```
for(var i = 0; i < document.images.length; i++)   // Loop through all images
    document.images[i].style.display = "none";    // ...and hide them.
```

You cannot invoke Array methods on NodeLists and HTMLCollections directly, but you can do so indirectly:

```
var content = Array.prototype.map.call(document.getElementsByTagName("p"),
                                       function(e) { return e.innerHTML; });
```

HTMLCollection objects may have additional named properties and can be indexed with strings as well as numbers.

For historical reasons, both NodeList and HTMLCollection objects can also be treated as functions: invoking them with a number or string argument is the same as indexing them with a number or string. Use of this quirk is discouraged.

Both the NodeList and HTMLCollection interfaces were designed with languages less dynamic than JavaScript in mind. Both define an `item()` method. It expects an integer and returns the element at that index. There is never any need to call this method in JavaScript because you can simply use array indexing instead. Similarly, HTMLCollection defines a `namedItem()` method that returns the value of a named property, but JavaScript programs can use array indexing or regular property access instead.

One of the most important and surprising features of NodeList and HTMLCollection is that they are not static snapshots of a historical document state but are generally *live* and the list of elements they contain can vary as the document changes. Suppose you call `getElementsByTagName('div')` on a document with no `<div>` elements. The return value is a NodeList with a `length` of 0. If you then insert a new `<div>` element into the document, that element automatically becomes a member of the NodeList, and the `length` property changes to 1.

Usually, the liveness of NodeLists and HTMLCollections is quite helpful. If you will be adding or removing elements from the document while iterating through a NodeList, however, you may want to make a static copy of the NodeList first:

```
var snapshot = Array.prototype.slice.call(nodelist, 0);
```

15.2.4 Selecting Elements by CSS Class

The `class` attribute of an HTML is a space-separated list of zero or more identifiers. It describes a way to define sets of related document elements: any elements that have the same identifier in their `class` attribute are part of the same set. `class` is a reserved word in JavaScript, so client-side JavaScript uses the `className` property to hold the value of the HTML `class` attribute. The `class` attribute is usually used in conjunction with a CSS stylesheet to apply the same presentation styles to all members of a set, and we'll see it again in Chapter 16. In addition, however, HTML5 defines a method, `getElements ByClassName()`, that allows us to select sets of document elements based on the identifiers in their `class` attribute.

Like `getElementsByTagName()`, `getElementsByClassName()` can be invoked on both HTML documents and HTML elements, and it returns a live NodeList containing all matching descendants of the document or element. `getElementsByClassName()` takes a single string argument, but the string may specify multiple space-separated identifiers. Only elements that include all of the specified identifiers in their `class` attribute are matched. The order of the identifiers does not matter. Note that both the `class` attribute and the `getElementsByClassName()` methods separate class identifiers with spaces, not with commas. Here are some examples of `getElementsByClassName()`:

```
// Find all elements that have "warning" in their class attribute
var warnings = document.getElementsByClassName("warning");
```

```
// Find all descendants of the element named "log" that have the class
// "error" and the class "fatal"
var log = document.getElementById("log");
var fatal = log.getElementsByClassName("fatal error");
```

Today's web browsers display HTML documents in "quirks mode" or "standards mode" depending on how strict the `<!DOCTYPE>` declaration at the start of the document is. Quirks mode exists for backward compatibility, and one of its quirks is that class identifiers in the `class` attribute and in CSS stylesheets are case-insensitive. `getElementsByClassName()` follows the matching algorithm used by stylesheets. If the document is rendered in quirks mode, the method performs a case-insensitive string comparison. Otherwise, the comparison is case sensitive.

At the time of this writing, `getElementsByClassName()` is implemented by all current browsers except IE8 and earlier. IE8 does support `querySelectorAll()`, described in the next section, and `getElementsByClassName()` can be implemented on top of that method.

15.2.5 Selecting Elements with CSS Selectors

CSS stylesheets have a very powerful syntax, known as *selectors*, for describing elements or sets of elements within a document. Full details of CSS selector syntax are beyond the scope of this book,[1] but some examples will demonstrate the basics. Elements can be described by ID, tag name, or class:

```
#nav                    // An element with id="nav"
div                     // Any <div> element
.warning                // Any element with "warning" in its class attribute
```

More generally, elements can be selected based on attribute values:

```
p[lang="fr"]            // A paragraph written in French: <p lang="fr">
*[name="x"]             // Any element with a name="x" attribute
```

These basic selectors can be combined:

```
span.fatal.error        // Any <span> with "fatal" and "error" in its class
span[lang="fr"].warning // Any warning in French
```

Selectors can also specify document structure:

```
#log span               // Any <span> descendant of the element with id="log"
#log>span               // Any <span> child of the element with id="log"
body>h1:first-child     // The first <h1> child of the <body>
```

Selectors can be combined to select multiple elements or multiple sets of elements:

```
div, #log               // All <div> elements plus the element with id="log"
```

As you can see, CSS selectors allow elements to be selected in all of the ways described above: by ID, by name, by tag name, and by class name. Along with the standardization of CSS3 selectors, another W3C standard, known as "Selectors API" defines JavaScript

1. CSS3 selectors are specified by *http://www.w3.org/TR/css3-selectors/*.

methods for obtaining the elements that match a given selector.[2] The key to this API is the Document method `querySelectorAll()`. It takes a single string argument containing a CSS selector and returns a NodeList that represents all elements in the document that match the selector. Unlike previously described element selection methods, the NodeList returned by `querySelectorAll()` is not live: it holds the elements that match the selector at the time the method was invoked, but it does not update as the document changes. If no elements match, `querySelectorAll()` returns an empty NodeList. If the selector string is invalid, `querySelectorAll()` throws an exception.

In addition to `querySelectorAll()`, the document object also defines `querySelector()`, which is like `querySelectorAll()`, but returns only the first (in document order) matching element or `null` if there is no matching element.

These two methods are also defined on Elements (and also on DocumentFragment nodes; see §15.6.4). When invoked on an element, the specified selector is matched against the entire document, and then the result set is filtered so that it only includes descendants of the specified element. This may seem counterintuitive, as it means that the selector string can include ancestors of the element against which it is matched.

Note that CSS defines `:first-line` and `:first-letter` pseudoelements. In CSS, these match portions of text nodes rather than actual elements. They will not match if used with `querySelectorAll()` or `querySelector()`. Also, many browsers will refuse to return matches for the `:link` and `:visited` pseudoclasses, as this could expose information about the user's browsing history.

All current browsers support `querySelector()` and `querySelectorAll()`. Note, however, that the specification of these methods does not require support for CSS3 selectors: browsers are encouraged to support the same set of selectors that they support in stylesheets. Current browsers other than IE support CSS3 selectors. IE7 and 8 support CSS2 selectors. (IE9 is expected to have CSS3 support.)

`querySelectorAll()` is the ultimate element selection method: it is a very powerful technique by which client-side JavaScript programs can select the document elements that they are going to manipulate. Fortunately, this use of CSS selectors is available even in browsers without native support for `querySelectorAll()`. The jQuery library (see Chapter 19) uses this kind of CSS selector-based query as its central programming paradigm. Web applications based on jQuery use a portable, cross-browser equivalent to `querySelectorAll()` named `$()`.

jQuery's CSS selector matching code has been factored out and released as a stand-alone library named Sizzle, which has been adopted by Dojo and other client-side libraries.[3] The advantage to using a library like Sizzle (or a library that uses Sizzle) is that

2. The Selectors API standard is not part of HTML5 but is closely affiliated with it. See *http://www.w3.org/TR/selectors-api/*.

3. A stand-alone version of Sizzle is available at *http://sizzlejs.com*.

selections work even on older browsers, and there is a baseline set of selectors that are guaranteed to work on all browsers.

15.2.6 document.all[]

Before the DOM was standardized, IE4 introduced the `document.all[]` collection that represented all elements (but not Text nodes) in the document. `document.all[]` has been replaced by standard methods like `getElementById()` and `getElementsByTagName()` and is now obsolete and should not be used. When introduced, however, it was revolutionary, and you may still see existing code that uses it in any of these ways:

```
document.all[0]            // The first element in the document
document.all["navbar"]     // Element (or elements) with id or name "navbar"
document.all.navbar        // Ditto
document.all.tags("div")   // All <div> elements in the document
document.all.tags("p")[0] // The first <p> in the document
```

15.3 Document Structure and Traversal

Once you have selected an Element from a Document, you sometimes need to find structurally related portions (parent, siblings, children) of the document. A Document can be conceptualized as a tree of Node objects, as illustrated in Figure 15-1. The Node type defines properties for traversing such a tree, which we'll cover in §15.3.1. Another API allows documents to be traversed as trees of Element objects. §15.3.2 covers this newer (and often easier-to-use) API.

15.3.1 Documents As Trees of Nodes

The Document object, its Element objects, and the Text objects that represent runs of text in the document are all Node objects. Node defines the following important properties:

`parentNode`
: The Node that is the parent of this one, or `null` for nodes like the Document object that have no parent.

`childNodes`
: A read-only array-like object (a NodeList) that is a live representation of a Node's child nodes.

`firstChild`, `lastChild`
: The first and last child nodes of a node, or `null` if the node has no children.

`nextSibling`, `previousSibling`
: The next and previous sibling node of a node. Two nodes with the same parent are siblings. Their order reflects the order in which they appear in the document. These properties connect nodes in a doubly linked list.

nodeType
: The kind of node this is. Document nodes have the value 9. Element nodes have the value 1. Text nodes have the value 3. Comments nodes are 8 and DocumentFragment nodes are 11.

nodeValue
: The textual content of a Text or Comment node.

nodeName
: The tag name of an Element, converted to uppercase.

Using these Node properties, the second child node of the first child of the Document can be referred to with expressions like these:

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Suppose the document in question is the following:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Then the second child of the first child is the <body> element. It has a nodeType of 1 and a nodeName of "BODY".

Note, however, that this API is extremely sensitive to variations in the document text. If the document is modified by inserting a single newline between the <html> and the <head> tag, for example, the Text node that represents that newline becomes the first child of the first child, and the second child is the <head> element instead of the <body> body.

15.3.2 Documents As Trees of Elements

When we are primarily interested in the Elements of a document instead of the text within them (and the whitespace between them), it is helpful to use an API that allows us to treat a document as a tree of Element objects, ignoring Text and Comment nodes that are also part of the document.

The first part of this API is the children property of Element objects. Like childNodes, this is a NodeList. Unlike childNodes, however, the children list contains only Element objects. The children property is nonstandard, but it works in all current browsers. IE has implemented it for a long time, and most other browsers have followed suit. The last major browser to adopt it was Firefox 3.5.

Note that Text and Comment nodes cannot have children, which means that the Node.parentNode property described above never returns a Text or Comment node. The parentNode of any Element will always be another Element, or, at the root of the tree, a Document or DocumentFragment.

The second part of an element-based document traversal API is Element properties that are analogs to the child and sibling properties of the Node object:

`firstElementChild`, `lastElementChild`
: Like `firstChild` and `lastChild`, but for Element children only.

`nextElementSibling`, `previousElementSibling`
: Like `nextSibling` and `previousSibling`, but for Element siblings only.

`childElementCount`
: The number of element children. Returns the same value as `children.length`.

These child and sibling properties are standardized and are implemented in all current browsers except IE.[4]

Because the API for element-by-element document traversal is not yet completely universal, you might want to define portable traversal functions like those in Example 15-2.

Example 15-2. Portable document traversal functions

```
/**
 * Return the nth ancestor of e, or null if there is no such ancestor
 * or if that ancestor is not an Element (a Document or DocumentFragment e.g.).
 * If n is 0 return e itself.  If n is 1 (or
 * omitted) return the parent.  If n is 2, return the grandparent, etc.
 */
function parent(e, n) {
    if (n === undefined) n = 1;
    while(n-- && e) e = e.parentNode;
    if (!e || e.nodeType !== 1) return null;
    return e;
}

/**
 * Return the nth sibling element of Element e.
 * If n is postive return the nth next sibling element.
 * If n is negative, return the -nth previous sibling element.
 * If n is zero, return e itself.
 */
function sibling(e,n) {
    while(e && n !== 0) {  // If e is not defined we just return it
        if (n > 0) {  // Find next element sibling
            if (e.nextElementSibling) e = e.nextElementSibling;
            else {
                for(e=e.nextSibling; e && e.nodeType !== 1; e=e.nextSibling)
                    /* empty loop */ ;
            }
            n--;
        }
        else {        // Find the previous element sibling
            if (e.previousElementSibing) e = e.previousElementSibling;
            else {
                for(e=e.previousSibling; e&&e.nodeType!==1; e=e.previousSibling)
                    /* empty loop */ ;
            }
            n++;
```

4. *http://www.w3.org/TR/ElementTraversal.*

```
        }
    }
    return e;
}

/**
 * Return the nth element child of e, or null if it doesn't have one.
 * Negative values of n count from the end. 0 means the first child, but
 * -1 means the last child, -2 means the second to last, and so on.
 */
function child(e, n) {
    if (e.children) {                    // If children array exists
        if (n < 0) n += e.children.length; // Convert negative n to array index
        if (n < 0) return null;          // If still negative, no child
        return e.children[n];            // Return specified child
    }

    // If e does not have a children array, find the first child and count
    // forward or find the last child and count backwards from there.
    if (n >= 0) { // n is non-negative: count forward from the first child
        // Find the first child element of e
        if (e.firstElementChild) e = e.firstElementChild;
        else {
            for(e = e.firstChild; e && e.nodeType !== 1; e = e.nextSibling)
                /* empty */;
        }
        return sibling(e, n); // Return the nth sibling of the first child
    }
    else { // n is negative, so count backwards from the end
        if (e.lastElementChild) e = e.lastElementChild;
        else {
            for(e = e.lastChild; e && e.nodeType !== 1; e=e.previousSibling)
                /* empty */;
        }
        return sibling(e, n+1); // +1 to convert child -1 to sib 0 of last
    }
}
```

Defining Custom Element Methods

All current browsers (including IE8, but not IE7 and before) implement the DOM so that types like Element and HTMLDocument[5] are classes like String and Array. They are not constructors (we'll see how to create new Element objects later in the chapter), but they have prototype objects and you can extend them with custom methods:

```
Element.prototype.next = function() {
    if (this.nextElementSibling) return this.nextElementSibling;
    var sib = this.nextSibling;
    while(sib && sib.nodeType !== 1) sib = sib.nextSibling;
    return sib;
};
```

5. IE8 supports extendable prototypes for Element, HTMLDocument, and Text, but not for Node, Document, HTMLElement, or any of the more specific HTMLElement subtypes.

The functions of Example 15-2 are not defined as Element methods because this technique is not supported by IE7.

This ability to extend DOM types is still useful, however, if you want to implement IE-specific features in browsers other than IE. As noted above, the nonstandard Element property `children` was introduced by IE and has been adopted by other browsers. You can use code like this to simulate it in browsers like Firefox 3.0 that do not support it:

```
// Simulate the Element.children property in non-IE browsers that don't have it
// Note that this returns a static array rather than a live NodeList
if (!document.documentElement.children) {
    Element.prototype.__defineGetter__("children", function() {
        var kids = [];
        for(var c = this.firstChild; c != null; c = c.nextSibling)
            if (c.nodeType === 1) kids.push(c);
        return kids;
    });
}
```

The `__defineGetter__` method (covered in §6.7.1) is completely nonstandard, but it is perfect for portability code like this.

15.4 Attributes

HTML elements consist of a tag name and a set of name/value pairs known as *attributes*. The `<a>` element that defines a hyperlink, for example, uses the value of its `href` attribute as the destination of the link. The attribute values of HTML elements are available as properties of the HTMLElement objects that represent those elements. The DOM also defines other APIs for getting and setting the values of XML attributes and nonstandard HTML attributes. The subsections that follow have details.

15.4.1 HTML Attributes As Element Properties

The HTMLElement objects that represent the elements of an HTML document define read/write properties that mirror the HTML attributes of the elements. HTMLElement defines properties for the universal HTTP attributes such as `id`, `title` `lang`, and `dir`, and event handler properties like `onclick`. Element-specific subtypes define attributes specific to those elements. To query the URL of an image, for example, you can use the `src` property of the HTMLElement that represents the `<img>` element:

```
var image = document.getElementById("myimage");
var imgurl = image.src;   // The src attribute is the URL of the image
image.id === "myimage"    // Since we looked up the image by id
```

Similarly, you might set the form-submission attributes of a `<form>` element with code like this:

```
var f = document.forms[0];                      // First <form> in the document
f.action = "http://www.example.com/submit.php"; // Set URL to submit it to.
f.method = "POST";                              // HTTP request type
```

HTML attributes are not case sensitive, but JavaScript property names are. To convert an attribute name to the JavaScript property, write it in lowercase. If the attribute is more than one word long, however, put the first letter of each word after the first in uppercase: `defaultChecked` and `tabIndex`, for example.

Some HTML attribute names are reserved words in JavaScript. For these, the general rule is to prefix the property name with "html". The HTML `for` attribute (of the `<label>` element), for example, becomes the JavaScript `htmlFor` property. "class" is a reserved (but unused) word in JavaScript, and the very important HTML `class` attribute is an exception to the rule above: it becomes `className` in JavaScript code. We'll see the `className` property again in Chapter 16.

The properties that represent HTML attributes usually have string value. When the attribute is a boolean or numeric value (the `defaultChecked` and `maxLength` attributes of an `<input>` element, for example), the properties values are booleans or numbers instead of strings. Event handler attributes always have Function objects (or `null`) as their values. The HTML5 specification defines a few attributes (such as the `form` attribute of `<input>` and related elements) that convert element IDs to actual Element objects. Finally, the value of the `style` property of any HTML element is a CSSStyleDeclaration object rather than a string. We'll see much more about this important property in Chapter 16.

Note that this property-based API for getting and setting attribute values does not define any way to remove an attribute from an element. In particular, the `delete` operator cannot be used for this purpose. The section that follows describes a method that you can use to accomplish this.

15.4.2 Getting and Setting Non-HTML Attributes

As described above, HTMLElement and its subtypes define properties that correspond to the standard attributes of HTML elements. The Element type also defines `getAttribute()` and `setAttribute()` methods that you can use to query and set nonstandard HTML attributes and to query and set attributes on the elements of an XML document:

```
var image = document.images[0];
var width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

The code above highlights two important differences between these methods and the property-based API described above. First, attribute values are all treated as strings. `getAttribute()` never returns a number, boolean, or object. Second, these methods use standard attribute names, even when those names are reserved words in JavaScript. For HTML elements, the attribute names are case insensitive.

Element also defines two related methods, `hasAttribute()` and `removeAttribute()`, which check for the presence of a named attribute and remove an attribute entirely. These methods are particularly useful with boolean attributes: these are attributes (such

as the disabled attribute of HTML form elements) whose presence or absence from an element matters but whose value is not relevant.

If you are working with XML documents that include attributes from other namespaces, you can use the namespaced variants of these four methods: getAttributeNS(), setAttributeNS(), hasAttributeNS(), and removeAttributeNS(). Instead of taking a single attribute name string, these methods take two. The first is the URI that identifies the namespace. The second is usually the unqualified local name of the attribute within the namespace. For setAttributeNS() only, however, the second argument is the qualified name of the attribute and includes the namespace prefix. You can read more about these namespace-aware attribute methods in Part IV.

15.4.3 Dataset Attributes

It is sometimes useful to attach additional information to HTML elements, typically when JavaScript code will be selecting those elements and manipulating them in some way. Sometimes this can be done by adding special identifiers to the class attribute. Other times, for more complex data, client-side programmers resort to the use of nonstandard attributes. As noted above, you can use the getAttribute() and setAttribute() methods to read and write the values of nonstandard attributes. The price you pay, however, is that your document will not be valid HTML.

HTML5 provides a solution. In an HTML5 document, any attribute whose name is lowercase and begins with the prefix "data-" is considered valid. These "dataset attributes" will not affect the presentation of the elements on which they appear and they define a standard way to attach additional data without compromising document validity.

HTML5 also defines a dataset property on Element objects. This property refers to an object, which has properties that correspond to the data- attributes with their prefix removed. Thus dataset.x would hold the value of the data-x attribute. Hyphenated attributes map to camel-case property names: the attribute data-jquery-test becomes the property dataset.jqueryTest.

As a more concrete example, suppose that a document contains the following markup:

```
<span class="sparkline" data-ymin="0" data-ymax="10">
1 1 1 2 2 3 4 5 5 4 3 5 6 7 7 4 2 1
</span>
```

A *sparkline* is a small graphic—often a line plot—designed to be displayed within the flow of text. In order to generate a sparkline, you might extract the value of the dataset attributes above with code like this:

```
// Assumes the ES5 Array.map() method (or a work-alike) is defined
var sparklines = document.getElementsByClassName("sparkline");
for(var i = 0; i < sparklines.length; i++) {
    var dataset = sparklines[i].dataset;
    var ymin = parseFloat(dataset.ymin);
    var ymax = parseFloat(dataset.ymax);
    var data = sparklines[i].textContent.split(" ").map(parseFloat);
```

```
    drawSparkline(sparklines[i], ymin, ymax, data);  // Not yet implemented
}
```

At the time of this writing, the dataset property is not implemented in current browsers, and the code above would have to be written like this:

```
var sparklines = document.getElementsByClassName("sparkline");
for(var i = 0; i < sparklines.length; i++) {
    var elt = sparklines[i];
    var ymin = parseFloat(elt.getAttribute("data-ymin"));
    var ymin = parseFloat(elt.getAttribute("data-ymax"));
    var points = elt.getAttribute("data-points");
    var data = elt.textContent.split(" ").map(parseFloat);
    drawSparkline(elt, ymin, ymax, data);  // Not yet implemented
}
```

Note that the dataset property is (or will be, when implemented) a live, two-way interface to the data- attributes of an element. Setting or deleting a property of dataset sets or removes the corresponding data- attribute of the element.

The drawSparkline() function in the above examples is fictitious, but Example 21-13 draws sparklines marked up like this using the <canvas> element.

15.4.4 Attributes As Attr Nodes

There is one more way to work with the attributes of an Element. The Node type defines an attributes property. This property is null for any nodes that are not Element objects. For Element objects, attributes is a read-only array-like object that represents all the attributes of the element. The attributes object is live in the way that NodeLists are. It can be indexed numerically, which means that you can enumerate all the attributes of an element. And it can also be indexed by attribute name:

```
document.body.attributes[0]        // The first attribute of the <body> elt
document.body.attributes.bgcolor   // The bgcolor attribute of the <body> elt
document.body.attributes["ONLOAD"] // The onload attribute of the <body> elt
```

The values obtained when you index the attributes object are Attr objects. Attr objects are a specialized kind of Node but are never really used like one. The name and value properties of an Attr return the name and value of the attribute.

15.5 Element Content

Take a look again at Figure 15-1, and ask yourself what the "content" of the <p> element is. There are three ways we might answer this question:

- The content is the HTML string "This is a simple document."
- The content is the plain-text string "This is a simple document."
- The content is a Text node, an Element node that has a Text node child, and another Text node.

Each of these are valid answers, and each answer is useful in its own way. The sections that follow explain how to work with the HTML representation, the plain-text representation, and the tree representation of element content.

15.5.1 Element Content As HTML

Reading the `innerHTML` property of an Element returns the content of that element as a string of markup. Setting this property on an element invokes the web browser's parser and replaces the element's current content with a parsed representation of the new string. (Despite its name, `innerHTML` can be used with XML elements as well as HTML elements.)

Web browsers are very good at parsing HTML and setting `innerHTML` is usually fairly efficient, even though the value you specify must be parsed. Note, however, that repeatedly appending bits of text to the `innerHTML` property with the `+=` operator is usually not efficient because it requires both a serialization step and a parsing step.

`innerHTML` was introduced in IE4. Although it has long been supported by all browsers, it has only become standardized with the advent of HTML5. HTML5 says that `innerHTML` should work on Document nodes as well as Element nodes, but this is not universally supported yet.

HTML5 also standardizes a property named `outerHTML`. When you query `outerHTML`, the string of HTML or XML markup that is returned includes the opening and closing tags of the element on which you queried it. When you set `outerHTML` on an element, the new content replaces the element itself. `outerHTML` is defined only for Element nodes, not Documents. At the time of this writing, `outerHTML` is supported by all current browsers except Firefox. (See Example 15-5, later in this chapter, for an `outerHTML` implementation based on `innerHTML`.)

Another feature introduced by IE and standardized by HTML5 is the `insertAdjacentHTML()` method, which allows you to insert a string of arbitrary HTML markup "adjacent" to the specified element. The markup is passed as the second argument to this method, and the precise meaning of "adjacent" depends on the value of the first argument. This first argument should be a string with one of the values "beforebegin", "afterbegin", "beforeend" or "afterend". These values correspond to insertion points that are illustrated in Figure 15-3.

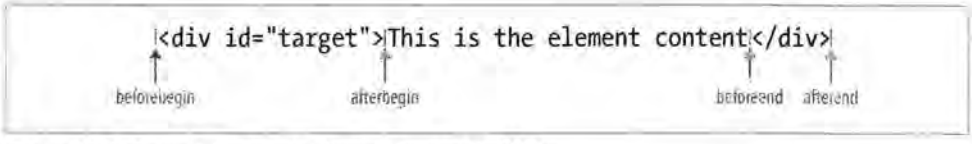


Figure 15-3. Insertion points for insertAdjacentHTML()

`insertAdjacentHTML()` is not supported by current versions of Firefox. Later in this chapter, Example 15-6 shows how to implement `insertAdjacentHTML()` using the

innerHTML property and also demonstrates how to write HTML insertion methods that do not require the insertion position to be specified with a string argument.

15.5.2 Element Content As Plain Text

Sometimes you want to query the content of an element as plain text, or to insert plain-text into a document (without having to escape the angle brackets and ampersands used in HTML markup). The standard way to do this is with the textContent property of Node:

```
var para = document.getElementsByTagName("p")[0];  // First <p> in the document
var text = para.textContent;          // Text is "This is a simple document."
para.textContent = "Hello World!";  // Alter paragraph content
```

The textContent property is supported by all current browsers except IE. In IE, you can use the Element property innerText instead. Microsoft introduced innerText in IE4, and it is supported by all current browsers except Firefox.

The textContent and innerText properties are similar enough that you can usually use them interchangeably. Be careful though to distinguish empty elements (the string "" is falsy in JavaScript) from undefined properties:

```
/**
 * With one argument, return the textContent or innerText of the element.
 * With two arguments, set the textContent or innerText of element to value.
 */
function textContent(element, value) {
    var content = element.textContent;  // Check if textContent is defined
    if (value === undefined) { // No value passed, so return current text
        if (content !== undefined) return content;
        else return element.innerText;
    }
    else {                      // A value was passed, so set text
        if (content !== undefined) element.textContent = value;
        else element.innerText = value;
    }
}
```

The textContent property is a straightforward concatenation of all Text node descendants of the specified element. innerText does not have a clearly specified behavior, but differs from textContent in a few ways. innerText does not return the content of <script> elements. It omits extraneous whitespace and attempts to preserve table formatting. Also, innerText is treated as a read-only property for certain table elements such as <table>, <tbody>, and <tr>.

Text in <script> elements

Inline <script> elements (i.e., those that do not have a src attribute) have a text property that you can use to retrieve their text. The content of a <script> element is never displayed by the browser, and the HTML parser ignores angle brackets and ampersands within a script. This makes a <script> element an ideal place to embed arbitrary textual data for use by your application. Simply set the type attribute of the element to some

value (such as "text/x-custom-data") that makes it clear that the script is not executable JavaScript code. If you do this, the JavaScript interpreter will ignore the script, but the element will exist in the document tree and its text property will return the data to you.

15.5.3 Element Content As Text Nodes

Another way to work with the content of an element is as a list of child nodes, each of which may have its own set of children. When thinking about element content, it is usually the Text nodes that are of interest. In XML documents, you must also be prepared to handle CDATASection nodes—they are a subtype of Text and represent the content of CDATA sections.

Example 15-3 shows a `textContent()` function that recursively traverses the children of an element and concatenates the text of all the Text node descendants. In order to understand the code, recall that the `nodeValue` property (defined by the Node type) holds the content of a Text node.

Example 15-3. Finding all Text node descendants of an element

```
// Return the plain-text content of element e, recursing into child elements.
// This method works like the textContent property
function textContent(e) {
    var child, type, s = "";  // s holds the text of all children
    for(child = e.firstChild; child != null; child = child.nextSibling) {
        type = child.nodeType;
        if (type === 3 || type === 4)  // Text and CDATASection nodes
            s += child.nodeValue;
        else if (type === 1)           // Recurse for Element nodes
            s += textContent(child);
    }
    return s;
}
```

The `nodeValue` property is read/write and you can set it to change the content displayed by a Text or CDATASection node. Both Text and CDATASection are subtypes of CharacterData, which you can look up in Part IV. CharacterData defines a `data` property, which is the same text as `nodeValue`. The following function converts the content of Text nodes to uppercase by setting the `data` property:

```
// Recursively convert all Text node descendants of n to uppercase.
function upcase(n) {
    if (n.nodeType == 3 || n.nodeTyep == 4) // If n is Text or CDATA
        n.data = n.data.toUpperCase();      // ...convert content to uppercase.
    else                                    // Otherwise, recurse on child nodes
        for(var i = 0; i < n.childNodes.length; i++)
            upcase(n.childNodes[i]);
}
```

CharacterData also defines infrequently used methods for appending, deleting, inserting, and replacing text within a Text or CDATASection node. Instead of altering the content of existing Text nodes, it is also possible to insert brand-new Text nodes into

an Element or to replace existing nodes with new Text nodes. Creating, inserting, and deleting nodes is the topic of the next section.

15.6 Creating, Inserting, and Deleting Nodes

We've seen how to query and alter document content using strings of HTML and of plain text. And we've also seen that we can traverse a Document to examine the individual Element and Text nodes that it is made of. It is also possible to alter a document at the level of individual nodes. The Document type defines methods for creating Element and Text objects, and the Node type defines methods for inserting, deleting, and replacing nodes in the tree. Example 13-4 demonstrated both node creation and node insertion, and that short example is duplicated here:

```
// Asynchronously load and execute a script from a specified URL
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; // Find document <head>
    var s = document.createElement("script");  // Create a <script> element
    s.src = url;                               // Set its src attribute
    head.appendChild(s);                       // Insert the <script> into head
}
```

The subsections that follow include more details and examples of node creation, of the insertion and deletion of nodes, and also of the use of DocumentFragment as a shortcut when working with multiple nodes.

15.6.1 Creating Nodes

As shown in the code above, you can create new Element nodes with the `createElement()` method of the Document object. Pass the tag name of the element as the method argument: this name is case-insensitive for HTML documents and case-sensitive for XML documents.

Text nodes are created with a similar method:

```
var newnode = document.createTextNode("text node content");
```

Document defines other factory methods, such as the infrequently used `createComment()`, as well. We'll use the `createDocumentFragment()` method in §15.6.4. When working with documents that use XML namespaces, you can use `createElementNS()` to specify both the namespace URI and the tag name of the Element to be created.

Another way to create new document nodes is to make copies of existing ones. Every node has a `cloneNode()` method that returns a new copy of the node. Pass `true` to recursively copy all descendants as well, or `false` to only make a shallow copy. In browsers other than IE, the Document object also defines a similar method named `importNode()`. If you pass it a node from another document, it returns a copy suitable for insertion into this document. Pass `true` as the second argument to recursively import all descendants.

15.6.2 Inserting Nodes

Once you have a new node, you can insert it into the document with the Node methods `appendChild()` or `insertBefore()`. `appendChild()` is invoked on the Element node that you want to insert into, and it inserts the specified node so that it becomes the `lastChild` of that node.

`insertBefore()` is like `appendChild()`, but it takes two arguments. The first argument is the node to be inserted. The second argument is the node before which that node is to be inserted. This method is invoked on the node that will be the parent of the new node, and the second argument must be a child of that parent node. If you pass `null` as that second argument, the `insertBefore()` behaves like `appendChild()` and inserts at the end.

Here is a simple function for inserting a node at a numerical index. It demonstrates both `appendChild()` and `insertBefore()`:

```
// Insert the child node into parent so that it becomes child node n
function insertAt(parent, child, n) {
    if (n < 0 || n > parent.childNodes.length) throw new Error("invalid index");
    else if (n == parent.childNodes.length) parent.appendChild(child);
    else parent.insertBefore(child, parent.childNodes[n]);
}
```

If you call `appendChild()` or `insertBefore()` to insert a node that is already in the document, that node will automatically be removed from its current position and reinserted at its new position: there is no need to explicitly remove the node. Example 15-4 shows a function for sorting the rows of a table based on the values of cells in a specified column. It doesn't create any new nodes but uses `appendChild()` to change the order of existing nodes.

Example 15-4. Sorting the rows of a table

```
// Sort the rows in first <tbody> of the specified table according to
// the value of nth cell within each row. Use the comparator function
// if one is specified. Otherwise, compare the values alphabetically.
function sortrows(table, n, comparator) {
    var tbody = table.tBodies[0]; // First <tbody>; may be implicitly created
    var rows = tbody.getElementsByTagName("tr"); // All rows in the tbody
    rows = Array.prototype.slice.call(rows,0);   // Snapshot in a true array

    // Now sort the rows based on the text in the nth <td> element
    rows.sort(function(row1,row2) {
        var cell1 = row1.getElementsByTagName("td")[n];  // Get nth cell
        var cell2 = row2.getElementsByTagName("td")[n];  // of both rows
        var val1 = cell1.textContent || cell1.innerText; // Get text content
        var val2 = cell2.textContent || cell2.innerText; // of the two cells
        if (comparator) return comparator(val1, val2);   // Compare them!
        if (val1 < val2) return -1;
        else if (val1 > val2) return 1;
        else return 0;
    });
```

```
    // Now append the rows into the tbody in their sorted order.
    // This automatically moves them from their current location, so there
    // is no need to remove them first. If the <tbody> contains any
    // nodes other than <tr> elements, those nodes will float to the top.
    for(var i = 0; i < rows.length; i++) tbody.appendChild(rows[i]);
}

// Find the <th> elements of the table (assuming there is only one row of them)
// and make them clickable so that clicking on a column header sorts
// by that column.
function makeSortable(table) {
    var headers = table.getElementsByTagName("th");
    for(var i = 0; i < headers.length; i++) {
        (function(n) {  // Nested funtion to create a local scope
            headers[i].onclick = function() { sortrows(table, n); };
        }(i));          // Assign value of i to the local variable n
    }
}
```

15.6.3 Removing and Replacing Nodes

The `removeChild()` method removes a node from the document tree. Be careful, however: this method isn't invoked on the node to be removed but (as the "child" part of its name implies) on the parent of that node. Invoke the method on the parent node and pass the child node that is to be removed as the method argument. To remove the node `n` from the document, you'd write:

```
n.parentNode.removeChild(n);
```

`replaceChild()` removes one child node and replaces it with a new one. Invoke this method on the parent node, passing the new node as the first argument and the node to be replaced as the second argument. To replace the node `n` with a string of text, for example, you could write:

```
n.parentNode.replaceChild(document.createTextNode("[ REDACTED ]"), n);
```

The following function demonstrates another use of `replaceChild()`:

```
// Replace the node n with a new <b> element and make n a child of that element.
function embolden(n) {
    // If we're passed a string instead of a node, treat it as an element id
    if (typeof n == "string") n = document.getElementById(n);
    var parent = n.parentNode;          // Get the parent of n
    var b = document.createElement("b"); // Create a <b> element
    parent.replaceChild(b, n);          // Replace n with the <b> element
    b.appendChild(n);                   // Make n a child of the <b> element
}
```

§15.5.1 introduced the `outerHTML` property of an element and explained that it was not implemented in current versions of Firefox. Example 15-5 shows how to implement this property in Firefox (and any other browser that supports `innerHTML`, has an extensible `Element.prototype` object, and has methods for defining property getters and setters). The code also demonstrates a practical use for the `removeChild()` and `cloneNode()` methods.

Example 15-5. Implementing the outerHTML property using innerHTML

```
// Implement the outerHTML property for browsers that don't support it.
// Assumes that the browser does support innerHTML, has an extensible
// Element.prototype, and allows getters and setters to be defined.
(function() {
    // If we already have outerHTML return without doing anything
    if (document.createElement("div").outerHTML) return;

    // Return the outer HTML of the element referred to by this
    function outerHTMLGetter() {
        var container = document.createElement("div"); // Dummy element
        container.appendChild(this.cloneNode(true));   // Copy this to dummy
        return container.innerHTML;                    // Return dummy content
    }

    // Set the outer HTML of the this element to the specified value
    function outerHTMLSetter(value) {
        // Create a dummy element and set its content to the specified value
        var container = document.createElement("div");
        container.innerHTML = value;
        // Move each of the nodes from the dummy into the document
        while(container.firstChild)  // Loop until container has no more kids
            this.parentNode.insertBefore(container.firstChild, this);
        // And remove the node that has been replaced
        this.parentNode.removeChild(this);
    }

    // Now use these two functions as getters and setters for the
    // outerHTML property of all Element objects. Use ES5 Object.defineProperty
    // if it exists and otherwise fall back on __defineGetter__ and Setter__.
    if (Object.defineProperty) {
        Object.defineProperty(Element.prototype, "outerHTML", {
                                get: outerHTMLGetter,
                                set: outerHTMLSetter,
                                enumerable: false, configurable: true
                              });
    }
    else {
        Element.prototype.__defineGetter__("outerHTML", outerHTMLGetter);
        Element.prototype.__defineSetter__("outerHTML", outerHTMLSetter);
    }
}());
```

15.6.4 Using DocumentFragments

A DocumentFragment is a special kind of Node that serves as a temporary container for other nodes. Create a DocumentFragment like this:

```
var frag = document.createDocumentFragment();
```

Like a Document node, a DocumentFragment stands alone and is not part of any other document. Its `parentNode` is always `null`. Like an Element, however, a DocumentFragment can have any number of children, which you can manipulate with `appendChild()`, `insertBefore()`, and so on.

The special thing about DocumentFragment is that it allows a set of nodes to be treated as a single node: if you pass a DocumentFragment to `appendChild()`, `insertBefore()`, or `replaceChild()`, it is the children of the fragment that are inserted into the document, not the fragment itself. (The children are moved from the fragment into the document, and the fragment becomes empty and ready for reuse.) The following function uses a DocumentFragment to reverse the order of the children of a node:

```
// Reverse the order of the children of Node n
function reverse(n) {
    // Create an empty DocumentFragment as a temporary container
    var f = document.createDocumentFragment();
    // Now loop backward through the children, moving each one to the fragment.
    // The last child of n becomes the first child of f, and vice-versa.
    // Note that appending a child to f automatically removes it from n.
    while(n.lastChild) f.appendChild(n.lastChild);

    // Finally, move the children of f all at once back to n, all at once.
    n.appendChild(f);
}
```

Example 15-6 implements the `insertAdjacentHTML()` method (see §15.5.1) using the `innerHTML` property and a DocumentFragment. It also defines logically named HTML insertion functions as an alternative to the confusing `insertAdjacentHTML()` API. The internal utility function `fragment()` is possibly the most useful part of this code: it returns a DocumentFragment that contains the parsed representation of a specified string of HTML text.

Example 15-6. Implementing insertAdjacentHTML() using innerHTML

```
// This module defines Element.insertAdjacentHTML for browsers that don't
// support it, and also defines portable HTML insertion functions that have
// more logical names than insertAdjacentHTML:
//     Insert.before(), Insert.after(), Insert.atStart(), Insert.atEnd()
var Insert = (function() {
    // If elements have a native insertAdjacentHTML, use it in four HTML
    // insertion functions with more sensible names.
    if (document.createElement("div").insertAdjacentHTML) {
        return {
            before: function(e,h) {e.insertAdjacentHTML("beforebegin",h);},
            after: function(e,h) {e.insertAdjacentHTML("afterend",h);},
            atStart: function(e,h) {e.insertAdjacentHTML("afterbegin",h);},
            atEnd: function(e,h) {e.insertAdjacentHTML("beforeend",h);}
        };
    }

    // Otherwise, we have no native insertAdjacentHTML. Implement the same
    // four insertion functions and then use them to define insertAdjacentHTML.

    // First, define a utility method that takes a string of HTML and returns
    // a DocumentFragment containing the parsed representation of that HTML.
    function fragment(html) {
        var elt = document.createElement("div");      // Create empty element
        var frag = document.createDocumentFragment(); // Create empty fragment
        elt.innerHTML = html;                         // Set element content
```

```
        while(elt.firstChild)                    // Move all nodes
            frag.appendChild(elt.firstChild);    //    from elt to frag
        return frag;                             // And return the frag
    }

    var Insert = {
        before: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html), elt);
        },
        after: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html),elt.nextSibling);
        },
        atStart: function(elt, html) {
            elt.insertBefore(fragment(html), elt.firstChild);
        },
        atEnd: function(elt, html) { elt.appendChild(fragment(html)); }
    };

    // Now implement insertAdjacentHTML based on the functions above
    Element.prototype.insertAdjacentHTML = function(pos, html) {
        switch(pos.toLowerCase()) {
        case "beforebegin": return Insert.before(this, html);
        case "afterend": return Insert.after(this, html);
        case "afterbegin": return Insert.atStart(this, html);
        case "beforeend": return Insert.atEnd(this, html);
        }
    };
    return Insert;  // Finally return the four insertion function
}());
```

15.7 Example: Generating a Table of Contents

Example 15-7 shows how to dynamically create a table of contents for a document. It demonstrates many of the document scripting concepts described in the sections above: element selection, document traversal, setting element attributes, setting the `innerHTML` property, and creating new nodes and inserting them into the document. The example is well-commented and you should have no trouble following the code.

Example 15-7. An automatically generated table of contents

```
/**
 * TOC.js: create a table of contents for a document.
 *
 * This module registers an anonymous function that runs automatically
 * when the document finishes loading. When it runs, the function first
 * looks for a document element with an id of "TOC". If there is no
 * such element it creates one at the start of the document.
 *
 * Next, the function finds all <h1> through <h6> tags, treats them as
 * section titles, and creates a table of contents within the TOC
 * element. The function adds section numbers to each section heading
 * and wraps the headings in named anchors so that the TOC can link to
 * them. The generated anchors have names that begin with "TOC", so
 * you should avoid this prefix in your own HTML.
```

```
 *
 * The entries in the generated TOC can be styled with CSS. All entries have
 * a class "TOCEntry". Entries also have a class that corresponds to the level
 * of the section heading. <h1> tags generate entries of class "TOCLevel1",
 * <h2> tags generate entries of class "TOCLevel2", and so on. Section numbers
 * inserted into headings have class "TOCSectNum".
 *
 * You might use this module with a stylesheet like this:
 *
 *   #TOC { border: solid black 1px; margin: 10px; padding: 10px; }
 *   .TOCEntry { font-family: sans-serif; }
 *   .TOCEntry a { text-decoration: none; }
 *   .TOCLevel1 { font-size: 16pt; font-weight: bold; }
 *   .TOCLevel2 { font-size: 12pt; margin-left: .5in; }
 *   .TOCSectNum:after { content: ": "; }
 *
 * That final line generates a colon and space after section numbers. To hide
 * the section numbers, use this:
 *
 *   .TOCSectNum { display: none }
 *
 * This module requires the onLoad() utility function.
 **/
onLoad(function() { // Anonymous function defines a local scope
    // Find the TOC container element.
    // If there isn't one, create one at the start of the document.
    var toc = document.getElementById("TOC");
    if (!toc) {
        toc = document.createElement("div");
        toc.id = "TOC";
        document.body.insertBefore(toc, document.body.firstChild);
    }

    // Find all section heading elements
    var headings;
    if (document.querySelectorAll) // Can we do it the easy way?
        headings = document.querySelectorAll("h1,h2,h3,h4,h5,h6");
    else   // Otherwise, find the headings the hard way
        headings = findHeadings(document.body, []);

    // Recursively traverse the document body looking for headings
    function findHeadings(root, sects) {
        for(var c = root.firstChild; c != null; c = c.nextSibling) {
            if (c.nodeType !== 1) continue;
            if (c.tagName.length == 2 && c.tagName.charAt(0) == "H")
                sects.push(c);
            else
                findHeadings(c, sects);
        }
        return sects;
    }

    // Initialize an array that keeps track of section numbers.
    var sectionNumbers = [0,0,0,0,0,0];

    // Now loop through the section header elements we found.
```

```
    for(var h = 0; h < headings.length; h++) {
        var heading = headings[h];

        // Skip the section heading if it is inside the TOC container.
        if (heading.parentNode == toc) continue;

        // Figure out what level heading it is.
        var level = parseInt(heading.tagName.charAt(1));
        if (isNaN(level) || level < 1 || level > 6) continue;

        // Increment the section number for this heading level
        // and reset all lower heading level numbers to zero.
        sectionNumbers[level-1]++;
        for(var i = level; i < 6; i++) sectionNumbers[i] = 0;

        // Now combine section numbers for all heading levels
        // to produce a section number like 2.3.1.
        var sectionNumber = sectionNumbers.slice(0,level).join(".")

        // Add the section number to the section header title.
        // We place the number in a <span> to make it styleable.
        var span = document.createElement("span");
        span.className = "TOCSectNum";
        span.innerHTML = sectionNumber;
        heading.insertBefore(span, heading.firstChild);

        // Wrap the heading in a named anchor so we can link to it.
        var anchor = document.createElement("a");
        anchor.name = "TOC"+sectionNumber;
        heading.parentNode.insertBefore(anchor, heading);
        anchor.appendChild(heading);

        // Now create a link to this section.
        var link = document.createElement("a");
        link.href = "#TOC" + sectionNumber; // Link destination
        link.innerHTML = heading.innerHTML; // Link text is same as heading

        // Place the link in a div that is styleable based on the level.
        var entry = document.createElement("div");
        entry.className = "TOCEntry TOCLevel" + level;
        entry.appendChild(link);

        // And add the div to the TOC container.
        toc.appendChild(entry);
    }
});
```

15.8 Document and Element Geometry and Scrolling

In this chapter so far we have thought about documents as abstract trees of elements and text nodes. But when a browser renders a document within a window, it creates a visual representation of the document in which each element has a position and a size. Often, web applications can treat documents as trees of elements and never have to

think about how those elements are rendered onscreen. Sometimes, however, it is necessary to determine the precise geometry of an element. We'll see in Chapter 16, for example, that the CSS can be used to specify the position of an element. If you want to use CSS to dynamically position an element (such as a tooltip or callout) next to some ordinary browser-positioned element, you need to be able to determine the location of that element.

This section explains how you can go back and forth between the abstract, tree-based *model* of a document and the geometrical, coordinate-based *view* of the document as it is laid out in a browser window. The properties and methods described in this section have been implemented in browsers for a long time (though some were, until recently, IE-specific and some were not implemented by IE until IE9). At the time of this writing, they are going through the W3C standardization process as the CSSOM-View Module (see *http://www.w3.org/TR/cssom-view/*).

15.8.1 Document Coordinates and Viewport Coordinates

The position of an element is measured in pixels, with the X coordinate increasing to the right and the Y coordinate increasing as we go down. There are two different points we can use as the coordinate system origin, however: the X and Y coordinates of an element can be relative to the top-left corner of the document or relative to the top-left corner of the *viewport* in which the document is displayed. In top-level windows and tabs, the "viewport" is the portion of the browser that actually displays document content: it excludes browser "chrome" such as menus, toolbars, and tabs. For documents displayed in frames, the viewport is the `<iframe>` element that defines the frame. In either case, when we talk about the position of an element, we must be clear whether we are using document coordinates or viewport coordinates. (Note that viewport coordinates are sometimes called window coordinates.)

If the document is smaller than the viewport, or if it has not been scrolled, the upper-left corner of the document is in the upper-left corner of the viewport and the document and viewport coordinate systems are the same. In general, however, to convert between the two coordinate systems, we must add or subtract the *scroll offsets*. If an element has a Y coordinate of 200 pixels in document coordinates, for example, and if the user has scrolled the browser down by 75 pixels, then that element has a Y coordinate of 125 pixels in viewport coordinates. Similarly, if an element has an X coordinate of 400 in viewport coordinates and the user has scrolled the viewport 200 pixels horizontally, the element's X coordinate in document coordinates is 600.

Document coordinates are more fundamental than viewport coordinates, and they do not change when the user scrolls. Nevertheless, it is quite common to use viewport coordinates in client-side programming. We use document coordinates when we specify an element position using CSS (see Chapter 16). But the simplest way of querying the position of an element (see §15.8.2) returns the position in viewport coordinates. Similarly, when we register handler functions for mouse events, the coordinates of the mouse pointer are reported in viewport coordinates.

In order to convert between coordinate systems, we need to be able to determine the scrollbar positions for the browser window. The `pageXOffset` and `pageYOffset` properties of the Window object provide these values in all browsers except IE versions 8 and before. IE (and all modern browsers) also make the scrollbar positions available through `scrollLeft` and `scrollTop` properties. The confusing thing is that you normally query these properties on the root element of the document (`document.documentElement`), but in quirks mode (see §13.4.4) you must query them on the `<body>` element (`document.body`) of the document instead. Example 15-8 shows how to portably query the scrollbar positions.

Example 15-8. Querying the scrollbar positions of a window

```
// Return the current scrollbar offsets as the x and y properties of an object
function getScrollOffsets(w) {
    // Use the specified window or the current window if no argument
    w = w || window;

    // This works for all browsers except IE versions 8 and before
    if (w.pageXOffset != null) return {x: w.pageXOffset, y:w.pageYOffset};

    // For IE (or any browser) in Standards mode
    var d = w.document;
    if (document.compatMode == "CSS1Compat")
        return {x:d.documentElement.scrollLeft, y:d.documentElement.scrollTop};

    // For browsers in Quirks mode
    return { x: d.body.scrollLeft, y: d.body.scrollTop };
}
```

It is sometimes useful to be able to determine the viewport size—to find what portions of the document are currently visible, for example. As with the scroll offsets, the easy way to query viewport size does not work in IE8 and before and the technique that works in IE depends on whether the browser is in quirks mode or standards mode. Example 15-9 shows how to portably query the viewport size. Note how similar the code is to Example 15-8.

Example 15-9. Querying the viewport size of a window

```
// Return the viewport size as w and h properties of an object
function getViewportSize(w) {
    // Use the specified window or the current window if no argument
    w = w || window;

    // This works for all browsers except IE8 and before
    if (w.innerWidth != null) return {w: w.innerWidth, h:w.innerHeight};

    // For IE (or any browser) in Standards mode
    var d = w.document;
    if (document.compatMode == "CSS1Compat")
```

```
        return { w: d.documentElement.clientWidth,
                 h: d.documentElement.clientHeight };

    // For browsers in Quirks mode
    return { w: d.body.clientWidth, h: d.body.clientWidth };
}
```

The two examples above have used the `scrollLeft`, `scrollTop`, `clientWidth`, and `clientHeight` properties. We'll encounter these properties again in §15.8.5.

15.8.2 Querying the Geometry of an Element

The easiest way to determine the size and position of an element is to call its `getBoundingClientRect()` method. This method was introduced in IE5 and is now implemented by all current browsers. It expects no arguments and returns an object with properties `left`, `right`, `top`, and `bottom`. The `left` and `top` properties give the X and Y coordinates of the upper-left corner of the element and the `right` and `bottom` properties give the coordinates of the lower-right corner.

This method returns element positions in viewport coordinates. (The word "client" in the method name `getBoundingClientRect()` is an oblique reference to the web browser client—specifically to the window and the viewport it defines.) To convert to document coordinates that remain valid even if the user scrolls the browser window, add the scroll offsets:

```
var box = e.getBoundingClientRect();  // Get position in viewport coordinates
var offsets = getScrollOffsets();     // Utility function defined above
var x = box.left + offsets.x;         // Convert to document coordinates
var y = box.top + offsets.y;
```

In many browsers (and in the W3C standard), the object returned by `getBoundingClientRect()` also has `width` and `height` properties, but the original IE implementation does not do this. For portability, you can compute the element width and height like this:

```
var box = e.getBoundingClientRect();
var w = box.width || (box.right - box.left);
var h = box.height || (box.bottom - box.top);
```

You'll learn in Chapter 16 that the content of an element is surrounded by an optional blank area known as *padding*. The padding is surrounded by an optional border, and the border is surrounded by optional margins. The coordinates returned by `getBoundingClientRect()` include the border and the padding of the element but do not include the element margins.

If the word "Client" in the method `getBoundingClientRect()` specifies the coordinate system of the returned rectangle, what explains the word "Bounding" in the method name? Block elements, such as images, paragraphs, and `<div>` elements are always rectangular when laid out by the browser. Inline elements, such as `<span>`, `<code>`, and `<b>` elements, however, may span multiple lines and may therefore consist of multiple rectangles. Imagine, for example, some italicized text (marked up with `<i>` and `</i>`

tags) that is broken across two lines. Its rectangles consist of the right-hand portion of the first line and the left-hand portion of the second line (assuming left-to-right text). If you call `getBoundingClientRect()` on an inline element, it returns the "bounding rectangle" of the individual rectangles. For the `<i>` element described above, the bounding rectangle would include the entire width of both lines.

If you want to query the individual rectangles of inline elements, call the `getClientRects()` method to obtain a read-only array-like object whose elements are rectangle objects like those returned by `getBoundingClientRect()`.

We've seen that DOM methods like `getElementsByTagName()` return "live" results that are updated as the document changes. The rectangle objects (and rectangle object lists) returned by `getBoundingClientRect()` and `getClientRects()` are *not* live. They are static snapshots of the visual state of document when the methods are called. They are not updated when the user scrolls or resizes the browser window.

15.8.3 Determining the Element at a Point

The `getBoundingClientRect()` method allows us to determine the current position of an element in a viewport. Sometimes we want to go in the other direction and determine which element is at a given location in the viewport. You can determine this with the `elementFromPoint()` method of the Document object. Pass X and Y coordinates (using viewport coordinates, not document coordinates) and this method returns an Element object that is at the specified position. At the time of this writing, the algorithm for selecting the element is not specified, but the intent of this method is that it returns the innermost and uppermost (see the CSS `z-index` attribute in §16.2.1.1) element at that point. If you specify a point that is outside of the viewport, `elementFromPoint()` will return `null` even if that point would be perfectly valid when converted to document coordinates.

`elementFromPoint()` seems like a very useful method, and the obvious use case is passing the coordinates of the mouse pointer to determine which element the mouse is over. As we'll learn in Chapter 17, however, mouse event objects already include this information in their `target` property. In practice, therefore, `elementFromPoint()` is not commonly used.

15.8.4 Scrolling

Example 15-8 showed how to query the scrollbar positions for a browser window. The `scrollLeft` and `scrollTop` properties used in that example can be set to make the browser scroll, but there is an easier way that has been supported since the earliest days of JavaScript. The `scrollTo()` method of the Window object (and its synonym `scroll()`) takes the X and Y coordinates of a point (in document coordinates) and sets these as the scrollbar offsets. That is, it scrolls the window so that the specified point is in the upper left corner of the viewport. If you specify a point that is too close to the bottom or too close to the right edge of the document, the browser will move it as close as possible to the upper left corner but won't be able to get it all the way there. The

following code scrolls the browser so that the bottom-most page of the document is visible:

```
// Get the height of the document and viewport. offsetHeight is explained below.
var documentHeight = document.documentElement.offsetHeight;
var viewportHeight = window.innerHeight; // Or use getViewportSize() above
// And scroll so the last "page" shows in the viewport
window.scrollTo(0, documentHeight - viewportHeight);
```

The `scrollBy()` method of the Window is similar to `scroll()` and `scrollTo()`, but its arguments are relative and are added to the current scrollbar offsets. Speed readers might like a bookmarklet (§13.2.5.1) like this one, for example:

```
// Scroll 10 pixels down every 200 ms. Note there is no way to turn this off!
javascript:void setInterval(function() {scrollBy(0,10)}, 200);
```

Often, instead of scrolling to a numeric location in document, we just want to scroll so that a certain element in the document is visible. You could compute the position of the element with `getBoundingClientRect()`, convert that position to document coordinates, and then use the `scrollTo()` method, but it is easier to just call the `scrollIntoView()` method on the desired HTML element. This method ensures that the element on which it is invoked is visible in the viewport. By default, it tries to put the top edge of the element at or near the top of the viewport. If you pass `false` as the only argument, it will try to put the bottom edge of the element at the bottom of the viewport. The browser will also scroll the viewport horizontally as needed to make the element visible.

The behavior of `scrollIntoView()` is similar to what the browser does when you set `window.location.hash` to the name of a named anchor (an `<a name="">` element).

15.8.5 More on Element Size, Position and Overflow

The `getBoundingClientRect()` method is defined in all current browsers, but if you need to support an older generation of browsers, you can't rely on this method and must use older techniques for determining element size and position. Element size is easy: the readonly `offsetWidth` and `offsetHeight` properties of any HTML element return its on screen size, in CSS pixels. The returned sizes include the element border and padding but not margins.

All HTML elements have `offsetLeft` and `offsetTop` properties that return the X and Y coordinates of the element. For many elements, these values are document coordinates and directly specify the position of the element. But for descendants of positioned elements and for some other elements, such as table cells, these properties return coordinates that are relative to an ancestor element rather than the document. The `offsetParent` property specifies which element the properties are relative to. If `offsetParent` is null, the properties are document coordinates. In general, therefore, computing the position of an element `e` using `offsetLeft` and `offsetTop` requires a loop:

```
function getElementPosition(e) {
    var x = 0, y = 0;
    while(e != null) {
        x += e.offsetLeft;
```

```
        y += e.offsetTop;
        e = e.offsetParent;
    }
    return {x:x, y:y};
}
```

By looping through the `offsetParent` chain and accumulating offsets, this function computes the document coordinates of the specified element. (Recall that `getBoundingClientRect()` returns viewport coordinates instead.) This is not the final word on element positioning, however—this `getElementPosition()` function does not always compute the correct values, and we'll see how to fix it below.

In addition to the set of `offset` properties, all document elements define two other sets of properties, one whose names begin with `client` and one whose names begin with `scroll`. That is, every HTML element has all of the following properties:

```
offsetWidth     clientWidth     scrollWidth
offsetHeight    clientHeight    scrollHeight
offsetLeft      clientLeft      scrollLeft
offsetTop       clientTop       scrollTop
offsetParent
```

In order to understand these `client` and `scroll` properties, you need to know that the content of an HTML element may be larger than the content box allocated to hold that content, and that therefore individual elements may have scrollbars (see the CSS `overflow` attribute in §16.2.6). The content area is a viewport, like the browser window is, and when the content is larger than the viewport, we need to take an element's scrollbar position into account.

`clientWidth` and `clientHeight` are like `offsetWidth` and `offsetHeight` except that they do not include the border size, only the content area and its padding. Also, if the browser has added scrollbars between the padding and the border, `clientWidth` and `clientHeight` do not include the scrollbar in their returned value. Note that `clientWidth` and `clientHeight` always return 0 for inline elements like `<i>`, `<code>`, and `<span>`.

`clientWidth` and `clientHeight` were used in the `getViewportSize()` method of Example 15-9. As a special case, when these properties are queried on the root element of a document (or the body element in quirks mode), they return the same values as the `innerWidth` and `innerHeight` properties of the window.

The `clientLeft` and `clientTop` properties are not very useful: they return the horizontal and vertical distance between the outside of an element's padding and the outside of its border. Usually these values are just the width of the left and top borders. If an element has scrollbars, however, and if the browser places those scrollbars on the left or top (which would be unusual), `clientLeft` and `clientTop` also include the scrollbar width. For inline elements, `clientLeft` and `clientTop` are always 0.

`scrollWidth` and `scrollHeight` are the size of an element's content area plus its padding plus any overflowing content. When the content fits within the content area without overflow, these properties are the same as `clientWidth` and `clientHeight`. But when

there is overflow, they include the overflowing content and return values larger than `clientWidth` and `clientHeight`.

Finally, `scrollLeft` and `scrollTop` give the scrollbar positions of an element. We queried them on the root element of the document in the `getScrollOffsets()` method (Example 15-8), but they are also defined on any element. Note that `scrollLeft` and `scrollTop` are writable properties and you can set them to scroll the content within an element. (HTML elements do not have a `scrollTo()` method like the Window object does.)

When a document contains scrollable elements with overflowing content, the `getElementPosition()` method defined above does not work correctly because it does not take scrollbar position into account. Here is a modified version that subtracts scrollbar positions from the accumulated offsets and, in so doing, converts the returned position from document coordinates to viewport coordinates:

```
function getElementPos(elt) {
    var x = 0, y = 0;
    // Loop to add up offsets
    for(var e = elt; e != null; e = e.offsetParent) {
        x += e.offsetLeft;
        y += e.offsetTop;
    }
    // Loop again, through all ancestor elements to subtract scroll offsets.
    // This subtracts the main scrollbars, too, and converts to viewport coords.
    for(var e=elt.parentNode; e != null && e.nodeType == 1; e=e.parentNode) {
        x -= e.scrollLeft;
        y -= e.scrollTop;
    }
    return {x:x, y:y};
}
```

In modern browsers, this `getElementPos()` method returns the same position values as `getBoundingClientRect()` does (but is much less efficient). Theoretically, a function such as `getElementPos()` could be used in browsers that do not support `getBoundingClientRect()`. In practice, however, browsers that do not support `getBoundingClientRect()` have a lot of element positioning incompatibilities and a function as simple as this one will not work reliably. Practical client-side libraries like jQuery include functions for computing element position that augment this basic position computation algorithm with a number of browser-specific bug fixes. If you need to compute element position and need your code to work in browsers that do not support `getBoundingClientRect()`, you should probably use a library like jQuery.

15.9 HTML Forms

The HTML `<form>` element, and the various form input elements, such as `<input>`, `<select>`, and `<button>`, have an important place in client-side programming. These HTML elements date from the very beginning of the Web and predate JavaScript itself. HTML forms are the mechanism behind the first generation of web applications, which required no JavaScript at all. User input is gathered in form elements; form submission

sends that input to the server; the server processes the input and generates a new HTML page (usually with new form elements) for display by the client.

HTML form elements are still a great way to gather input from the user, even when form data is processed entirely by client-side JavaScript and never submitted to the server. With server-side programs, a form isn't useful unless it has a Submit button. In client-side programming, on the other hand, a Submit button is never necessary (though it may still be useful). Server-side programs are based on form submissions—they process data in form-sized chunks—and this limits their interactivity. Client-side programs are event based—they can respond to events on individual form elements—and this allows them to be much more responsive. A client-side program might validate the user's input as she types it, for example. Or it might respond to a click on a checkbox by enabling a set of options that are only meaningful when that box is checked.

The subsections that follow explain how to do these kinds of things with HTML forms. Forms are composed of HTML elements, just like any other part of an HTML document, and you can manipulate them with the DOM techniques already explained in this chapter. But form elements were the first ones to be made scriptable, in the earliest days of client-side programming, and they also support some APIs that predate the DOM.

Note that this section is about scripting HTML forms, not about the HTML itself. It assumes that you are already somewhat familiar with the HTML elements (`<input>`, `<textarea>`, `<select>`, and so on) used to define those forms. Nevertheless, Table 15-1 is a quick reference to the most commonly used form elements. You can read more about the form and form element APIs in Part IV, under the entries `Form`, `Input`, `Option`, `Select`, and `TextArea`.

Table 15-1. HTML form elements

HTML element	Type property	Event handler	Description and events
<input type="button"> or <button type="button">	"button"	onclick	A push button
<input type="checkbox">	"checkbox"	onchange	A toggle button without radio button behavior
<input type="file">	"file"	onchange	An input field for entering the name of a file to upload to the web server; `value` property is read-only
<input type="hidden">	"hidden"	none	Data submitted with the form but not visible to the user
<option>	none	none	A single item within a Select object; event handlers are on the Select object, not on individual Option objects
<input type="password">	"password"	onchange	An input field for password entry—typed characters are not visible

HTML element	Type property	Event handler	Description and events
<input type="radio">	"radio"	onchange	A toggle button with radio button behavior—only one selected at a time
<input type="reset"> or <button type="reset">	"reset"	onclick	A push button that resets a form
<select>	"select-one"	onchange	A list or drop-down menu from which one item may be selected (also see <option>)
<select multiple>	"select-multiple"	onchange	A list from which multiple items may be selected (also see <option>)
<input type="submit"> or <button type="submit">	"submit"	onclick	A push button that submits a form
<input type="text">	"text"	onchange	A single-line text entry field; the default <input> element if type attribute is omitted or unrecognized
<textarea>	"textarea"	onchange	A multiline text entry field

15.9.1 Selecting Forms and Form Elements

Forms and the elements they contain can be selected from a document using standard methods like `getElementById()` and `getElementsByTagName()`:

```
var fields = document.getElementById("address").getElementsByTagName("input");
```

In browsers that support `querySelectorAll()`, you might select all radio buttons, or all elements with the same name, from a form with code like this:

```
// All radio buttons in the form with id "shipping"
document.querySelectorAll('#shipping input[type="radio"]');
// All radio buttons with name "method" in form with id "shipping"
document.querySelectorAll('#shipping input[type="radio"][name="method"]');
```

As described in §14.7, §15.2.2, and §15.2.3, however, a `<form>` element with a `name` or `id` attribute can be selected in a number of other ways. A `<form>` with a `name="address"` attribute can be selected in any of these ways:

```
window.address            // Brittle: do not use
document.address          // Only works for forms with name attribute
document.forms.address    // Explicit access to a form with name or id
document.forms[n]         // Brittle: n is the form's numerical position
```

§15.2.3 explained that `document.forms` is an HTMLCollection object that allows form elements to be selected by numerical order, by `id`, or by `name`. Form objects themselves act like HTMLCollections of form elements and can be indexed by name or number. If a form with name "address" has a first element with name "street", you can refer to that form element with any of these expressions:

```
document.forms.address[0]
document.forms.address.street
document.address.street        // only for name="address", not id="address"
```

If you want to be explicit about selecting a form element, you can index the elements property of the form object instead:

```
document.forms.address.elements[0]
document.forms.address.elements.street
```

The id attribute is the generally preferred way to name specific document elements. The name attribute, however, has a special purpose for HTML form submission, and is much more commonly used with forms than with other elements. It is typical for groups of related checkboxes and mandatory for mutually exclusive groups of radioboxes to share a value of the name attribute. Remember that when you index an HTMLCollection with a name and more than one element shares that name, the returned value is an array-like object that contains all matching elements. Consider this form that contains radio buttons for selecting a shipping method:

```
<form name="shipping">
  <fieldset><legend>Shipping Method</legend>
    <label><input type="radio" name="method" value="1st">First-class</label>
    <label><input type="radio" name="method" value="2day">2-day Air</label>
    <label><input type="radio" name="method" value="overnite">Overnight</label>
  </fieldset>
</form>
```

With this form, you might refer to the array of radio button elements like this:

```
var methods = document.forms.shipping.elements.method;
```

Note that <form> elements have an HTML attribute and corresponding JavaScript property named "method", so in this case, we must use the elements property of the form instead of directly accessing the method property. In order to determine which shipping method the user has selected, we'd loop through the form elements in the array and check the checked property of each:

```
var shipping_method;
for(var i = 0; i < methods.length; i++)
    if (methods[i].checked) shipping_method = methods[i].value;
```

We'll see more about the properties, such as checked and value, of form elements in the next section.

15.9.2 Form and Element Properties

The elements[] array described above is the most interesting property of a Form object. The remaining properties of the Form object are of less importance. The action, encoding , method, and target properties correspond directly to the action, encoding, method, and target attributes of the <form> element. These properties and attributes are all used to control how form data is submitted to the web server and where the results are displayed. Client-side JavaScript can set the value of these properties, but they are only useful when the form is actually submitted to a server-side program.

In the days before JavaScript, a form was submitted with a special-purpose Submit button, and form elements had their values reset with a special-purpose Reset button.

The JavaScript Form object supports two methods, `submit()` and `reset()`, that serve the same purpose. Invoking the `submit()` method of a Form submits the form, and invoking `reset()` resets the form elements.

All (or most) form elements have the following properties in common. Some elements have other special-purpose properties that are described later when various types of form elements are considered individually:

`type`
: A read-only string that identifies the type of the form element. For form elements that are defined by an `<input>` tag, this is simply the value of the `type` attribute. Other form elements (such as `<textarea>` and `<select>`) define a `type` property so that they can easily be identified by the same test that distinguishes between `<input>` elements. The second column of Table 15-1 lists the value of this property for each form element.

`form`
: A read-only reference to the Form object in which the element is contained, or `null` if the element is not contained within a `<form>` element.

`name`
: A read-only string specified by the HTML `name` attribute.

`value`
: A read/write string that specifies the "value" contained or represented by the form element. This is the string that is sent to the web server when the form is submitted, and it is only sometimes of interest to JavaScript programs. For Text and Textarea elements, this property contains the text that the user entered. For button elements created with an `<input>` tag (but not those created with a `<button>` tag) this property specifies the text displayed within the button. For radio and checkbox elements, however, the `value` property is not edited or displayed to the user in any way. It is simply a string set by the HTML `value` attribute. It is intended for use in form submission, but it can also be a useful way to associate extra data with a form element. The `value` property is discussed further in the sections on the different categories of form elements, later in this chapter.

15.9.3 Form and Element Event Handlers

Each Form element has an `onsubmit` event handler to detect form submission and an `onreset` event handler to detect form resets. The `onsubmit` handler is invoked just before the form is submitted; it can cancel the submission by returning `false`. This provides an opportunity for a JavaScript program to check the user's input for errors in order to avoid submitting incomplete or invalid data over the network to a server-side program. Note that the `onsubmit` handler is triggered only by a genuine click on a Submit button. Calling the `submit()` method of a form does not trigger the `onsubmit` handler.

The `onreset` event handler is similar to the `onsubmit` handler. It is invoked just before the form is reset, and it can prevent the form elements from being reset by returning

false. Reset buttons are rarely necessary in forms, but if you have one, you might want to make the user confirm the reset:

```
<form...
      onreset="return confirm('Really erase ALL input and start over?')">
  ...
  <button type="reset">Clear and Start Over</button>
</form>
```

Like the onsubmit handler, onreset is triggered only by a genuine Reset button. Calling the reset() method of a form does not trigger onreset.

Form elements typically fire a click or change event when the user interacts with them, and you can handle these events by defining an onclick or onchange event handler. The third column of Table 15-1 specifies the primary event handler for each form element. In general, form elements that are buttons fire a click event when activated (even when this activation happens through the keyboard rather than via an actual mouse click). Other form elements fire a change event when the user changes the value represented by the element. This happens when the user enters text in a text field or selects an option from a drop-down list. Note that this event is not fired every time the user types a key in a text field. It is fired only when the user changes the value of an element and then moves the input focus to some other form element. That is, the invocation of this event handler indicates a completed change. Radio buttons and checkboxes are buttons that have a state, and they fire both click and change events; the change event is the more useful of the two.

Form elements also fire a focus event when they receive keyboard focus and a blur event when they lose it.

An important thing to know about event handlers is that within the code of an event handler, the this keyword refers to the document element that triggered the event (we'll talk about this again in Chapter 17). Since elements within a <form> element have a form property that refers to the containing form, the event handlers of these elements can always refer to the Form object as this.form. Going a step further, this means that an event handler for one form element can refer to a sibling form element named x as this.form.x.

15.9.4 Push Buttons

Buttons are one the most commonly used form elements because they provide a clear visual way to allow the user to trigger some scripted action. A button element has no default behavior of its own, and it is never useful unless it has an onclick event handler. Buttons defined as <input> elements display the plain text of the value attribute. Buttons defined as <button> elements display whatever the element content.

Note that hyperlinks provide the same onclick event handler that buttons do. Use a link when the action to be triggered by the onclick handler can be conceptualized as "following a link"; otherwise, use a button.

Submit and reset elements are just like button elements, but they have default actions (submitting and resetting a form) associated with them. If the onclick event handler returns false, the default action of these buttons is not performed. You can use the onclick handler of a submit element to perform form validation, but it is more common to do this with the onsubmit handler of the Form object itself.

Part IV does not include a Button entry. See Input for details on all form element push buttons, including those created with the <button> element.

15.9.5 Toggle Buttons

The checkbox and radio elements are toggle buttons, or buttons that have two visually distinct states: they can be checked or unchecked. The user can change the state of a toggle button by clicking on it. Radio elements are designed to be used in groups of related elements, all of which have the same value for the HTML name attribute. Radio elements created in this way are mutually exclusive: when you check one, the one that was previously checked becomes unchecked. Checkboxes are also often used in groups that share a name attribute, and when you select these elements using the name as a form property you must remember that you get an array-like object rather than a single element.

Radio and checkbox elements both define a checked property. This read/write boolean value specifies whether the element is currently checked. The defaultChecked property is a boolean that has the value of the HTML checked attribute; it specifies whether the element is checked when the page is first loaded.

Radio and checkbox elements do not display any text themselves and are typically displayed with adjacent HTML text (or with an associated <label> element.) This means that setting the value property of a checkbox or radio element does not alter the visual appearance of the element. You can set value, but this changes only the string that is sent to the web server when the form is submitted.

When the user clicks on a toggle button, the radio or checkbox element triggers its onclick handlers. If the toggle button changes state as the result of the click, it also triggers the onchange event handlers. (Note, however, that radio buttons that change state when the user clicks on a different radio button do not fire an onchange handler.)

15.9.6 Text Fields

Text input fields are probably the most commonly used element in HTML forms and JavaScript programs. They allow the user to enter a short, single-line string of text. The value property represents the text the user has entered. You can set this property to specify explicitly the text that should be displayed in the field.

In HTML5, the placeholder attribute specifies a prompt to be displayed within the field before the user enters anything:

```
Arrival Date: <input type="text" name="arrival" placeholder="yyyy-mm-dd">
```

A text field's onchange event handler is triggered when the user enters new text or edits existing text and then indicates that he is finished editing by moving input focus out of the text field.

The Textarea element is like a text input field element, except that it allows the user to input (and your JavaScript programs to display) multiline text. Textarea elements are created with a `<textarea>` tag using a syntax significantly different from the `<input>` tag that creates a text field. (See TextArea in Part IV.) Nevertheless, the two types of elements behave quite similarly. You can use the value property and onchange event handler of a Textarea element just as you can for a Text element.

An `<input type="password">` element is a modified input field that displays asterisks as the user types into it. As the name indicates, this is useful to allow a user to enter passwords without worrying about others reading over his shoulder. Note that the Password element protects the user's input from prying eyes, but when the form is submitted, that input is not encrypted in any way (unless it is submitted over a secure HTTPS connection), and it may be visible as it is transmitted over the network.

Finally, an `<input type="file">` element allows the user to enter the name of a file to be uploaded to the web server. It is a text field combined with a button that opens a file-chooser dialog box. This file selection element has an onchange event handler, like a regular input field. Unlike an input field, however, the value property of a file selection element is read-only. This prevents malicious JavaScript programs from tricking the user into uploading a file that should not be shared.

The various text input elements define onkeypress, onkeydown, and onkeyup event handlers. You can return `false` from the onkeypress or onkeydown event handlers to prevent the user's keystroke from being recorded. This can be useful, for example, if you want to force the user to enter only digits into a particular text input field. See Example 17-6 for a demonstration of this technique.

15.9.7 Select and Option Elements

The Select element represents a set of options (represented by Option elements) from which the user can select. Browsers typically render Select elements in drop-down menus, but if you specify a `size` attribute with a value greater than 1, they will display the options in a (possibly scrollable) list instead. The Select element can operate in two very distinct ways, and the value of the type property depends on how it is configured. If the `<select>` element has the `multiple` attribute, the user is allowed to select multiple options, and the type property of the Select object is "select-multiple". Otherwise, if the `multiple` attribute is not present, only a single item can be selected, and the type property is "select-one".

In some ways, a select-multiple element is like a set of checkbox elements, and a select-one element is like a set of radio elements. The options displayed by a Select element are not toggle buttons, however: they are defined by `<option>` elements instead. A Select

element defines an options property which is an array-like object that contains Option elements.

When the user selects or deselects an option, the Select element triggers its onchange event handler. For select-one Select elements, the read/write selectedIndex property specifies which one of the options is currently selected. For select-multiple elements, the single selectedIndex property is not sufficient to represent the complete set of selected options. In this case, to determine which options are selected, you must loop through the elements of the options[] array and check the value of the selected property for each Option object.

In addition to its selected property, each Option object has a text property that specifies the string of plain text that appears in the Select element for that option. You can set this property to change the text that is displayed to the user. The value property is also a read/write string that specifies the text to be sent to the web server when the form is submitted. Even if you are writing a pure client-side program and your form never gets submitted, the value property (or its corresponding HTML value attribute) can be a useful place to store any data that you'll need if the user selects a particular option. Note that Option elements do not have form-related event handlers: use the onchange handler of the containing Select element instead.

In addition to setting the text property of Option objects, you can dynamically change the options displayed in a Select element using special features of the options property that date to the early days of client-side scripting. You can truncate the array of Option elements by setting options.length to the desired number of options, and you can remove all Option objects by setting options.length to 0. You can remove an individual Option object from the Select element by setting its spot in the options[] array to null. This deletes the Option object, and any higher elements in the options[] array automatically get moved down to fill the empty spot.

To add new options to a Select element, create an Option object with the Option() constructor and append it to the options[] property with code like this:

```
// Create a new Option object
var zaire = new Option("Zaire",  // The text property
                       "zaire",  // The value property
                       false,    // The defaultSelected property
                       false);   // The selected property

// Display it in a Select element by appending it to the options array:
var countries = document.address.country;  // Get the Select object
countries.options[countries.options.length] = zaire;
```

Keep in mind that these special-purpose Select element APIs are very old. You can more clearly insert and remove option elements with standard calls to Document.createElement(), Node.insertBefore(), Node.removeChild(), and so on.

15.10 Other Document Features

This chapter began with the assertion that it is one of the most important in the book. It is also, by necessity, one of the longest. This final section rounds out the chapter by covering a number of miscellaneous features of the Document object.

15.10.1 Document Properties

This chapter has already introduced Document properties such as `body`, `document Element` , and `forms` that refer to special elements of the document. Documents also define a few other properties of interest:

`cookie`
: A special property that allows JavaScript programs to read and write HTTP cookies. This property is covered in Chapter 20.

`domain`
: A property that allows mutually trusted web servers within the same Internet domain to collaboratively relax same-origin policy security restrictions on interactions between their web pages (see §13.6.2.1).

`lastModified`
: A string that contains the modification date of the document.

`location`
: This property refers to the same Location object as the `location` property of the Window object.

`referrer`
: The URL of the document containing the link, if any, that brought the browser to the current document. This property has the same content as the HTTP `Referer` header, but it is spelled with a double r.

`title`
: The text between the `<title>` and `<title>` tags for this document.

`URL`
: The URL of the document as a read-only String rather than as a Location object. The value of this property is the same as the initial value of `location.href`, but it is not dynamic like the Location object is. If the user navigates to a new fragment identifier within the document, for example, `location.href` will change, but `document.URL` will not.

`referrer` is one of the most interesting of these properties: it contains the URL of the document from which the user linked to the current document. You might use this property with code like this:

```
if (document.referrer.indexOf("http://www.google.com/search?") == 0) {
    var args = document.referrer.substring(ref.indexOf("?")+1).split("&");
    for(var i = 0; i < args.length; i++) {
        if (args[i].substring(0,2) == "q=") {
```

```
            document.write("<p>Welcome Google User. ");
            document.write("You searched for: " +
                           unescape(args[i].substring(2)).replace('+', ' ');
            break;
        }
    }
}
```

The `document.write()` method used in the code above is the subject of the next section.

15.10.2 The document.write() Method

The `document.write()` method was one of the very first scriptable APIs implemented by the Netscape 2 web browser. It was introduced well before the DOM and was the only way to display computed text in a document. It is no longer needed in new code, but you are likely to see it in existing code.

`document.write()` concatenates its string arguments and inserts the resulting string into the document at the location of the script element that invoked it. When the script finishes running, the browser parses the generated output and displays it. The following code, for example, uses `write()` to dynamically output information into an otherwise static HTML document:

```
<script>
  document.write("<p>Document title: " + document.title);
  document.write("<br>URL: " + document.URL);
  document.write("<br>Referred by: " + document.referrer);
  document.write("<br>Modified on: " + document.lastModified);
  document.write("<br>Accessed on: " + new Date());
</script>
```

It is important to understand that you can use the `write()` method to output HTML to the current document only while that document is being parsed. That is, you can call `document.write()` from within top-level code in `<script>` elements only because these scripts are executed as part of the document parsing process. If you place a `document.write()` call within a function definition and then call that function from an event handler, it will not work as you expect—in fact, it will erase the current document and the scripts it contains! (You'll see why shortly.) For similar reasons, you should not use `document.write()` in scripts that have the `defer` or `async` attributes set.

Example 13-3 in Chapter 13 used `document.write()` in this way to generate more complicated output.

You can also use the `write()` method to create entirely new documents in other windows or frames. (When working with multiple windows or frames, however, you must be careful not to violate the same-origin policy.) Your first call to the `write()` method of another document will erase all content in that document. You can call `write()` more than once to build up the new content of the document. The content you pass to `write()` may be buffered (and not displayed) until you terminate the sequence of writes by calling the `close()` method of the document object. This, in essence, tells the HTML

parser that it has reached the end-of-file for the document and that it should finish parsing and display the new document.

It is worth noting that the Document object also supports a `writeln()` method, which is identical to the `write()` method in every way except that it appends a newline after outputting its arguments. This can be useful if you are outputting preformatted text within a `<pre>` element, for example.

The `document.write()` method is not commonly used in modern code: the `innerHTML` property and other DOM techniques provide a better way of adding content to a document. On the other hand, some algorithms do lend themselves nicely to a stream-style I/O API like that provided by the `write()` method. If you are writing code that computes and outputs text while it runs, you might be interested in Example 15-10, which wraps simple `write()` and `close()` methods around the `innerHTML` property of a specified element.

Example 15-10. A streaming API for the innerHTML property

```
// Define a simple "streaming" API for setting the innerHTML of an element.
function ElementStream(elt) {
    if (typeof elt === "string") elt = document.getElementById(elt);
    this.elt = elt;
    this.buffer = "";
}

// Concatenate all arguments and append to the buffer
ElementStream.prototype.write = function() {
    this.buffer += Array.prototype.join.call(arguments, "");
};

// Just like write(), but add a newline
ElementStream.prototype.writeln = function() {
    this.buffer += Array.prototype.join.call(arguments, "") + "\n";
};

// Set element content from buffer and empty the buffer.
ElementStream.prototype.close = function() {
    this.elt.innerHTML = this.buffer;
    this.buffer = "";
};
```

15.10.3 Querying Selected Text

It is sometimes useful to be able to determine what text the user has selected within a document. You can do that with a function like this:

```
function getSelectedText() {
    if (window.getSelection)       // The HTML5 standard API
        return window.getSelection().toString();
    else if (document.selection)   // This is the IE-specific technique.
        return document.selection.createRange().text;
}
```

The standard `window.getSelection()` method returns a Selection object that describes the current selection as a sequence of one or more Range objects. Selection and Range define a fairly complex API that is not commonly used and is not documented in this book. The most important and widely implemented (except in IE) feature of the Selection object is that it has a `toString()` method that returns the plain text content of the selection.

IE defines a different API that is also left undocumented in this book. `document.selection` returns an object that represents the selection. The `createRange()` method of that object returns an IE-specific TextRange object, and the `text` property of that object contains the selected text.

Code like the above can be particularly useful in bookmarklets (§13.2.5.1) that operate on the selected text by looking up a word with a search engine or reference site. The following HTML link, for example, looks up the currently selected text in Wikipedia. When bookmarked, this link and the JavaScript URL it contains become a bookmarklet:

```
<a href="javascript: var q;
      if (window.getSelection) q = window.getSelection().toString();
      else if (document.selection) q = document.selection.createRange().text;
      void window.open('http://en.wikipedia.org/wiki/' + q);">
  Look Up Selected Text In Wikipedia
</a>
```

There is an incompatibility in the selection querying code shown above: the `getSelection()` method of the Window object does not return selected text if it is within an `<input>` or `<textarea>` form element: it only returns text selected from the body of the document itself. The IE `document.selection` property, on the other hand, returns selected text from anywhere in the document.

To obtain the selected text from a text input field or `<textarea>` element, use this code:

```
elt.value.substring(elt.selectionStart, elt.selectionEnd);
```

The `selectionStart` and `selectionEnd` properties are not supported in IE8 or earlier.

15.10.4 Editable Content

We've seen that HTML form elements include text fields and textarea elements that allow the user to enter and edit plain text. Following the lead of IE, all current web browsers also support simple HTML editing functionality: you may have seen this in use on pages (such as blog comment pages) that embed a rich-text editor that includes a toolbar of buttons for setting typographic styles (bold, italic), setting justification, and inserting images and links.

There are two ways to enable this editing functionality. Set the `contenteditable` HTML attribute of any tag or set the `contenteditable` JavaScript property on the corresponding Element to make the content of that element editable. When the user clicks on the

content inside that element, an insertion cursor will appear and the user's keystrokes will be inserted. Here is an HTML element that creates an editable region:

```
<div id="editor" contenteditable>
Click to edit
</div>
```

Browsers may support automatic spell-checking for form fields and `contenteditable` elements. In browsers that support this, checking may be on by default or off by default. Add the `spellcheck` attribute to explicitly turn checking on in browsers that support it. And use `spellcheck=false` to explicitly disable checking (when, for example, a `<textarea>` will display source code or other content with identifiers that do not appear in dictionaries).

You can also make an entire document editable by setting the `designMode` property of the Document object to the string "on". (Set it to "off" to revert to a read-only document.) The `designMode` property does not have a corresponding HTML attribute. You might make the document within an `<iframe>` editable like this (note the use of the `onLoad()` function from Example 13-5):

```
<iframe id="editor" src="about:blank"></iframe>     // Empty iframe
<script>
onLoad(function() {                                 // When document loads,
    var editor = document.getElementById("editor"); // get the iframe document
    editor.contentDocument.designMode = "on";       // and turn editing on.
});
</script>
```

All current browsers support `contenteditable` and `designMode`. They are less compatible, however, when it comes to their actual editing behavior. All browsers allow you to insert and delete text and move the cursor using the mouse and keyboard. In all browsers, the Enter key begins a new line, but different browsers produce different markup. Some begin a new paragraph and others simply insert a `<br/>` element.

Some browsers allow keyboard shortcuts such as Ctrl-B to convert the currently selected text to bold. In other browsers (such as Firefox), standard word processor shortcuts such as Ctrl-B and Ctrl-I are bound to other, browser-related functions and are not available to the text editor.

Browsers define a number of text-editing commands, most of which do not have keyboard shortcuts. To execute these commands, you instead use the `execCommand()` method of the Document object. (Note that this is a method of the Document, not of the element on which the `contenteditable` attribute is set. If a document contains more than one editable element, the command applies to whichever one holds the selection or the insertion cursor.) Commands executed by `execCommand()` are named by strings such as "bold", "subscript", "justifycenter," or "insertimage". The command name is the first argument to `execCommand()`. Some commands require a value argument—"createlink", for example, requires the hyperlink URL. In theory, if the second argument to `execCommand()` is `true`, the browser will automatically prompt the user for whatever value is required. For portability, however, you should prompt the user

yourself, pass `false` as the second argument, and pass the value as the third argument. Here are two example functions that perform edits using `execCommand()`:

```
function bold() { document.execCommand("bold", false, null); }
function link() {
  var url = prompt("Enter link destination");
  if (url) document.execCommand("createlink", false, url);
}
```

The commands supported by `execCommand()` are typically triggered by buttons in a toolbar. A good UI will disable buttons when the command they trigger is not available. Pass a command name to `document.queryCommandSupported()` to find out if it is supported by the browser. Call `document.queryCommandEnabled()` to find out if the command can currently be used. (A command that expects a selected range of text, for example, might be disabled when there is no selection.) Some commands, such as the "bold" and "italic" commands, have a boolean state and can be on or off depending on the current selection or cursor location. These commands are typically represented with a toggle button in a toolbar. Use `document.queryCommandState()` to determine the current state of such a command. Finally, some commands, such as "fontname," have an associated value (a font family name). Query this value with `document.queryCommandValue()`. If the current selection includes text using two different font families, the value of "fontname" will be indeterminate. Use `document.queryCommandIndeterm()` to check for this case.

Different browsers implement different sets of editing commands. A few, such as "bold", "italic", "createlink", "undo," and "redo", are well supported.[6] The HTML5 draft current at the time of this writing defines the following commands. Because they are not universally supported, however, they are not documented here in any detail:

```
bold              insertLineBreak          selectAll
createLink        insertOrderedList        subscript
delete            insertUnorderedList      superscript
formatBlock       insertParagraph          undo
forwardDelete     insertText               unlink
insertImage       italic                   unselect
insertHTML        redo
```

If you need rich-text editing functionality for your web application, you will probably want to adopt a prebuilt solution that addresses the various differences between browsers. A number of such editor components can be found online.[7] It is worth noting that the editing functionality built into browsers is powerful enough to allow a user to enter small amounts of rich text, but it is too simple for any kind of serious document editing. In particular, note that the HTML markup generated by these editors is likely to be quite messy.

6. See *http://www.quirksmode.org/dom/execCommand.html* for a list of interoperable commands.

7. The YUI and Dojo frameworks include editor components. A list of other choices can be found at *http://en.wikipedia.org/wiki/Online_rich-text_editor*.

Once the user has edited the content of an element that has the `contenteditable` attribute set, you can use the `innerHTML` property to obtain the HTML markup of the edited content. What you do with that rich text is up to you. You might store it in a hidden form field and send it to a server by submitting the form. You might use the techniques described in Chapter 18 to send the edited text directly to a server. Or you might use the techniques shown in Chapter 20 to save the user's edits locally.

CHAPTER 16
Scripting CSS

Cascading Style Sheets (CSS) is a standard for specifying the visual presentation of HTML documents. CSS is intended for use by graphic designers: it allows a designer to precisely specify fonts, colors, margins, indentation, borders, and even the position of document elements. But CSS is also of interest to client-side JavaScript programmers because CSS styles can be scripted. Scripted CSS enables a variety of interesting visual effects: you can create animated transitions where document content "slides in" from the right, for example, or create an expanding and collapsing outline list in which the user can control the amount of information that is displayed. When first introduced, scripted visual effects like these were revolutionary. The JavaScript and CSS techniques that produced them were loosely referred to as Dynamic HTML or DHTML, a term that has since fallen out of favor.

CSS is a complex standard that, at the time of this writing, is undergoing active development. CSS is a book-length topic of its own and complete coverage is well beyond the scope of *this* book.[1] In order to understand CSS scripting, however, it is necessary to be familiar with CSS basics and with the most commonly scripted styles, so this chapter begins with a concise overview of CSS, followed by an explanation of key styles that are most amenable to scripting. After these two introductory sections, the chapter moves on to explain how to script CSS. §16.3 explains the most common and important technique: altering the styles that apply to individual document elements using the HTML `style` attribute. Although an element's `style` attribute can be used to set styles, it is not useful for querying an element's style. §16.4 explains how to query the *computed style* of any element. §16.5 explains how to alter many styles at once by altering the `style` attribute of an element. It is also possible, though less common, to script stylesheets directly, and §16.6 shows how to enable and disable stylesheets, alter the rules of existing stylesheets, and add new stylesheets.

1. But see *CSS: The Definitive Guide* by Eric Meyer (O'Reilly), for example.

16.1 Overview of CSS

There are many variables in the visual display of an HTML document: fonts, colors, spacing, and so on. The CSS standard enumerates these variables and calls them style *properties*. CSS defines properties that specify fonts, colors, margins, borders, background images, text alignment, element size, and element position. To define the visual appearance of HTML elements, we specify the value of CSS properties. To do this, follow the name of a property with a colon and a value:

```
font-weight: bold
```

In order to fully describe the visual presentation of an element, we usually need to specify the value of more than one property. When multiple name:value pairs are required, they are separated from one another by semicolons:

```
margin-left: 10%;      /* left margin is 10% of page width */
text-indent: .5in;     /* indent by 1/2 inch */
font-size: 12pt;       /* 12 point font size */
```

As you can see, CSS ignores comments between /* and */. It does not support comments that begin with //, however.

There are two ways to associate a set of CSS property values with the HTML elements whose presentation they define. The first is by setting the `style` attribute of an individual HTML element. This is called the inline style:

```
<p style="margin: 20px; border: solid red 2px;">
This paragraph has increased margins and is
surrounded by a rectangular red border.
</p>
```

It is usually much more useful, however, to separate CSS styles from individual HTML elements and define them in a *stylesheet*. A stylesheet associates sets of style properties with sets of HTML elements that are described using *selectors*. A selector specifies or "selects" one or more elements of a document, based on element ID, class, or tag name, or on more specialized criteria. Selectors were introduced in §15.2.5, which also showed how to use `querySelectorAll()` to obtain the set of elements that match the selector.

The basic element of a CSS stylesheet is a style rule, which consists of a selector followed by a set of CSS properties and their values, enclosed in curly braces. A stylesheet can contain any number of style rules:

```
p {                     /* the selector "p" matches all <p> elements */
    text-indent: .5in;  /* indent the first line by .5 inches */
}

.warning {                    /* Any element with class="warning" */
    background-color: yellow; /* gets a yellow background */
    border: solid black 5px;  /* and a big black border */
}
```

A CSS stylesheet can be associated with an HTML document by enclosing it within <style> and </style> tags within the <head> of a document. Like the <script> element, the <style> element parses its content specially and does not treat it as HTML:

```
<html>
<head><title>Test Document</title>
<style>
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
p { font-size: 24px; }
</style>
</head>
<body><p>Testing, testing</p>
</html>
```

When a stylesheet is to be used by more than one page on a website, it is usually better to store it in its own file, without any enclosing HTML tags. This CSS file can then be included in the HTML page. Unlike the <script> element, however, the <style> element does not have a src attribute. To include a stylesheet in an HTML page, use a <link> in the <head> of a document:

```
<head>
  <title>Test Document</title>
  <link rel="stylesheet" href="mystyles.css" type="text/css">
</head>
```

That, in a nutshell, is how CSS works. There are a few other points about CSS that are worth understanding, however, and the subsections that follow explain them.

16.1.1 The Cascade

Recall that the C in CSS stands for "cascading." This term indicates that the style rules that apply to any given element in a document can come from a "cascade" of different sources:

- The web browser's default stylesheet
- The document's stylesheets
- The style attribute of individual HTML elements

Styles from the style attribute override styles from stylesheets. And styles from a document's stylesheets override the browser's default styles, of course. The visual presentation of any given element may be a combination of style properties from all three sources. An element may even match more than one selector within a stylesheet, in which case the style properties associated with all of those selectors are applied to the element. (If different selectors define different values for the same style property, the value associated with the most specific selector overrides the value associated with less specific selectors, but the details are beyond the scope of this book.)

To display any document element, the web browser must combine the style attribute of that element with styles from all the matched selectors in the document stylesheets. The result of this computation is the actual set of style properties and values that are

used to display the element. This set of values is known as the *computed style* of the element.

16.1.2 CSS History

CSS is a relatively old standard. CSS1 was adopted in December 1996 and defines properties for specifying colors, fonts, margins, borders, and other basic styles. Browsers as old as Netscape 4 and Internet Explorer 4 include substantial support for CSS1. The second edition of the standard, CSS2, was adopted in May 1998; it defines a number of more advanced features, most notably support for absolute positioning of elements. CSS2.1 clarifies and corrects CSS2, and it removes features that browser vendors never actually implemented. Current browsers have essentially complete support for CSS2.1, although versions of IE prior to IE8 have notable omissions.

Work continues on CSS. For version 3, the CSS specification has been broken into various specialized modules that are going through the standardization process separately. You can find the CSS specifications and working drafts at *http://www.w3.org/Style/CSS/current-work*.

16.1.3 Shortcut Properties

Certain style properties that are commonly used together can be combined using special shortcut properties. For example, the `font-family`, `font-size`, `font-style`, and `font-weight` properties can all be set at once using a single `font` property with a compound value:

```
font: bold italic 24pt helvetica;
```

Similarly, the `border`, `margin`, and `padding` properties are shortcuts for properties that specify borders, margins, and padding (space between the border and element content) for each of the individual sides of an element. For example, instead of using the `border` property, you can use `border-left`, `border-right`, `border-top`, and `border-bottom` properties to specify the border of each side separately. In fact, each of these properties is itself a shortcut. Instead of specifying `border-top`, you can specify `border-top-color`, `border-top-style`, and `border-top-width`.

16.1.4 Nonstandard Properties

When browser vendors implement nonstandard CSS properties, they prefix the property names with a vendor-specific string. Firefox uses `moz-`, Chrome uses `-webkit-`, and IE uses `-ms-`. Browser vendors do this even when implementing properties that are intended for future standardization. One example is the `border-radius` property, which specifies rounded corners for elements. This was implemented experimentally in Firefox 3 and Safari 4 using prefixes. Once the standard had matured sufficiently, Firefox 4 and Safari 5 removed the prefix and supported `border-radius` directly. (Chrome and Opera have supported it for a long time with no prefix. IE9 also supports it without a prefix, but IE8 did not support it, even with a prefix.)

When working with CSS properties that have different names in different browsers, you may find it helpful to define a class for that property:

```
.radius10 {
  border-radius: 10px;          /* for current browsers */
  -moz-border-radius: 10px;     /* for Firefox 3.x */
  -webkit-border-radius: 10px;  /* For Safari 3.2 and 4 */
}
```

With a class like this defined, you can add "radius10" to the `class` attribute of any element to give it a border radius of 10 pixels.

16.1.5 CSS Example

Example 16-1 is an HTML file that defines and uses a stylesheet. It demonstrates tag name, class, and ID-based selectors, and also has an example of an inline style defined with the `style` attribute. Figure 16-1 shows how this example is rendered in a browser.

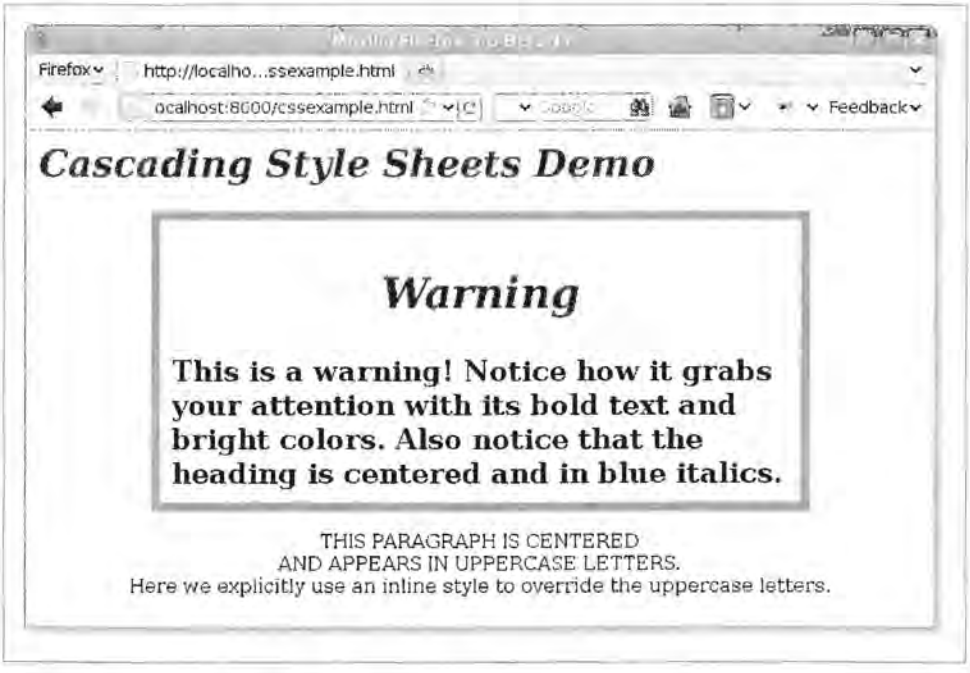


Figure 16-1. A web page styled with CSS

Example 16-1. Defining and using Cascading Style Sheets

```
<head>
<style type="text/css">
/* Specify that headings display in blue italic text. */
h1, h2 { color: blue; font-style: italic }

/*
```

```
 * Any element of class="WARNING" displays in big bold text with large margins
 * and a yellow background with a fat red border.
 */
.WARNING {
          font-weight: bold;
          font-size: 150%;
          margin: 0 1in 0 1in; /* top right bottom left */
          background-color: yellow;
          border: solid red 8px;
          padding: 10px;       /* 10 pixels on all 4 sides */
}

/*
 * Text within an h1 or h2 heading within an element with class="WARNING"
 * should be centered, in addition to appearing in blue italics.
 */
.WARNING h1, .WARNING h2 { text-align: center }

/* The single element with id="special" displays in centered uppercase. */
#special {
      text-align: center;
      text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Cascading Style Sheets Demo</h1>

<div class="WARNING">
<h2>Warning</h2>
This is a warning!
Notice how it grabs your attention with its bold text and bright colors.
Also notice that the heading is centered and in blue italics.
</div>

<p id="special">
This paragraph is centered<br>
and appears in uppercase letters.<br>
<span style="text-transform: none">
Here we explicitly use an inline style to override the uppercase letters.
</span>
</p>
```

Cutting-Edge CSS

As I write this chapter, CSS is in the midst of a revolution, with browser vendors implementing powerful new styles such as `border-radius`, `text-shadow`, `box-shadow`, and `column-count`. Another revolutionary new feature of CSS is *web fonts*: the ability to download and use custom fonts with a CSS `@font-face` rule. (See *http://code.google.com/webfonts* for a selection of fonts free for use on the Web and an easy mechanism for downloading them from Google's servers.)

Another revolutionary development in CSS is CSS Transitions. This is a draft specification that can automatically turn any scripted change to a CSS style into a smoothly

animated transition. (When widely implemented, it will largely obviate the need for CSS animation code like that shown in §16.3.1.) CSS Transitions is implemented in current browsers other than IE, but its style properties are still prefixed with vendor-specific strings. CSS Animations is a related proposal that uses CSS Transitions as a starting point for defining more complex animation sequences. CSS Animations are currently only implemented by Webkit browsers. Neither transitions nor animations are covered in this chapter, but they are technologies that you, as a web developer, should be aware of.

Another CSS draft that web developers should be aware of is CSS Transforms, which allows arbitrary 2D affine transforms (rotation, scaling, translation, or any combination expressed as a matrix) to be applied to any element. All current browsers (including IE9 and later) support the draft, with vendor prefixes. Safari even supports an extension that allows 3D transformations, but it is unclear whether other browsers will follow their lead.

16.2 Important CSS Properties

For client-side JavaScript programmers, the most important features of CSS are the properties that specify the visibility, size, and precise position of individual elements of a document. Other CSS properties allow you to specify stacking order, transparency, clipping region, margins, padding, borders, and colors. In order to script CSS, it is important to understand how these style properties work. They are summarized in Table 16-1 and documented in more detail in the sections that follow.

Table 16-1. Important CSS style properties

Property	Description
`position`	Specifies the type of positioning applied to an element
`top, left`	Specify the position of the top and left edges of an element
`bottom, right`	Specify the position of the bottom and right edges of an element
`width, height`	Specify the size of an element
`z-index`	Specifies the "stacking order" of an element relative to any overlapping elements; defines a third dimension of element positioning
`display`	Specifies how and whether an element is displayed
`visibility`	Specifies whether an element is visible
`clip`	Defines a "clipping region" for an element; only portions of the element within this region are displayed
`overflow`	Specifies what to do if an element is bigger than the space allotted for it
`margin, border, padding`	Specify spacing and borders for an element.
`background`	Specifies the background color or image of an element.
`opacity`	Specifies how opaque (or translucent) an element is. This is a CSS3 property, supported by some browsers. A working alternative exists for IE.

16.2.1 Positioning Elements with CSS

The CSS `position` property specifies the type of positioning applied to an element. Here are the four possible values for this property:

`static`
: This is the default value and specifies that the element is positioned according to the normal flow of document content (for most Western languages, this is left to right and top to bottom). Statically positioned elements cannot be positioned with `top`, `left`, and other properties. To use CSS positioning techniques with a document element, you must first set its `position` property to one of the other three values.

`absolute`
: This value allows you to specify the position of an element relative to its containing element. Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements. An absolutely positioned element is positioned either relative to its nearest positioned ancestor or relative to the document itself.

`fixed`
: This value allows you to specify an element's position with respect to the browser window. Elements with `fixed` positioning are always visible and do not scroll with the rest of the document. Like absolutely positioned elements, fixed-position elements are independent of all others and are not part of the document flow. Fixed positioning is supported in most modern browsers but is not available in IE6.

`relative`
: When the `position` property is set to `relative`, an element is laid out according to the normal flow, and its position is then adjusted relative to its position in the normal flow. The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in that space, nor are they "pushed away" from the new position of the element.

Once you have set the `position` property of an element to something other than `static`, you can specify the position of that element with some combination of the `left`, `top`, `right`, and `bottom` properties. The most common positioning technique uses the `left` and `top` properties to specify the distance from the left edge of the containing element (usually the document itself) to the left edge of the element and the distance from the top edge of the container to the top edge of the element. For example, to place an element 100 pixels from the left and 100 pixels from the top of the document, you can specify CSS styles in a `style` attribute as follows:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

If an element uses absolute positioning, its `top` and `left` properties are interpreted relative to the closest ancestor element that has its `position` property set to something other than `static`. If an absolutely positioned element has no positioned ancestor, the `top` and `left` properties are measured in document coordinates—they are offsets from

the top-left corner of the document. If you wish to absolutely position an element relative to a container that is part of the normal document flow, use `position:relative` for the container and specify a `top` and `left` position of `0px`. This makes the container dynamically positioned but leaves it at its normal place in the document flow. Any absolutely positioned children are then positioned relative to the container position.

Although it is most common to specify the position of the upper-left corner of an element with `left` and `top`, you can also use `right` and `bottom` to specify the position of the bottom and right edges of an element relative to the bottom and right edges of the containing element. For example, to position an element so that its bottom-right corner is at the bottom-right of the document (assuming it is not nested within another dynamic element), use the following styles:

```
position: absolute; right: 0px; bottom: 0px;
```

To position an element so that its top edge is 10 pixels from the top of the window and its right edge is 10 pixels from the right of the window, and so that it does not scroll with the document, you might use these styles:

```
position: fixed; right: 10px; top: 10px;
```

In addition to the position of elements, CSS allows you to specify their size. This is most commonly done by providing values for the `width` and `height` style properties. For example, the following HTML creates an absolutely positioned element with no content. Its `width`, `height`, and `background-color` properties make it appear as a small blue square:

```
<div style="position: absolute; top: 10px; left: 10px;
            width: 10px; height: 10px; background-color: blue">
</div>
```

Another way to specify the width of an element is to specify a value for both the `left` and `right` properties. Similarly, you can specify the height of an element by specifying both `top` and `bottom`. If you specify a value for `left`, `right`, and `width`, however, the `width` property overrides the `right` property; if the height of an element is overconstrained, `height` takes priority over `bottom`.

Bear in mind that it is not necessary to specify the size of every dynamic element. Some elements, such as images, have an intrinsic size. Furthermore, for dynamic elements that contain text or other flowed content, it is often sufficient to specify the desired width of the element and allow the height to be determined automatically by the layout of the element's content.

CSS requires position and dimension properties to be specified with a unit. In the examples above, the position and size properties were specified with the suffix "px," which stands for pixels. You can also use inches ("in"), centimeters ("cm"), points ("pt"), and ems ("em"; a measure of the line height for the current font).

Instead of specifying absolute positions and sizes using the units shown above, CSS also allows you to specify the position and size of an element as a percentage of the size of the containing element. For example, the following HTML creates an empty element with a black border that is half as wide and half as high as the containing element (or the browser window) and centered within that element:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
            border: 2px solid black">
</div>
```

16.2.1.1 The third dimension: z-index

You've seen that the `left`, `top`, `right`, and `bottom` properties can specify the X and Y coordinates of an element within the two-dimensional plane of the containing element. The `z-index` property defines a kind of third dimension: it allows you to specify the stacking order of elements and indicate which of two or more overlapping elements is drawn on top of the others. The `z-index` property is an integer. The default value is zero, but you may specify positive or negative values. When two or more elements overlap, they are drawn in order from lowest to highest `z-index`; the element with the highest `z-index` appears on top of all the others. If overlapping elements have the same `z-index`, they are drawn in the order in which they appear in the document so that the last overlapping element appears on top.

Note that `z-index` stacking applies only to sibling elements (i.e., elements that are children of the same container). If two elements that are not siblings overlap, setting their individual `z-index` properties does not allow you to specify which one is on top. Instead, you must specify the `z-index` property for the two sibling containers of the two overlapping elements.

Nonpositioned elements (i.e., elements with default `position:static` positioning) are always laid out in a way that prevents overlaps, so the `z-index` property does not apply to them. Nevertheless, they have a default `z-index` of zero, which means that positioned elements with a positive `z-index` appear on top of the normal document flow and positioned elements with a negative `z-index` appear beneath the normal document flow.

16.2.1.2 CSS positioning example: Shadowed text

The CSS3 specification includes a `text-shadow` property to produce drop-shadow effects under text. It is supported by a number of current browsers, but you can use CSS positioning properties to achieve a similar effect, as long as you are willing to repeat and restyle the text to produce a shadow:

```
<!-- The text-shadow property produces shadows automatically -->
<span style="text-shadow: 3px 3px 1px #888">Shadowed</span>

<!-- Here's how we can produce a similar effect with positioning. -->
<span style="position:relative;">
  Shadowed     <!-- This is the text that casts the shadow. -->
```

```
  <span style="position:absolute; top:3px; left:3px; z-index:-1; color: #888">
    Shadowed   <!-- This is the shadow -->
  </span>
</span>
```

The text to be shadowed is enclosed in a relatively positioned element. There are no position properties set, so the text appears at its normal position in the flow. The shadow is in an absolutely positioned inside (and therefore positioned relatively to) the relatively positioned . The z-index property ensures that the shadow appears underneath the text that produces it.

16.2.2 Borders, Margins and Padding

CSS allows you to specify borders, margins, and padding around any element. The border of an element is a rectangle (or rounded rectangle in CSS3) drawn around (or partially around) it. CSS properties allow you to specify the style, color, and thickness of the border:

```
border: solid black 1px; /* border is drawn with a solid, black 1-pixel line */
border: 3px dotted red;  /* border is drawn in 3-pixel red dots */
```

It is possible to specify the border width, style, and color using individual CSS properties, and it is also possible to specify the border for individual sides of an element. To draw a line beneath an element, for example, simply specify its `border-bottom` property. It is even possible to specify the width, style, or color of a single side of an element with properties such as `border-top-width` and `border-left-color`.

In CSS3, you can round all corners of a border with the `border-radius` property, and you can round individual corners with more explicit property names. For example:

```
border-top-right-radius: 50px;
```

The `margin` and `padding` properties both specify blank space around an element. The important difference is that `margin` specifies space outside the border, between the border and adjacent elements, and `padding` specifies space inside the border, between the border and the element content. A margin provides visual space between a (possibly bordered) element and its neighbors in the normal document flow. Padding keeps element content visually separated from its border. If an element has no border, padding is typically not necessary. If an element is dynamically positioned, it is not part of the normal document flow, and its margins are irrelevant.

You can specify the margin and padding of an element with the `margin` and `padding` properties:

```
margin: 5px; padding: 5px;
```

You can also specify margins and paddings for individual sides of an element:

```
margin-left: 25px;
padding-bottom: 5px;
```

Or you can specify margin and padding values for all four edges of an element with the `margin` and `padding` properties. You specify the top values first and then proceed

clockwise: top, right, bottom, and left. For example, the following code shows two equivalent ways to set different padding values for each of the four sides of an element:

```
padding: 1px 2px 3px 4px;
/* The previous line is equivalent to the following lines. */
padding-top: 1px;
padding-right: 2px;
padding-bottom: 3px;
padding-left: 4px;
```

The `margin` property works in the same way.

16.2.3 The CSS Box Model and Positioning Details

The `margin`, `border`, and `padding` style properties described above are not properties that you are likely to script very frequently. The reason that they are mentioned here is that margins, borders, and padding are part of the CSS *box model*, and you have to understand the box model in order to truly understand the CSS positioning properties.

Figure 16-2 illustrates the CSS box model and visually explains the meaning of the `top`, `left`, `width`, and `height` for elements that have borders and padding.

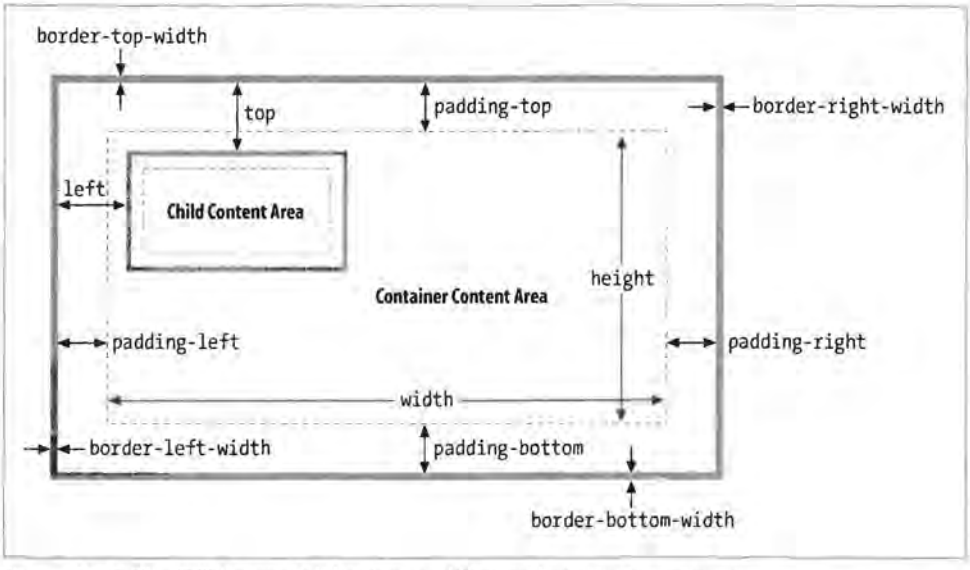


Figure 16-2. The CSS box model: borders, padding, and positioning properties

Figure 16-2 shows an absolutely positioned element nested inside a positioned container element. Both the container and the contained elements have borders and padding, and the figure illustrates the CSS properties that specify padding and border width for each side of the container element. Notice that no margin properties are shown: margins aren't relevant to absolutely positioned elements.

Figure 16-2 contains other, more important information as well. First, `width` and `height` specify the size of an element's content area only; they do not include any additional space required for the element's padding or border (or margins). To determine the full on-screen size of an element with a border, you must add the left and right padding and left and right border widths to the element width, and you must add the top and bottom padding and top and bottom border widths to the element's height.

Second, the `left` and `top` properties specify the distance from the inside of the container's border to the outside of the positioned element's border. These properties do not measure from the upper-left corner of the content area of the container, but from the upper-left corner of the container's padding. Similarly, the `right` and `bottom` properties measure from the lower-right corner of the padding.

Here's an example that might make this clearer. Suppose you've created a dynamically positioned container element that has 10 pixels of padding all the way around its content area and a 5-pixel border all the way around the padding. Now suppose you dynamically position a child element inside this container. If you set the `left` property of the child to "0 px", you'll discover that the child is positioned with its left edge right up against the inner edge of the container's border. With this setting, the child overlaps the container's padding, which presumably was supposed to remain empty (since that is the purpose of padding). If you want to position the child element in the upper left corner of the container's content area, you should set both the `left` and `top` properties to "10px".

16.2.3.1 The border-box model and the box-sizing property

The standard CSS box model specifies that the `width` and `height` style properties give the size of the content area and do not include padding and borders. We might call this box model the "content-box model." There are exceptions to the content-box model in old versions of IE and also in new versions of CSS. Before IE6, and when IE6, 7, or 8 displays a page in "quirks mode" (when the page has no `<!DOCTYPE>` or has an insufficiently strict doctype), the `width` and `height` properties do include the padding and border widths.

IE's behavior is a bug, but IE's nonstandard box model is often quite useful. Recognizing this, CSS3 introduces a `box-sizing` property. The default value is `content-box`, which specifies the standard box model described above. If you instead specify `box-sizing:border-box`, the browser will use the IE box model for that element, and the `width` and `height` properties will include border and padding. The `border-box` model is particularly useful when you want to specify the overall size of an element as a percentage but also want to specify the border and padding size in pixels:

```
<div style="box-sizing:border-box; width: 50%;
            padding: 10px; border: solid black 2px;">
```

The box-sizing property is supported by all current browsers but is not yet universally implemented without a prefix. In Chrome and Safari, use -webkit-box-sizing. In Firefox, use -moz-box-sizing. In Opera and IE8 and later, you can use box-sizing without any prefix.

A future CSS3 alternative to the border-box model is the use of calculated values for box dimensions:

```
<div style="width: calc(50%-12px); padding: 10px; border: solid black 2px;">
```

Calculated CSS values with calc() are supported in IE9 and in Firefox 4 (as -moz-calc()).

16.2.4 Element Display and Visibility

Two CSS properties affect the visibility of a document element: visibility and display . The visibility property is simple: when the property is set to the value hidden, the element is not shown; when it is set to the value visible, the element is shown. The display property is more general and is used to specify the type of display an item receives. It specifies whether an element is a block element, an inline element, a list item, or so on. When display is set to none, however, the affected element is not displayed, or even laid out, at all.

The difference between the visibility and display style properties has to do with their effect on elements that use static or relative positioning. For an element that appears in the normal layout flow, setting visibility to hidden makes the element invisible but reserves space for it in the document layout. Such an element can be repeatedly hidden and shown without changing the document layout. If an element's display property is set to none, however, no space is allocated for it in the document layout; elements on either side of it close up as if it were not there. The display property is useful, for example, when creating expanding and collapsing outlines.

visibility and display have equivalent effects when used with absolute- or fixed-position elements because these elements are not part of the document layout. The visibility property is generally preferred for hiding and showing positioned elements, however.

Note that it doesn't make much sense to use visibility or display to make an element invisible unless you are going to use JavaScript to dynamically set these properties and make the element visible at some point! We'll see how to do this later in the chapter.

16.2.5 Color, Transparency, and Translucency

You can specify the color of text contained in a document element with the CSS color property. And you can specify the background color of any element with the background-color property. Earlier, we saw that you can specify the color of an element's border with border-color or with the shortcut property border.

The discussion of borders included examples that specified border colors using the English names of common colors such as "red" and "black". CSS supports a number of these English color names, but the more general syntax for specifying colors in CSS is to use hexadecimal digits to specify the red, green, and blue components of a color. You can use either one or two digits per component. For example:

```
#000000      /* black */
#fff         /* white */
#f00         /* bright red */
#404080      /* dark unsaturated blue */
#ccc         /* light gray */
```

CSS3 also defines syntaxes for specifying colors in the RGBA color space (red, green, and blue values plus an *alpha* value that specifies the transparency of the color). RGBA is supported by all modern browsers except IE, and support is expected in IE9. CSS3 also defines support for HSL (hue-saturation-value) and HSLA color specifications. Again, these are supported by Firefox, Safari, and Chrome, but not IE.

CSS allows you to specify the exact position, size, background color, and border color of elements; this gives you a rudimentary graphics capability for drawing rectangles and (when the height and width are reduced) horizontal and vertical lines. The last edition of this book included a bar chart example using CSS graphics, but it has been replaced in this book by extended coverage of the `<canvas>` element. (See Chapter 21 for more on scripted client-side graphics.)

In addition to the `background-color` property, you can also specify images to be used as the background of an element. The `background-image` property specifies the image to use, and the `background-attachment`, `background-position`, and `background-repeat` properties specify further details about how this image is drawn. The shortcut property `background` allows you to specify these properties together. You can use these background image properties to create interesting visual effects, but those are beyond the scope of this book.

It is important to understand that if you do not specify a background color or image for an element, that element's background is usually transparent. For example, if you absolutely position a `<div>` over some existing text in the normal document flow, that text will, by default, show through the `<div>` element. If the `<div>` contains its own text, the letters may overlap and become an illegible jumble. Not all elements are transparent by default, however. Form elements don't look right with a transparent background, for example, and elements such as `<button>` have a default background color. You can override this default with the `background-color` property, and you can even explicitly set it to "transparent" if you desire.

The transparency we've been discussing so far is all-or-none: an element either has a transparent background or an opaque background. It is also possible to specify that an element (both its background and its foreground content) is translucent. (See Figure 16-3 for an example.) You do this with the CSS3 `opacity` property. The value of this property is a number between 0 and 1, where 1 means 100 percent opaque (the

default) and 0 means 0% opaque (or 100% transparent). The opacity property is supported by all current browsers except IE. IE provides a work-alike alternative through its IE-specific filter property. To make an element 75 percent opaque, you can use the following CSS styles:

```
opacity: .75;              /* standard CSS3 style for transparency */
filter: alpha(opacity=75); /* transparency for IE; note no decimal point */
```

16.2.6 Partial Visibility: overflow and clip

The visibility property allows you to completely hide a document element. The overflow and clip properties allow you to display only part of an element. The overflow property specifies what happens when the content of an element exceeds the size specified (with the width and height style properties, for example) for the element. The allowed values and their meanings for this property are as follows:

visible
: Content may overflow and be drawn outside of the element's box if necessary. This is the default.

hidden
: Content that overflows is clipped and hidden so that no content is ever drawn outside the region defined by the size and positioning properties.

scroll
: The element's box has permanent horizontal and vertical scrollbars. If the content exceeds the size of the box, the scrollbars allow the user to scroll to view the extra content. This value is honored only when the document is displayed on a computer screen; when the document is printed on paper, for example, scrollbars obviously do not make sense.

auto
: Scrollbars are displayed only when content exceeds the element's size rather than being permanently displayed.

While the overflow property allows you to specify what happens when an element's content is bigger than the element's box, the clip property allows you to specify exactly which portion of an element should be displayed, whether or not the element overflows. This property is especially useful for scripted effects in which an element is progressively displayed or uncovered.

The value of the clip property specifies the clipping region for the element. In CSS2, clipping regions are rectangular, but the syntax of the clip property leaves open the possibility that future versions of the standard will support clipping shapes other than rectangles. The syntax of the clip property is:

```
rect(top right bottom left)
```

The *top*, *right*, *bottom*, and *left* values specify the boundaries of the clipping rectangle relative to the upper-left corner of the element's box. For example, to display only a 100 × 100-pixel portion of an element, you can give that element this style attribute:

```
style="clip: rect(0px 100px 100px 0px);"
```

Note that the four values within the parentheses are length values and must include a unit specification, such as px for pixels. Percentages are not allowed. Values can be negative to specify that the clipping region extends beyond the box specified for the element. You can also use the auto keyword for any of the four values to specify that the edge of the clipping region is the same as the corresponding edge of the element's box. For example, you can display just the leftmost 100 pixels of an element with this style attribute:

```
style="clip: rect(auto 100px auto auto);"
```

Note that there are no commas between the values, and the edges of the clipping region are specified in clockwise order from the top edge. To turn clipping off, set the clip property to auto.

16.2.7 Example: Overlapping Translucent Windows

This section concludes with an example that demonstrates many of the CSS properties discussed here. Example 16-2 uses CSS to create the visual effect of scrolling, overlapping, translucent windows within the browser window. Figure 16-3 shows how it looks.

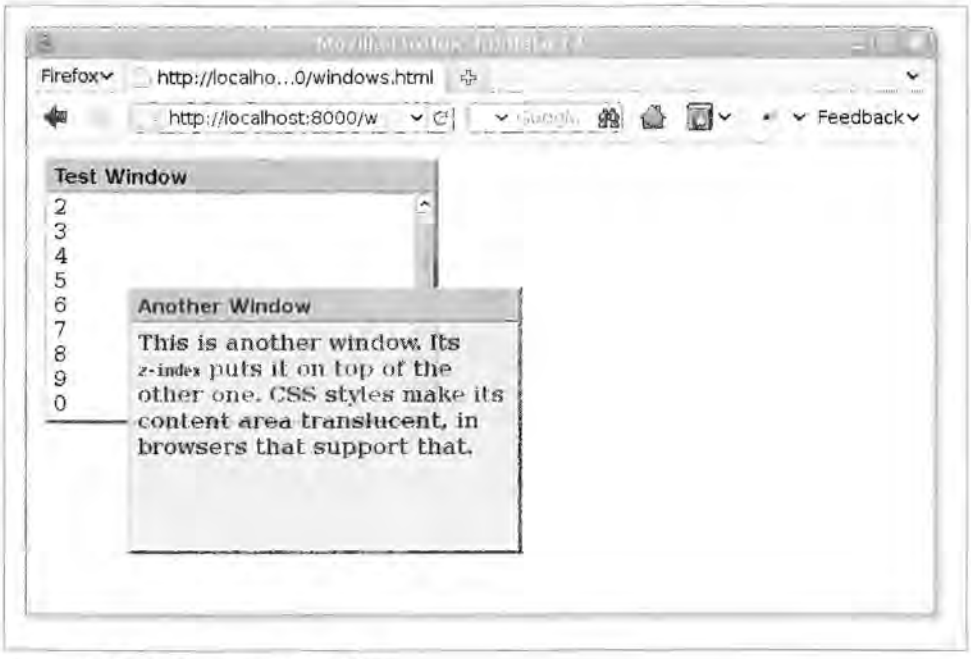


Figure 16-3. Windows created with CSS

The example contains no JavaScript code and no event handlers, so there is no way to interact with the windows (other than to scroll them), but it is a useful demonstration of the powerful effects that can be achieved with CSS.

Example 16-2. Displaying windows with CSS

```
<!DOCTYPE html>
<head>
<style type="text/css">
/**
 * This is a CSS stylesheet that defines three style rules that we use
 * in the body of the document to create a "window" visual effect.
 * The rules use positioning properties to set the overall size of the window
 * and the position of its components. Changing the size of the window
 * requires careful changes to positioning properties in all three rules.
 **/
div.window {  /* Specifies size and border of the window */
    position: absolute;         /* The position is specified elsewhere */
    width: 300px; height: 200px;/* Window size, not including borders */
    border: 3px outset gray;    /* Note 3D "outset" border effect */
}

div.titlebar {  /* Specifies position, size, and style of the titlebar */
    position: absolute;         /* It's a positioned element */
    top: 0px; height: 18px;     /* Titlebar is 18px + padding and borders */
    width: 290px;               /* 290 + 5px padding on left and right = 300 */
    background-color: #aaa;     /* Titlebar color */
    border-bottom: groove gray 2px;  /* Titlebar has border on bottom only */
    padding: 3px 5px 2px 5px;   /* Values clockwise: top, right, bottom, left */
    font: bold 11pt sans-serif; /* Title font */
}

div.content {  /* Specifies size, position and scrolling for window content */
    position: absolute;         /* It's a positioned element */
    top: 25px;                  /* 18px title+2px border+3px+2px padding */
    height: 165px;              /* 200px total - 25px titlebar - 10px padding*/
    width: 290px;               /* 300px width - 10px of padding */
    padding: 5px;               /* Allow space on all four sides */
    overflow: auto;             /* Give us scrollbars if we need them */
    background-color: #fff;     /* White background by default */
}

div.translucent { /* this class makes a window partially transparent */
    opacity: .75;               /* Standard style for transparency */
    filter: alpha(opacity=75);  /* Transparency for IE */
}
</style>
</head>

<body>
<!-- Here is how we define a window: a "window" div with a titlebar and -->
<!-- content div nested inside. Note how position is specified with -->
<!-- a style attribute that augments the styles from the stylesheet. -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Test Window</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- Lots of lines to -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- demonstrate scrolling-->
</div>
</div>
```

```
<!-- Here's another window with different position, color, and font weight -->
<div class="window" style="left: 75px; top: 110px; z-index: 20;">
<div class="titlebar">Another Window</div>
<div class="content translucent"
     style="background-color:#ccc; font-weight:bold;">
This is another window. Its <tt>z-index</tt> puts it on top of the other one.
CSS styles make its content area translucent, in browsers that support that.
</div>
</div>
```

The major shortcoming of this example is that the stylesheet specifies a fixed size for all windows. Because the titlebar and content portions of the window must be precisely positioned within the overall window, changing the size of a window requires changing the value of various positioning properties in all three rules defined by the stylesheet. This is difficult to do in a static HTML document, but it would not be so difficult if you could use a script to set all the necessary properties. This topic is explored in the next section.

16.3 Scripting Inline Styles

The most straightforward way to script CSS is to alter the `style` attribute of individual document elements. Like most HTML attributes, `style` is also a property of the Element object, and you can manipulate it in JavaScript. The `style` property is unusual, however: its value is not a string, but a CSSStyleDeclaration object. The JavaScript properties of this style object represent the CSS properties specified by the HTML `style` attribute. To make the text of an element `e` big, bold, and blue, for example, you can use the following code to set the JavaScript properties that correspond to the `font-size` `font-weight` and `color` style properties:

```
e.style.fontSize = "24pt";
e.style.fontWeight = "bold";
e.style.color = "blue";
```

Naming Conventions: CSS Properties in JavaScript

Many CSS style properties, such as `font-size`, contain hyphens in their names. In JavaScript, a hyphen is interpreted as a minus sign, so it is not possible to write an expression like:

```
e.style.font-size = "24pt"; // Syntax error!
```

Therefore, the names of the properties of the CSSStyleDeclaration object are slightly different from the names of actual CSS properties. If a CSS property name contains one or more hyphens, the CSSStyleDeclaration property name is formed by removing the hyphens and capitalizing the letter immediately following each hyphen. Thus, the CSS property `border-left-width` is accessed through the JavaScript `borderLeftWidth` property, and you can access the CSS `font-family` property with code like this:

```
e.style.fontFamily = "sans-serif";
```

> Also, when a CSS property, such as the `float` property, has a name that is a reserved word in JavaScript, that name is prefixed with "css" to create a legal CSSStyleDeclaration name. Thus, to set or query the value of the CSS `float` property of an element, use the `cssFloat` property of the CSSStyleDeclaration object.

When working with the style properties of the CSSStyleDeclaration object, remember that all values must be specified as strings. In a stylesheet or `style` attribute, you can write:

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

To accomplish the same thing for an element e with JavaScript, you have to quote all of the values:

```
e.style.position = "absolute";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Note that the semicolons go outside the strings. These are just normal JavaScript semicolons; the semicolons you use in CSS stylesheets are not required as part of the string values you set with JavaScript.

Furthermore, remember that all the positioning properties require units. Thus, it is not correct to set the `left` property like this:

```
e.style.left = 300;    // Incorrect: this is a number, not a string
e.style.left = "300";  // Incorrect: the units are missing
```

Units are required when setting style properties in JavaScript, just as they are when setting style properties in stylesheets. The correct way to set the value of the `left` property of an element e to 300 pixels is:

```
e.style.left = "300px";
```

If you want to set the `left` property to a computed value, be sure to append the units at the end of the computation:

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

Notice that the numeric result of the computation will be converted to a string as a side effect of appending the units string.

Recall that some CSS properties, such as `margin`, are shortcuts for other properties, such as `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. The CSSStyleDeclaration object has properties that correspond to these shortcut properties. For example, you might set the `margin` property like this:

```
e.style.margin = topMargin + "px " + rightMargin + "px " +
                 bottomMargin + "px " + leftMargin + "px";
```

Arguably, it is easier to set the four margin properties individually:

```
e.style.marginTop = topMargin + "px";
e.style.marginRight = rightMargin + "px";
```

```
e.style.marginBottom = bottomMargin + "px";
e.style.marginLeft = leftMargin + "px";
```

The `style` attribute of an HTML element is its *inline* style, and it overrides any style specifications in a stylesheet. Inline styles are generally useful for setting style values, and that is what the examples above have all done. You can read the properties of a CSSStyleDeclaration object that represents inline styles, but they return meaningful values only if they've previously been set by your JavaScript code or if the HTML element with which you are working has an inline `style` attribute that sets the desired properties. For example, your document may include a stylesheet that sets the left margin for all paragraphs to 30 pixels, but if you read the `marginLeft` property of one of your paragraph elements, you'll get the empty string unless that paragraph has a `style` attribute that overrides the stylesheet setting.

Reading the inline style of an element is particularly difficult for style properties that require units and for shortcut properties: your code has to include nontrivial CSS parsing capabilities to really make use of those values. In general, the inline style of an element is only useful for setting styles. If you need to query the style of an element, use the computed style, which is discussed in §16.4.

Sometimes, you may find it easier to set or query the inline style of an element as a single string value rather than as a CSSStyleDeclaration object. To do that, you can use the Element `getAttribute()` and `setAttribute()` methods, or you can use the `cssText` property of the CSSStyleDeclaration object:

```
// Set the style attribute of e to the string s with either of these lines:
e.setAttribute("style", s);
e.style.cssText = s;

// Query the inline style string of the element e with either of these:
s = e.getAttribute("style");
s = e.style.cssText;
```

16.3.1 CSS Animations

One of the most common uses of scripted CSS is to produce animated visual effects. This can be achieved by using `setTimeout()` or `setInterval()` (see §14.1) to repeatedly invoke a function that alters the inline style of an element. Example 16-3 demonstrates with two functions `shake()` and `fadeOut()`. `shake()` quickly moves or "shakes" an element side to side. It might be used to grab the user's attention if he enters invalid data, for example. `fadeOut()` decreases the opacity of an element over a specified amount of time (500 milliseconds, by default), causing it to fade out and vanish.

Example 16-3. CSS animations

```
// Convert element e to relative positioning and "shake" it left and right.
// The first argument can be an element object or the id of an element.
// If a function is passed as the second argument, it will be invoked
// with e as an argument when the animation is complete.
// The 3rd argument specifies how far to shake e. The default is 5 pixels.
// The 4th argument specifies how long to shake for. The default is 500 ms.
```

```
function shake(e, oncomplete, distance, time) {
    // Handle arguments
    if (typeof e === "string") e = document.getElementById(e);
    if (!time) time = 500;
    if (!distance) distance = 5;

    var originalStyle = e.style.cssText;       // Save the original style of e
    e.style.position = "relative";             // Make e relatively positioned
    var start = (new Date()).getTime();        // Note the animation start time
    animate();                                 // Start the animation

    // This function checks the elapsed time and updates the position of e.
    // If the animation is complete, it restores e to its original state.
    // Otherwise, it updates e's position and schedules itself to run again.
    function animate() {
        var now = (new Date()).getTime();      // Get current time
        var elapsed = now-start;               // How long since we started
        var fraction = elapsed/time;           // What fraction of total time?

        if (fraction < 1) {     // If the animation is not yet complete
            // Compute the x position of e as a function of animation
            // completion fraction. We use a sinusoidal function, and multiply
            // the completion fraction by 4pi, so that it shakes back and
            // forth twice.
            var x = distance * Math.sin(fraction*4*Math.PI);
            e.style.left = x + "px";

            // Try to run again in 25ms or at the end of the total time.
            // We're aiming for a smooth 40 frames/second animation.
            setTimeout(animate, Math.min(25, time-elapsed));
        }
        else {                  // Otherwise, the animation is complete
            e.style.cssText = originalStyle  // Restore the original style
            if (oncomplete) oncomplete(e);   // Invoke completion callback
        }
    }
}

// Fade e from fully opaque to fully transparent over time milliseconds.
// Assume that e is fully opaque when this function is invoked.
// oncomplete is an optional function that will be invoked with e as its
// argument when the animation is done. If time is omitted, use 500ms.
// This function does not work in IE, but could be modified to animate
// IE's nonstandard filter property in addition to opacity.
function fadeOut(e, oncomplete, time) {
    if (typeof e === "string") e = document.getElementById(e);
    if (!time) time = 500;

    // We use Math.sqrt as a simple "easing function" to make the animation
    // subtly nonlinear: it fades quickly at first and then slows down some.
    var ease = Math.sqrt;

    var start = (new Date()).getTime();    // Note the animation start time
    animate();                             // And start animating

    function animate() {
```

```
        var elapsed = (new Date()).getTime()-start; // elapsed time
        var fraction = elapsed/time;                // As a fraction of total
        if (fraction < 1) {     // If the animation is not yet complete
            var opacity = 1 - ease(fraction);  // Compute element opacity
            e.style.opacity = String(opacity); // Set it on e
            setTimeout(animate,                // Schedule another frame
                       Math.min(25, time-elapsed));
        }
        else {                  // Otherwise, we're done
            e.style.opacity = "0";          // Make e fully transparent
            if (oncomplete) oncomplete(e);  // Invoke completion callback
        }
    }
}
```

Both `shake()` and `fadeOut()` accept an optional callback function as their second argument. If specified, this function will be invoked when the animation is complete. The element that was animated is passed as an argument to the callback. The following HTML creates a button that, when clicked, shakes from side to side and then fades away:

```
<button onclick="shake(this, fadeOut);">Shake and Fade</button>
```

Note that the `shake()` and `fadeOut()` example functions are quite similar to each other, and both can serve as templates for similar animations of other CSS properties. Client-side libraries, such as jQuery usually support pre-defined visual effects, however, so you may never actually need to write an animation function like `shake()`, unless you want to create a particularly complex visual effect. One early and noteworthy effects library is Scriptaculous, which is designed for use with the Prototype framework. Visit *http://script.aculo.us/* and *http://scripty2.com/* to learn more.

The CSS3 Transitions module defines a way to specify animated effects in stylesheets, obviating the need for any scripting. Instead of defining a function like `fadeOut()`, for example, you might use CSS like this:

```
.fadeable { transition: opacity .5s ease-in }
```

This specifies that any time the opacity of a "fadeable" element changed, that change will be animated (from the current value to the new value) over a period of half a second, using a nonlinear easing function. CSS Transitions is not yet standard but has been implemented for some time in Safari and Chrome using the `-webkit-transition` property. At the time of this writing, Firefox 4 has added support using `-moz-transition`.

16.4 Querying Computed Styles

The `style` property of an element is that element's *inline* style. It overrides all stylesheets, and it is the perfect place to set CSS properties to change the visual appearance of an element. It is not, however, generally useful for querying the styles that actually apply to an element. For that, you want the *computed* style. The computed style for an element is the set of property values that the browser derives (or computes) from the

inline style plus all applicable style rules in all linked stylesheets: it is the set of properties actually used to display the element. Like inline styles, computed styles are represented with a CSSStyleDeclaration object. Unlike inline styles, however, computed styles are read-only. You can't set these styles, but the computed CSSStyleDeclaration object for an element lets you determine exactly what style property values the browser used when rendering that element.

Obtain the computed style for an element with the `getComputedStyle()` method of the Window object. The first argument to this method is the element whose computed style is desired. The second argument is required and is usually `null` or the empty string, but it can also be a string that names a CSS pseudoelement, such as ":before", ":after", ":first-line", or ":first-letter":

```
var title = document.getElementById("section1title");
var titlestyles = window.getComputedStyle(element, null);
```

The return value of `getComputedStyle()` is a CSSStyleDeclaration object that represents all the styles that apply to the specified element (or pseudoelement). There are a number of important differences between a CSSStyleDeclaration object that represents inline styles and one that represents computed styles:

- Computed style properties are read-only.
- Computed style properties are *absolute*: relative units like percentages and points are converted to absolute values. Any property that specifies a size (such as a margin size or a font size) will have a value measured in pixels. This value will be a string with a "px" suffix, so you'll still need to parse it, but you won't have to worry about parsing or converting units. Properties whose values are colors will be returned in "rgb(#,#,#)" or "rgba(#,#,#,#)" format.
- Shortcut properties are not computed, only the fundamental properties that they are based on. Don't query the `margin` property, for example, but use `marginLeft`, `marginTop`, and so on.
- The `cssText` property of the computed style is undefined.

Computed styles and inline styles can be used together. Example 16-4 defines functions `scale()` and `scaleColor()`. One queries and parses the computed text size of a specified element; the other queries and parses the computed background color of an element. Both functions then scale the resulting value and set the scaled value as an inline style of the element. (These functions do not work in IE8 and earlier: as we'll discuss below, those versions of IE do not support `getComputedStyle()`.)

Example 16-4. Querying computed styles and setting inline styles

```
// Scale the text size of element e by the specified factor
function scale(e, factor) {
    // Use the computed style to query the current size of the text
    var size = parseInt(window.getComputedStyle(e,"").fontSize);
    // And use the inline style to enlarge that size
    e.style.fontSize = factor*size + "px";
}
```

```
// Alter the background color of element e by the specified amount.
// Factors > 1 lighten the color and factors < 1 darken it.
function scaleColor(e, factor) {
    var color = window.getComputedStyle(e, "").backgroundColor;  // Query
    var components = color.match(/[\d\.]+/g);   // Parse r,g,b, and a components
    for(var i = 0; i < 3; i++) {                // Loop through r, g and b
        var x = Number(components[i]) * factor;         // Scale each one
        x = Math.round(Math.min(Math.max(x, 0), 255));  // Round and set bounds
        components[i] = String(x);
    }
    if (components.length == 3)  // A rgb() color
        e.style.backgroundColor = "rgb(" + components.join() + ")";
    else                         // A rgba() color
        e.style.backgroundColor = "rgba(" + components.join() + ")";
}
```

Computed styles can be tricky, and querying them does not always provide the information you might expect. Consider the `font-family` property: it accepts a comma-separated list of desired font families for cross-platform portability. When you query the `fontFamily` property of a computed style, you're simply getting the value of the most specific `font-family` style that applies to the element. This may return a value such as "arial,helvetica,sans-serif", which does not tell you which typeface is actually in use. Similarly, if an element is not absolutely positioned, attempting to query its position and size through the `top` and `left` properties of its computed style often returns the value "auto". This is a perfectly legal CSS value, but it is probably not what you were looking for.

`getComputedStyle()` is not implemented by IE8 and earlier, but it is expected in IE9. In IE, every HTML element has a `currentStyle` property whose value is a CSSStyleDeclaration. IE's `currentStyle` combines inline styles with stylesheets, but it is not a true computed style because relative values are not converted to absolute values. Querying the properties of IE's current style can return sizes with relative units like "%" or "em" or colors with imprecise names like "red".

Although CSS can be used to precisely specify the position and size of document elements, querying the computed style of an element is not the preferred way to determine the element's size and position. See §15.8.2 for a simpler, portable alternative.

16.5 Scripting CSS Classes

An alternative to scripting individual CSS styles through the inline `style` property is to script the value of the HTML `class` attribute. Changing an element's `class` changes the set of stylesheet selectors that apply to the element and can cause multiple CSS properties to change at the same time. Suppose, for example, that you want a way to draw the user's attention to individual paragraphs (or other elements) of a document. You might start by defining attention-grabbing styles for any elements that have a class name of "attention":

```
.attention {  /* Styles to grab the user's attention */
    background-color: yellow;    /* Yellow highlight background */
    font-weight: bold;           /* Bold text */
    border: solid black 2px;     /* Black box */
}
```

The identifier `class` is a reserved word in JavaScript, so the HTML `class` attribute is available to JavaScript code using the name `className`. Here is code that sets and clears the `className` property of an element to add and remove the "attention" class for that element:

```
function grabAttention(e) { e.className = "attention"; }
function releaseAttention(e) { e.className = ""; }
```

HTML elements can be members of more than one CSS class and the `class` attribute holds a space-separated list of class names. The `className` property has a misleading name: `classNames` would have been a much better choice. The functions above assume that the `className` property will specify zero or one class name and do not work when more than one class is in use. If an element already has a class assigned, calling the `grabAttention()` function for that element will overwrite the existing class.

HTML5 addresses this issue by defining a `classList` property for every element. The value of this property is known as a DOMTokenList: a read-only array-like object (§7.11) whose elements contain the individual classnames for the element. More important than its array elements, however, are the methods defined by DOMTokenList. `add()` and `remove()` add and remove individual class names from the element's `class` attribute. `toggle()` adds a classname if it is not already present and removes it otherwise. Finally, the `contains()` method tests whether the `class` attribute contains a specified classname.

Like other DOM collection types, a DOMTokenList is a "live" representation of the element's set of classes, not a static snapshot of the classes at the time the `classList` property is queried. If you obtain a DOMTokenList from the `classList` property of an element and then change the `className` property of that element, those changes are immediately visible through the token list. Similarly, any changes you make through the token list are immediately visible through the `className` property.

At the time of this writing, the `classList` property is not supported by all current browsers. This important functionality is easy to approximate, however, with code like that of Example 16-5. Using code like this that allows an element's `class` attribute to be treated as a set of classnames makes many CSS scripting tasks much easier.

Example 16-5. classList(): treat className as a set of CSS classes

```
/*
 * Return the classList property of e, if it has one.
 * Otherwise, return an object that simulates the DOMTokenList API for e.
 * The returned object has contains(), add(), remove(), toggle() and toString()
 * methods for testing and altering the set of classes of the element e.
 * If the classList property is natively supported, the returned object is
 * array-like and has length and array index properties. The simulated
```

```
 * DOMTokenList is not array-like, but has a toArray() method that returns
 * a true-array snapshot of the element's class names.
 */
function classList(e) {
    if (e.classList) return e.classList;   // Return e.classList if it exists
    else return new CSSClassList(e);       // Otherwise try to fake it
}

// CSSClassList is a JavaScript class that simulates DOMTokenList
function CSSClassList(e) { this.e = e; }

// Return true if e.className contains the class c, false otherwise
CSSClassList.prototype.contains = function(c) {
    // Check that c is a valid class name
    if (c.length === 0 || c.indexOf(" ") != -1)
        throw new Error("Invalid class name: '" + c + "'");
    // Check common cases first
    var classes = this.e.className;
    if (!classes) return false;       // e has no classes at all
    if (classes === c) return true;   // e has one class that matches exactly

    // Otherwise, use a RegExp to search for c as a word by itself
    // \b in a regular expression requires a match at a word boundary.
    return classes.search("\\b" + c + "\\b") != -1;
};

// Add c to the e.className if it is not already present
CSSClassList.prototype.add = function(c) {
    if (this.contains(c)) return;             // Do nothing if already present
    var classes = this.e.className;
    if (classes && classes[classes.length-1] != " ")
        c = " " + c;                          // Add a space if we need one
    this.e.className += c;                    // Add c to the className
};

// Remove all occurrences of c from e.className
CSSClassList.prototype.remove = function(c) {
    // Make sure c is a valid class name
    if (c.length === 0 || c.indexOf(" ") != -1)
        throw new Error("Invalid class name: '" + c + "'");
    // Remove all occurances of c as a word, plus any trailing space
    var pattern = new RegExp("\\b" + c + "\\b\\s*", "g");
    this.e.className = this.e.className.replace(pattern, "");
};

// Add c to e.className if it is not already present and return true.
// Otherwise, remove all occurrences of c from e.className and return false.
CSSClassList.prototype.toggle = function(c) {
    if (this.contains(c)) {  // If e.className contains c
        this.remove(c);      // then remove it.
        return false;
    }
    else {                   // Otherwise:
        this.add(c);         // add it.
        return true;
    }
```

```
};

// Return e.className itself
CSSClassList.prototype.toString = function() { return this.e.className; };

// Return of the names in e.className
CSSClassList.prototype.toArray = function() {
    return this.e.className.match(/\b\w+\b/g) || [];
};
```

16.6 Scripting Stylesheets

So far, we've seen how to set and query the CSS styles and classes of individual elements. It is also possible to script CSS stylesheets themselves. This is not commonly done, but it can occasionally be useful and this section sketches out the technique.

When scripting stylesheets, there are two kinds of objects you may need to work with. The first kind are the Element objects that represent `<style>` and `<link>` elements that contain or reference your stylesheets. These are regular document elements and if you give them `id` attributes, you can select them with `document.getElementById()`. The second kind of object is a CSSStyleSheet object that represents the stylesheet itself. The `document.styleSheets` property is a read-only array-like object containing CSSStyleSheet objects that represent the stylesheets associated with the document. If you set the `title` attribute of the `<style>` or `<link>` element that defines or references the stylesheet, that title will be available as the `title` property of the corresponding CSSStyleSheet object.

The subsections that follow explain what you can do with these style and link elements and stylesheet objects.

16.6.1 Enabling and Disabling Stylesheets

The simplest stylesheet scripting technique is also the most portable and robust. `<style>` elements, `<link>` elements, and CSSStyleSheet objects all define a `disabled` property that you can query and set in JavaScript. As its name implies, if the `disabled` property is `true`, then the stylesheet is disabled and is ignored by the browser.

The `disableStylesheet()` function below demonstrates this. If passed a number, it treats it as an index into the `document.styleSheets` array. If passed a string, it treats it as a CSS selector and passes it to `document.querySelectorAll()` (see §15.2.5), then sets the `disabled` property of all returned elements:

```
function disableStylesheet(ss) {
    if (typeof ss === "number")
        document.styleSheets[ss].disabled = true;
    else {
        var sheets = document.querySelectorAll(ss);
        for(var i = 0; i < sheets.length; i++)
```

```
            sheets[i].disabled = true;
        }
    }
```

16.6.2 Querying, Inserting and Deleting Stylesheet Rules

In addition to disabling and enabling stylesheets, the CSSStyleSheet object also defines an API for querying, inserting, and deleting the style rules of a stylesheet. IE8 and before implement a slightly different API than the standard API implemented by other browsers.

Manipulating stylesheets directly is not normally a useful thing to do. Instead of editing or adding new rules to a stylesheet, it is typically better to leave your stylesheets static and script the `className` property of your elements. On the other hand, if you want to allow the user complete control over the styles used on your pages, you might need to dynamically manipulate a stylesheet.

The elements of the `document.styleSheets[]` array are CSSStyleSheet objects. A CSSStyleSheet object has a `cssRules[]` array that contains the rules of the stylesheet:

```
var firstRule = document.styleSheets[0].cssRules[0];
```

IE uses the property name `rules` instead of `cssRules`.

The elements of the `cssRules[]` or `rules[]` arrays are CSSRule objects. In the standard API, a CSSRule object may represent any kind of CSS rule, including *at-rules* such as `@import` and `@page` directives. In IE, however, the `rules[]` array contains only the actual style rules of the stylesheet.

CSSRule objects have two properties that can be used portably. (In the standard API, a rule that is not a style rule will not have these properties defined, and you probably want to skip over it when traversing the stylesheet.) `selectorText` is the CSS selector for the rule, and `style` refers to a writable CSSStyleDeclaration object that describes the styles associated with that selector. Recall that CSSStyleDeclaration is the same type used to represent inline and computed styles. You can use this `CSSStyleDeclaration` object to query the style values or to set new styles for the rule. Often, when traversing a stylesheet, you are interested in the text of the rule rather than a parsed representation of the rule. In this case, use the `cssText` property of the CSSStyleDeclaration object to obtain the text representation of the rules.

In addition to querying and altering the existing rules of a stylesheet, you can also add rules to and remove rules from a stylesheet. The standard API interface defines `insertRule()` and `deleteRule()` methods for adding and removing rules:

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

IE does not support `insertRule()` and `deleteRule()` but defines largely equivalent `addRule()` and `removeRule()` functions. The only real difference (aside from the different names) is that `addRule()` expects the selector text and styles text as two separate arguments.

The following code loops through the rules of a stylesheet, demonstrating the API by making some dubious changes to the stylesheet:

```
var ss = document.styleSheets[0];               // Get the first stylesheet
var rules = ss.cssRules?ss.cssRules:ss.rules;  // Get the stylesheet rules

for(var i = 0; i < rules.length; i++) {         // Loop through those rules
    var rule = rules[i];
    if (!rule.selectorText) continue;  // Skip @import and other nonstyle rules

    var selector = rule.selectorText;  // The selector
    var ruleText = rule.style.cssText; // The styles, in text form

    // If the rule applies to h1 elements, apply it to h2 elements as well
    // Note this only works if the selector is literally "h1"
    if (selector == "h1") {
        if (ss.insertRule) ss.insertRule("h2 {" + ruleText + "}", rules.length);
        else if (ss.addRule) ss.addRule("h2", ruleText, rules.length);
    }

    // If the rule sets the text-decoration property, delete it.
    if (rule.style.textDecoration) {
        if (ss.deleteRule) ss.deleteRule(i);
        else if (ss.removeRule) ss.removeRule(i);
        i--; // Adjust the loop index since the former rule i+1 is now rule i
    }
}
```

16.6.3 Creating New Stylesheets

Finally, it is possible to create entirely new stylesheets and add them to your document. In most browsers, this is done with standard DOM techniques: just create a new `<style>` element and insert it into the document head, then use its `innerHTML` property to set stylesheet content. In IE8 and before, however, a new CSSStyleSheet object is created with the nonstandard method `document.createStyleSheet()`, and stylesheet text is specified using the `cssText` property. Example 16-6 demonstrates.

Example 16-6. Creating a new stylesheet

```
// Add a stylesheet to the document and populate it with the specified styles.
// The styles argument can be a string or an object. If it is a string, it
// is treated as the text of the stylesheet. If it is an object, then each
// property defines a style rule to be added to the stylesheet. Property
// names are selectors and their values are the corresponding styles
function addStyles(styles) {
    // First, create a new stylesheet
    var styleElt, styleSheet;
    if (document.createStyleSheet) { // If the IE API is defined, use it
        styleSheet = document.createStyleSheet();
    }
    else {
        var head = document.getElementsByTagName("head")[0]
        styleElt = document.createElement("style"); // New <style> element
        head.appendChild(styleElt);                 // Insert it into <head>
```

```
        // Now the new stylesheet should be the last one
        styleSheet = document.styleSheets[document.styleSheets.length-1]
    }

    // Now insert the styles into it
    if (typeof styles === "string") {
        // The argument is stylesheet text
        if (styleElt) styleElt.innerHTML = styles;
        else styleSheet.cssText = styles;           // The IE API
    }
    else {
        // The argument is an object of individual rules to insert
        var i = 0;
        for(selector in styles) {
            if (styleSheet.insertRule) {
                var rule = selector + " {" + styles[selector] + "}";
                styleSheet.insertRule(rule, i++);
            }
            else {
                styleSheet.addRule(selector, styles[selector], i++);
            }
        }
    }
}
```

CHAPTER 17
Handling Events

Client-side JavaScript programs use an asynchronous event-driven programming model (introduced in §13.3.2). In this style of programming, the web browser generates an *event* whenever something interesting happens to the document or browser or to some element or object associated with it. For example, the web browser generates an event when it finishes loading a document, when the user moves the mouse over a hyperlink, or when the user strikes a key on the keyboard. If a JavaScript application cares about a particular type of event, it can register one or more functions to be invoked when events of that type occur. Note that this is not unique to web programming: all applications with graphical user interfaces are designed this way—they sit around waiting for something to happen (i.e., they wait for events to occur), and then they respond.

Note that the word *event* is not itself a technical word that requires definition. Events are simply occurrences that a web browser will notify your program about. Events are not JavaScript objects and have no manifestation in the source code of your program. There are, of course, a number of event-related objects that do appear in your source code, and these do require technical explanation. We'll begin this chapter, therefore, with some important definitions.

The *event type* is a string that specifies what kind of event occurred. The type "mousemove", for example, means that the user moved the mouse. The type "keydown" means that a key on the keyboard was pushed down. And the type "load" means that a document (or some other resource) has finished loading from the network. Because the type of an event is just a string, it is sometimes called an *event name*, and indeed, we use this name to identify the specific kind of event we're talking about. Modern web browsers support many event types. §17.1 has an overview.

The *event target* is the object on which the event occurred or with which the event is associated. When we speak of an event, we must specify both the type and the target. A load event on a Window, for example, or a click event on a `<button>` Element. Window, Document, and Element objects are the most common event targets in client-side JavaScript applications, but some events are triggered on other kinds of objects. In Chapter 18 we'll see a readystatechange event that is triggered on an XMLHttpRequest object, for example.

An *event handler* or *event listener* is a function that handles or responds to an event.[1] Applications register their event handler functions with the web browser, specifying an event type and an event target. When an event of the specified type occurs on the specified target, the browser invokes the handler. When event handlers are invoked for an object, we sometimes say that the browser has "fired", "triggered", or "dispatched" the event. There are a number of ways to register event handlers, and the details of handler registration and invocation are explained in §17.2 and §17.3.

An *event object* is an object that is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function (except in IE8 and before where they are sometimes only available through the global variable `event`). All event objects have a `type` property that specifies the event type and a `target` property that specifies the event target. (In IE8 and before, use `srcElement` instead of `target`.) Each event type defines a set of properties for its associated event object. The object associated with a mouse event includes the coordinates of the mouse pointer, for example, and the object associated with a keyboard event contains details about the key that was pressed and the modifier keys that were held down. Many event types define only a few standard properties—such as `type` and `target`—and do not carry much other useful information. For those events it is the simple occurrence of the event, not the event details, that matter. This chapter does not have a specific section covering the Event object. Instead, it explains event object properties when describing specific event types. You can read more about the event object under the name `Event` in the reference section.[2]

Event propagation is the process by which the browser decides which objects to trigger event handlers on. For events that are specific to a single object (such as the load event on the Window object), no propagation is required. When certain kinds of events occur on document elements, however, they propagate or "bubble" up the document tree. If the user moves the mouse over a hyperlink, the mousemove event is first fired on the `<a>` element that defines that link. Then it is fired on the containing elements: perhaps a `<p>` element, a `<div>` element, and the Document object itself. It is sometimes more convenient to register a single event handler on a Document or other container element than to register handlers on each individual element you're interested in. An event handler can stop the propagation of an event, so that it will not continue to bubble and will not trigger handlers on containing elements. Handlers do this by invoking a method or setting a property of the event object. Event propagation is covered in detail in §17.3.6.

1. Some sources, including the HTML5 specification, make a technical distinction between handlers and listeners, based on the way in which they are registered. In this book we treat the two terms as synonyms.

2. Standards define a hierarchy of event object interfaces for different types of events. The Event interface describes "plain" events with no extra details. The MouseEvent subinterface describes the additional fields available in the event objects passed with mouse events, and the KeyEvent subinterface describes the fields you can use with keyboard events, for example. In this book, the reference section collapses all those common event interfaces into a single Event reference page.

In another form of event propagation, known as *event capturing*, handlers specially registered on container elements have the opportunity to intercept (or "capture") events before they are delivered to their actual target. Event capturing is not supported by IE8 and earlier, and is not, therefore, commonly used. The ability to capture or "grab" mouse events is required when processing mouse drag events, however, and we'll see how to do this in Example 17-2.

Some events have *default actions* associated with them. When a click event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page. Event handlers can prevent this default action by returning an appropriate value, invoking a method of the event object, or by setting a property of the event object. This is sometimes called "canceling" the event and is covered in §17.3.7.

With those terms defined, we can now move on to study events and event handling in detail. The first section that follows is an overview of the many event types supported by web browsers. It doesn't cover any single kind of event in detail, but it lets you know what kinds of events are available for use in your web applications. This section includes cross-references to other parts of this book that demonstrate some of the events in action.

After the introductory section on event types, the next two sections explain how to register event handlers and how the browser invokes those event handlers. Because of the historical evolution of the JavaScript event model and because of IE's lack of standards support prior to IE9, both of these topics are more complicated than you might expect.

The chapter ends with a series of examples that demonstrate how to work with a specific types of events. These event-type-specific sections cover:

- Document loading and readiness events
- Mouse events
- Mouse wheel events
- Drag-and-drop events
- Key events
- Text input events

17.1 Types of Events

In the early days of the Web, client-side programmers made do with only a small set of events: "load", "click", "mouseover", and the like. These legacy event types are well supported by all browsers and are the subject of §17.1.1. As the web platform has grown to include more powerful APIs, the set of events has grown large. No single standard defines a complete set of events, and at the time of this writing, the number of events supported by browsers is growing rapidly. These new events come from three sources:

- The DOM Level 3 Events specification, which after a long period of inactivity is being actively worked on under the auspices of the W3C. DOM events are covered in §17.1.2.
- Many new APIs in the HTML5 specification (and related spin-off specifications) define new events for things like history management, drag-and-drop, cross-document messaging, and audio and video playback. §17.1.3 gives an overview of these events.
- The advent of touch-based and JavaScript-enabled mobile devices, such as the iPhone, have required the definition of new touch and gesture event types. See §17.1.4 for some Apple-specific examples.

Note that many of these new event types are not yet widely implemented and are defined by standards that are still in draft stage. The subsections that follow provide an overview of the events, but do not document each one in detail. The rest of this chapter covers the event handling model comprehensively and includes lots of examples of working with events that are well supported. If you understand how to work with events generally, you will be able to easily handle new event types as new web APIs are defined and implemented.

Event Categories

Events can be grouped into some general categories, and knowing what these categories are will help you to understand and organize the long list of events that follows:

Device-dependent input events
: These are events that are directly tied to a specific input device, such as the mouse or keyboard. They include legacy event types such as "mousedown", "mousemove", "mouseup", "keydown", "keypress", and "keyup" and also new touch-specific events like "touchmove" and "gesturechange".

Device-independent input events
: These are input events that are not directly tied to a specific input device. The click event, for example, indicates that a link or button (or other document element) has been activated. This is often done via a mouse click, but it could also be done by keyboard or (on touch-sensitive devices) by gesture. The textinput event (which is not yet widely implemented) is a device-independent alternative to the keypress event and supports keyboard input as well as alternatives such as cut-and-paste and handwriting recognition.

User interface events
: UI events are higher-level events, often on HTML form elements that define a user interface for a web application. They include the focus event (when a text input field gains keyboard focus), the change event when the user changes the value displayed by a form element, and the submit event when the user clicks a Submit button in a form.

State-change events
: Some events are not triggered directly by user activity, but by network or browser activity, and indicate some kind of lifecycle or state-related change. The load event,

fired on the Window object when the document is fully loaded, is probably the most commonly used of these events. The DOMContentLoaded event (discussed in §13.3.4) is another such event. The HTML5 history management mechanism (§22.2) fires the popstate event in response to the browser's Back button. The HTML5 offline web application API (§20.4) includes online and offline events. Chapter 18 shows how to use a readystatechange event to be notified when data requested from a server is ready. Similarly, the new API for reading user-selected local files (§22.6.5) uses events like "loadstart", "progress", and "loadend" for asynchronous notification of I/O progress.

API-specific events
: A number of web APIs defined by HTML5 and related specifications include their own event types. The drag-and-drop API (§17.7) defines events such as "dragstart", "dragenter", "dragover", and "drop". Applications that want to define custom drag sources or drop targets must handle some of these events. The HTML5 `<video>` and `<audio>` elements (§21.2) define a long list of associated event types such as "waiting", "playing", "seeking", "volumechange", and so on. These events are usually only of interest to web apps that want to define custom controls for video or audio playback.

Timers and error handlers
: Timers and error handlers (both of which are described in Chapter 14) are part of client-side JavaScript's asynchronous programming model and are similar to events. Although timers and error handlers are not discussed in this chapter, it is useful to think of them as related to event handling, and you may find it interesting to reread §14.1 and §14.6 in the context of this chapter.

17.1.1 Legacy Event Types

The events you'll use most often in your web apps are generally the ones that have been around the longest and are universally supported: events for dealing with the mouse, the keyboard, HTML forms, and the Window object. The sections below explain many important details about these kinds of events.

17.1.1.1 Form events

Forms and hyperlinks were the first scriptable elements in a web page, way back in the early days of the Web and of JavaScript. This means that form events are some of the most stable and well-supported of all event types. `<form>` elements fire submit events when the form is submitted and reset events when the form is reset. Button-like form elements (including radio buttons and checkboxes) fire click events when the user interacts with them. Form elements that maintain some kind of state generally fire change events when the user changes their state by entering text, selecting an item, or checking a box. For text input fields, a change event is not fired until the user has finished interacting with a form element and has tabbed or clicked to move focus to another element. Form elements respond to keyboard focus changes by firing focus and blur events when they gain and lose the focus.

These form-related events are all covered in more detail in §15.9.3. A few more notes are in order here, however.

The submit and reset events have default actions that can be canceled by event handlers, and some click events do, too. The focus and blur events do not bubble, but all the other form events do. IE defines focusin and focusout events that do bubble as a useful alternative to focus and blur. The jQuery library (see Chapter 19) emulates focusin and focusout events for browsers that do not support them, and the DOM Level 3 Events specification is standardizing them as well.

Finally, note that browsers other than IE trigger an input event on `<textarea>` and other text-input form elements whenever the user enters text (via the keyboard or cut-and-paste) into the element. Unlike the change event, these input events are triggered for each insertion. Unfortunately, the event object of an input event does not specify what text has been input. (The new textinput event described later will be a useful alternative to this event.)

17.1.1.2 Window events

Window events represent occurrences related to the browser window itself, rather than any specific document content displayed inside the window. (For some of these events, however, an event with the same name can be fired on document elements.)

The load event is the most important of these events: it is fired when a document and all of its external resources (such as images) are fully loaded and displayed to the user. The load event was discussed throughout Chapter 13. DOMContentLoaded and readystatechange are alternatives to the load event: they are triggered sooner, when the document and its elements are ready to manipulate, but before external resources are fully loaded. §17.4 has examples of these document load-related events.

The unload event is the opposite of load: it is triggered when the user is navigating away from a document. An unload event handler might be used to save the user's state, but it cannot be used to cancel navigation. The beforeunload event is similar to unload but gives you the opportunity to ask the user to confirm that they really want to navigate away from your web page. If a handler for beforeunload returns a string, that string will be displayed to the user in a confirmation dialog before the new page is loaded, and the user will have the opportunity to cancel her navigation and remain at your page.

The `onerror` property of the Window object is something like an event handler, and it is triggered in response to JavaScript errors. It isn't a true event handler, however, because it is invoked with different arguments. See §14.6 for details.

Individual document elements, such as `<img>` elements, can also register handlers for load and error events. These are triggered when an external resource (the image, for example) is fully loaded, or when an error occurs that prevents it from loading. Some browsers also support (and HTML5 standardizes) an abort event, which is triggered when an image (or other network resource) fails to load because the user stopped the loading process.

The focus and blur events described above for form elements are also used as Window events: they are triggered on a window when that browser window receives or loses keyboard focus from the operating system.

Finally, the resize and scroll events are fired on a Window when the user resizes or scrolls the browser window. Scroll events can also be fired on any scrollable document element, such as those with the CSS `overflow` property (§16.2.6) set. The event object passed to resize and scroll event handlers is just an ordinary Event object and does not have properties that specify how much resizing or scrolling occurred—you can determine the new window size and scrollbar position using the techniques shown in §15.8.

17.1.1.3 Mouse events

Mouse events are generated when the user moves or clicks the mouse over a document. These events are triggered on the most deeply nested element that the mouse pointer is over, but they bubble up through the document. The event object passed to mouse event handlers has properties set that describe the position and button state of the mouse and also specify whether any modifier keys were held down when the event occurred. The `clientX` and `clientY` properties specify the position of the mouse in window coordinates. The `button` and `which` properties specify which mouse button (if any) was pressed. (See the `Event` reference page, however, because these properties are difficult to use portably.) The `altKey`, `ctrlKey`, `metaKey`, and `shiftKey` properties are set to `true` when the corresponding keyboard modifier keys are held down. And for click events, the `detail` property specifies whether this was a single, double, or triple click.

The mousemove event is triggered any time the user moves or drags the mouse. These events occur frequently, so mousemove handlers must not trigger computationally intensive tasks. The mousedown and mouseup events are triggered when the user presses and releases a mouse button. By registering a mousedown handler that registers a mousemove handler, you can detect and respond to mouse drags. Doing this properly involves being able to capture mouse events so that you continue to receive mousemove events even when the mouse has moved out of the element it started in. §17.5 includes an example of handling drags.

After a mousedown and mouseup event sequence, the browser also triggers a click event. The click event was described above as a device-independent form event, but it is actually triggered on any document element, not just form elements, and it is passed an event object with all of the extra mouse-related fields described above. If the user clicks a mouse button twice in a row (within a sufficiently short amount of time), the second click event will be followed by a dblclick event. Browsers often display a context menu when the right mouse button is clicked. They generally fire a contextmenu event before displaying the menu, and if you cancel the event, you can prevent the display of the menu. This is also an easy way to be notified of right mouse button clicks.

When the user moves the mouse so that it goes over a new element, the browser fires a mouseover event on that element. When the mouse moves so that it is no longer over an element, the browser fires a mouseout event on that element. For these events, the

event object will have a relatedTarget property that specifies the other element involved in the transition. (See the Event reference page for the IE equivalent of the relatedTarget property.) mouseover and mouseout events bubble like all of the mouse events described here. This is often inconvenient, because when a mouseout handler is triggered, you have to check whether the mouse actually left the element you are interested in or if it merely transitioned from one child of the element to another. Because of this, IE supports nonbubbling versions of these events known as mouseenter and mouseleave. jQuery emulates support for these events in browsers other than IE (see Chapter 19), and the DOM Level 3 Events specification standardizes them.

When the user rotates the mouse wheel, browsers trigger a mousewheel event (or, in Firefox, a DOMMouseScroll event). The event object passed with these events includes properties that specify how much, and in which direction, the wheel was rotated. The DOM Level 3 Events specification is standardizing a more general multidimensional wheel event that, if implemented, will supersede both mousewheel and DOMMouseScroll. §17.6 includes a mousewheel event example.

17.1.1.4 Key events

When the web browser has keyboard focus, it generates events each time the user presses or releases a key on the keyboard. Keyboard shortcuts that have meaning to the operating system or to the browser itself are often "eaten" by the OS or browser and may not be visible to JavaScript event handlers, however. Keyboard events are triggered on whatever document element has keyboard focus, and they bubble up to the document and window. If no element has the focus, the events are triggered directly on the document. Keyboard event handlers are passed an event object with a keyCode field that specifies what key was pressed or released. In addition to keyCode, the event object for key events also has altKey, ctrlKey, metaKey, and shiftKey that describe the state of the keyboard modifier keys.

The keydown and keyup events are low-level keyboard events: they are triggered whenever a key (even a modifier key) is pressed or released. When a keydown event generates a printable character, an additional keypress event is triggered after the keydown but before the keyup. (In the case of a key that is held down until it repeats, there may be many keypress events before the keyup event.) The keypress event is a higher-level text event, and its event object specifies the character that was generated, not the key that was pressed.

The keydown, keyup, and keypress events are supported by all browsers, but there are some interoperability problems because the values of the keyCode property of the event object have never been standardized. The DOM Level 3 Events specification, described below, attempts to addresses these interoperability problems, but has not yet been implemented. §17.9 includes an example of handling keydown events and §17.8 includes an example of processing keypress events.

17.1.2 DOM Events

The DOM Level 3 Events specification has been under development by the W3C for about a decade. At the time of this writing, it has undergone substantial revision to bring it in line with current browser reality and it is finally in the "last call working draft" stage of standardization. It standardizes many of the legacy events described above and adds some new ones described here. These new event types are not yet widely supported, but browser vendors are expected to implement them once the standard is final.

As noted above, the DOM Level 3 Events specification standardizes the focusin and focusout events as bubbling alternatives to the focus and blur events and standardizes the mouseenter and mouseleave events as nonbubbling alternatives to mouseover and mouseout. This version of the standard also deprecates a number of event types that were defined by Level 2 but never widely implemented. Browsers are still allowed to generate events like DOMActivate, DOMFocusIn, and DOMNodeInserted, but these are no longer required, and they are not documented in this book.[3]

What is new in the DOM Level 3 Events specification is standardized support for two-dimensional mouse wheels via the wheel event and better support for text input events with a textInput event and with a new KeyboardEvent object that is passed as the argument to handlers for keydown, keyup, and keypress events.

A handler for a wheel event receives an event object with all the usual mouse event properties, and also `deltaX`, `deltaY`, and `deltaZ` properties that report rotation around three different mouse wheel axes. (Most mouse wheels are one or two dimensional and do not use `deltaZ`.) See §17.6 for more on mousewheel events.

DOM Level 3 Events defines the keypress event described above, but deprecates it in favor of a new event named textinput. Rather than a hard-to-use numeric `keyCode` value, the event object passed to a textinput event handler has a `data` property that specifies the string of text that was entered. The textinput event is not a keyboard-specific event: it is triggered whenever text input occurs, whether via the keyboard, cut-and-paste, drag-and-drop, and so on. The specification defines an `inputMethod` property on the event object and a set of constants representing different kinds of text input (keyboard, paste or drop, handwriting or voice recognition, and so on). At the time of this writing, Safari and Chrome support a version of this event using the mixed-case name `textInput`. Its event object includes the `data` property but not the `inputMethod` property. §17.8 includes an example that makes use of this textInput event.

This new DOM standard also simplifies keydown, keyup, and keypress events by adding new `key` and `char` properties to the event object. Both of these properties are strings. For key events that generate printable characters, `key` and `char` will be equal to the generated text. For control keys, the `key` property will be a string like "Enter", "Delete",

3. The only event in common use with "DOM" in its name is DOMContentLoaded. This event was introduced by Mozilla and was never part of the DOM Events standard.

or "Left" that identifies the key. The char property will either be null, or, for control keys like Tab that have a character code, it will be the string generated by the key. At the time of this writing, no browsers support these key and char properties, but Example 17-8 will use the key property if and when it is implemented.

17.1.3 HTML5 Events

HTML5 and related standards define a host of new APIs for web applications (see Chapter 22). Many of these APIs define events. This section lists and briefly describes these HTML5 and web application events. Some of these events are ready to be used now and are explained in more detail elsewhere in the book. Others are not yet widely implemented and are not documented in any detail.

One of the widely advertised features of HTML is inclusion of <audio> and <video> elements for playing sound and video. These elements have a long list of events that they trigger to send notifications about network events, data buffering status, and playback state:

```
canplay          loadeddata      playing         stalled
canplaythrough   loadedmetadata  progress        suspend
durationchange   loadstart       ratechange      timeupdate
emptied          pause           seeked          volumechange
ended            play            seeking         waiting
```

These media events are passed an ordinary event object with no special properties. The target property identifies the <audio> or <video> element, however, and that element has many relevant properties and methods. See §21.2 for more details on these elements, their properties, and their events.

The HTML5 drag-and-drop API allows JavaScript applications to participate in OS-based drag-and-drop operations, transferring data between web applications and native applications. The API defines the following seven event types:

```
dragstart    drag        dragend
dragenter    dragover    dragleave
drop
```

These drag-and-drop events are triggered with an event object like those sent with mouse events. One additional property, dataTransfer, holds a DataTransfer object that contains information about the data being transferred and the formats in which it is available. The HTML5 drag-and-drop API is explained and demonstrated in §17.7.

HTML5 defines a history management mechanism (§22.2) that allows web applications to interact with the browser's Back and Forward buttons. This mechanism involves events named hashchange and popstate. These events are lifecycle notification events like load and unload and are fired at the Window object rather than any individual document element.

HTML5 defines a lot of new features for HTML forms. In addition to standardizing the form input event described earlier, HTML5 also defines a form validation mechanism, which includes an invalid event fired on form elements that have failed validation.

Browser vendors other than Opera have been slow to implement HTML5's new form features and events, however, and this book does not cover them.

HTML5 includes support for offline web applications (see §20.4) that can be installed locally in an application cache so that they can run even when the browser is offline (as when a mobile device is out of network range). The two most important events associated with this are the offline and online events: they are triggered on the Window object whenever the browser loses or gains a network connection. A number of additional events are defined to provide notification of application download progress and application cache updates:

```
cached      checking     downloading    error
noupdate    obsolete     progress       updateready
```

A number of new web application APIs use a message event for asynchronous communication. The Cross-Document Messaging API (§22.3) allows scripts in a document from one server to exchange messages with scripts in a document from another server. This works around the limitations of the same-origin policy (§13.6.2) in a secure way. Each message that is sent triggers a message event on the Window of the receiving document. The event object passed to the handler includes a `data` property that holds the content of the message as well as `source` and `origin` policies that identify the sender of the message. The message event is used in similar ways for communication with Web Workers (§22.4) and for network communication via Server-Sent Events (§18.3) and WebSockets (§22.9).

HTML5 and related standards define some events that are triggered on objects other than windows, documents, and document elements. Version 2 of the XMLHttpRequest specification, as well as the File API specification, define a series of events that track the progress of asynchronous I/O. They trigger events on an XMLHttpRequest or FileReader object. Each read operation begins with a loadstart event, followed by progress events and a loadend event. Additionally, each operation ends with a load, error, or abort event just before the final loadend event. See §18.1.4 and §22.6.5 for details.

Finally, HTML5 and related standards define a few miscellaneous event types. The Web Storage (§20.1) API defines a storage event (on the Window object) that provides notification of changes to stored data. HTML5 also standardizes the beforeprint and afterprint events that were originally introduced by Microsoft in IE. As their names imply, these events are triggered on a Window immediately before and immediately after its document is printed and provide an opportunity to add or remove content such as the date and time that the document was printed. (These events should not be used to change the presentation of a document for printing because CSS media types already exist for that purpose.)

17.1.4 Touchscreen and Mobile Events

The widespread adoption of powerful mobile devices, particularly those with touchscreens, has required the creation of new categories of events. In many cases, touchscreen events are mapped to traditional event types such as click and scroll. But

not every interaction with a touchscreen UI emulates a mouse, and not all touches can be treated as mouse events. This section briefly explains the gesture and touch events generated by Safari when running on Apple's iPhone and iPad devices and also covers the orientationchange event generated when the user rotates the device. At the time of this writing, there are no standards for these events, but the W3C has begun work on a "Touch Events Specification" that uses Apple's touch event as a starting point. These events are not documented in the reference section of this book, but you can find more information at the Apple Developer Center (*http://developer.apple.com/*).

Safari generates gesture events for two-finger scaling and rotation gestures. The gesturestart event is fired when the gesture begins and gestureend is fired when it ends. Between these two events are a sequence of gesturechange events that track the progress of the gesture. The event object sent with these events has numeric `scale` and `rotation` properties. The `scale` property is the ratio of the current distance between the two fingers to the initial distance between the fingers. A "pinch close" gesture has a `scale` less than 1.0, and a "pinch open" gesture has a `scale` greater than 1.0. The `rotation` property is the angle of finger rotation since the start of the event. It is reported in degrees, with positive values indicating clockwise rotation.

Gesture events are high-level events that notify you of a gesture that has already been interpreted. If you want to implement your own custom gestures, you can listen for low-level touch events. When a finger touches the screen a touchstart event is triggered. When the finger moves, a touchmove event is triggered. And when the finger is lifted from the screen, a touchend event is triggered. Unlike mouse events, touch events do not directly report the coordinates of the touch. Instead, the object sent with a touch event has a `changedTouches` property. This property is an array-like object whose elements each describe the position of a touch.

The orientationchanged event is triggered on the Window object by devices that allow the user to rotate the screen from portrait to landscape mode. The object passed with an orientationchanged event is not useful itself. In mobile Safari, however, the `orientation` property of the Window object gives the current orientation as one of the numbers 0, 90, 180, or -90.

17.2 Registering Event Handlers

There are two basic ways to register event handlers. The first, from the early days of the Web, is to set a property on the object or document element that is the event target. The second, newer and more general, technique is to pass the handler to a method of the object or element. To complicate matters, there are two versions of each technique. You can set an event handler property in JavaScript code, or for document elements, you can set the corresponding attribute directly in HTML. For handler registration by method invocation, there is a standard method, named `addEventListener()`, that is supported by all browsers except IE8 and before, and a different method, named `attachEvent()`, for all versions of IE before IE9.

17.2.1 Setting Event Handler Properties

The simplest way to register an event handler is by setting a property of the event target to the desired event handler function. By convention, event handler properties have names that consist of the word "on" followed by the event name: `onclick`, `onchange`, `onload`, `onmouseover`, and so on. Note that these property names are case sensitive and are written in all lowercase, even when the event type (such as "readystatechange" consists of multiple words. Here are two example event handler registrations:

```
// Set the onload property of the Window object to a function.
// The function is the event handler: it is invoked when the document loads.
window.onload = function() {
    // Look up a <form> element
    var elt = document.getElementById("shipping_address");
    // Register an event handler function that will be invoked right
    // before the form is submitted.
    elt.onsubmit = function() { return validate(this); }
}
```

This event handler registration technique works in all browsers for all commonly used event types. In general, all widely implemented web APIs that define events allow handlers to be registered by setting event handler properties.

The shortcoming of event handler properties is that they are designed around the assumption that event targets will have at most one handler for each type of event. If you are writing library code for use in arbitrary documents, it is better to register event handlers using a technique (such as `addEventListener()`) that will not modify or overwrite any previously registered handlers.

17.2.2 Setting Event Handler Attributes

The event handler properties of a document element can also be set as attributes on the corresponding HTML tag. If you do this, the attribute value should be a string of JavaScript code. That code should be the *body* of the event handler function, not a complete function declaration. That is, your HTML event handler code should not be surrounded by curly braces and prefixed with the `function` keyword. For example:

```
<button onclick="alert('Thank you');">Click Here</button>
```

If an HTML event handler attribute contains multiple JavaScript statements, you must remember to separate those statements with semicolons or to break the attribute value across multiple lines.

Some event types are directed at the browser as a whole, rather than at any particular document element. In JavaScript, handlers for these events are registered on the Window object. In HTML, we place them on the `<body>` tag, but the browser registers them on the Window. The following is the complete list of such event handlers as defined by the draft HTML5 specification:

```
onafterprint   onfocus        ononline      onresize
onbeforeprint  onhashchange   onpagehide    onstorage
```

```
onbeforeunload  onload          onpageshow      onundo
onblur          onmessage       onpopstate      onunload
onerror         onoffline       onredo
```

When you specify a string of JavaScript code as the value of an HTML event handler attribute, the browser converts your string into a function that looks something like this:

```
function(event) {
    with(document) {
        with(this.form || {}) {
            with(this) {
                /* your code here */
            }
        }
    }
}
```

If the browser supports ES5, the function is defined in non-strict mode (see §5.7.3). We'll see more about the `event` argument and the `with` statements when we consider event handler invocation in §17.3.

A common style in client-side programming involves keeping HTML content separate from JavaScript behavior. Programmers who follow this discipline shun (or at least avoid) HTML event handler attributes, since they directly mix JavaScript and HTML.

17.2.3 addEventListener()

In the standard event model supported by all browsers other than IE8 and earlier, any object that can be an event target—this includes the Window and Document objects and all document Elements—defines a method named `addEventListener()` that you can use to register an event handler for that target. `addEventListener()` takes three arguments. The first is the event type for which the handler is being registered. The event type (or name) is a string and it should not include the "on" prefix that is used when setting event handler properties. The second argument to `addEventListener()` is the function that should be invoked when the specified type of event occurs. The final argument to `addEventListener()` is a boolean value. Normally, you'll pass `false` for this argument. If you pass `true` instead, your function is registered as a *capturing* event handler and is invoked at a different phase of event dispatch. We'll cover event capturing in §17.3.6. You ought to be able to omit the third argument instead of passing `false`, and the specification may eventually change to allow this, but at the time of this writing, omitting that argument is an error in some current browsers.

The code below registers two handlers for the click event on a `<button>` element. Note the differences between the two techniques used:

```
<button id="mybutton">Click me</button>
<script>
var b = document.getElementById("mybutton");
b.onclick = function() { alert("Thanks for clicking me!"); };
b.addEventListener("click", function() { alert("Thanks again!"); }, false);
</script>
```

Calling addEventListener() with "click" as its first argument does not affect the value of the onclick property. In the code above, a button click will generate two alert() dialog boxes. More importantly, you can call addEventListener() multiple times to register more than one handler function for the same event type on the same object. When an event occurs on an object, all of the handlers registered for that type of event are invoked, in the order in which they were registered. Invoking addEventListener() more than once on the same object with the same arguments has no effect—the handler function remains registered only once, and the repeated invocation does not alter the order in which handlers are invoked.

addEventListener() is paired with a removeEventListener() method that expects the same three arguments but removes an event handler function from an object rather than adding it. It is often useful to temporarily register an event handler and then remove it soon afterward. For example, when you get a mousedown event, you might register temporary capturing event handlers for mousemove and mouseup events so that you can see if the user drags the mouse. You'd then deregister these handlers when the mouseup event arrives. In such a situation, your event handler removal code might look like this:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

17.2.4 attachEvent()

Internet Explorer, prior to IE9, does not support addEventListener() and removeEventListener() . In IE5 and later, it defines similar methods attachEvent() and detachEvent().

The attachEvent() and detachEvent() methods work like addEventListener() and removeEventListener(), with the following exceptions:

- Since the IE event model does not support event capturing, attachEvent() and detachEvent() expect only two arguments: the event type and the handler function.
- The first argument to the IE methods is an event handler property name, with the "on" prefix, rather than the unprefixed event type. For example, pass "onclick" to attachEvent() where you would pass "click" to addEventListener().
- attachEvent() allows the same event handler function to be registered more than once. When an event of the specified type occurs, the registered function will be invoked as many times as it was registered.

It is common to see event handler registration code that uses addEventListener() in browsers that support it and otherwise uses attachEvent():

```
var b = document.getElementById("mybutton");
var handler = function() { alert("Thanks!"); };
if (b.addEventListener)
    b.addEventListener("click", handler, false);
else if (b.attachEvent)
    b.attachEvent("onclick", handler);
```

17.3 Event Handler Invocation

Once you've registered an event handler, the web browser will invoke it automatically when an event of the specified type occurs on the specified object. This section describes event handler invocation in detail, explaining event handler arguments, the invocation context (the `this` value), the invocation scope, and the meaning of the return value of an event handler. Unfortunately, some of these details are different for IE8 and before than for other browsers.

In addition to describing how individual handlers are invoked, this section also explains how events *propagate*: how a single event can trigger the invocation of multiple handlers on the original event target and also on containing elements of the document.

17.3.1 Event Handler Argument

Event handlers are normally (there is one exception, described below) invoked with an event object as their single argument. The properties of the event object provide details about the event. The `type` property, for example, specifies the type of the event that occurred. §17.1 mentioned a number of other event object properties for various event types.

In IE8 and before, event handlers registered by setting a property are not passed an event object when they are invoked. Instead, the event object is available through the global variable `window.event`. For portability, you can write event handlers like this, so that they use the `window.event` if no argument is supplied:

```
function handler(event) {
    event = event || window.event;
    // Handler code goes here
}
```

Event handlers registered with `attachEvent()` are passed an event object, but they can also use `window.event`.

Recall from §17.2.2 that when you register an event handler by setting an HTML attribute, the browser converts your string of JavaScript code into a function. Browsers other than IE construct a function with a single argument named `event`. IE constructs a function that expects no argument. If you use the identifier `event` in such a function, you are referring to `window.event`. In either case, HTML event handlers can refer to the event object as `event`.

17.3.2 Event Handler Context

When you register an event handler by setting a property, it looks as if you are defining a new method on the document element:

```
e.onclick = function() { /* handler code */ };
```

It isn't surprising, therefore, that event handlers are invoked (with one IE-related exception, described below) as methods of the object on which they are defined. That is, within the body of an event handler, the `this` keyword refers to the event target.

Handlers are invoked with the target as their `this` value even when registered using `addEventListener()`. Unfortunately, however, this is not true for `attachEvent()`: handlers registered with `attachEvent()` are invoked as functions, and their `this` value is the global (Window) object. You can work around this with code like this:

```
/*
 * Register the specified handler function to handle events of the specified
 * type on the specified target. Ensure that the handler will always be
 * invoked as a method of the target.
 */
function addEvent(target, type, handler) {
    if (target.addEventListener)
        target.addEventListener(type, handler, false);
    else
        target.attachEvent("on" + type,
                           function(event) {
                               // Invoke the handler as a method of target,
                               // passing on the event object
                               return handler.call(target, event);
                           });
}
```

Note that event handlers registered using this method cannot be removed, since the wrapper function passed to `attachEvent()` is not retained anywhere to be passed to `detachEvent()`.

17.3.3 Event Handler Scope

Like all JavaScript functions, event handlers are lexically scoped. They are executed in the scope in which they are defined, not the scope from which they are invoked, and they can access any local variables from that scope. (This is demonstrated in the `addEvent()` function above, for example.)

Event handlers registered as HTML attributes are a special case, however. They are converted into top-level functions that have access to global variables but not to any local variables. But, for historical reasons, they run with a modified scope chain. Event handlers defined by HTML attributes can use the properties of the target object, the containing `<form>` object (if there is one), and the Document object as if they are local variables. §17.2.2 shows how an event handler function is created from an HTML event handler attribute, and the code there approximates this modified scope chain using `with` statements.

HTML attributes are not natural places to include long strings of code, and this modified scope chain allows helpful shortcuts. You can use `tagName` instead of `this.tagName`. You can use `getElementById` instead of `document.getElementById`. And, for document elements that are inside a `<form>`, you can refer to any other form element by ID, using `zipcode`, for example, instead of `this.form.zipcode`.

On the other hand, the modified scope chain of HTML event handlers is a source of pitfalls, since the properties of each of the objects in the chain shadow any properties of the same name in the global object. The Document object defines a (rarely used) `open()` method, for example, so an HTML event handler that wants to invoke the `open()` method of the Window object must explicitly write `window.open` instead of `open`. There is a similar (but more pernicious) problem with forms, because the names and IDs of form elements define properties on the containing form element (see §15.9.1). So if a form contains an element with the ID "location", for example, all HTML event handlers within that form must use `window.location` instead of `location` if they want to refer to the window's Location object.

17.3.4 Handler Return Value

The return value of an event handler registered by setting an object property or an HTML attribute is sometimes significant. In general, a return value of `false` tells the browser that it should not perform the default action associated with the event. The `onclick` handler of a Submit button in a form, for example, can return `false` to prevent the browser from submitting the form. (This is useful if the user's input fails client-side validation.) Similarly, an `onkeypress` handler on an input field can filter keyboard input by returning `false` if the user types an inappropriate character. (Example 17-6 filters keyboard input in this way.)

The return value of the `onbeforeunload` handler of the Window object is also significant. This event is triggered when the browser is about to navigate to a new page. If this event handler returns a string, it will be displayed in a modal dialog box that asks the user to confirm that she wants to leave the page.

It is important to understand that event handler return values are significant only for handlers registered as properties. We'll see below that event handlers registered with `addEventListener()` or `attachEvent()` must instead call the `preventDefault()` method or set the `returnValue` property of the event object.

17.3.5 Invocation Order

A document element or other object may have more than one event handler registered for a particular type of event. When an appropriate event occurs, the browser must invoke all of the handlers, following these rules of invocation order:

- Handlers registered by setting an object property or HTML attribute, if any, are always invoked first.
- Handlers registered with `addEventListener()` are invoked in the order in which they were registered.[4]

4. The DOM Level 2 standard leaves the invocation order undefined, but current browsers all invoke the handlers in registration order and the current DOM Level 3 draft standardizes this behavior.

- Handlers registered with `attachEvent()` may be invoked in any order and your code should not depend on sequential invocation.

17.3.6 Event Propagation

When the target of an event is the Window object, or some other standalone object (such as an XMLHttpRequest), the browser responds to an event simply by invoking the appropriate handlers on that one object. When the event target is a Document or document Element, however, the situation is more complicated.

After the event handlers registered on the target element are invoked, most events "bubble" up the DOM tree. The event handlers of the target's parent are invoked. Then the handlers registered on the target's grandparent are invoked. This continues up to the Document object, and then beyond to the Window object. Event bubbling provides an alternative to registering handlers on lots of individual document elements: instead you can register a single handler on a common ancestor element and handle events there. You might register an "change" handler on a `<form>` element, for example, instead of registering a "change" handler for every element in the form.

Most events that occur on document elements bubble. Notable exceptions are the focus, blur, and scroll events. The load event on document elements bubbles, but it stops bubbling at the Document object and does not propagate on to the Window object. The load event of the Window object is triggered only when the entire document has loaded.

Event bubbling is the third "phase" of event propagation. The invocation of the event handlers of the target object itself is the second phase. The first phase, which occurs even before the target handlers are invoked, is called the "capturing" phase. Recall that `addEventListener()` takes a boolean value as its third argument. If that argument is `true`, the event handler is registered as a capturing event handler for invocation during this first phase of event propagation. Event bubbling is universally supported: it works in all browsers including IE, and it works for all handlers, regardless of how they are registered (unless they are registered as capturing event handlers). Event capturing, by contrast, only works with event handlers registered with `add Event Listener()` when the third argument is `true`. This means that event capturing is not available in IE prior to IE9, and is not, at the time of this writing, a commonly used technique.

The capturing phase of event propagation is like the bubbling phase in reverse. The capturing handlers of the Window object are invoked first, then the capturing handlers of the Document object, then of the body object, and so on down the DOM tree until the capturing event handlers of the parent of the event target are invoked. Capturing event handlers registered on the event target itself are not invoked.

Event capturing provides an opportunity to peek at events before they are delivered to their target. A capturing event handler can be used for debugging, or it can be used along with the event cancellation technique described below to filter events so that the target event handlers are never actually invoked. One common use for event capturing

is handling mouse drags, where mouse motion events need to be handled by the object being dragged, not the document elements over which it is dragged. See Example 17-2 for an example.

17.3.7 Event Cancellation

§17.3.4 explained that the return value of event handlers registered as properties can be used to cancel the browser's default action for the event. In browsers that support `addEventListener()`, you can also cancel the default action for an event by invoking the `preventDefault()` method of the event object. In IE prior to IE9, however, you do the same by setting the `returnValue` property of the event object to `false`. The following code shows a dummy event handler that uses all three cancellation techniques:

```
function cancelHandler(event) {
    var event = event || window.event;  // For IE

    /* Do something to handle the event here */

    // Now cancel the default action associated with the event
    if (event.preventDefault) event.preventDefault();  // Standard technique
    if (event.returnValue) event.returnValue = false;  // IE
    return false;        // For handlers registered as object properties
}
```

The current DOM Events module draft defines a property of the Event object named `defaultPrevented`. It is not yet widely supported, but the intent is that this property will normally be `false` but will become `true` if `preventDefault()` is called.[5]

Canceling the default action associated with an event is only one kind of event cancellation. We can also cancel the propagation of events. In browsers that support `addEventListener()`, the event object has a `stopPropagation()` method that you can invoke to prevent the continued propagation of the event. If there are other handlers defined on the same object, the rest of those handlers will still be invoked, but no event handlers on any other object will be invoked after `stopPropagation()` is called. The `stopPropagation()` method can be called at any time during event propagation. It works during the capturing phase, at the event target itself, and during the bubbling phase.

Prior to IE9, IE does not support the `stopPropagation()` method. Instead, the IE event object has a property named `cancelBubble`. Set this property to `true` to prevent any further propagation. (IE8 and before do not support the capturing phase of event propagation, so bubbling is the only kind of propagation to be canceled.)

The current draft DOM Events specification defines another method on the Event object, named `stopImmediatePropagation()`. Like `stopPropagation()`, this method prevents the propagation of the event to any other objects. But it also prevents the invocation of any other event handlers registered on the same object. At the time of this writing, some browsers support `stopImmediatePropagation()` and some do not. Some

5. The jQuery (see Chapter 19) event object has a `defaultPrevented()` method instead of a property.

utility libraries, like jQuery and YUI, define `stopImmediatePropagation()` in a cross-platform way.

17.4 Document Load Events

Now that we've covered the fundamentals of JavaScript event handling, we'll start looking in more detail at specific categories of events. We begin, in this section, with document load events.

Most web applications need notification from the web browser to tell them when the document has been loaded and is ready to be manipulated. The load event on the Window object serves this purpose and was discussed in detail in Chapter 13, which included an `onLoad()` utility function in Example 13-5. The load event does not fire until a document and all of its images are fully loaded. It is usually safe, however, to start running your scripts after the document is fully parsed but before images are downloaded. You can improve the startup time of your web applications if you trigger your scripts on events other than "load".

The DOMContentLoaded event is fired when the document has been loaded and parsed and any deferred scripts have been executed. Images and `async` scripts may still be loading, but the document is ready to be manipulated. (Deferred and async scripts are explained in §13.3.1.) This event was introduced by Firefox, and it has been adopted by all other browser vendors, including Microsoft in IE9. Despite the "DOM" in its name, it is not part of the DOM Level 3 event standard, but it is standardized by HTML5.

As described in §13.3.4, the `document.readyState` property changes as the document loads. In IE, each change in state is accompanied by a readystatechange event on the Document object, and it is possible to use this event to determine when IE reaches the "complete" state. HTML5 standardizes the readystatechange event, but fires it immediately before the load event, so it is not clear that much advantage is gained by listening for "readystatechange" instead of "load".

Example 17-1 defines a `whenReady()` function that is much like the `onLoad()` function of Example 13-5. Functions passed to `whenReady()` will be invoked (as methods of the Document object) when the document is ready to be manipulated. Unlike the earlier `onLoad()` function, `whenReady()` listens for DOMContentLoaded and readystatechange events, and uses load events only as backup for older browsers that do not support the earlier events. Some of the examples that follow (in this and subsequent chapters) use this `whenReady()` function.

Example 17-1. Invoking functions when the document is ready

```
/*
 * Pass a function to whenReady() and it will be invoked (as a method of the
 * document) when the document is parsed and ready for manipulation. Registered
 * functions are triggered by the first DOMContentLoaded, readystatechange, or
 * load event that occurs. Once the document is ready and all functions have
```

```
 * been invoked, any functions passed to whenReady() will be invoked
 * immediately.
 */
var whenReady = (function() { // This function returns the whenReady() function
    var funcs = [];     // The functions to run when we get an event
    var ready = false; // Switches to true when the handler is triggered

    // The event handler invoked when the document becomes ready
    function handler(e) {
        // If we've already run once, just return
        if (ready) return;

        // If this was a readystatechange event where the state changed to
        // something other than "complete", then we're not ready yet
        if (e.type === "readystatechange" && document.readyState !== "complete")
            return;

        // Run all registered functions.
        // Note that we look up funcs.length each time, in case calling
        // one of these functions causes more functions to be registered.
        for(var i = 0; i < funcs.length; i++)
            funcs[i].call(document);

        // Now set the ready flag to true and forget the functions
        ready = true;
        funcs = null;
    }

    // Register the handler for any event we might receive
    if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", handler, false);
        document.addEventListener("readystatechange", handler, false);
        window.addEventListener("load", handler, false);
    }
    else if (document.attachEvent) {
        document.attachEvent("onreadystatechange", handler);
        window.attachEvent("onload", handler);
    }

    // Return the whenReady function
    return function whenReady(f) {
        if (ready) f.call(document); // If already ready, just run it
        else funcs.push(f);          // Otherwise, queue it for later.
    }
}());
```

17.5 Mouse Events

There are quite a few mouse-related events. Table 17-1 lists them all. All mouse events except "mouseenter" and "mouseleave" bubble. Click events on links and Submit buttons have default actions that can be prevented. You may be able to cancel a context menu event to prevent the display of a context menu, but some browsers have configuration options that make context menus noncancelable.

Table 17-1. Mouse events

Type	Description
click	A higher-level event fired when the user presses and releases a mouse button or otherwise "activates" an element.
contextmenu	A cancelable event fired when a contextmenu is about to be popped up. Current browsers display context menus on right mouse clicks, so this event can also be used like the click event.
dblclick	Fired when the user double-clicks the mouse
mousedown	Fired when the user presses a mouse button
mouseup	Fired when the user releases a mouse button
mousemove	Fired when the user moves the mouse.
mouseover	Fired when the mouse enters an element. `relatedTarget` (or `fromElement` in IE); specifies what element the mouse is coming from.
mouseout	Fired when the mouse leaves an element. `relatedTarget` (or `toElement` in IE); specifies what element the mouse is going to.
mouseenter	Like "mouseover", but does not bubble. Introduced by IE and standardized in HTML5 but not yet widely implemented.
mouseleave	Like "mouseout", but does not bubble. Introduced by IE and standardized in HTML5 but not yet widely implemented.

The object passed to mouse event handlers has `clientX` and `clientY` properties that specify the coordinates of the mouse pointer relative to the containing window. Add the window's scroll offsets (see Example 15-8) to convert this position to document coordinates.

The `altKey`, `ctrlKey`, `metaKey`, and `shiftKey` properties specify whether various keyboard modifier keys were held down when the event occurred: this allows you to distinguish an ordinary click from a Shift-click, for example.

The `button` property specifies which mouse button, if any, was held down when the event occurred. Different browsers assign different values to this property, however, so it is difficult to use portably. See the Event reference page for details. Some browsers only fire click events for left button clicks. You should use mousedown and mouseup if you need to detect clicks of other buttons. The contextmenu event usually signals a right-button click, but as noted above, it may be impossible to prevent the appearance of a context menu when this event occurs.

The event object for mouse events has a few other mouse-specific properties, but they are not as commonly used as these. See the Event reference page for a list.

Example 17-2 shows a JavaScript function, `drag()`, that, when invoked from a `mousedown` event handler, allows an absolutely positioned document element to be dragged by the user. `drag()` works with both the DOM and IE event models.

`drag()` takes two arguments. The first is the element that is to be dragged. This may be the element on which the mousedown event occurred or a containing element (e.g.,

you might allow the user to drag on an element that looks like a titlebar to move the containing element that looks like a window). In either case, however, it must be a document element that is absolutely positioned using the CSS `position` attribute. The second argument is the event object from the triggering mousedown event. Here's a simple example that uses `drag()`. It defines an `<img>` that the user can drag if the Shift key is held down:

```
<img src="draggable.gif"
     style="position:absolute; left:100px; top:100px;"
     onmousedown="if (event.shiftKey) drag(this, event);">
```

The `drag()` function converts the position of the mousedown event to document coordinates in order to compute the distance between the mouse pointer and the upper-left corner of the element being moved. It uses `getScrollOffsets()` from Example 15-8 to help with the coordinate conversion. Next, `drag()` registers event handlers for the mousemove and mouseup events that follow the mousedown event. The mousemove handler is responsible for moving the document element, and the mouseup handler is responsible for deregistering itself and the mousemove handler.

It is important to note that the mousemove and mouseup handlers are registered as capturing event handlers. This is because the user may move the mouse faster than the document element can follow it, and if that happens, some of the mousemove events occur outside the original target element. Without capturing, those events will not be dispatched to the correct handlers. The IE event model does not support event capturing the way the standard event model does, but it does have a special-purpose `setCapture()` method for capturing mouse events in cases just like this. The example code shows how it works.

Finally, note that the `moveHandler()` and `upHandler()` functions are defined within `drag()`. Because they are defined in this nested scope, they can use the arguments and local variables of `drag()`, which considerably simplifies their implementation.

Example 17-2. Dragging document elements

```
/**
 * Drag.js: drag absolutely positioned HTML elements.
 *
 * This module defines a single drag() function that is designed to be called
 * from an onmousedown event handler. Subsequent mousemove events will
 * move the specified element. A mouseup event will terminate the drag.
 * This implementation works with both the standard and IE event models.
 * It requires the getScrollOffsets() function from elsewhere in this book.
 *
 * Arguments:
 *
 *   elementToDrag: the element that received the mousedown event or
 *     some containing element. It must be absolutely positioned. Its
 *     style.left and style.top values will be changed based on the user's
 *     drag.
 *
 *   event: the Event object for the mousedown event.
```

```
 **/
function drag(elementToDrag, event) {
    // The initial mouse position, converted to document coordinates
    var scroll = getScrollOffsets();  // A utility function from elsewhere
    var startX = event.clientX + scroll.x;
    var startY = event.clientY + scroll.y;

    // The original position (in document coordinates) of the element
    // that is going to be dragged.  Since elementToDrag is absolutely
    // positioned, we assume that its offsetParent is the document body.
    var origX = elementToDrag.offsetLeft;
    var origY = elementToDrag.offsetTop;

    // Compute the distance between the mouse down event and the upper-left
    // corner of the element. We'll maintain this distance as the mouse moves.
    var deltaX = startX - origX;
    var deltaY = startY - origY;

    // Register the event handlers that will respond to the mousemove events
    // and the mouseup event that follow this mousedown event.
    if (document.addEventListener) {  // Standard event model
        // Register capturing event handlers on the document
        document.addEventListener("mousemove", moveHandler, true);
        document.addEventListener("mouseup", upHandler, true);
    }
    else if (document.attachEvent) {  // IE Event Model for IE5-8
        // In the IE event model, we capture events by calling
        // setCapture() on the element to capture them.
        elementToDrag.setCapture();
        elementToDrag.attachEvent("onmousemove", moveHandler);
        elementToDrag.attachEvent("onmouseup", upHandler);
        // Treat loss of mouse capture as a mouseup event.
        elementToDrag.attachEvent("onlosecapture", upHandler);
    }

    // We've handled this event. Don't let anybody else see it.
    if (event.stopPropagation) event.stopPropagation();  // Standard model
    else event.cancelBubble = true;                      // IE

    // Now prevent any default action.
    if (event.preventDefault) event.preventDefault();   // Standard model
    else event.returnValue = false;                     // IE

    /**
     * This is the handler that captures mousemove events when an element
     * is being dragged. It is responsible for moving the element.
     **/
    function moveHandler(e) {
        if (!e) e = window.event;  // IE event Model

        // Move the element to the current mouse position, adjusted by the
        // position of the scrollbars and the offset of the initial click.
        var scroll = getScrollOffsets();
        elementToDrag.style.left = (e.clientX + scroll.x - deltaX) + "px";
        elementToDrag.style.top = (e.clientY + scroll.y - deltaY) + "px";
```

```
        // And don't let anyone else see this event.
        if (e.stopPropagation) e.stopPropagation();  // Standard
        else e.cancelBubble = true;                 // IE
    }

    /**
     * This is the handler that captures the final mouseup event that
     * occurs at the end of a drag.
     **/
    function upHandler(e) {
        if (!e) e = window.event;  // IE Event Model

        // Unregister the capturing event handlers.
        if (document.removeEventListener) {  // DOM event model
            document.removeEventListener("mouseup", upHandler, true);
            document.removeEventListener("mousemove", moveHandler, true);
        }
        else if (document.detachEvent) {  // IE 5+ Event Model
            elementToDrag.detachEvent("onlosecapture", upHandler);
            elementToDrag.detachEvent("onmouseup", upHandler);
            elementToDrag.detachEvent("onmousemove", moveHandler);
            elementToDrag.releaseCapture();
        }

        // And don't let the event propagate any further.
        if (e.stopPropagation) e.stopPropagation();  // Standard model
        else e.cancelBubble = true;                 // IE
    }
}
```

The following code shows how you can use `drag()` in an HTML file (it's a simplified version of Example 16-2, with the addition of dragging):

```
<script src="getScrollOffsets.js"></script> <!-- drag() requires this -->
<script src="Drag.js"></script>             <!-- defines drag() -->
<!-- The element to be dragged -->
<div style="position:absolute; left:100px; top:100px; width:250px;
            background-color: white; border: solid black;">
  <!-- The "titlebar" to drag it with. Note the onmousedown attribute. -->
  <div style="background-color: gray; border-bottom: dotted black;
              padding: 3px; font-family: sans-serif; font-weight: bold;"
       onmousedown="drag(this.parentNode, event);">
    Drag Me  <!-- The content of the titlebar -->
  </div>
<!-- Content of the draggable element -->
  <p>This is a test. Testing, testing, testing.</p><p>Test</p><p>Test</p>
</div>
```

The key here is the `onmousedown` attribute of the inner `<div>` element. Note that it uses `this.parentNode` to specify that the entire container element is to be dragged.

17.6 Mousewheel Events

All modern browsers support mouse wheels and fire events when the user rotates the mousewheel. Browsers often use the mousewheel to scroll the document or to zoom in or out, but you can cancel the mousewheel event to prevent those default actions.

There are a number of interoperability issues that affect mousewheel events, but it is possible to write code that works on all platforms. At the time of this writing, all browsers but Firefox support an event named "mousewheel". Firefox uses "DOMMouseScroll" instead. And the DOM Level 3 Events draft proposes an event named "wheel" instead of "mousewheel". In addition to the differences in event names, the objects passed to these various events use different property names to specify the amount of wheel rotation that occurred. Finally, note that there are fundamental hardware distinctions between mouse wheels as well. Some allow 1-dimensional rotation forward and back and some (particularly on Macs) also allow rotation left and right (on these mice the "wheel" is really a trackball). The DOM Level 3 standard even includes support for 3-dimensional mouse "wheels" that can report clockwise or counter-clockwise rotation in addition to forward/back and left/right.

The event object passed to a "mousewheel" handler has a `wheelDelta` property that specifies how far the user turned the wheel. One mousewheel "click" away from the user is generally a delta of 120 and one click toward the user is -120. In Safari and Chrome, to support Apple's mice that include a two-dimensional trackball instead of a one-dimensional mousewheel, the event object has `wheelDeltaX` and `wheelDeltaY` properties in addition to `wheelDelta`, and `wheelDelta` and `wheelDeltaY` are always the same value.

In Firefox you can the nonstandard DOMMouseScroll event instead of mousewheel and use the `detail` property of the event object instead of `wheelDelta`. The scaling and sign of this `detail` property is different than `wheelDelta`, however: multiply `detail` by -40 to compute the equivalent `wheelDelta` value.

At the time of this writing, the DOM Level 3 Events draft standard defines a wheel event as the standardized version of mousewheel and DOMMouseScroll. The object passed to a wheel event handler will have `deltaX`, `deltaY`, and `deltaZ` properties to specify rotation in three dimensions. You must multiply these values by -120 to match the value and sign of a mousewheel event.

For all of these event types, the event object is like a mouse event object: it includes mouse pointer coordinates and the state of the keyboard modifier keys.

Example 17-3 demonstrates how to work with mouse wheel events and how to do so interoperably across platforms. It defines a function named `enclose()` that wraps a "frame" or "viewport" of the specified size around a larger content element (such as an image) and defines a mouse wheel event handler that allows the user to pan the content element within the viewport and also to resize the viewport. You might use this `enclose()` function with code like this:

```
<script src="whenReady.js"></script>
<script src="Enclose.js"></script>
<script>
whenReady(function() {
    enclose(document.getElementById("content"),400,200,-200,-300);
});
</script>
<style>div.enclosure {  border: solid black 10px; margin: 10px; }</style>
<img id="content" src="testimage.jpg"/>
```

In order to work correctly in all common browsers, Example 17-3 must perform some browser testing (§13.4.5). The example anticipates the DOM Level 3 Events specification and includes code to use the wheel event when browsers implement it.[6] It also includes some future-proofing to stop using the DOMMouseScroll event when Firefox starts firing wheel or mousewheel. Note that Example 17-3 is also a practical example of the element geometry and CSS positioning techniques explained in §15.8 and §16.2.1.

Example 17-3. Handling mousewheel events

```
// Enclose the content element in a frame or viewport of the specified width
// and height (minimum 50x50). The optional contentX and contentY arguments
// specify the initial offset of the content relative to the frame. (If
// specified, they must be <= 0.) The frame has mousewheel event handlers that
// allow the user to pan the element, and to shrink or enlarge the frame.
function enclose(content, framewidth, frameheight, contentX, contentY) {
    // These arguments aren't just the initial values: they maintain the
    // current state and are used and modified by the mousewheel handler.
    framewidth = Math.max(framewidth, 50);
    frameheight = Math.max(frameheight, 50);
    contentX = Math.min(contentX, 0) || 0;
    contentY = Math.min(contentY, 0) || 0;

    // Create the frame element and set a CSS classname and styles
    var frame = document.createElement("div");
    frame.className = "enclosure"; // So we can define styles in a stylesheet
    frame.style.width = framewidth + "px";       // Set the frame size.
    frame.style.height = frameheight + "px";
    frame.style.overflow = "hidden";             // No scrollbars, no overflow
    frame.style.boxSizing = "border-box";        // Border-box simplifies the
    frame.style.webkitBoxSizing = "border-box"; // calculations for resizing
    frame.style.MozBoxSizing = "border-box";     // the frame.

    // Put the frame in the document and move the content elt into the frame.
    content.parentNode.insertBefore(frame, content);
    frame.appendChild(content);

    // Position the element relative to the frame
    content.style.position = "relative";
    content.style.left = contentX + "px";
    content.style.top = contentY + "px";
```

6. This is risky: if future implementations do not match the draft specification current as I write this, this will backfire and the example will break.

```
// We'll need to work around some browser-specific quirks below
var isMacWebkit = (navigator.userAgent.indexOf("Macintosh") !== -1 &&
                   navigator.userAgent.indexOf("WebKit") !== -1);
var isFirefox = (navigator.userAgent.indexOf("Gecko") !== -1);

// Register mousewheel event handlers.
frame.onwheel = wheelHandler;       // Future browsers
frame.onmousewheel = wheelHandler;  // Most current browsers
if (isFirefox)                      // Firefox only
    frame.addEventListener("DOMMouseScroll", wheelHandler, false);

function wheelHandler(event) {
    var e = event || window.event;  // Standard or IE event object

    // Extract the amount of rotation from the event object, looking
    // for properties of a wheel event object, a mousewheel event object
    // (in both its 2D and 1D forms), and the Firefox DOMMouseScroll event.
    // Scale the deltas so that one "click" toward the screen is 30 pixels.
    // If future browsers fire both "wheel" and "mousewheel" for the same
    // event, we'll end up double-counting it here. Hopefully, however,
    // cancelling the wheel event will prevent generation of mousewheel.
    var deltaX = e.deltaX*-30 ||  // wheel event
                 e.wheelDeltaX/4 ||  // mousewheel
                           0;     // property not defined
    var deltaY = e.deltaY*-30 ||  // wheel event
                 e.wheelDeltaY/4 ||  // mousewheel event in Webkit
(e.wheelDeltaY===undefined &&      // if there is no 2D property then
                 e.wheelDelta/4) ||  // use the 1D wheel property
                    e.detail*-10 ||  // Firefox DOMMouseScroll event
                           0;     // property not defined

    // Most browsers generate one event with delta 120 per mousewheel click.
    // On Macs, however, the mousewheels seem to be velocity-sensitive and
    // the delta values are often larger multiples of 120, at
    // least with the Apple Mouse. Use browser-testing to defeat this.
    if (isMacWebkit) {
        deltaX /= 30;
        deltaY /= 30;
    }

    // If we ever get a mousewheel or wheel event in (a future version of)
    // Firefox, then we don't need DOMMouseScroll anymore.
    if (isFirefox && e.type !== "DOMMouseScroll")
        frame.removeEventListener("DOMMouseScroll", wheelHandler, false);

    // Get the current dimensions of the content element
    var contentbox = content.getBoundingClientRect();
    var contentwidth = contentbox.right - contentbox.left;
    var contentheight = contentbox.bottom - contentbox.top;

    if (e.altKey) {  // If Alt key is held down, resize the frame
        if (deltaX) {
            framewidth -= deltaX; // New width, but not bigger than the
            framewidth = Math.min(framwidth, contentwidth);  // content
            framewidth = Math.max(framewidth,50);   // and no less than 50.
```

```
            frame.style.width = framewidth + "px";  // Set it on frame
        }
        if (deltaY) {
            frameheight -= deltaY;  // Do the same for the frame height
            frameheight = Math.min(frameheight, contentheight);
            frameheight = Math.max(frameheight-deltaY, 50);
            frame.style.height = frameheight + "px";
        }
    }
    else { // Without the Alt modifier, pan the content within the frame
        if (deltaX) {
            // Don't scroll more than this
            var minoffset = Math.min(framewidth-contentwidth, 0);
            // Add deltaX to contentX, but don't go lower than minoffset
            contentX = Math.max(contentX + deltaX, minoffset);
            contentX = Math.min(contentX, 0);      // or higher than 0
            content.style.left = contentX + "px"; // Set new offset
        }
        if (deltaY) {
            var minoffset = Math.min(frameheight - contentheight, 0);
            // Add deltaY to contentY, but don't go lower than minoffset
            contentY = Math.max(contentY + deltaY, minoffset);
            contentY = Math.min(contentY, 0);      // Or higher than 0
            content.style.top = contentY + "px";  // Set the new offset.
        }
    }

    // Don't let this event bubble. Prevent any default action.
    // This stops the browser from using the mousewheel event to scroll
    // the document. Hopefully calling preventDefault() on a wheel event
    // will also prevent the generation of a mousewheel event for the
    // same rotation.
    if (e.preventDefault) e.preventDefault();
    if (e.stopPropagation) e.stopPropagation();
    e.cancelBubble = true;  // IE events
    e.returnValue = false;  // IE events
    return false;
  }
}
```

17.7 Drag and Drop Events

Example 17-2 showed how to respond to mouse drags within an application. It is possible to use techniques like that to allow elements to be dragged and "dropped" within a web page, but true "drag-and-drop" is something else. Drag-and-drop (or DnD) is a user interface for transferring data between a "drag source" and "drop target" that may be in the same application or in different applications. DnD is a complex human/computer interaction, and APIs for implementing DnD are always complicated:

- They have to tie into the underlying OS so that they can work between unrelated applications.

- They must accommodate "move", "copy", and "link" data-transfer operations, allow the drag source and drop target to restrict the set of allowed operations, and then allow the user to choose (usually using keyboard modifiers) among the permitted set.
- They must provide a way for a drag source to specify the icon or image to be dragged.
- They must provide event-based notification to both the drag source and the drop target of the progress of the DnD interaction.

Microsoft introduced a DnD API into early versions of IE. It was not a well-designed or well-documented API, but other browsers have attempted to replicate it, and HTML5 standardizes something like the IE API and then adds new features that make the API much easier to use. This new easy-to-use DnD API is not implemented at the time of this writing, so this section covers the IE API, as blessed by the HTML5 standard.

The IE DnD API is tricky to use and implementation differences in current browsers make it impossible to use some of the more sophisticated parts of the API interoperably, but it does allow web applications to participate in interapplication DnD like regular desktop applications can. Browsers have always been able to perform simple DnD. If you select text in a web browser, it is easy to drag that text into a word processor. And if you select a URL in a word processor, you can drag it to the browser to make the browser visit the URL. What this section demonstrates is how to create custom drag sources that transfer data other than their textual content and custom drop targets that respond to dropped data in some way other than simply displaying it.

DnD is always event-based and the JavaScript API involves two sets of events: one set that is fired on the drag source and another set that is fired on the drop target. All DnD event handlers are passed an event object that is like a mouse event object, with the addition of a `dataTransfer` property. This property refers to a DataTransfer object that defines the methods and properties of the DnD API.

Drag source events are relatively simple and we'll begin with them. Any document element that has the HTML `draggable` attribute is a drag source. When the user begins a mouse drag over a drag source, the browser does not select the element content: instead, it fires a `dragstart` event on the element. Your handler for that event should call `dataTransfer.setData()` to specify the data (and the type of that data) that the drag source is making available. (When the new HTML5 API is implemented, you might call `dataTransfer.items.add()` instead.) Your handler may also want to set `dataTransfer.effectAllowed` to specify which of the "move", "copy", and "link" transfer operations are supported and it might want to call `dataTransfer.setDragImage()` or `dataTransfer.addElement()` (in browsers that support those methods) to specify an image or document element to be used as the visual representation of the drag.

While the drag progresses, the browser fires drag events on the drag source. You can listen for these events if you want to update the drag image or alter the data being offered, but it is not generally necessary to register "drag" handlers.

When a drop occurs, the dragend event is fired. If your drag source supports a "move" operation, it should check `dataTransfer.dropEffect` to see if a move operation was actually performed. If so, the data has been transferred elsewhere, and you should delete it from the drag source.

The dragstart event is the only one you need to implement simple custom drag sources. Example 17-4 is an example. It displays the current time in "hh:mm" format in a `<span>` element and updates the time once a minute. If this was all the example did, the user could select the text displayed in the clock and then drag the time. But the JavaScript code in this example makes the clock into a custom drag source by setting the `draggable` property of this clock element to `true` and defining an `ondragstart` event handler function. The event handler uses `dataTransfer.setData()` to specify a complete timestamp string (including the date, seconds, and timezone information) as the data to be dragged. It also calls `dataTransfer.setDragIcon()` to specify an image (a clock icon) to be dragged.

Example 17-4. A custom drag source

```
<script src="whenReady.js"></script>
<script>
whenReady(function() {
    var clock = document.getElementById("clock");  // The clock element
    var icon = new Image();                        // An image to drag
    icon.src = "clock-icon.png";                   // Image URL

    // Display the time once every minute
    function displayTime() {
        var now = new Date();                 // Get current time
        var hrs = now.getHours(), mins = now.getMinutes();
        if (mins < 10) mins = "0" + mins;
        clock.innerHTML = hrs + ":" + mins; // Display current time
        setTimeout(displayTime, 60000);     // Run again in 1 minute
    }
    displayTime();

    // Make the clock draggable
    // We can also do this with an HTML attribute: <span draggable="true">...
    clock.draggable = true;

    // Set up drag event handlers
    clock.ondragstart = function(event) {
        var event = event || window.event; // For IE compatability

        // The dataTransfer property is key to the drag-and-drop API
        var dt = event.dataTransfer;

        // Tell the browser what is being dragged.
        // The Date() constructor used as a function returns a timestamp string
        dt.setData("Text", Date() + "\n");

        // Tell the browser to drag our icon to represent the timestamp, in
        // browsers that support that. Without this line, the browser may
        // use an image of the clock text as the value to drag.
```

```
        if (dt.setDragImage) dt.setDragImage(icon, 0, 0);
    };
});
</script>
<style>
#clock { /* Make the clock look nice */
    font: bold 24pt sans; background: #ddf; padding: 10px;
    border: solid black 2px; border-radius: 10px;
}
</style>
<h1>Drag timestamps from the clock</h1>
<span id="clock"></span>  <!-- The time is displayed here -->
<textarea cols=60 rows=20></textarea> <!-- You can drop timestamps here -->
```


Drop targets are trickier than drag sources. Any document element can be a drop target: there is no need to set an HTML attribute as there is for drag sources; you simply need to define appropriate event listeners. (With the new HTML5 DnD API, however, you will be able to define a `dropzone` attribute on the drop target instead of defining some of the event handlers described below.) There are four events that are fired at drop targets. When a dragged object enters a document element, the browser fires a dragenter event on that element. Your drop target should use the `dataTransfer.types` property to determine whether the dragged object has data available in a format that it can understand. (You might also want to check `dataTransfer.effectAllowed` to ensure that the drag source and your drop target can agree on one of the move, copy, and link operations.) If these checks succeed, your drop target must let both the user and the browser know that it is interested in a drop. You can give this feedback to the user by changing its border or background color. Surprisingly, a drop target tells the browser that it is interested in a drop by canceling the event.

If an element does not cancel the dragenter event the browser sends it, the browser will not treat it as a drop target for this drag and will not send it any more events. But if a drop target does cancel the dragenter event, the browser will send dragover events as the user continues to drag the object over that target. Surprisingly (again) a drop target must listen for and cancel all of these events to indicate its continued interest in the drop. If the drop target wants to specify that it only allows move, copy, or link operations, it should use this dragover event handler to set `dataTransfer.dropEffect`.

If the user moves the dragged object off of a drop target that has indicated interest by canceling events, then the dragleave event will be fired on the drop target. The handler for this event should restore the element's border or background color or undo any other visual feedback performed in response to the dragenter event. Unfortunately, both the dragenter and dragleave events bubble and if a drop target has elements nested within it, it is difficult to know whether a dragleave event means that the drag has left the drop target for an event outside of the target or for an event inside the target.

Finally, if the user drops an object on a drop target, the drop event is fired on the drop target. The handler for this event should use the `dataTransfer.getData()` to obtain the data that was transferred and do something appropriate with it. Alternatively, if the user dropped one or more files on the drop target, the `dataTransfer.files` property

will be an array-like object of File objects. (See Example 18-11 for a demonstration.) With the new HTML5 API, drop event handlers will be able to loop through the elements of `dataTransfer.items[]` to examine both files and nonfile data.

Example 17-5 demonstrates how to make `<ul>` elements into drop targets and how to make the `<li>` elements within them into drag sources. The example is a piece of unobtrusive JavaScript that looks for `<ul>` elements with a `class` attribute that includes "dnd" and registers DnD event handlers on any such lists it finds. The event handlers make the list itself into a drop target: any text you drop onto the list is turned into a new list item and is inserted at the end of the list. The event handlers also listen for drags on the items within the list and make the text of each list item available for transfer. The drag source handlers allow "copy" and "move" operations and delete list items that are dropped in move operations. (Note, however, that not all browsers support move operations interoperably.)

Example 17-5. A list as drop target and drag source

```
/*
 * The DnD API is quite complicated, and browsers are not fully interoperable.
 * This example gets the basics right, but each browser is a little different
 * and each one seems to have its own unique bugs. This code does not attempt
 * browser-specific workarounds.
 */
whenReady(function() {  // Run this function when the document is ready

    // Find all <ul class='dnd'> elements and call the dnd() function on them
    var lists = document.getElementsByTagName("ul");
    var regexp = /\bdnd\b/;
    for(var i = 0; i < lists.length; i++)
        if (regexp.test(lists[i].className)) dnd(lists[i]);

    // Add drag-and-drop handlers to a list element
    function dnd(list) {
        var original_class = list.className;  // Remember original CSS class
        var entered = 0;                      // Track enters and leaves

        // This handler is invoked when a drag first enters the list. It checks
        // that the drag contains data in a format it can process and, if so,
        // returns false to indicate interest in a drop. In that case, it also
        // highlights the drop target to let the user know of that interest.
        list.ondragenter = function(e) {
            e = e || window.event;  // Standard or IE event
            var from = e.relatedTarget;

            // dragenter and dragleave events bubble, which makes it tricky to
            // know when to highlight or unhighlight the element in a case like
            // this where the <ul> element has <li> children. In browsers that
            // define relatedTarget we can track that.
            // Otherwise, we count enter/leave pairs

            // If we entered from outside the list or if
            // this is the first entrance then we need to do some stuff
            entered++;
```

```
    if ((from && !ischild(from, list)) || entered == 1) {

        // All the DnD info is in this dataTransfer object
        var dt = e.dataTransfer;

        // The dt.types object lists the types or formats that the data
        // being dragged is available in. HTML5 says the type has a
        // contains() method. In some browsers it is an array with an
        // indexOf method. In IE8 and before, it simply doesn't exist.
        var types = dt.types;     // What formats data is available in

        // If we don't have any type data or if data is
        // available in plain text format, then highlight the
        // list to let the user know we're listening for drop
        // and return false to let the browser know.
        if (!types ||                                          // IE
            (types.contains && types.contains("text/plain")) || //HTML5
            (types.indexOf && types.indexOf("text/plain")!=-1)) //Webkit
        {
            list.className = original_class + " droppable";
            return false;
        }
        // If we don't recognize the data type, we don't want a drop
        return;   // without canceling
    }
    return false; // If not the first enter, we're still interested
};

// This handler is invoked as the mouse moves over the list.
// We have to define this handler and return false or the drag
// will be canceled.
list.ondragover = function(e) { return false; };

// This handler is invoked when the drag moves out of the list
// or out of one of its children. If we are actually leaving the list
// (not just going from one list item to another), then unhighlight it.
list.ondragleave = function(e) {
    e = e || window.event;
    var to = e.relatedTarget;

    // If we're leaving for something outside the list or if this leave
    // balances out the enters, then unhighlight the list
    entered--;
    if ((to && !ischild(to,list)) || entered <= 0) {
        list.className = original_class;
        entered = 0;
    }
    return false;
};

// This handler is invoked when a drop actually happens.
// We take the dropped text and make it into a new <li> element
list.ondrop = function(e) {
    e = e || window.event;       // Get the event

    // Get the data that was dropped in plain text format.
```

```
    // "Text" is a nickname for "text/plain".
    // IE does not support "text/plain", so we use "Text" here.
    var dt = e.dataTransfer;        // dataTransfer object
    var text = dt.getData("Text"); // Get dropped data as plain text.

    // If we got some text, turn it into a new item at list end.
    if (text) {
        var item = document.createElement("li"); // Create new <li>
        item.draggable = true;                   // Make it draggable
        item.appendChild(document.createTextNode(text)); // Add text
        list.appendChild(item);                  // Add it to the list

        // Restore the list's original style and reset the entered count
        list.className = original_class;
        entered = 0;

        return false;
    }
};

// Make all items that were originally in the list draggable
var items = list.getElementsByTagName("li");
for(var i = 0; i < items.length; i++)
    items[i].draggable = true;

// And register event handlers for dragging list items.
// Note that we put these handlers on the list and let events
// bubble up from the items.

// This handler is invoked when a drag is initiated within the list.
list.ondragstart = function(e) {
    var e = e || window.event;
    var target = e.target || e.srcElement;
    // If it bubbled up from something other than a <li>, ignore it
    if (target.tagName !== "LI") return false;
    // Get the all-important dataTransfer object
    var dt = e.dataTransfer;
    // Tell it what data we have to drag and what format it is in
    dt.setData("Text", target.innerText || target.textContent);
    // Tell it we know how to allow copies or moves of the data
    dt.effectAllowed = "copyMove";
};

// This handler is invoked after a successful drop occurs
list.ondragend = function(e) {
    e = e || window.event;
    var target = e.target || e.srcElement;

    // If the drop was a move, then delete the list item.
    // In IE8, this will be "none" unless you explicitly set it to
    // move in the ondrop handler above.  But forcing it to "move" for
    // IE prevents other browsers from giving the user a choice of a
    // copy or move operation.
    if (e.dataTransfer.dropEffect === "move")
        target.parentNode.removeChild(target);
}
```

```
        // This is the utility function we used in ondragenter and ondragleave.
        // Return true if a is a child of b.
        function ischild(a,b) {
            for(; a; a = a.parentNode) if (a === b) return true;
            return false;
        }
    }
});
```


17.8 Text Events

Browsers have three legacy events for keyboard input. The keydown and keyup events are low-level events that are covered in the next section. The keypress event, however, is a higher-level event that signals that a printable character has been generated. The DOM Level 3 Events draft specification defines a more general textinput event triggered whenever the user inputs text regardless of the source (a keyboard, data transfer in the form of a paste or a drop, an Asian-language input method, or a voice or handwriting recognition system, for example). The textinput event is not supported at the time of this writing, but Webkit browsers support a very similar "textInput" (with a capital letter I) event.

The proposed textinput event and the currently implemented textInput event are passed a simple event object with a `data` property that holds the input text. (Another property, `inputMethod`, is proposed to specify the source of the input, but it has not yet been implemented.) For keyboard input, the `data` property will usually hold only a single character, but input from other sources may often include multiple characters.

The event object passed with keypress events is more confusing. A keypress event represents a single character of input. The event object specifies that character as a numeric Unicode codepoint, and you must use `String.fromCharCode()` to convert it to a string. In most browsers, the `keyCode` property of the event object specifies the codepoint of the input character. For historical reasons, however, Firefox uses the `charCode` property instead. Most browser only fire keypress events when a printable character is generated. Firefox, however, also fires "keypress" for nonprinting characters. To detect this case (so you can ignore the nonprinting characters), you can look for an event object with a `charCode` property that is defined but set to 0.

The textinput, textInput, and keypress events can be canceled to prevent the character from being input. This means you can use these events to filter input. You might want to prevent a user from entering letters into a field intended for numeric data, for example. Example 17-6 is an unobtrusive module of JavaScript code that allows exactly this sort of filtering. It looks for `<input type=text>` elements that have an additional (nonstandard) attribute named `data-allowed-chars`. The module registers handlers for textinput, textInput, and keypress events on any such text field to restrict input to characters that appear in the value of the allowed attribute. The initial comment at the top of Example 17-6 includes some sample HTML that uses the module.

Example 17-6. Filtering user input

```
/**
 * InputFilter.js: unobtrusive filtering of keystrokes for <input> elements
 *
 * This module finds all <input type="text"> elements in the document that
 * have an "data-allowed-chars" attribute. It registers keypress, textInput, and
 * textinput event handlers for any such element to restrict the user's input
 * so that only characters that appear in the value of the attribute may be
 * entered. If the <input> element also has an attribute named "data-messageid",
 * the value of that attribute is taken to be the id of another document
 * element. If the user types a character that is not allowed, the message
 * element is made visible. If the user types a character that is allowed, the
 * message element is hidden. This message id element is intended to offer
 * an explanation to the user of why her keystroke was rejected. It should
 * typically be styled with CSS so that it is initially invisible.
 *
 * Here is sample HTML that uses this module.
 *   Zipcode: <input id="zip" type="text"
 *                   data-allowed-chars="0123456789" data-messageid="zipwarn">
 *   <span id="zipwarn" style="color:red;visibility:hidden">Digits only</span>
 *
 * This module is purely unobtrusive: it does not define any symbols in
 * the global namespace.
 */
whenReady(function () {  // Run this function when the document is loaded
    // Find all <input> elements
    var inputelts = document.getElementsByTagName("input");
    // Loop through them all
    for(var i = 0 ; i < inputelts.length; i++) {
        var elt = inputelts[i];
        // Skip those that aren't text fields or that don't have
        // a data-allowed-chars attribute.
        if (elt.type != "text" || !elt.getAttribute("data-allowed-chars"))
            continue;

        // Register our event handler function on this input element
        // keypress is a legacy event handler that works everywhere.
        // textInput (mixed-case) is supported by Safari and Chrome in 2010.
        // textinput (lowercase) is the version in the DOM Level 3 Events draft.
        if (elt.addEventListener) {
            elt.addEventListener("keypress", filter, false);
            elt.addEventListener("textInput", filter, false);
            elt.addEventListener("textinput", filter, false);
        }
        else { // textinput not supported versions of IE w/o addEventListener()
            elt.attachEvent("onkeypress", filter);
        }
    }

    // This is the keypress and textInput handler that filters the user's input
    function filter(event) {
        // Get the event object and the target element target
        var e = event || window.event;         // Standard or IE model
        var target = e.target || e.srcElement; // Standard or IE model
        var text = null;                       // The text that was entered
```

```
        // Get the character or text that was entered
        if (e.type === "textinput" || e.type === "textInput") text = e.data;
        else {  // This was a legacy keypress event
            // Firefox uses charCode for printable key press events
            var code = e.charCode || e.keyCode;

            // If this keystroke is a function key of any kind, do not filter it
            if (code < 32 ||           // ASCII control character
                e.charCode == 0 ||     // Function key (Firefox only)
                e.ctrlKey || e.altKey) // Modifier key held down
                return;                // Don't filter this event

            // Convert character code into a string
            var text = String.fromCharCode(code);
        }

        // Now look up information we need from this input element
        var allowed = target.getAttribute("data-allowed-chars"); // Legal chars
        var messageid = target.getAttribute("data-messageid");   // Message id
        if (messageid)  // If there is a message id, get the element
            var messageElement = document.getElementById(messageid);

        // Loop through the characters of the input text
        for(var i = 0; i < text.length; i++) {
            var c = text.charAt(i);
            if (allowed.indexOf(c) == -1) { // Is this a disallowed character?
                // Display the message element, if there is one
                if (messageElement) messageElement.style.visibility = "visible";

                // Cancel the default action so the text isn't inserted
                if (e.preventDefault) e.preventDefault();
                if (e.returnValue) e.returnValue = false;
                return false;
            }
        }

        // If all the characters were legal, hide the message if there is one.
        if (messageElement) messageElement.style.visibility = "hidden";
    }
});
```

The keypress and textinput events are triggered before the newly input text is actually inserted into the focused document element, which is why handlers for these events can cancel the event and prevent the insertion of the text. Browsers also implement an input event type that is fired after text is inserted into an element. These events cannot be canceled, and they do not specify what the new text was in their event object, but they do provide notification that the textual content of an element has changed in some way. If you wanted to ensure that any text entered into an input field was in uppercase, for example, you might use the input event like this:

```
SURNAME: <input type="text" oninput="this.value = this.value.toUpperCase();">
```

HTML 5 standardizes the input event and it is supported by all modern browsers except IE. You can achieve a similar effect in IE by using the nonstandard propertychange event to detect changes to the value property of a text input element. Example 17-7 shows how you might force all input to uppercase in a cross-platform way.

Example 17-7. Using the propertychange event to detect text input

```
function forceToUpperCase(element) {
    if (typeof element === "string") element = document.getElementById(element);
    element.oninput = upcase;
    element.onpropertychange = upcaseOnPropertyChange;

    // Easy case: the handler for the input event
    function upcase(event) { this.value = this.value.toUpperCase(); }
    // Hard case: the handler for the propertychange event
    function upcaseOnPropertyChange(event) {
        var e = event || window.event;
        // If the value property changed
        if (e.propertyName === "value") {
            // Remove onpropertychange handler to avoid recursion
            this.onpropertychange = null;
            // Change the value to all uppercase
            this.value = this.value.toUpperCase();
            // And restore the original propertychange handler
            this.onpropertychange = upcaseOnPropertyChange;
        }
    }
}
```

17.9 Keyboard Events

The keydown and keyup events are fired when the user presses or releases a key on the keyboard. They are generated for modifier keys, function keys, and alphanumeric keys. If the user holds the key down long enough for it to begin repeating, there will be multiple keydown events before the keyup event arrives.

The event object associated with these events has a numeric `keyCode` property that specifies which key was pressed. For keys that generate printable characters, the `keyCode` is generally the Unicode encoding of the primary character that appears on the key. Letter keys always generate uppercase `keyCode` values, regardless of the state of the Shift key since that is what appears on the physical key. Similarly, number keys always generate `keyCode` values for the digit that appears on the key, even if you are holding down Shift in order to type a punctuation character. For nonprinting keys, the `keyCode` property will be some other value. These `keyCode` values have never been standardized, but reasonable cross-browser compatibility is possible, and Example 17-8 includes a mapping from `keyCode` values to function key names.

Like mouse event objects, key event objects have `altKey`, `ctrlKey`, `metaKey`, and `shiftKey` properties, which are set to `true` if the corresponding modifier key is held down when the event occurs.

The keydown and keyup events and the keyCode property have been in use for more than a decade but have never been standardized. The DOM Level 3 Events draft standard does standardize the keydown and keyup event types, but it does not attempt to standardize keyCode. Instead, it defines a new key property that contains the key name as a string. If the key corresponds to a printable character, the key property will just be that printable character. If the key is a function key, the key property will be a value like "F2", "Home", or "Left".

The DOM Level 3 key property is not yet implemented in any browsers at the time of this writing. The Webkit-based browsers Safari and Chrome define a keyIdentifier property in the event object for these events, however. Like key, keyIdentifier is a string rather than a number and it has useful values like "Shift" and "Enter" for function keys. For printing keys, this property holds a less useful string representation of the Unicode encoding of the character. It is "U+0041" for the A key, for example.

Example 17-8 defines a Keymap class that maps keystroke identifiers such as "PageUp", "Alt_Z", and "ctrl+alt+shift+F5" to JavaScript functions that are invoked in response to those keystrokes. Pass key bindings to the Keymap() constructor in the form of a JavaScript object in which property names are keystroke identifiers and property values are handler functions. Add and remove bindings with the bind() and unbind() methods. Install a Keymap on an HTML element (often the Document object) with the install() method. Installing a keymap on an element registers a keydown event handler on that element. Each time a key is pressed, the handler checks to see if there is a function associated with that keystroke. If there is, it invokes it. The keydown handler uses the DOM Level 3 key property, if it is defined. If not, it looks for the Webkit keyIdentifier property and uses that. Otherwise, it falls back on the nonstandard keyCode property. Example 17-8 begins with a long comment that explains the module in more detail.

Example 17-8. A Keymap class for keyboard shortcuts

```
/*
 * Keymap.js: bind key events to handler functions.
 *
 * This module defines a Keymap class. An instance of this class represents a
 * mapping of key identifiers (defined below) to handler functions. A Keymap
 * can be installed on an HTML element to handle keydown events. When such an
 * event occurs, the Keymap uses its mapping to invoke the appropriate handler.
 *
 * When you create a Keymap, you can pass a JavaScript object that represents
 * the initial set of bindings for the Keymap. The property names of this object
 * are key identifers, and the property values are the handler functions.
 * After a Keymap has been created, you can add new bindings by passing a key
 * identifer and handler function to the bind() method. You can remove a
 * binding by passing a key identifier to the unbind() method.
 *
 * To make use of a Keymap, call its install() method, passing an HTML element,
 * such as the document object. install() adds an onkeydown event handler to
 * the specified object. When this handler is invoked, it determines the key
 * identifier of the pressed key and invokes the handler function, if any,
```

```
 * bound to that key identifier. A single Keymap may be installed on more than
 * one HTML element.
 *
 * Key Identifiers
 *
 * A key identifier is a case-insensitive string representation of a key plus
 * any modifier keys that are held down at the same time. The key name is
 * usually the (unshifted) text on the key. Legal key names include "A", "7",
 * "F2", "PageUp", "Left", "Backspace", and "Esc".
 *
 * See the Keymap.keyCodeToKeyName object in this module for a list of names.
 * These are a subset of the names defined by the DOM Level 3 standard and
 * this class will use the key property of the event object when implemented.
 *
 * A key identifier may also include modifier key prefixes. These prefixes are
 * Alt, Ctrl, Meta, and Shift. They are case-insensitive, and must be separated
 * from the key name and from each other with spaces or with an underscore,
 * hyphen, or +. For example: "SHIFT+A", "Alt_F2", "meta-v", and "ctrl alt left".
 * On Macs, Meta is the Command key and Alt is the Option key. Some browsers
 * map the Windows key to the Meta modifier.
 *
 * Handler Functions
 *
 * Handlers are invoked as methods of the document or document element on which
 * the keymap is installed and are passed two arguments:
 *   1) the event object for the keydown event
 *   2) the key identifier of the key that was pressed
 * The handler return value becomes the return value of the keydown handler.
 * If a handler function returns false, the keymap will stop bubbling and
 * cancel any default action associated with the keydown event.
 *
 * Limitations
 *
 * It is not possible to bind a handler function to all keys. The operating
 * system traps some key sequences (Alt-F4, for example). And the browser
 * itself may trap others (Ctrl-S, for example). This code is browser, OS,
 * and locale-dependent. Function keys and modified function keys work well,
 * and unmodified alphanumeric keys work well. The combination of Ctrl and Alt
 * with alphanumeric characters is less robust.
 *
 * Most punctuation characters that do not require the Shift key (`=[];',./\
 * but not hyphen) on standard US keyboard layouts are supported. But they are
 * not particularly portable to other keyboard layouts and should be avoided.
 */

// This is the constructor function
function Keymap(bindings) {
    this.map = {};     // Define the key identifier->handler map
    if (bindings) {    // Copy initial bindings into it
        for(name in bindings) this.bind(name, bindings[name]);
    }
}

// Bind the specified key identifier to the specified handler function
Keymap.prototype.bind = function(key, func) {
    this.map[Keymap.normalize(key)] = func;
```

```
};

// Delete the binding for the specified key identifier
Keymap.prototype.unbind = function(key) {
    delete this.map[Keymap.normalize(key)];
};

// Install this Keymap on the specified HTML element
Keymap.prototype.install = function(element) {
    // This is the event-handler function
    var keymap = this;
    function handler(event) { return keymap.dispatch(event, element); }

    // Now install it
    if (element.addEventListener)
        element.addEventListener("keydown", handler, false);
    else if (element.attachEvent)
        element.attachEvent("onkeydown", handler);
};

// This method dispatches key events based on the keymap bindings.
Keymap.prototype.dispatch = function(event, element) {
    // We start off with no modifiers and no key name
    var modifiers = ""
    var keyname = null;

    // Build the modifier string in canonical lowercase alphabetical order.
    if (event.altKey) modifiers += "alt_";
    if (event.ctrlKey) modifiers += "ctrl_";
    if (event.metaKey) modifiers += "meta_";
    if (event.shiftKey) modifiers += "shift_";

    // The keyname is easy if the DOM Level 3 key property is implemented:
    if (event.key) keyname = event.key;
    // Use the keyIdentifier on Safari and Chrome for function key names
    else if (event.keyIdentifier && event.keyIdentifier.substring(0,2) !== "U+")
        keyname = event.keyIdentifier;
    // Otherwise, use the keyCode property and the code-to-name map below
    else keyname = Keymap.keyCodeToKeyName[event.keyCode];

    // If we couldn't figure out a key name, just return and ignore the event.
    if (!keyname) return;

    // The canonical key id is modifiers plus lowercase key name
    var keyid = modifiers + keyname.toLowerCase();

    // Now see if the key identifier is bound to anything
    var handler = this.map[keyid];

    if (handler) {  // If there is a handler for this key, handle it
        // Invoke the handler function
        var retval = handler.call(element, event, keyid);

        // If the handler returns false, cancel default and prevent bubbling
        if (retval === false) {
            if (event.stopPropagation) event.stopPropagation();  // DOM model
```

```
            else event.cancelBubble = true;                    // IE model
            if (event.preventDefault) event.preventDefault();  // DOM
            else event.returnValue = false;                    // IE
        }

        // Return whatever the handler returned
        return retval;
    }
};

// Utility function to convert a key identifier to canonical form.
// On non-Macintosh hardware, we could map "meta" to "ctrl" here, so that
// Meta-C would be "Command-C" on the Mac and "Ctrl-C" everywhere else.
Keymap.normalize = function(keyid) {
    keyid = keyid.toLowerCase();           // Everything lowercase
    var words = keyid.split(/\s+|[\-+_]/); // Split modifiers from name
    var keyname = words.pop();             // keyname is the last word
    keyname = Keymap.aliases[keyname] || keyname; // Is it an alias?
    words.sort();                          // Sort remaining modifiers
    words.push(keyname);                   // Add the normalized name back
    return words.join("_");                // Concatenate them all
};

Keymap.aliases = {        // Map common key aliases to their "official"
    "escape":"esc",       // key names used by DOM Level 3 and by
    "delete":"del",       // the key code to key name map below.
    "return":"enter",     // Both keys and values must be lowercase here.
    "ctrl":"control",
    "space":"spacebar",
    "ins":"insert"
};

// The legacy keyCode property of the keydown event object is not standardized
// But the following values seem to work for most browsers and OSes.
Keymap.keyCodeToKeyName = {
    // Keys with words or arrows on them
    8:"Backspace", 9:"Tab", 13:"Enter", 16:"Shift", 17:"Control", 18:"Alt",
    19:"Pause", 20:"CapsLock", 27:"Esc", 32:"Spacebar", 33:"PageUp",
    34:"PageDown", 35:"End", 36:"Home", 37:"Left", 38:"Up", 39:"Right",
    40:"Down", 45:"Insert", 46:"Del",

    // Number keys on main keyboard (not keypad)
    48:"0",49:"1",50:"2",51:"3",52:"4",53:"5",54:"6",55:"7",56:"8",57:"9",

    // Letter keys. Note that we don't distinguish upper and lower case
    65:"A", 66:"B", 67:"C", 68:"D", 69:"E", 70:"F", 71:"G", 72:"H", 73:"I",
    74:"J", 75:"K", 76:"L", 77:"M", 78:"N", 79:"O", 80:"P", 81:"Q", 82:"R",
    83:"S", 84:"T", 85:"U", 86:"V", 87:"W", 88:"X", 89:"Y", 90:"Z",

    // Keypad numbers and punctuation keys. (Opera does not support these.)
    96:"0",97:"1",98:"2",99:"3",100:"4",101:"5",102:"6",103:"7",104:"8",105:"9",
    106:"Multiply", 107:"Add", 109:"Subtract", 110:"Decimal", 111:"Divide",

    // Function keys
    112:"F1", 113:"F2", 114:"F3", 115:"F4", 116:"F5", 117:"F6",
    118:"F7", 119:"F8", 120:"F9", 121:"F10", 122:"F11", 123:"F12",
```

```
    124:"F13", 125:"F14", 126:"F15", 127:"F16", 128:"F17", 129:"F18",
    130:"F19", 131:"F20", 132:"F21", 133:"F22", 134:"F23", 135:"F24",

    // Punctuation keys that don't require holding down Shift
    // Hyphen is nonportable: FF returns same code as Subtract
    59:";", 61:"=", 186:";", 187:"=", // Firefox and Opera return 59,61
    188:",", 190:".", 191:"/", 192:"`", 219:"[", 220:"\\", 221:"]", 222:"'"
};
```

CHAPTER 18
Scripted HTTP

The Hypertext Transfer Protocol (HTTP) specifies how web browsers get documents from and post form contents to web servers, and how web servers respond to those requests and posts. Web browsers obviously handle a lot of HTTP. Usually HTTP is not under the control of scripts and instead occurs when the user clicks on a link, submits a form, or types a URL.

It *is* possible for JavaScript code to script HTTP, however. HTTP requests are initiated when a script sets the `location` property of a window object or calls the `submit()` method of a form object. In both cases, the browser loads a new page. This kind of trivial HTTP scripting can be useful in a multiframed web page but is not the topic we'll cover here. Instead, this chapter explains how scripts can communicate with a web server without causing the web browser to reload the content of any window or frame.

The term *Ajax* describes an architecture for web applications that prominently features scripted HTTP.[1] The key feature of an Ajax application is that it uses scripted HTTP to initiate data exchange with a web server without causing pages to reload. The ability to avoid page reloads (which were the norm in the early days of the Web) results in responsive web applications that feel more like traditional desktop applications. A web application might use Ajax technologies to log user interaction data to the server or to improve its start-up time by displaying only a simple page at first and then downloading additional data and page components on an as-needed basis.

The term *Comet* refers to a related web application architecture that uses scripted HTTP.[2] In a sense, Comet is the reverse of Ajax: in Comet, it is the web server that initiates the communication, asynchronously sending messages to the client. If the web

1. Ajax is an (uncapitalized) acronym for Asynchronous JavaScript and XML. The term was coined by Jesse James Garrett and first appeared in his February 2005 essay "Ajax: A New Approach to Web Applications" (*http://www.adaptivepath.com/publications/essays/archives/000385.php*). "Ajax" was a popular buzzword for many years; now it is simply a useful term for an web application architecture based on scripted HTTP requests.

2. The name *Comet* was coined by Alex Russell in "Comet: Low Latency Data for the Browser" (*http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/*). The name is likely a play on Ajax: both Comet and Ajax are US brands of scouring powder.

application needs to respond to these messages sent by the server, it can then use Ajax techniques to send or request data. In Ajax, the client "pulls" data from the server. With Comet, the server "pushes" data to the client. Other names for Comet include "Server Push," "Ajax Push," and "HTTP Streaming."

There are a number of ways to implement Ajax and Comet, and these underlying implementations are sometimes known as *transports*. The `<img>` element, for example, has a `src` property. When a script sets this property to a URL, an HTTP GET request is initiated to download an image from that URL. A script can therefore pass information to a web server by encoding that information into the query-string portion of the URL of an image and setting the `src` property of an `<img>` element. The web server must actually return some image as the result of this request, but it can be invisible: a transparent 1-pixel-by-1-pixel image, for instance.[3]

An `<img>` element does not make a good Ajax transport, because the data exchange is one-way: the client can send data to the server, but the server's response will always be an image that the client can't easily extract information from. The `<iframe>` element is more versatile, however. To use an `<iframe>` as an Ajax transport, the script first encodes information for the web server into a URL and then sets the `src` property of the `<iframe>` to that URL. The server creates an HTML document containing its response and sends it back to the web browser, which displays it in the `<iframe>`. The `<iframe>` need not be visible to the user; it can be hidden with CSS, for example. A script can access the server's response by traversing the document object of the `<iframe>`. Note, though, that this traversal is subject to the constraints of the same-origin policy described in §13.6.2.

Even the `<script>` element has a `src` property that can be set to initiate an HTTP GET request. Doing HTTP scripting with `<script>` elements is particularly attractive because they are not subject to the same-origin policy and can be used for cross-domain communication. Usually, with a `<script>`-based Ajax transport, the server's response takes the form of JSON-encoded (see §6.9) data that is automatically "decoded" when the script is executed by the JavaScript interpreter. Because of its use of the JSON data format, this Ajax transport is known as "JSONP."

Although Ajax techniques can be implemented on top of an `<iframe>` or `<script>` transport, there is usually an easier way to do it. For some time, all browsers have supported an XMLHttpRequest object that defines an API for scripted HTTP. The API includes the ability to make POST requests, in addition to regular GET requests, and can return the server's response as text or as a Document object. Despite its name, the XMLHttpRequest API is not limited to use with XML documents: it can fetch any kind of text document. §18.1 covers the XMLHttpRequest API and takes up most of the chapter. Most of the Ajax examples in this chapter will use the XMLHttpRequest object

3. Images of this sort are sometimes called *web bugs*. Privacy concerns arise when web bugs are used to communicate information to a server other than the one from which the web page was loaded. One common use of this kind of third-party web bug is for hit counting and website traffic analysis.

as their transport, but we'll also demonstrate how to use the <script>-based transport in §18.2 because of the <script> element's ability to circumvent same-origin restrictions.

The XML Is Optional

The X in "Ajax" stands for XML, the primary client-side API for HTTP (XMLHttpRequest) features XML in its name, and we'll see later that one of the properties of the XMLHttpRequest object is named `responseXML`. It would seem that XML is an important part of scripted HTTP. But it's not: these names are the historical legacy of the days when XML was a powerful buzzword. Ajax techniques work with XML documents, of course, but the use of XML is purely optional and has actually become relatively rare. The XMLHttpRequest specification lays out the inadequacies of the name we're stuck with:

> The name of the object is XMLHttpRequest for compatibility with the Web, though each component of this name is potentially misleading. First, the object supports any text based format, including XML. Second, it can be used to make requests over both HTTP and HTTPS (some implementations support protocols in addition to HTTP and HTTPS, but that functionality is not covered by this specification). Finally, it supports "requests" in a broad sense of the term as it pertains to HTTP; namely all activity involved with HTTP requests or responses for the defined HTTP methods.

Transport mechanisms for Comet are trickier than Ajax, but all require the client to establish (and re-establish as necessary) a connection to the server, and require the server to keep that connection open so that it can send asynchronous messages over it. A hidden <iframe> can serve as a Comet transport, for example, if the server sends each message in the form of a <script> element to be executed in the <iframe>. A more reliably cross-platform approach to implementing Comet is for the client to establish a connection to the server (using an Ajax transport) and for the server to keep this connection open until it needs to push a message. Each time the server sends a message, it closes the connection, which helps to ensure that the message is properly received by the client. After processing the message, the client then immediately establishes a new connection for future messages.

Implementing a reliable cross-platform Comet transport is hard to do, and most web app developers who use the Comet architecture rely on the transports in web framework libraries such as Dojo. At the time of this writing, browsers are beginning to implement an HTML5-related draft specification known as Server-Sent Events that defines a simple Comet API in the form of an EventSource object. §18.3 covers the EventSource API and demonstrates a simple emulation of it using XMLHttpRequest.

It is possible to build higher-level communication protocols on top of Ajax and Comet. These client/server communication techniques can be used as the basis of an RPC (remote procedure call) mechanism or a publish/subscribe event system, for example.

This chapter does not describe higher-level protocols like this, however, and instead focuses on the APIs that enable Ajax and Comet.

18.1 Using XMLHttpRequest

Browsers define their HTTP API on an XMLHttpRequest class. Each instance of this class represents a single request/response pair, and the properties and methods of the object allow you to specify request details and extract response data. XMLHttpRequest has been supported by web browsers for many years, and the API is in the final stages of standardization through the W3C. At the same time, the W3C is working on a draft "XMLHttpRequest Level 2" standard. This section covers the core XMLHttpRequest API and also those portions of the Level 2 draft (which I'll call XHR2) that are currently implemented by at least two browsers.

The first thing you must do to use this HTTP API, of course, is to instantiate an XMLHttpRequest object:

```
var request = new XMLHttpRequest();
```

You can also reuse an existing XMLHttpRequest object, but note that doing so will abort any request pending through that object.

XMLHttpRequest in IE6

Microsoft introduced the XMLHttpRequest object to the world in IE5, and in IE5 and IE6 it is available only as an ActiveX object. The now-standard `XMLHttpRequest()` constructor is not supported before IE7, but it can be emulated like this:

```
// Emulate the XMLHttpRequest() constructor in IE5 and IE6
if (window.XMLHttpRequest === undefined) {
    window.XMLHttpRequest = function() {
        try {
            // Use the latest version of the ActiveX object if available
            return new ActiveXObject("Msxml2.XMLHTTP.6.0");
        }
        catch (e1) {
            try {
                // Otherwise fall back on an older version
                return new ActiveXObject("Msxml2.XMLHTTP.3.0");
            }
            catch(e2) {
                // Otherwise, throw an error
                throw new Error("XMLHttpRequest is not supported");
            }
        }
    };
}
```

An HTTP request consists of four parts:

- the HTTP request method or "verb"
- the URL being requested
- an optional set of request headers, which may include authentication information
- an optional request body

The HTTP response sent by a server has three parts:

- a numeric and textual status code that indicates the success or failure of the request
- a set of response headers
- the response body

The first two subsections below demonstrate how to set each of the parts of an HTTP request and how to query each of the parts of an HTTP response. Those key sections are followed by coverage of more specialized topics.

The basic request/response architecture of HTTP is pretty simple and easy to work with. In practice, however, there are all sorts of complications: clients and server exchange cookies, servers redirect browsers to other servers, some resources are cached and others are not, some clients send all their requests through proxy servers, and so on. XMLHttpRequest is not a protocol-level HTTP API but instead a browser-level API. The browser takes care of cookies, redirects, caching, and proxies and your code need worry only about requests and responses.

XMLHttpRequest and Local Files

The ability to use relative URLs in web pages usually means that we can develop and test our HTML using the local file system and then deploy it unchanged to a web server. This is generally not possible when doing Ajax programming with XMLHttpRequest, however. XMLHttpRequest is designed to work with the HTTP and HTTPS protocols. In theory, it could be made to work with other protocols, such as FTP, but parts of the API, such as the request method and the response status code, are HTTP-specific. If you load a web page from a local file, the scripts in that page will not be able to use XMLHttpRequest with relative URLs, since those URLs will be relative to a `file://` URL rather than an `http://` URL. And the same-origin policy will often prevent you from using absolute `http://` URLs. (But see §18.1.6.) The upshot is that when working with XMLHttpRequest, you generally have to upload your files to a web server (or run a server locally) in order to test them.

18.1.1 Specifying the Request

After creating an XMLHttpRequest object, the next step in making an HTTP request is to call the `open()` method of your XMLHttpRequest object to specify the two required parts of the request, the method and the URL:

```
request.open("GET",       // Begin a HTTP GET request
             "data.csv"); // For the contents of this URL
```

The first argument to `open()` specifies the HTTP method or verb. This is a case-insensitive string, but uppercase letters are typically used to match the HTTP protocol. The "GET" and "POST" methods are universally supported. "GET" is used for most "regular" requests, and it is appropriate when the URL completely specifies the requested resource, when the request has no side effects on the server, and when the server's response is cacheable. The "POST" method is what is typically used by HTML forms. It includes additional data (the form data) in the request body and that data is often stored in a database on the server (a side effect). Repeated POSTs to the same URL may result in different responses from the server, and requests that use this method should not be cached.

In addition to "GET" and "POST", the XMLHttpRequest specification also allows "DELETE", "HEAD", "OPTIONS", and "PUT" as the first argument to `open()`. (The "HTTP CONNECT", "TRACE", and "TRACK" methods are explicitly forbidden as security risks.) Older browsers may not support all of these methods, but "HEAD", at least, is widely supported and Example 18-13 demonstrates its use.

The second argument to `open()` is the URL that is the subject of the request. This is relative to the URL of the document that contains the script that is calling `open()`. If you specify an absolute URL, the protocol, host, and port must generally match those of the containing document: cross-origin HTTP requests normally cause an error. (But the XMLHttpRequest Level 2 specification allows cross-origin requests when the server explicitly allows it; see §18.1.6.)

The next step in the request process is to set the request headers, if any. POST requests, for example, need a "Content-Type" header to specify the MIME type of the request body:

```
request.setRequestHeader("Content-Type", "text/plain");
```

If you call `setRequestHeader()` multiple times for the same header, the new value does not replace the previously specified value: instead, the HTTP request will include multiple copies of the header or the header will specify multiple values.

You cannot specify the "Content-Length", "Date", "Referer", or "User-Agent" headers yourself: XMLHttpRequest will add those automatically for you and will not allow you to spoof them. Similarly, XMLHttpRequest object automatically handles cookies, and connection lifetime, charset, and encoding negotiations, so you're not allowed to pass any of these headers to `setRequestHeader()`:

```
Accept-Charset      Content-Transfer-Encoding   TE
Accept-Encoding     Date                        Trailer
Connection          Expect                      Transfer-Encoding
Content-Length      Host                        Upgrade
Cookie              Keep-Alive                  User-Agent
Cookie2             Referer                     Via
```

You can specify an "Authorization" header with your request, but you do not normally need to do so. If you are requesting a password-protected URL, pass the username and password as the fourth and fifth arguments to `open()`, and XMLHttpRequest will set appropriate headers for you. (We'll learn about the optional third argument to `open()` below. The optional username and password arguments are described in the reference section.)

The final step in making an HTTP request with XMLHttpRequest is to specify the optional request body and send it off to the server. Do this with the `send()` method:

```
request.send(null);
```

GET requests never have a body, so you should pass `null` or omit the argument. POST requests do generally have a body, and it should match the "Content-Type" header you specified with `setRequestHeader()`.

Order Matters

The parts of an HTTP request have a specific order: the request method and URL must come first, then the request headers, and finally the request body. XMLHttpRequest implementations generally do not initiate any networking until the `send()` method is called. But the XMLHttpRequest API is designed as if each method was writing to a network stream. This means that the XMLHttpRequest method must be called in an order that matches the structure of an HTTP request. `setRequestHeader()`, for example, must be called after you call `open()` and before you call `send()` or it will throw an exception.

Example 18-1 uses each of the XMLHttpRequest methods we've described so far. It POSTs a string of text to a server and ignores any response the server sends.

Example 18-1. POSTing plain text to a server

```
function postMessage(msg) {
    var request = new XMLHttpRequest();      // New request
    request.open("POST", "/log.php");        // POST to a server-side script
    // Send the message, in plain-text, as the request body
    request.setRequestHeader("Content-Type", // Request body will be plain text
                             "text/plain;charset=UTF-8");
    request.send(msg);                       // Send msg as the request body
    // The request is done. We ignore any response or any error.
}
```

Note that the `send()` method in Example 18-1 initiates the request and then returns: it does not block while waiting for the server's response. HTTP responses are almost always handled asynchronously, as demonstrated in the following section.

18.1.2 Retrieving the Response

A complete HTTP response consists of a status code, a set of response headers, and a response body. These are available through properties and methods of the XMLHttpRequest object:

- The `status` and `statusText` properties return the HTTP status in numeric and textual forms. These properties hold standard HTTP values like 200 and "OK" for successful requests and 404 and "Not Found" for URLs that don't match any resource on the server.
- The response headers can be queried with `getResponseHeader()` and `getAllResponseHeaders()`. XMLHttpRequest handles cookies automatically: it filters cookie headers out of the set returned by `getAllResponseHeaders()` and returns `null` if you pass "Set-Cookie" or "Set-Cookie2" to `getResponseHeader()`.
- The response body is available in textual form from the `responseText` property or in Document form from the `responseXML` property. (The name of that property is historical: it actually works for XHTML documents as well as XML documents, and XHR2 says that it should work for ordinary HTML documents as well.) See §18.1.2.2 for more on `responseXML`.

The XMLHttpRequest object is usually (but see §18.1.2.1) used asynchronously: the `send()` method returns immediately after sending the request, and the response methods and properties listed above aren't valid until the response is received. To be notified when the response is ready, you must listen for readystatechange events (or the new XHR2 progress events described in §18.1.4) on the XMLHttpRequest object. But to understand this event type, you must first understand the `readyState` property.

`readyState` is an integer that specifies the status of an HTTP request, and its possible values are enumerated in Table 18-1. The symbols in the first column are constants defined on the XMLHttpRequest constructor. These constants are part of the XMLHttpRequest specification, but older browsers and IE8 do not define them, and you'll often see code that hardcodes the value 4 instead of `XMLHttpRequest.DONE`.

Table 18-1. XMLHttpRequest readyState values

Constant	Value	Meaning
UNSENT	0	open() has not been called yet
OPENED	1	open() has been called
HEADERS_RECEIVED	2	Headers have been received
LOADING	3	The response body is being received
DONE	4	The response is complete

In theory, the readystatechange event is triggered every time the `readyState` property changes. In practice, the event may not be fired when `readyState` changes to 0 or 1. It is often fired when `send()` is called, even though `readyState` remains at `OPENED` when

that happens. Some browsers fire the event multiple times during the LOADING state to give progress feedback. All browsers do fire the readystatechange event when readyState has changed to the value 4 and the server's response is complete. Because the event is also fired before the response is complete, however, event handlers should always test the readyState value.

To listen for readystatechange events, set the onreadystatechange property of the XMLHttpRequest object to your event handler function. You can also use addEventListener() (or attachEvent() in IE8 and before), but you generally need only one handler per request and it is easier to simply set onreadystatechange.

Example 18-2 defines a getText() function that demonstrates how to listen for readystatechange events. The event handler first ensures that the request is complete. If so, it checks the response status code to ensure that the request was successful. Then it looks at the "Content-Type" header to verify that the response was of the expected type. If all three conditions are satisfied, it passes the response body (as text) to a specified callback function.

Example 18-2. Getting an HTTP response onreadystatechange

```
// Issue an HTTP GET request for the contents of the specified URL.
// When the response arrives successfully, verify that it is plain text
// and if so, pass it to the specified callback function
function getText(url, callback) {
    var request = new XMLHttpRequest();          // Create new request
    request.open("GET", url);                    // Specify URL to fetch
    request.onreadystatechange = function() {    // Define event listener
        // If the request is compete and was successful
        if (request.readyState === 4 && request.status === 200) {
            var type = request.getResponseHeader("Content-Type");
            if (type.match(/^text/))             // Make sure response is text
                callback(request.responseText); // Pass it to callback
        }
    };
    request.send(null);                          // Send the request now
}
```

18.1.2.1 Synchronous responses

By their very nature, HTTP responses are best handled asynchronously. Nevertheless, XMLHttpRequest also supports synchronous responses. If you pass false as the third argument to open(), the send() method will block until the request completes. In this case, there is no need to use an event handler: once send() returns, you can just check the status and responseText properties of the XMLHttpRequest object. Compare this synchronous code to the getText() function in Example 18-2:

```
// Issue a synchronous HTTP GET request for the contents of the specified URL.
// Return the response text or throw an error if the request was not successful
// or if the response was not text.
function getTextSync(url) {
    var request = new XMLHttpRequest();  // Create new request
    request.open("GET", url, false);     // Pass false for synchronous
```

```
    request.send(null);                 // Send the request now

    // Throw an error if the request was not 200 OK
    if (request.status !== 200) throw new Error(request.statusText);

    // Throw an error if the type was wrong
    var type = request.getResponseHeader("Content-Type");
    if (!type.match(/^text/))
        throw new Error("Expected textual response; got: " + type);

    return request.responseText;
}
```

Synchronous requests are tempting, but they should be avoided. Client-side JavaScript is single-threaded and when the `send()` method blocks, it typically freezes the entire browser UI. If the server you are connecting to is responding slowly, your user's browser will freeze up. See §22.4 for one context in which it is acceptable to make synchronous requests, however.

18.1.2.2 Decoding the response

In the examples above, we assume that the server has sent a textual response, with a MIME type like "text/plain", "text/html", or "text/css", and we retrieve it with the `responseText` property of the XMLHttpRequest object.

There are other ways to handle the server's response, however. If the server sends an XML or XHTML document as its response, you can retrieve a parsed representation of the XML document through the `responseXML` property. The value of this property is a Document object, and you can search and traverse it using the techniques shown in Chapter 15. (The XHR2 draft specification says that browsers should also automatically parse responses of type "text/html" and make them available as Document objects through `responseXML` as well, but browsers current at the time of this writing do not do that.)

If the server wants to send structured data, such as an object or array, as its response, it might transmit that data as a JSON-encoded (§6.9) string. When you receive it, you would then pass the `responseText` property to `JSON.parse()`. Example 18-3 is a generalization of Example 18-2: it makes a GET request for the specified URL and passes the contents of that URL to the specified callback function when they are ready. But instead of always passing text, it passes a Document, or an object decoded with `JSON.parse()`, or a string.

Example 18-3. Parsing the HTTP response

```
// Issue an HTTP GET request for the contents of the specified URL.
// When the response arrives, pass it to the callback function as a
// parsed XML Document object, a JSON-parsed object, or a string.
function get(url, callback) {
    var request = new XMLHttpRequest();          // Create new request
    request.open("GET", url);                    // Specify URL to fetch
    request.onreadystatechange = function() {    // Define event listener
```

```
        // If the request is compete and was successful
        if (request.readyState === 4 && request.status === 200) {
            // Get the type of the response
            var type = request.getResponseHeader("Content-Type");
            // Check type so we don't get HTML documents in the future
            if (type.indexOf("xml") !== -1 && request.responseXML)
                callback(request.responseXML);              // Document response
            else if (type === "application/json")
                callback(JSON.parse(request.responseText)); // JSON response
            else
                callback(request.responseText);             // String response
        }
    };
    request.send(null);                         // Send the request now
}
```

Example 18-3 checks the "Content-Type" header of the response and handles "application/json" responses specially. Another response type that you might want to "decode" specially is "application/javascript" or "text/javascript". You can use an XMLHttpRequest to request a JavaScript script, and then use a global `eval()` (§4.12.2) to execute that script. Using an XMLHttpRequest object is unnecessary in this case, however, since the HTTP scripting capabilities of the `<script>` element itself are sufficient to download and execute a script. See Example 13-4, and keep in mind that the `<script>` element can make cross-origin HTTP requests that are prohibited to the XMLHttpRequest API.

Web servers often respond to HTTP requests with binary data (image files, for example). The `responseText` property is for text only, and it cannot properly handle binary responses, even if you use the `charCodeAt()` method of the resulting string. XHR2 defines a way to handle binary responses, but at the time of this writing, browser vendors have not implemented it. See §22.6.2 for further details.

Proper decoding of a server's response assumes that the server sends a "Content-Type" header with the correct MIME type for the response. If a server sends an XML document without setting the appropriate MIME type, for example, the XMLHttpRequest object will not parse it and set the responseXML property. Or if a server includes an incorrect "charset" parameter in the content-type header, the XMLHttpRequest will decode the response using the wrong encoding and the characters in `responseText` may be wrong. XHR2 defines an `overrideMimeType()` method to address this problem and a number of browsers have already implemented it. If you know the MIME type of a resource better than the server does, pass the type of `overrideMimeType()` before you call `send()`—this will make XMLHttpRequest ignore the content-type header and use the type you specify instead. Suppose you're downloading an XML file that you're planning to treat as plain text. You can use `setOverrideMimeType()` to let the XMLHttpRequest know that it does not need to parse the file into an XML document:

```
// Don't process the response as an XML document
request.overrideMimeType("text/plain; charset=utf-8")
```

18.1.3 Encoding the Request Body

HTTP POST requests include a request body that contains data the client is passing to the server. In Example 18-1, the request body was simply a string of text. Often, however, we want to send more complicated data along with an HTTP request. This section demonstrates a number of ways to do that.

18.1.3.1 Form-encoded requests

Consider HTML forms. When the user submits a form, the data in the form (the names and values of each of the form elements) is encoded into a string and sent along with the request. By default, HTML forms are POSTed to the server, and the encoded form data is used as the body of the request. The encoding scheme used for form data is relatively simple: perform normal URI encoding (replacing special characters with hexadecimal escape codes) on the name and value of each form element, separate the encoded name and value with an equals sign, and separate these name/value pairs with ampersands. The encoding of a simple form might look like this:

```
find=pizza&zipcode=02134&radius=1km
```

This form data encoding format has a formal MIME type:

```
application/x-www-form-urlencoded
```

You must set the "Content-Type" request header to this value when POSTing form data of this sort.

Note that this kind of encoding does not require an HTML form, and we won't actually work directly with forms in this chapter. In Ajax applications, you are likely to have a JavaScript object that you want to send to the server. (That object may be derived from the user input in an HTML form, but that does not matter here.) The data shown above might be the form-encoded representation of this JavaScript object:

```
{
    find: "pizza",
    zipcode: 02134,
    radius: "1km"
}
```

Form encoding is so widely used on the Web, and so well supported in all server-side programming languages, that form-encoding your nonform data is often the easiest thing to do. Example 18-4 demonstrates how to form-encode the properties of an object.

Example 18-4. Encoding an object for an HTTP request

```
/**
 * Encode the properties of an object as if they were name/value pairs from
 * an HTML form, using application/x-www-form-urlencoded format
 */
function encodeFormData(data) {
    if (!data) return "";    // Always return a string
```

```
    var pairs = [];          // To hold name=value pairs
    for(var name in data) {                                  // For each name
        if (!data.hasOwnProperty(name)) continue;            // Skip inherited
        if (typeof data[name] === "function") continue;      // Skip methods
        var value = data[name].toString();                   // Value as string
        name = encodeURIComponent(name).replace("%20","+");  // Encode name
        value = encodeURIComponent(value).replace("%20", "+"); // Encode value
        pairs.push(name + "=" + value);   // Remember name=value pair
    }
    return pairs.join('&'); // Return joined pairs separated with &
}
```

With this `encodeFormData()` function defined, we can easily write utilities like the `postData()` function of Example 18-5. Note that, for simplicity, this `postData()` function (and similar functions in the examples that follow) does not process the server's response. When the response is complete, it passes the entire XMLHttpRequest object to the specified callback function. That callback is responsible for checking the response status code and extracting the response text.

Example 18-5. Making an HTTP POST request with form-encoded data

```
function postData(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);                    // POST to the specified url
    request.onreadystatechange = function() {     // Simple event handler
        if (request.readyState === 4 && callback) // When response is complete
            callback(request);                    // call the callback.
    };
    request.setRequestHeader("Content-Type",      // Set Content-Type
                             "application/x-www-form-urlencoded");
    request.send(encodeFormData(data));           // Send form-encoded data
}
```

Form data can also be submitted using a GET request, and when the purpose of a form submission is to make a read-only query, GET is more appropriate than POST. GET requests never have a body, so the "payload" of form-encoded data has to be sent to the server as the query portion of the URL (following a question mark). The `encodeFormData()` utility can also be useful for this kind of GET request, and Example 18-6 demonstrates how to use it.

Example 18-6. Making a GET request with form-encoded data

```
function getData(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("GET", url +                     // GET the specified url
                 "?" + encodeFormData(data));     // with encoded data added
    request.onreadystatechange = function() {     // Simple event handler
        if (request.readyState === 4 && callback) callback(request);
    };
    request.send(null);                           // Send the request
}
```

HTML forms use form-encoded query sections to encode data into a URL, but using XMLHttpRequest gives us the freedom to encode our data however we want. With appropriate support on the server, our pizza query data might be encoded into a more legible URL like this one:

```
http://restaurantfinder.example.com/02134/1km/pizza
```

18.1.3.2 JSON-encoded requests

The use of form encoding in the body of POST requests is a common convention, but it is not a requirement of the HTTP protocol by any means. In recent years, the JSON format has gained popularity as a web interchange format. Example 18-7 shows how you might encode a request body using `JSON.stringify()` (§6.9). Note that this example differs from Example 18-5 only in the last two lines.

Example 18-7. Making an HTTP POST request with a JSON-encoded body

```
function postJSON(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);                    // POST to the specified url
    request.onreadystatechange = function() {     // Simple event handler
        if (request.readyState === 4 && callback) // When response is complete
            callback(request);                    // call the callback.
    };
    request.setRequestHeader("Content-Type", "application/json");
    request.send(JSON.stringify(data));
}
```

18.1.3.3 XML-encoded requests

XML is sometimes also used as an encoding for data transfer. Instead of expressing our pizza query as a form-encoded or JSON-encoded version of a JavaScript object, we could represent it as an XML document. It might look like this, for example:

```
<query>
  <find zipcode="02134" radius="1km">
    pizza
  </find>
</query>
```

In all the examples we've shown so far, the argument to the XMLHttpRequest `send()` method has been a string or `null`. In fact, you can also pass an XML Document object here. Example 18-8 demonstrates how to create a simple XML Document object and use it as the body of an HTTP request.

Example 18-8. An HTTP POST request with an XML document as its body

```
// Encode what, where, and radius in an XML document and post them to the
// specified url, invoking callback when the response is received
function postQuery(url, what, where, radius, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);                  // POST to the specified url
    request.onreadystatechange = function() {   // Simple event handler
```

```
        if (request.readyState === 4 && callback) callback(request);
    };

    // Create an XML document with root element <query>
    var doc = document.implementation.createDocument("", "query", null);
    var query = doc.documentElement;            // The <query> element
    var find = doc.createElement("find");       // Create a <find> element
    query.appendChild(find);                    // And add it to the <query>
    find.setAttribute("zipcode", where);        // Set attributes on <find>
    find.setAttribute("radius", radius);
    find.appendChild(doc.createTextNode(what)); // And set content of <find>

    // Now send the XML-encoded data to the server.
    // Note that the Content-Type will be automatically set.
    request.send(doc);
}
```

Note that Example 18-8 does not ever set the "Content-Type" header for the request. When you pass an XML document to the `send()` method, without previously specifying a `Content-Type` header, the XMLHttpRequest object automatically sets an appropriate header for you. (Similarly, if you pass a string to `send()` and have not specified a Content-Type, the XMLHttpRequest will add a "text/plain; charset=UTF-8" header for you. The code in Example 18-1 sets this header explicitly, but that is not actually required for plain-text request bodies.

18.1.3.4 Uploading a file

One of the features of HTML forms is that when the user selects a file through an `<input type="file">` element, the form will send the content of that file in the body of the POST request it generates. HTML forms have always been able to upload files, but until recently it was not possible to do the same thing with the XMLHttpRequest API. The XHR2 API, however, allows you to upload files by passing a File object to the `send()` method.

There is no `File()` object constructor: scripts can only obtain File objects that represent files the user has selected. In browsers that support File objects, every `<input type="file">` element has a `files` property that is an array-like object of File objects. The drag-and-drop API (§17.7) also allows access to files that the user "drops" over an element, through the `dataTransfer.files` property of the drop event. We'll see more about the File object in §22.6 and §22.7. For now, we can treat it as a completely opaque representation of a user-selected file, suitable for upload through `send()`. Example 18-9 is a an unobtrusive JavaScript function that adds an change event handler to certain file upload elements so that they automatically POST the contents of any selected file to a specified URL.

Example 18-9. File upload with an HTTP POST request

```
// Find all <input type="file"> elements with a data-uploadto attribute
// and register an onchange handler so that any selected file is
// automatically POSTED to the specified "uploadto" URL. The server's
```

```
// response is ignored.
whenReady(function() {                          // Run when the document is ready
    var elts = document.getElementsByTagName("input"); // All input elements
    for(var i = 0; i < elts.length; i++) {              // Loop through them
        var input = elts[i];
        if (input.type !== "file") continue;  // Skip all but file upload elts
        var url = input.getAttribute("data-uploadto"); // Get upload URL
        if (!url) continue;                   // Skip any without a url

        input.addEventListener("change", function() {  // When user selects file
            var file = this.files[0];         // Assume a single file selection
            if (!file) return;                // If no file, do nothing
            var xhr = new XMLHttpRequest();   // Create a new request
            xhr.open("POST", url);            // POST to the URL
            xhr.send(file);                   // Send the file as body
        }, false);
    }
});
```

As we'll see in §22.6, the File type is a subtype of the more general Blob type. XHR2 allows you to pass any Blob object to the `send()` method. The `type` property of the Blob will be used to set the `Content-Type` header for the upload, if you do not set that header explicitly yourself. If you need to upload binary data that you have generated, you can use the techniques shown in §22.5 and §22.6.3 to convert the data to a Blob and use it as a request body.

18.1.3.5 multipart/form-data requests

When HTML forms include file upload elements and other elements as well, the browser cannot use ordinary form encoding and must POST the form using a special content-type known as "multipart/form-data". This encoding involves the use of long "boundary" strings to separate the body of the request into multiple parts. For textual data, it is possible to create "multipart/form-data" request bodies by hand, but it is tricky.

XHR2 defines a new FormData API that makes multipart request bodies simple. First, create a FormData object with the `FormData()` constructor and then call the `append()` method of that object as many times as necessary to add the individual "parts" (these can be strings or File or Blob objects) to the request. Finally, pass the FormData object to the `send()` method. The `send()` method will define an appropriate boundary string and set the "Content-Type" header for the request. Example 18-10 demonstrates the use of FormData, and we'll see it again in Example 18-11.

Example 18-10. POSTing multipart/form-data request body

```
function postFormData(url, data, callback) {
    if (typeof FormData === "undefined")
        throw new Error("FormData is not implemented");

    var request = new XMLHttpRequest();          // New HTTP request
    request.open("POST", url);                   // POST to the specified url
```

```
    request.onreadystatechange = function() {      // A simple event handler.
        if (request.readyState === 4 && callback)  // When response is complete
            callback(request);                     // ...call the callback.
    };
    var formdata = new FormData();
    for(var name in data) {
        if (!data.hasOwnProperty(name)) continue;  // Skip inherited properties
        var value = data[name];
        if (typeof value === "function") continue; // Skip methods
        // Each property becomes one "part" of the request.
        // File objects are allowed here
        formdata.append(name, value);              // Add name/value as one part
    }
    // Send the name/value pairs in a multipart/form-data request body. Each
    // pair is one part of the request. Note that send automatically sets
    // the Content-Type header when you pass it a FormData object
    request.send(formdata);
}
```

18.1.4 HTTP Progress Events

In the examples above, we've used the readystatechange event to detect the completion of an HTTP request. The XHR2 draft specification defines a more useful set of events and these have already been implemented by Firefox, Chrome, and Safari. In this new event model, the XMLHttpRequest object triggers different types of events at different phases of the request so that it is no longer necessary to check the `readyState` property.

In browsers that support them, these new events are triggered as follows. When the `send()` method is called, a single loadstart event is fired. While the server's response is being downloaded, the XMLHttpRequest object fires progress events, typically every 50 milliseconds or so, and you can use these events to give the user feedback about the progress of the request. If a request completes very quickly, it may never fire a progress event. When a request is complete, a load event is fired.

A complete request is not necessarily a successful request, and your handler for the load event should check the `status` code of the XMLHttpRequest object to ensure that you received a HTTP "200 OK" response rather than a "404 Not Found" response, for example.

There are three ways that an HTTP request can fail to complete, and three corresponding events. If a request times out, the timeout event is triggered. If a request is aborted, the abort event is triggered. (Timeouts and the `abort()` method will be covered in §18.1.5.) Finally, other network errors, such as too many redirects, can prevent the completion of a request, and the error event is triggered when this happens.

A browser will fire only one of the load, abort, timeout, or error events for any given request. The XHR2 draft says that browsers should trigger a loadend event once one of these events has occurred. At the time of this writing, however, browsers do not implement loadend.

You can call the addEventListener() method of the XMLHttpRequest object register handlers for each of these progress events. If you have only one handler for each kind of event, it is generally easier to just set the corresponding handler property, such as onprogress and onload. You can even use the existence of these event properties to test whether a browser supports progress events:

```
if ("onprogress" in (new XMLHttpRequest())) {
    // Progress events are supported
}
```

The event object associated with these progress events has three useful properties in addition to the normal Event object properties like type and timestamp. The loaded property is the number of bytes that have been transferred so far. The total property is the total length (in bytes) of the data to be transferred, from the "Content-Length" header, or 0 if the content length is not known. Finally, the lengthComputable property is true if the content length is known and is false otherwise. Obviously, the total and loaded properties are particularly useful in progress event handlers:

```
request.onprogress = function(e) {
    if (e.lengthComputable)
        progress.innerHTML = Math.round(100*e.loaded/e.total) + "% Complete";
}
```

18.1.4.1 Upload progress events

In addition to defining these useful events for monitoring the download of an HTTP response, XHR2 also allows the events to be used to monitor the upload of an HTTP request. In browsers that have implemented this feature, the XMLHttpRequest object will have an upload property. The value of the upload property is an object that defines an addEventListener() method and defines a full set of progress event properties, such as onprogress and onload. (The upload object does not define an onreadystatechange property, however: uploads only trigger the new event types.)

You can use the upload event handlers just as you would use the regular progress event handlers. For an XMLHttpRequest object x, set x.onprogress to monitor the download progress of the response. And set x.upload.onprogress to monitor the upload progress of the request.

Example 18-11 demonstrates how to use upload progress events to present upload progress feedback to the user. This example also demonstrates how to obtain File objects from the Drag-and-Drop API and how to upload multiple files in a single XMLHttpRequest request with the FormData API. These features are still in draft form at the time of this writing and the example does not work in all browsers.

Example 18-11. Monitoring HTTP upload progress

```
// Find all elements of class "fileDropTarget" and register DnD event handlers
// to make them respond to file drops.  When files are dropped, upload them to
// the URL specified in the data-uploadto attribute.
whenReady(function() {
    var elts = document.getElementsByClassName("fileDropTarget");
```

```
for(var i = 0; i < elts.length; i++) {
    var target = elts[i];
    var url = target.getAttribute("data-uploadto");
    if (!url) continue;
    createFileUploadDropTarget(target, url);
}

function createFileUploadDropTarget(target, url) {
    // Keep track of whether we're currently uploading something so we can
    // reject drops. We could handle multiple concurrent uploads, but
    // that would make progress notification too tricky for this example.
    var uploading = false;

    console.log(target, url);

    target.ondragenter = function(e) {
        console.log("dragenter");
        if (uploading) return;  // Ignore drags if we're busy
        var types = e.dataTransfer.types;
        if (types &&
            ((types.contains && types.contains("Files")) ||
             (types.indexOf && types.indexOf("Files") !== -1))) {
            target.classList.add("wantdrop");
            return false;
        }
    };
    target.ondragover = function(e) { if (!uploading) return false; };
    target.ondragleave = function(e) {
        if (!uploading) target.classList.remove("wantdrop");
    };
    target.ondrop = function(e) {
        if (uploading) return false;
        var files = e.dataTransfer.files;
        if (files && files.length) {
            uploading = true;
            var message = "Uploading files:<ul>";
            for(var i = 0; i < files.length; i++)
                message += "<li>" + files[i].name + "</li>";
            message += "</ul>";

            target.innerHTML = message;
            target.classList.remove("wantdrop");
            target.classList.add("uploading");

            var xhr = new XMLHttpRequest();
            xhr.open("POST", url);
            var body = new FormData();
            for(var i = 0; i < files.length; i++) body.append(i, files[i]);
            xhr.upload.onprogress = function(e) {
                if (e.lengthComputable) {
                    target.innerHTML = message +
                        Math.round(e.loaded/e.total*100) +
                        "% Complete";
                }
            };
            xhr.upload.onload = function(e) {
```

```
                uploading = false;
                target.classList.remove("uploading");
                target.innerHTML = "Drop files to upload";
            };
            xhr.send(body);

            return false;
        }
        target.classList.remove("wantdrop");
    }
}
});
```

18.1.5 Aborting Requests and Timeouts

You can cancel an HTTP request in process by calling the `abort()` method of the XMLHttpRequest object. The `abort()` method is available in all versions of XMLHttpRequest, and in XHR2, calling `abort()` triggers an abort event on the object. (Some browsers support abort events at the time of this writing. You can test for the presence of an "onabort" property on the XMLHttpRequest object.)

The primary reason to call `abort()` is to cancel or time-out requests that have taken too long to complete or when the responses become irrelevant. Suppose you're using XMLHttpRequest to request auto-complete suggestions for a text input field. If the user types a new character into the field before the server's suggestions can arrive, then the pending request is no longer interesting and can be aborted.

XHR2 defines a `timeout` property that specifies a time in milliseconds after which a request will automatically be aborted and also defines a timeout event that is supposed to be triggered (instead of the abort event) when such a timeout occurs. At the time of this writing, browsers do not implement these automatic timeouts (and their XMLHttpRequest objects do not have `timeout` or `ontimeout` properties). You can implement your own timeouts, however, with `setTimeout()` (§14.1) and the `abort()` method. Example 18-12 demonstrates how to do this.

Example 18-12. Implementing timeouts

```
// Issue an HTTP GET request for the contents of the specified URL.
// If the response arrives successfully, pass responseText to the callback.
// If the response does not arrive in less than timeout ms, abort the request.
// Browsers may fire "readystatechange" after abort(), and if a partial
// request has been received, the status property may even be set, so
// we need to set a flag so that we don't invoke the callback for a partial,
// timed-out response. This problem does not arise if we use the load event.
function timedGetText(url, timeout, callback) {
    var request = new XMLHttpRequest();         // Create new request.
    var timedout = false;                       // Whether we timed out or not.
    // Start a timer that will abort the request after timeout ms.
    var timer = setTimeout(function() {         // Start a timer. If triggered,
                           timedout = true; // set a flag and then
                           request.abort(); // abort the request.
                       },
```

```
                        timeout);                // How long before we do this
    request.open("GET", url);                    // Specify URL to fetch
    request.onreadystatechange = function() {    // Define event listener.
        if (request.readyState !== 4) return;    // Ignore incomplete requests.
        if (timedout) return;                    // Ignore aborted requests.
        clearTimeout(timer);                     // Cancel pending timeout.
        if (request.status === 200)              // If request was successful
            callback(request.responseText);      // pass response to callback.
    };
    request.send(null);                          // Send the request now
}
```


18.1.6 Cross-Origin HTTP Requests

As part of the same-origin security policy (§13.6.2), the XMLHttpRequest object can normally issue HTTP requests only to the server from which the document that uses it was downloaded. This restriction closes security holes, but it is heavy-handed and also prevents a number of legitimate uses for cross-origin requests. You can use cross-origin URLs with `<form>` and `<iframe>` elements, and the browser will display the resulting cross-origin document. But because of the same-origin policy, the browser won't allow the original script to inspect the contents of the cross-origin document. With XMLHttpRequest, document contents are always exposed through the `responseText` property, so the same-origin policy cannot allow XMLHttpRequest to make cross-origin requests. (Note that the `<script>` element has never really been subject to the same-origin policy: it will download and execute any script, regardless of origin. As we'll see in §18.2, this freedom to make cross-origin requests makes the `<script>` element an attractive Ajax transport alternative to XMLHttpRequest.)

XHR2 allows cross-origin requests to websites that opt-in by sending appropriate CORS (Cross-Origin Resource Sharing) headers in their HTTP responses. At the time of this writing, current versions of Firefox, Safari, and Chrome support CORS and IE8 supports it through a proprietary XDomainRequest object that is not documented here. As a web programmer, there is nothing special you need to do to make this work: if the browser supports CORS for XMLHttpRequest and if the website you are trying to make a cross-origin request to has decided to allow cross-origin requests with CORS, the same-origin policy will be relaxed and your cross-origin requests will just work.

Although there is nothing you have to do to make CORS-enabled cross-origin requests work, there are a few security details worth understanding. First, if you pass a username and password to the XMLHttpRequest `open()` method, they will never be sent with a cross-origin request (that would enable distributed password-cracking attempts). In addition, cross-origin requests do not normally include any other user credentials either: cookies and HTTP authentication tokens are not normally sent as part of the request and any cookies received as part of a cross-origin response are discarded. If your cross-origin request requires these kinds of credentials to succeed, you must set the `withCredentials` property of the XMLHttpRequest to `true` before you `send()` the request. It is uncommon to have to do this, but testing for the presence of the `withCredentials` property is a way to test for CORS support in your browser.

Example 18-13 is unobtrusive JavaScript code that uses XMLHttpRequest to make HTTP HEAD requests to download type, size, and date information about the resources linked to by the <a> elements in a document. The HEAD requests are made on demand, and the resulting link information is displayed in tooltips. The example assumes that information will not be available for cross-origin links, but on CORS-enabled browsers it attempts to download it anyway.

Example 18-13. Requesting link details with HEAD and CORS

```
/**
 * linkdetails.js
 *
 * This unobtrusive JavaScript module finds all <a> elements that have an href
 * attribute but no title attribute and adds an onmouseover event handler to
 * them. The event handler makes an XMLHttpRequest HEAD request to fetch
 * details about the linked resource, and then sets those details in the title
 * attribute of the link so that they will be displayed as a tooltip.
 */
whenReady(function() {
    // Is there any chance that cross-origin requests will succeed?
    var supportsCORS = (new XMLHttpRequest()).withCredentials !== undefined;

    // Loop through all links in the document
    var links = document.getElementsByTagName('a');
    for(var i = 0; i < links.length; i++) {
        var link = links[i];
        if (!link.href) continue; // Skip anchors that are not hyperlinks
        if (link.title) continue; // Skip links that already have tooltips

        // If this is a cross-origin link
        if (link.host !== location.host || link.protocol !== location.protocol)
        {
            link.title = "Off-site link";  // Assume we can't get any more info
            if (!supportsCORS) continue;   // Quit now if no CORS support
            // Otherwise, we might be able to learn more about the link
            // So go ahead and register the event handlers so we can try.
        }

        // Register event handler to download link details on mouse over
        if (link.addEventListener)
            link.addEventListener("mouseover", mouseoverHandler, false);
        else
            link.attachEvent("onmouseover", mouseoverHandler);
    }

    function mouseoverHandler(e) {
        var link = e.target || e.srcElement;      // The <a> element
        var url = link.href;                      // The link URL

        var req = new XMLHttpRequest();           // New request
        req.open("HEAD", url);                    // Ask for just the headers
        req.onreadystatechange = function() {     // Event handler
            if (req.readyState !== 4) return;     // Ignore incomplete requests
            if (req.status === 200) {             // If successful
                var type = req.getResponseHeader("Content-Type");   // Get
```

```
                var size = req.getResponseHeader("Content-Length"); // link
                var date = req.getResponseHeader("Last-Modified");  // details
                // Display the details in a tooltip.
                link.title = "Type: " + type + "   \n" +
                    "Size: " + size + "   \n" + "Date: " + date;
            }
            else {
                // If request failed, and the link doesn't already have an
                // "Off-site link" tooltip, then display the error.
                if (!link.title)
                    link.title = "Couldn't fetch details: \n" +
                        req.status + " " + req.statusText;
            }
        };
        req.send(null);

        // Remove handler: we only want to fetch these headers once.
        if (link.removeEventListener)
            link.removeEventListener("mouseover", mouseoverHandler, false);
        else
            link.detachEvent("onmouseover", mouseoverHandler);
    }
});
```


18.2 HTTP by <script>: JSONP

The introduction to this chapter mentioned that a `<script>` element can be used as an Ajax transport mechanism: simply set the `src` attribute of a `<script>` (and insert it into the document if it isn't already there) and the browser will generate an HTTP request to download the URL you specify. `<script>` elements are useful Ajax transports for one primary reason: they are not subject to the same origin policy, so you can use them to request data from servers other than your own. A secondary reason to use `<script>` elements is that they automatically decode (i.e., execute) response bodies that consist of JSON-encoded data.

Scripts and Security

In order to use a `<script>` element as an Ajax transport, you have to allow your web page to run whatever JavaScript code the remote server chooses to send you. This means that you *must not* use the technique described here with untrusted servers. And when you do use it with trusted servers, keep in mind that if an attacker can hack into that server, then the hacker can take over your web page, run any code she wants and display any content she wants, and that content will appear to come from your site.

With that said, note that it has become commonplace for websites to use trusted third-party scripts, especially to embed advertising or "widgets" into a page. Using a `<script>` as an Ajax transport to communicate with a trusted web service is no more dangerous than that.

The technique of using a <script> element as an Ajax transport has come to be known as JSONP: it works when the response body of the HTTP request is JSON-encoded. The "P" stands for "padding" or "prefix"—this will be explained in a moment.[4]

Suppose you've written a service that handles GET requests and returns JSON-encoded data. Same-origin documents can use it with XMLHttpRequest and `JSON.parse()` with code like that in Example 18-3. If you enable CORS on your server, cross-origin documents in new browsers can also use your service with XMLHttpRequest. Cross-origin documents in older browsers that do not support CORS can only access your service with a <script> element, however. Your JSON response body is (by definition) valid JavaScript code, and the browser will execute it when it arrives. Executing JSON-encoded data decodes it, but the result is still just data, and it doesn't *do* anything.

This is where the P part of JSONP comes in. When invoked through a <script> element, your service must "pad" its response by surrounding it with parentheses and prefixing it with the name of a JavaScript function. Instead of just sending JSON data like this:

```
[1, 2, {"buckle": "my shoe"}]
```

It sends a padded-JSON response like this:

```
handleResponse(
[1, 2, {"buckle": "my shoe"}]
)
```

As the body of a <script> element, this padded response does something valuable: it evaluates the JSON-encoded data (which is nothing more than one big JavaScript expression, after all) and then passes it to the function `handleResponse()`, which, we assume, the containing document has defined to do something useful with the data.

In order for this to work, we have to have some way to tell the service that it is being invoked from a <script> element and must send a JSONP response instead of a plain JSON response. This can be done by adding a query parameter to the URL: appending `?json` (or `&json`), for example.

In practice, services that support JSONP do not dictate a function name like "handleResponse" that all clients must implement. Instead, they use the value of a query parameter to allow the client to specify a function name, and then use that function name as the padding in the response. Example 18-14 uses a query parameter named "jsonp" to specify the name of the callback function. Many services that support JSONP recognize this parameter name. Another common name is "callback", and you might have to modify the code shown here to make it work with the particular requirements of the service you need to use.

Example 18-14 defines a function `getJSONP()` that makes a JSONP request. This example is a little tricky, and there are some things you should note about it. First, notice how it creates a new <script> element, sets its URL, and inserts it into the document.

4. Bob Ippolito coined the term "JSONP" (*http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/*) in 2005.

It is this insertion that triggers the HTTP request. Second, notice that the example creates a new internal callback function for each request, storing the function as a property of `getJSONP()` itself. Finally, note that callback performs some necessary cleanup: it removes the script element and deletes itself.

Example 18-14. Making a JSONP request with a script element

```
// Make a JSONP request to the specified URL and pass the parsed response
// data to the specified callback. Add a query parameter named "jsonp" to
// the URL to specify the name of the callback function for the request.
function getJSONP(url, callback) {
    // Create a unique callback name just for this request
    var cbnum = "cb" + getJSONP.counter++; // Increment counter each time
    var cbname = "getJSONP." + cbnum;      // As a property of this function

    // Add the callback name to the url query string using form-encoding
    // We use the parameter name "jsonp".  Some JSONP-enabled services
    // may require a different parameter name, such as "callback".
    if (url.indexOf("?") === -1)   // URL doesn't already have a query section
        url += "?jsonp=" + cbname; // add parameter as the query section
    else                           // Otherwise,
        url += "&jsonp=" + cbname; // add it as a new parameter.

    // Create the script element that will send this request
    var script = document.createElement("script");

    // Define the callback function that will be invoked by the script
    getJSONP[cbnum] = function(response) {
        try {
            callback(response); // Handle the response data
        }
        finally {               // Even if callback or response threw an error
            delete getJSONP[cbnum];               // Delete this function
            script.parentNode.removeChild(script); // Remove script
        }
    };

    // Now trigger the HTTP request
    script.src = url;                   // Set script url
    document.body.appendChild(script); // Add it to the document
}

getJSONP.counter = 0;  // A counter we use to create unique callback names
```

18.3 Comet with Server-Sent Events

The Server-Sent Events draft standard defines an EventSource object that makes Comet applications trivial to write. Simply pass a URL to the `EventSource()` constructor and then listen for message events on the returned object:

```
var ticker = new EventSource("stockprices.php");
ticker.onmessage = function(e) {
    var type = e.type;
    var data = e.data;
```

```
        // Now process the event type and event data strings.
    }
```

The event object associated with a message event has a `data` property that holds whatever string the server sent as the payload for this event. The event object also has a `type` property like all event objects do. The default value is "message", but the event source can specify a different string for the property. A single `onmessage` event handler receives all events from a given server event source, and can dispatch them, if necessary, based on their `type` property.

The Server-Sent Event protocol is straightforward. The client initiates a connection to the server (when it creates the `EventSource` object) and the server keeps this connection open. When an event occurs, the server writes lines of text to the connection. An event going over the wire might look like this:

```
event: bid    sets the type of the event object
data: GOOG    sets the data property
data: 999     appends a newline and more data
              a blank line triggers the message event
```

There are some additional details to the protocol that allow events to be given IDs and allow a reconnecting client to tell the server what the ID of the last event it received was, so that a server can resend any events it missed. Those details are not important here, however.

One obvious application for the Comet architecture is online chat: a chat client can post new messages to the chat room with XMLHttpRequest and can subscribe to the stream of chatter with an EventSource object. Example 18-15 demonstrates how easy it is to write a chat client like this with EventSource.

Example 18-15. A simple chat client, using EventSource

```
<script>
window.onload = function() {
    // Take care of some UI details
    var nick = prompt("Enter your nickname");      // Get user's nickname
    var input = document.getElementById("input"); // Find the input field
    input.focus();                                 // Set keyboard focus

    // Register for notification of new messages using EventSource
    var chat = new EventSource("/chat");
    chat.onmessage = function(event) {             // When a new message arrives
        var msg = event.data;                      // Get text from event object
        var node = document.createTextNode(msg);  // Make it into a text node
        var div = document.createElement("div");  // Create a <div>
        div.appendChild(node);                     // Add text node to div
        document.body.insertBefore(div, input);   // And add div before input
        input.scrollIntoView();                    // Ensure input elt is visible
    }

    // Post the user's messages to the server using XMLHttpRequest
    input.onchange = function() {                  // When user strikes return
```

```
        var msg = nick + ": " + input.value;      // Username plus user's input
        var xhr = new XMLHttpRequest();           // Create a new XHR
        xhr.open("POST", "/chat");                // to POST to /chat.
        xhr.setRequestHeader("Content-Type",      // Specify plain UTF-8 text
                             "text/plain;charset=UTF-8");
        xhr.send(msg);                            // Send the message
        input.value = "";                         // Get ready for more input
    }
};
</script>
<!-- The chat UI is just a single text input field -->
<!-- New chat messages will be inserted before this input field -->
<input id="input" style="width:100%"/>
```


At the time of this writing, EventSource is supported in Chrome and Safari, and Mozilla is expected to implement it in the first release after Firefox 4.0. In browsers (like Firefox) whose XMLHttpRequest implementation fires a readystatechange event (for `readyState` 3) whenever there is download progress, it is relatively easy to emulate EventSource with XMLHttpRequest, and Example 18-16 shows how this can be done. With this emulation module, Example 18-15 works in Chrome, Safari, *and* Firefox. (Example 18-16 does not work in IE or Opera, since their XMLHttpRequest implementations do not generate events on download progress.)

Example 18-16. Emulating EventSource with XMLHttpRequest

```
// Emulate the EventSource API for browsers that do not support it.
// Requires an XMLHttpRequest that sends readystatechange events whenever
// there is new data written to a long-lived HTTP connection. Note that
// this is not a complete implementation of the API: it does not support the
// readyState property, the close() method, nor the open and error events.
// Also event registration for message events is through the onmessage
// property only--this version does not define an addEventListener method.
if (window.EventSource === undefined) {     // If EventSource is not defined,
    window.EventSource = function(url) {    // emulate it like this.
        var xhr;                            // Our HTTP connection...
        var evtsrc = this;                  // Used in the event handlers.
        var charsReceived = 0;              // So we can tell what is new.
        var type = null;                    // To check property response type.
        var data = "";                      // Holds message data
        var eventName = "message";          // The type field of our event objects
        var lastEventId = "";               // For resyncing with the server
        var retrydelay = 1000;              // Delay between connection attempts
        var aborted = false;                // Set true to give up on connecting

        // Create an XHR object
        xhr = new XMLHttpRequest();

        // Define an event handler for it
        xhr.onreadystatechange = function() {
            switch(xhr.readyState) {
            case 3: processData(); break;   // When a chunk of data arrives
            case 4: reconnect(); break;     // When the request closes
            }
        };
```

```
// And establish a long-lived connection through it
connect();

// If the connection closes normally, wait a second and try to restart
function reconnect() {
    if (aborted) return;            // Don't reconnect after an abort
    if (xhr.status >= 300) return;  // Don't reconnect after an error
    setTimeout(connect, retrydelay); // Wait a bit, then reconnect
};

// This is how we establish a connection
function connect() {
    charsReceived = 0;
    type = null;
    xhr.open("GET", url);
    xhr.setRequestHeader("Cache-Control", "no-cache");
    if (lastEventId) xhr.setRequestHeader("Last-Event-ID", lastEventId);
    xhr.send();
}

// Each time data arrives, process it and trigger the onmessage handler
// This function handles the details of the Server-Sent Events protocol
function processData() {
    if (!type) {   // Check the response type if we haven't already
        type = xhr.getResponseHeader('Content-Type');
        if (type !== "text/event-stream") {
            aborted = true;
            xhr.abort();
            return;
        }
    }
    // Keep track of how much we've received and get only the
    // portion of the response that we haven't already processed.
    var chunk = xhr.responseText.substring(charsReceived);
    charsReceived = xhr.responseText.length;

    // Break the chunk of text into lines and iterate over them.
    var lines = chunk.replace(/(\r\n|\r|\n)$/, "").split(/\r\n|\r|\n/);
    for(var i = 0; i < lines.length; i++) {
        var line = lines[i], pos = line.indexOf(":"), name, value="";
        if (pos == 0) continue;              // Ignore comments
        if (pos > 0) {                       // field name:value
            name = line.substring(0,pos);
            value = line.substring(pos+1);
            if (value.charAt(0) == " ") value = value.substring(1);
        }
        else name = line;                    // field name only

        switch(name) {
        case "event": eventName = value; break;
        case "data": data += value + "\n"; break;
        case "id": lastEventId = value; break;
        case "retry": retrydelay = parseInt(value) || 1000; break;
        default: break;  // Ignore any other line
        }
```

```
                if (line === "") {  // A blank line means send the event
                    if (evtsrc.onmessage && data !== "") {
                        // Chop trailing newline if there is one
                        if (data.charAt(data.length-1) == "\n")
                            data = data.substring(0, data.length-1);
                        evtsrc.onmessage({    // This is a fake Event object
                            type: eventName,  // event type
                            data: data,       // event data
                            origin: url       // the origin of the data
                        });
                    }
                    data = "";
                    continue;
                }
            }
        }
    };
}
```

We conclude this exploration of the Comet architecture with a server example. Example 18-17 is a custom HTTP server written in server-side JavaScript for the Node (§12.2) server-side environment. When a client requests the root URL "/", it sends the chat client code shown in Example 18-15 and the emulation code from Example 18-16. When a client makes a GET request for the URL "/chat", it saves the response stream in an array and keeps that connection open. And when a client makes a POST request to "/chat", it uses the body of the request as a chat message and writes it, prefixed with the Server-Sent Events "data:" prefix, to each of the open response streams. If you install Node, you can run this server example locally. It listens on port 8000, so after starting the server, you'd point your browser to `http://localhost:8000` to connect and begin chatting with yourself.

Example 18-17. A custom Server-Sent Events chat server

```
// This is server-side JavaScript, intended to be run with NodeJS.
// It implements a very simple, completely anonymous chat room.
// POST new messages to /chat, or GET a text/event-stream of messages
// from the same URL. Making a GET request to / returns a simple HTML file
// that contains the client-side chat UI.
var http = require('http');  // NodeJS HTTP server API

// The HTML file for the chat client. Used below.
var clientui = require('fs').readFileSync("chatclient.html");
var emulation = require('fs').readFileSync("EventSourceEmulation.js");

// An array of ServerResponse objects that we're going to send events to
var clients = [];

// Send a comment to the clients every 20 seconds so they don't
// close the connection and then reconnect
setInterval(function() {
    clients.forEach(function(client) {
        client.write(":ping\n");
```

```
    });
}, 20000);

// Create a new server
var server = new http.Server();

// When the server gets a new request, run this function
server.on("request", function (request, response) {
    // Parse the requested URL
    var url = require('url').parse(request.url);

    // If the request was for "/", send the client-side chat UI.
    if (url.pathname === "/") {  // A request for the chat UI
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("<script>" + emulation + "</script>");
        response.write(clientui);
        response.end();
        return;
    }
    // Send 404 for any request other than "/chat"
    else if (url.pathname !== "/chat") {
        response.writeHead(404);
        response.end();
        return;
    }

    // If the request was a post, then a client is posting a new message
    if (request.method === "POST") {
        request.setEncoding("utf8");
        var body = "";
        // When we get a chunk of data, add it to the body
        request.on("data", function(chunk) { body += chunk; });

        // When the request is done, send an empty response
        // and broadcast the message to all listening clients.
        request.on("end", function() {
            response.writeHead(200);   // Respond to the request
            response.end();

            // Format the message in text/event-stream format
            // Make sure each line is prefixed with "data:" and that it is
            // terminated with two newlines.
            message = 'data: ' + body.replace('\n', '\ndata: ') + "\r\n\r\n";
            // Now send this message to all listening clients
            clients.forEach(function(client) { client.write(message); });
        });
    }
    // Otherwise, a client is requesting a stream of messages
    else {
        // Set the content type and send an initial message event
        response.writeHead(200, {'Content-Type': "text/event-stream" });
        response.write("data: Connected\n\n");

        // If the client closes the connection, remove the corresponding
        // response object from the array of active clients
        request.connection.on("end", function() {
```

```
            clients.splice(clients.indexOf(response), 1);
            response.end();
        });

        // Remember the response object so we can send future messages to it
        clients.push(response);
    }
});

// Run the server on port 8000. Connect to http://localhost:8000/ to use it.
server.listen(8000);
```

CHAPTER 19
The jQuery Library

JavaScript has an intentionally simple core API and an overly complicated client-side API that is marred by major incompatibilities between browsers. The arrival of IE9 eliminates the worst of those incompatibilities, but many programmers find it easier to write web applications using a JavaScript framework or utility library to simplify common tasks and hide the differences between browsers. At the time of this writing, one of the most popular and widely used such libraries is jQuery.[1]

Because the jQuery library has become so widely used, web developers should be familiar with it: even if you don't use it in your own code, you are likely to encounter it in code written by others. Fortunately, jQuery is stable and small enough to document in this book. You'll find a comprehensive introduction in this chapter, and Part IV includes a `jQuery` quick reference. jQuery methods do not have individual entries in the reference section, but the `jQuery` gives a synopsis of each method.

jQuery makes it easy to find the elements of a document that you care about and then manipulate those elements by adding content, editing HTML attributes and CSS properties, defining event handlers, and performing animations. It also has Ajax utilities for dynamically making HTTP requests and general-purpose utility functions for working with objects and arrays.

As its name implies, the jQuery library is focused on *queries*. A typical query uses a CSS selector to identify a set of document elements and returns an object that represents those elements. This returned object provides many useful methods for operating on the matching elements as a group. Whenever possible, these methods return the object on which they are invoked, which allows a succinct method chaining idiom to be used. These features are at the heart of jQuery's power and utility:

- An expressive syntax (CSS selectors) for referring to elements in the document
- An efficient query method for finding the set of document elements that match a CSS selector

1. Other commonly used libraries not covered in this book include Prototype, YUI, and dojo. Search the Web for "JavaScript libraries" to find many more.

- A useful set of methods for manipulating selected elements
- Powerful functional programming techniques for operating on sets of elements as a group, rather than one at a time
- A succinct idiom (method chaining) for expressing sequences of operations

This chapter begins with an introduction to jQuery that shows how to make simple queries and work with the results. The sections that follow explain:

- How to set HTML attributes, CSS styles and classes, HTML form values and element content, geometry, and data
- How to alter the structure of a document by inserting, replacing, wrapping, and deleting elements
- How to use jQuery's cross-browser event model
- How to produce animated visual effects with jQuery
- jQuery's Ajax utilities for making scripted HTTP requests
- jQuery's utility functions
- The full syntax of jQuery's selectors, and how to use jQuery's advanced selection methods
- How to extend jQuery by using and writing plug-ins
- The jQuery UI library

19.1 jQuery Basics

The jQuery library defines a single global function named `jQuery()`. This function is so frequently used that the library also defines the global symbol `$` as a shortcut for it. These are the only two symbols jQuery defines in the global namespace.[2]

This single global function with two names is the central query function for jQuery. Here, for example, is how we ask for the set of all `<div>` elements in a document:

```
var divs = $("div");
```

The value returned by this function represents a set of zero or more DOM elements and is known as a jQuery object. Note that `jQuery()` is a factory function rather than a constructor: it returns a newly created object but is not used with the `new` keyword. jQuery objects define many methods for operating on the sets of elements they represent, and most of this chapter is devoted to explaining those methods. Below, for example, is code that finds, highlights, and quickly displays all hidden `<p>` elements that have a class of "details":

```
$("p.details").css("background-color", "yellow").show("fast");
```

2. If you use `$` in your own code, or are using another library, such as Prototype, that uses `$`, you can call `jQuery.noConflict()` to restore `$` to its original value.

The `css()` method operates on the jQuery object returned by `$()`, and returns that same object, so that the `show()` method can be invoked next in a compact "method chain." This method chaining idiom is common in jQuery programming. As another example, the code below finds all elements in the document that have the CSS class "clicktohide" and registers an event handler on each one. That event handler is invoked when the user clicks on the element and makes the element slowly "slide up" and disappear:

```
$(".clicktohide").click(function() { $(this).slideUp("slow"); });
```


Obtaining jQuery

The jQuery library is free software. You can download it from *http://jquery.com*. Once you have the code, you can include it in your web pages with a `<script>` element like this:

```
<script src="jquery-1.4.2.min.js"></script>
```

The "min" in the filename above indicates that this is the minimized version of the library, with unnecessary comments and whitespace removed, and internal identifiers replaced with shorter ones.

Another way to use jQuery in your web applications is to allow a content distribution network to serve it using a URL like one of these:

```
http://code.jquery.com/jquery-1.4.2.min.js
http://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.min.js
http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js
```

This chapter documents jQuery version 1.4. If you are using a different version, you should replace the "1.4.2" version number in the URLs above as necessary.[3] If you use the Google CDN, you can use "1.4" to get the latest release in the 1.4.x series, or just "1" to get the most current release less than 2.0. The major advantage of loading `jQuery` from well-known URLs like these is that, because of jQuery's popularity, visitors to your website will likely already have a copy of the library in their browser's cache and no download will be necessary.

19.1.1 The jQuery() Function

The `jQuery()` function (a.k.a. `$()`) is the most important one in the jQuery library. It is heavily overloaded, however, and there are four different ways you can invoke it.

The first, and most common, way to invoke `$()` is to pass a CSS selector (a string) to it. When called this way, it returns the set of elements from the current document that match the selector. jQuery supports most of the CSS3 selector syntax, plus some extensions of its own. Complete details of the jQuery selector syntax are in §19.8.1. If you pass an element or a jQuery object as the second argument to `$()`, it returns only

3. When this chapter was written, the current version of jQuery was 1.4.2. As the book goes to press, jQuery 1.5 has just been released. The changes in jQuery 1.5 mostly involve the Ajax utility function, and they will be mentioned in passing in §19.6.

matching descendants of the specified element or elements. This optional second argument value defines the starting point (or points) for the query and is often called the *context*.

The second way to invoke `$()` is to pass it an Element or Document or Window object. Called like this, it simply wraps the element, document, or window in a jQuery object and returns that object. Doing this allows you to use jQuery methods to manipulate the element rather than using raw DOM methods. It is common to see jQuery programs call `$(document)` or `$(this)`, for example. jQuery objects can represent more than one element in a document, and you can also pass an array of elements to `$()`. In this case, the returned jQuery object represents the set of elements in your array.

The third way to invoke `$()` is to pass it a string of HTML text. When you do this, jQuery creates the HTML element or elements described by that text and then returns a jQuery object representing those elements. jQuery does not automatically insert the newly created elements into the document, but the jQuery methods described in §19.3 allow you to easily insert them where you want them. Note that you cannot pass plain text when you invoke `$()` in this way, or jQuery will think you are passing a CSS selector. For this style of invocation, the string you pass to `$()` must include at least one HTML tag with angle brackets.

When invoked in this third way, `$()` accepts an optional second argument. You can pass a Document object to specify the document with which the elements are to be associated. (If you are creating elements to be inserted into an `<iframe>`, for example, you'll need to explicitly specify the document object of that frame.) Or you can pass an object as the second argument. If you do this, the object properties are assumed to specify the names and values of HTML attributes to be set on the object. But if the object includes properties with any of the names "css", "html", "text", "width", "height", "offset", "val", or "data", or properties that have the same name as any of the jQuery event handler registration methods, jQuery will invoke the method of the same name on the newly created element and pass the property value to it. (Methods like `css()`, `html()`, and `text()` are covered in §19.2 and event handler registration methods are in §19.4. For example:

```
var img = $("<img/>",                  // Create a new <img> element
           { src:url,                  // with this HTML attribute,
             css: {borderWidth:5},     // this CSS style,
             click: handleClick        // and this event handler.
           });
```

Finally, the fourth way to invoke `$()` is to pass a function to it. If you do this, the function you pass will be invoked when the document has been loaded and the DOM is ready to be manipulated. This is the jQuery version of the `onLoad()` function from Example 13-5. It is very common to see jQuery programs written as anonymous functions defined within a call to `jQuery()`:

```
jQuery(function() {  // Invoked when the document has loaded
    // All jQuery code goes here
});
```

You'll sometimes see `$(f)` written using the older and more verbose form: `$(document).ready(f)`.

The function you pass to `jQuery()` will be invoked with the document object as its `this` value and with the `jQuery` function as its single argument. This means that you can undefine the global `$` function and still use that convenient alias locally with this idiom:

```
jQuery.noConflict();  // Restore $ to its original state
jQuery(function($) {  // Use $ as a local alias for the jQuery object
    // Put all your jQuery code here
});
```

jQuery triggers functions registered through `$()` when the DOMContentLoaded event is fired (§13.3.4) or, in browsers that don't support that event, when the load event is fired. This means that the document will be completely parsed, but that external resources such as images may not be loaded yet. If you pass a function to `$()` after the DOM is ready, that function will be invoked immediately, before `$()` returns.

The jQuery library also uses the `jQuery()` function as its namespace and defines a number of utility functions and properties under it. The `jQuery.noConflict()` function mentioned above is one such utility function. Others include `jQuery.each()` for general-purpose iteration and `jQuery.parseJSON()` for parsing JSON text. §19.7 lists general-purpose utility functions, and other jQuery functions are described throughout this chapter.

jQuery Terminology

Let's pause here to define some important terms and phrases that you'll see throughout this chapter:

"the jQuery function"
: The jQuery function is the value of `jQuery` or of `$`. This is the function that creates jQuery objects, registers handlers to be invoked when the DOM is ready, and that also serves as the jQuery namespace. I usually refer to it as `$()`. Because it serves as a namespace, the jQuery function might also be called "the global jQuery object," but it is very important not to confuse it with "a jQuery object."

"a jQuery object"
: A jQuery object is an object returned by the jQuery function. A jQuery object represents a set of document elements and can also be called a "jQuery result," a "jQuery set," or a "wrapped set."

"the selected elements"
: When you pass a CSS selector to the jQuery function, it returns a jQuery object that represents the set of document elements that match that selector. When describing the methods of the jQuery object, I'll often use the phrase "the selected elements" to refer to those matching elements. For example, to explain the `attr()` method, I might write "the `attr()` method sets HTML attributes on the selected elements." This is instead of a more precise but awkward description like "the `attr()` method sets HTML attributes on the elements of the jQuery object on

which it was invoked." Note that the word "selected" refers to the CSS selector and has nothing to do with any selection performed by the user.

"a jQuery function"
: A jQuery function is a function like `jQuery.noConflict()` that is defined in the namespace of *the* jQuery function. jQuery functions might also be described as "static methods."

"a jQuery method"
: A jQuery method is a method of a jQuery object returned by the jQuery function. The most important part of the jQuery library is the powerful methods it defines.

The distinction between jQuery functions and methods is sometimes tricky because a number of functions and methods have the same name. Note the differences between these two lines of code:

```
// Call the jQuery function each() to
// invoke the function f once for each element of the array a
$.each(a,f);

// Call the jQuery() function to obtain a jQuery object that represents all
// <a> elements in the document. Then call the each() method of that jQuery
// object to invoke the function f once for each selected element.
$("a").each(f);
```

The official jQuery documentation at *http://jquery.com* uses names like `$.each` to refer to jQuery functions and names like `.each` (with a period but without a dollar sign) to refer to jQuery methods. In this book, I'll use the terms "function" and "method" instead. Usually it will be clear from the context which is being discussed.

19.1.2 Queries and Query Results

When you pass a CSS selector string to `$()`, it returns a jQuery object that represents the set of matched (or "selected") elements. CSS selectors were introduced in §15.2.5, and you can review that section for examples—all of the examples shown there work when passed to `$()`. The specific selector syntax supported by jQuery is detailed in §19.8.1. Rather than focus on those advanced selector details now, however, we're going to first explore what you can do with the results of a query.

The value returned by `$()` is a jQuery object. jQuery objects are array-like: they have a `length` property and have numeric properties from 0 to `length-1`. (See §7.11 for more on array-like objects.) This means that you can access the contents of the jQuery object using standard square-bracket array notation:

```
$("body").length    // => 1: documents have only a single body element
$("body")[0]        // This the same as document.body
```

If you prefer not to use array notation with jQuery objects, you can use the `size()` method instead of the `length` property and the `get()` method instead of indexing with square brackets. If you need to convert a jQuery object to a true array, call the `toArray()` method.

In addition to the length property, jQuery objects have three other properties that are sometimes of interest. The selector property is the selector string (if any) that was used when the jQuery object was created. The context property is the context object that was passed as the second argument to $(), or the Document object otherwise. Finally, all jQuery objects have a property named jquery, and testing for the existence of this property is a simple way to distinguish jQuery objects from other array-like objects. The value of the jquery property is the jQuery version number as a string:

```
// Find all <script> elements in the document body
var bodyscripts = $("script", document.body);
bodyscripts.selector   // => "script"
bodyscripts.context    // => document.body
bodyscripts.jquery     // => "1.4.2"
```


$() versus querySelectorAll()

The $() function is similar to the Document method querySelectorAll() that was described in §15.2.5: both take a CSS selector as their argument and return an array-like object that holds the elements that match the selector. The jQuery implementation uses querySelectorAll() in browsers that support it, but there are good reasons to use $() instead of querySelectorAll() in your own code:

- querySelectorAll() has only recently been implemented by browser vendors. $() works in older browsers as well as new ones.
- Because jQuery can perform selections "by hand," the CSS3 selectors supported by $() work in all browsers, not just those browsers that support CSS3.
- The array-like object returned by $() (a jQuery object) is much more useful than the array-like object (a NodeList) returned by querySelectorAll().

If you want to loop over all elements in a jQuery object, you can call the each() method instead of writing a for loop. The each() method is something like the ECMAScript 5 (ES5) forEach() array method. It expects a callback function as its sole argument, and it invokes that callback function once for each element in the jQuery object (in document order). The callback is invoked as a method of the matched element, so within the callback the this keyword refers to an Element object. each() also passes the index and the element as the first and second arguments to the callback. Note that this and the second argument are raw document elements, not jQuery objects; if you want to use a jQuery method to manipulate the element, you'll need to pass it to $() first.

jQuery's each() method has one feature that is quite different than forEach(): if your callback returns false for any element, iteration is terminated after that element (this is like using the break keyword in a normal loop). each() returns the jQuery object on which it is called, so that it can be used in method chains. Here is an example (it uses the prepend() method that will be explained in §19.3):

```
// Number the divs of the document, up to and including div#last
$("div").each(function(idx) { // find all <div>s, and iterate through them
    $(this).prepend(idx + ": ");          // Insert index at start of each
```

```
    if (this.id === "last") return false; // Stop at element #last
});
```

Despite the power of the each() method, it is not very commonly used, since jQuery methods usually iterate implicitly over the set of matched elements and operate on them all. You typically only need to use each() if you need to manipulate the matched elements in different ways. Even then, you may not need to call each(), since a number of jQuery methods allow you to pass a callback function.

The jQuery library predates the ES5 array methods, and it defines a couple of other methods that provide functionality similar to the ES5 methods. The jQuery method map() works much like the Array.map() method. It accepts a callback function as its argument and invokes that function once for each element of the jQuery object, collecting the return values of those invocations, and returning a new jQuery object holding those return values. map() invokes the callback in the same way that the each() method does: the element is passed as the this value and as the second argument and the index of the element is passed as the first argument. If the callback returns null or undefined, that value is ignored and nothing is added to the new jQuery object for that invocation. If the callback returns an array or an array-like object (such as a jQuery object), it is "flattened" and its elements are added individually to the new jQuery object. Note that the jQuery object returned by map() may not hold document elements, but it still works as an array-like object. Here is an example:

```
// Find all headings, map to their ids, convert to a true array, and sort it.
$(":header").map(function() { return this.id; }).toArray().sort();
```

Along with each() and map(), another fundamental jQuery method is index(). This method expects an element as its argument and returns the index of that element in the jQuery object, or –1 if it is not found. In typical jQuery fashion, however, this index() method is overloaded. If you pass a jQuery object as the argument, index() searches for the first element of that object. If you pass a string, index() uses it as a CSS selector and returns the index of the first element of this jQuery object in the set of elements matching that selector. And if you pass no argument, index() returns the index of the first element of this jQuery object within its sibling elements.

The final general-purpose jQuery method we'll discuss here is is(). It takes a selector as its argument and returns true if at least one of the selected elements also matches the specified selector. You might use it in an each() callback function, for example:

```
$("div").each(function() {   // For each <div> element
    if ($(this).is(":hidden")) return;  // Skip hidden elements
    // Do something with the visible ones here
});
```

19.2 jQuery Getters and Setters

Some of the simplest, and most common, operations on jQuery objects are those that get or set the value of HTML attributes, CSS styles, element content, or element

geometry. This section describes those methods. First, however, it is worth making some generalizations about getter and setter methods in jQuery:

- Rather than defining a pair of methods, jQuery uses a single method as both getter and setter. If you pass a new value to the method, it sets that value; if you don't specify a value, it returns the current value.
- When used as setters, these methods set values on every element in the jQuery object, and then return the jQuery object to allow method chaining.
- When used as getters, these methods query only the first element of the set of elements and return a single value. (Use `map()` if you want to query all elements.) Since getters do not return the jQuery object they are invoked on, they can only appear at the end of a method chain.
- When used as setters, these methods often accept object arguments. In this case, each property of the object specifies a name and a value to be set.
- When used as setters, these methods often accept functions as values. In this case, the function is invoked to compute the value to be set. The element that the value is being computed for is the `this` value, the element index is passed as the first argument to the function, and the current value is passed as the second argument.

Keep these generalizations about getters and setters in mind as you read the rest of this section. Each subsection below explains an important category of jQuery getter/setter methods.

19.2.1 Getting and Setting HTML Attributes

The `attr()` method is the jQuery getter/setter for HTML attributes, and it adheres to each of the generalizations described above. `attr()` handles browser incompatibilities and special cases and allows you to use either HTML attribute names or their JavaScript property equivalents (where they differ). For example, you can use either "for" or "htmlFor" and either "class" or "className". `removeAttr()` is a related function that completely removes an attribute from all selected elements. Here are some examples:

```
$("form").attr("action");               // Query the action attr of 1st form
$("#icon").attr("src", "icon.gif");     // Set the src attribute
$("#banner").attr({src:"banner.gif",    // Set 4 attributes at once
                   alt:"Advertisement",
                   width:720, height:64});
$("a").attr("target", "_blank");        // Make all links load in new windows
$("a").attr("target", function() {      // Load local links locally and load
    if (this.host == location.host) return "_self"
    else return "_blank";               // off-site links in a new window
});
$("a").attr({target: function() {...}}); // We can also pass functions like this
$("a").removeAttr("target");            // Make all links load in this window
```

19.2.2 Getting and Setting CSS Attributes

The `css()` method is very much like the `attr()` method, but it works with the CSS styles of an element rather than the HTML attributes of the element. When querying style values, `css()` returns the current (or "computed"; see §16.4) style of the element: the returned value may come from the `style` attribute or from a stylesheet. Note that it is not possible to query compound styles such as "font" or "margin". You must instead query individual styles such as "font-weight", "font-family", "margin-top", or "margin-left". When setting styles, the `css()` method simply adds the style to the element's `style` attribute. `css()` allows you to use hyphenated CSS style names ("background-color") or camel-case JavaScript style names ("backgroundColor"). When querying style values, `css()` returns numeric values as strings, with units suffixes included. When setting, however, it converts numbers to strings and adds a "px" (pixels) suffix to them when necessary:

```
$("h1").css("font-weight");              // Get font weight of first <h1>
$("h1").css("fontWeight");               // Camel case works, too
$("h1").css("font");                     // Error: can't query compound styles
$("h1").css("font-variant",              // Set a style on all <h1> elements
            "smallcaps");
$("div.note").css("border",              // Okay to set compound styles
                  "solid black 2px");
$("h1").css({ backgroundColor: "black",  // Set multiple styles at once
              textColor: "white",        // camelCase names work better
              fontVariant: "small-caps", // as object properties
              padding: "10px 2px 4px 20px",
              border: "dotted black 4px" });
// Increase all <h1> font sizes by 25%
$("h1").css("font-size", function(i,curval) {
              return Math.round(1.25*parseInt(curval));
            });
```

19.2.3 Getting and Setting CSS Classes

Recall that the value of the `class` attribute (accessed via the `className` property in JavaScript) is interpreted as a space-separated list of CSS class names. Usually, we want to add, remove, or test for the presence of a single name in the list rather than replacing one list of classes with another. For this reason, jQuery defines convenience methods for working with the `class` attribute. `addClass()` and `removeClass()` add and remove classes from the selected elements. `toggleClass()` adds classes to elements that don't already have them and removes classes from those that do. `hasClass()` tests for the presence of a specified class. Here are some examples:

```
// Adding CSS classes
$("h1").addClass("hilite");              // Add a class to all <h1> elements
$("h1+p").addClass("hilite first");      // Add 2 classes to <p> elts after <h1>
$("section").addClass(function(n) {      // Pass a function to add a custom class
    return "section" + n;                //  to each matched element
});

// Removing CSS classes
```

```
$("p").removeClass("hilite");          // Remove a class from all <p> elements
$("p").removeClass("hilite first");    // Multiple classes are allowed
$("section").removeClass(function(n) { // Remove custom classes from elements
    return "section" + n;
});
$("div").removeClass();                // Remove all classes from all <div>s

// Toggling CSS classes
$("tr:odd").toggleClass("oddrow");     // Add the class if it is not there
                                       // or remove it if it is
$("h1").toggleClass("big bold");       // Toggle two classes at once
$("h1").toggleClass(function(n) {      // Toggle a computed class or classes
    return "big bold h1-" + n;
});
$("h1").toggleClass("hilite", true);   // Works like addClass
$("h1").toggleClass("hilite", false);  // Works like removeClass

// Testing for CSS classes
$("p").hasClass("first")               // Does any p element have this class?
$("#lead").is(".first")                // This does the same thing
$("#lead").is(".first.hilite")         // is() is more flexible than hasClass()
```

Note that the `hasClass()` method is less flexible than `addClass()`, `removeClass()`, and `toggleClass()`. `hasClass()` works for only a single class name and does not support function arguments. It returns `true` if any of the selected elements has the specified CSS class and returns `false` if none of them do. The `is()` method (described in §19.1.2) is more flexible and can be used for the same purpose.

These jQuery methods are like the `classList` methods described in §16.5, but the jQuery methods work in all browsers, not just those that support the HTML5 `class List` property. Also, of course, the jQuery methods work for multiple elements and can be chained.

19.2.4 Getting and Setting HTML Form Values

`val()` is a method for setting and querying the `value` attribute of HTML form elements and also for querying and setting the selection state of checkboxes, radio buttons, and `<select>` elements:

```
$("#surname").val()              // Get value from the surname text field
$("#usstate").val()              // Get single value from <select>
$("select#extras").val()         // Get array of values from <select multiple>

$("input:radio[name=ship]:checked").val()  // Get val of checked radio button
$("#email").val("Invalid email address")   // Set value of a text field
$("input:checkbox").val(["opt1", "opt2"])  // Check any checkboxes with
                                           // these names or values
$("input:text").val(function() { // Reset all text fields to their default
    return this.defaultValue;
})
```

19.2.5 Getting and Setting Element Content

The `text()` and `html()` methods query and set the plain-text or HTML content of an element or elements. When invoked with no arguments, `text()` returns the plain-text content of all descendant text nodes of all matched elements. This works even in browsers that do not support the `textContent` or `innerText` properties (§15.5.2).

If you invoke the `html()` method with no arguments, it returns the HTML content of just the first matched element. jQuery uses the `innerHTML` property to do this: `x.html()` is effectively the same as `x[0].innerHTML`.

If you pass a string to `text()` or `html()`, that string will be used for the plain-text or HTML-formatted text content of the element, and it will replace all existing content. As with the other setter methods we've seen, you can also pass a function, which will be used to compute the new content string:

```
var title = $("head title").text()   // Get document title
var headline = $("h1").html()        // Get html of first <h1> element
$("h1").text(function(n,current) {   // Give each heading a section number
    return "§" + (n+1) + ": " + current
});
```

19.2.6 Getting and Setting Element Geometry

We learned in §15.8 that it can be tricky to correctly determine the size and position of an element, especially in browsers that do not support `getBoundingClientRect()` (§15.8.2). jQuery simplifies these computations with methods that work in any browser. Note that all of the methods described here are getters, but only some can also be used as setters.

To query or set the position of an element, use the `offset()` method. This method measures positions relative to the document and returns them in the form of an object with `left` and `top` properties that hold the X and Y coordinates. If you pass an object with these properties to the method, it sets the position you specify. It sets the CSS `position` attribute as necessary to make elements positionable:

```
var elt = $("#sprite");      // The element we want to move
var position = elt.offset(); // Get its current position
position.top += 100;         // Change its Y coordinate
elt.offset(position);        // Set the new position

// Move all <h1> elements to the right by a distance that depends on their
// position in the document
$("h1").offset(function(index,curpos) {
    return {left: curpos.left + 25*index, top:curpos.top};
});
```

The `position()` method is like `offset()`, except that it is a getter only and it returns element positions relative to their offset parent, rather than relative to the document as a whole. In §15.8.5, we saw that every element has an `offsetParent` property that its position is relative to. Positioned elements always serve as the offset parents for their

descendants, but some browsers also make other elements, such as table cells, into offset parents. jQuery only considers positioned elements to be offset parents, and the `offset Parent()` method of a jQuery object maps each element to the nearest positioned ancestor element or to the `<body>` element. Note the unfortunate naming mismatch for these methods: `offset()` returns the absolute position of an element, in document coordinates. And `position()` returns the offset of an element relative to its `offsetParent()`.

There are three getters for querying the width of an element and three for querying the height. The `width()` and `height()` methods return the basic width and height and do not include padding, borders, or margins. `innerWidth()` and `innerHeight()` return the width and height of an element plus the width and height of its padding (the word "inner" refers to the fact that these methods return the dimensions measured to the inside of the border). `outerWidth()` and `outerHeight()` normally return the element's dimensions plus its padding and border. If you pass the value `true` to either of these methods, they also add in the size of the element's margins. The code below shows four different widths that you can compute for an element:

```
var body = $("body");
var contentWidth = body.width();
var paddingWidth = body.innerWidth();
var borderWidth = body.outerWidth();
var marginWidth = body.outerWidth(true);
var padding = paddingWidth-contentWidth; // sum of left and right padding
var borders = borderWidth-paddingWidth;  // sum of left and right border widths
var margins = marginWidth-borderWidth;   // sum of left and right margins
```

The `width()` and `height()` methods have features that the other four methods (the inner and outer methods) do not. First, if the first element of the jQuery object is a Window or Document object, they return the size of the window's viewport or the full size of the document. The other methods only work for elements, not windows or documents.

The other feature of the `width()` and `height()` methods is that they are setters as well as getters. If you pass a value to these methods, they set the width or height of every element in the jQuery object. (Note, however, that they cannot set the width or height of Window and Document objects.) If you pass a number, it is taken as a dimension in pixels. If you pass a string value, it is used as the value of the CSS `width` or `height` attribute and can therefore use any CSS unit. Finally, as with other setters, you can pass a function that will be called to compute the width or height.

There is a minor asymmetry between the getter and setter behavior of `width()` and `height()`. When used as getters, these methods return the dimensions of an element's content box, excluding padding, borders, and margins. When you use them as setters, however, they simply set the CSS `width` and `height` attributes. By default, those attributes also specify the size of the content box. But if an element has its CSS `box-sizing` attribute (§16.2.3.1) set to `border-box`, the `width()` and `height()` methods set dimensions that include the padding and border. For an element `e` that uses the content-box box model, calling `$(e).width(x).width()` returns the value `x`. For elements that use the border-box model, however, this is not generally the case.

The final pair of geometry-related jQuery methods are `scrollTop()` and `scrollLeft()`, which query the scrollbar positions for an element or set the scrollbar positions for all elements. These methods work for the Window object as well as for document elements, and when invoked on a Document, they query or set the scrollbar positions of the Window that holds the document. Unlike with other setters, you cannot pass a function to `scrollTop()` or `scrollLeft()`.

We can use `scrollTop()` as a getter and a setter, along with the `height()` method to define a method that scrolls the window up or down by the number of pages you specify:

```
// Scroll the window by n pages. n can be fractional or negative
function page(n) {
    var w = $(window);                  // Wrap the window in a jQuery object
    var pagesize = w.height();          // Get the size of a page
    var current = w.scrollTop();        // Get the current scrollbar position
    w.scrollTop(current + n*pagesize);  // Set new scrollbar position
}
```

19.2.7 Getting and Setting Element Data

jQuery defines a getter/setter method named `data()` that sets or queries data associated with any document element or with the Document or Window objects. The ability to associate data with any element is an important and powerful one: it is the basis for jQuery's event handler registration and effects queuing mechanisms, and you may sometimes want to use the `data()` method in your own code.

To associate data with the elements in a jQuery object, call `data()` as a setter method, passing a name and a value as the two arguments. Alternatively, you can pass a single object to the `data()` setter and each property of that object will be used as a name/value pair to associate with the element or elements of the jQuery object. Note, however, that when you pass an object to `data()`, the properties of that object replace any data previously associated with the element or elements. Unlike many of the other setter methods we've seen, `data()` does not invoke functions you pass. If you pass a function as the second argument to `data()`, that function is stored, just as any other value would be.

The `data()` method can also serve as a getter, of course. When invoked with no arguments, it returns an object containing all name/value pairs associated with the first element in the jQuery object. When you invoke `data()` with a single string argument, it returns the value associated with that string for the first element.

Use the `removeData()` method to remove data from an element or elements. (Using `data()` to set a named value to `null` or `undefined` is not the same thing as actually deleting the named value.) If you pass a string to `removeData()`, the method deletes any value associated with that string for the element or elements. If you call `removeData()` with no arguments, it removes all data associated with the element or elements:

```
$("div").data("x", 1);              // Set some data
$("div.nodata").removeData("x");    // Remove some data
var x = $('#mydiv').data("x");      // Query some data
```

jQuery also defines utility function forms of the `data()` and `removeData()` methods. You can associate data with an individual element `e` using either the method or function form of `data()`:

```
$(e).data(...)   // The method form
$.data(e, ...)   // The function form
```

jQuery's data framework does not store element data as properties of the elements themselves, but it does need to add one special property to any element that has data associated with it. Some browsers do not allow properties to be added to `<applet>`, `<object>`, and `<embed>` elements, so jQuery simply does not allow data to be associated with elements of these types.

19.3 Altering Document Structure

In §19.2.5 we saw the `html()` and `text()` methods for setting element content. This section covers methods for making more complex changes to a document. Because HTML documents are represented as a tree of nodes rather than a linear sequence of characters, insertions, deletions, and replacements are not as simple as they are for strings and arrays. The subsections that follow explain the various jQuery methods for document modification.

19.3.1 Inserting and Replacing Elements

Let's begin with basic methods for insertions and replacements. Each of the methods demonstrated below takes an argument that specifies the content that is to be inserted into the document. This can be a string of plain text or of HTML to specify new content, or it can be a jQuery object or an Element or text Node. The insertion is made into or before or after or in place of (depending on the method) each of the selected elements. If the content to be inserted is an element that already exists in the document, it is moved from its current location. If it is to be inserted more than once, the element is cloned as necessary. These methods all return the jQuery object on which they are called. Note, however, that after `replaceWith()` runs, the elements in the jQuery object are no longer in the document:

```
$("#log").append("<br/>"+message); // Add content at end of the #log element
$("h1").prepend("§");              // Add section sign at start of each <h1>
$("h1").before("<hr/>");           // Insert a rule before each <h1>
$("h1").after("<hr/>");            // And after as well
$("hr").replaceWith("<br/>");      // Replace <hr/> elements with <br/>
$("h2").each(function() {          // Replace <h2> with <h1>, keeping content
               var h2 = $(this);
               h2.replaceWith("<h1>" + h2.html() + "</h1>");
             });
// after() and before() can be called on text nodes, as well
// This is another way to add a section sign at the start of each <h1>
$("h1").map(function() { return this.firstChild; }).before("§");
```

Each of these five structure-altering methods can also be passed a function that will be invoked to compute the value to be inserted. As usual, if you supply such a function, it will be invoked once for each selected element. The `this` value will be that element and the first argument will be the index of that element within the jQuery object. For `append()`, `prepend()`, and `replaceWith()`, the second argument is the current content of the element as an HTML string. For `before()` and `after()`, the function is invoked with no second argument.

The five methods demonstrated above are all invoked on target elements and are passed the content that is to be inserted as an argument. Each of those five methods can be paired with another method that works the other way around: invoked on the content and passed the target elements as the argument. This table shows the method pairs:

Operation	`$(target).method(content)`	`$(content).method(target)`
insert content at end of target	`append()`	`appendTo()`
insert content at start of target	`prepend()`	`prependTo()`
insert content after target	`after()`	`insertAfter()`
insert content before target	`before()`	`insertBefore()`
replace target with content	`replaceWith()`	`replaceAll()`

The methods demonstrated in the example code above are the ones in the second column. The methods in the third column are demonstrated below. There are a couple of important things to understand about these pairs of methods:

- If you pass a string to one of the second column methods, it is taken as a string of HTML to insert. If you pass a string to one of the third column methods, it is taken as a selector that identifies the target elements. (You can also identify the target elements directly by passing a jQuery object or Element or text node.)
- The third column methods do not accept function arguments like the second column methods do.
- The second column methods return the jQuery object on which they were invoked. The elements in that jQuery object may have new content or new siblings, but they are not themselves altered. The third column methods are invoked on the content that is being inserted and they return a new jQuery object that represents the new content after its insertion. In particular, note that if content is inserted at multiple locations, the returned jQuery object will include one element for each location.

With those differences listed, the code below performs the same operations as the code above, using the methods in the third column instead of the methods in the second column. Notice that in the second line we can't pass plain text (without angle brackets to identify it as HTML) to the `$()` method—it thinks we're specifying a selector. For this reason, we must explicitly create the text node that we want to insert:

```
$("<br/>+message").appendTo("#log");              // Append html to #log
$(document.createTextNode("§")).prependTo("h1"); // Append text node to <h1>s
```

```
$("<hr/>").insertBefore("h1");          // Insert rule before <h1>s
$("<hr/>").insertAfter("h1");           // Insert rule after <h1>s
$("<br/>").replaceAll("hr");            // Replace <hr/> with <br/>
```

19.3.2 Copying Elements

As noted above, if you insert elements that are already part of the document, those elements will simply be moved, not copied, to their new location. If you are inserting the elements in more than one place, jQuery will make copies as needed, but copies are not made when only one insertion is done. If you want to copy elements to a new location instead of moving them, you must first make a copy with the `clone()` method. `clone()` makes and returns a copy of each selected element (and of all of the descendants of those elements). The elements in the returned jQuery object are not part of the document yet, but you can insert them with one of the methods above:

```
// Append a new div, with id "linklist", to the end of the document
$(document.body).append("<div id='linklist'><h1>List of Links</h1></div>");
// Copy all links in the document and insert them into that new div
$("a").clone().appendTo("#linklist");
// Insert <br/> elements after each link so they display on separate lines
$("#linklist > a").after("<br/>");
```

`clone()` does not normally copy event handlers (§19.4) or other data you have associated with elements (§19.2.7); pass `true` if you want to clone that additional data as well.

19.3.3 Wrapping Elements

Another type of insertion into an HTML document involves wrapping a new element (or elements) around one or more elements. jQuery defines three wrapping functions. `wrap()` wraps each of the selected elements. `wrapInner()` wraps the contents of each selected element. And `wrapAll()` wraps the selected elements as a group. These methods are usually passed a newly created wrapper element or a string of HTML used to create a wrapper. The HTML string can include multiple nested elements, if desired, but there must be a single innermost element. If you pass a function to any of these methods, it will be invoked once in the context of each element (with the element index as its only argument) and should return the wrapper string, Element, or jQuery object. Here are some examples:

```
// Wrap all <h1> elements with <i> elements
$("h1").wrap(document.createElement("i")); // Produces <i><h1>...</h1></i>
// Wrap the content of all <h1> elements. Using a string argument is easier.
$("h1").wrapInner("<i/>");                 // Produces <h1><i>...</i></h1>
// Wrap the first paragraph in one anchor and div
$("body>p:first").wrap("<a name='lead'><div class='first'></div></a>");
// Wrap all the other paragraphs in another div
$("body>p:not(:first)").wrapAll("<div class='rest'></div>");
```

19.3.4 Deleting Elements

Along with insertions and replacements, jQuery also defines methods for deleting elements. `empty()` removes all children (including text nodes) of each of the selected elements, without altering the elements themselves. The `remove()` method, by contrast, removes the selected elements (and all of their content) from the document. `remove()` is normally invoked with no arguments and removes all elements in the jQuery object. If you pass an argument, however, that argument is treated as a selector, and only elements of the jQuery object that also match the selector are removed. (If you just want to remove elements from the set of selected elements, without removing them from the document, use the `filter()` method, which is covered in §19.8.2.) Note that it is not necessary to remove elements before reinserting them into the document: you can simply insert them at a new location and they will be moved.

The `remove()` method removes any event handlers (see §19.4) and other data (§19.2.7) you may have bound to the removed elements. The `detach()` method works just like `remove()` but does not remove event handlers and data. `detach()` may be more useful when you want to temporarily remove elements from the document for later reinsertion.

Finally, the `unwrap()` method performs element removal in a way that is the opposite of the `wrap()` or `wrapAll()` method: it removes the parent element of each selected element without affecting the selected elements or their siblings. That is, for each selected element, it replaces the parent of that element with its children. Unlike `remove()` and `detach()`, `unwrap()` does not accept an optional selector argument.

19.4 Handling Events with jQuery

As we saw in Chapter 17, one of the difficulties of working with events is that IE (until IE9) implements a different event API than all other browsers do. To address this difficulty, jQuery defines a uniform event API that works in all browsers. In its simple form, the jQuery API is easier to use than the standard or IE event APIs. And in its more complex full-featured form, the jQuery API is more powerful than the standard API. The subsections below have all the details.

19.4.1 Simple Event Handler Registration

jQuery defines simple event-registration methods for each of the commonly used and universally implemented browser events. To register an event handler for click events, for example, just call the `click()` method:

```
// Clicking on any <p> gives it a gray background
$("p").click(function() { $(this).css("background-color", "gray"); });
```

Calling a jQuery event-registration method registers your handler on all of the selected elements. This is typically much easier than one-at-a-time event handler registration with `addEventListener()` or `attachEvent()`.

These are the simple event handler registration methods jQuery defines:

`blur()`	`focusin()`	`mousedown()`	`mouseup()`
`change()`	`focusout()`	`mouseenter()`	`resize()`
`click()`	`keydown()`	`mouseleave()`	`scroll()`
`dblclick()`	`keypress()`	`mousemove()`	`select()`
`error()`	`keyup()`	`mouseout()`	`submit()`
`focus()`	`load()`	`mouseover()`	`unload()`

Most of these registration methods are for common event types you are already familiar with from Chapter 17. A few notes are in order, however. Focus and blur events do not bubble, but the focusin and focusout events do, and jQuery ensures that these events work in all browsers. Conversely, the mouseover and mouseout events do bubble, and this is often inconvenient because it is difficult to know whether the mouse has left the element you're interested in or whether it has simply moved out of one of the descendants of that element. mouseenter and mouseleave are nonbubbling events that solve this problem. These event types were originally introduced by IE, and jQuery ensures that they work correctly in all browsers.

The resize and unload event types are only ever fired on the Window object, so if you want to register handlers for these event types, you should invoke the `resize()`and `unload()` methods on `$(window)`. The `scroll()` method is also most often used on `$(window)`, but it can also be used on any element that has scrollbars (such as when the CSS `overflow` attribute is set to "scroll" or "auto"). The `load()` method can be called on `$(window)` to register a load event handler for the window, but it is usually better to pass your initialization function directly to `$()` as shown in §19.1.1. You can use the `load()` method on iframes and images, however. Note that when invoked with different arguments, `load()` is also used to load new content (via scripted HTTP) into an element—see §19.6.1. The `error()` method can be used on `<img>` elements to register handlers that are invoked if an image fails to load. It should not be used to set the Window `onerror` property that was described in §14.6.

In addition to these simple event registration methods, there are two special forms that are sometimes useful. The `hover()` method registers handlers for mouseenter and mouseleave events. Calling `hover(f,g)` is like calling `mouseenter(f)` and then calling `mouseleave(g)`. If you pass just one argument to `hover()`, that function is used as the handler for both enter and leave events.

The other special event registration method is `toggle()`. This method binds event handler functions to the click event. You specify two or more handler functions and jQuery invokes one of them each time a click event occurs. If you call `toggle(f,g,h)`, for example, the function `f()` is invoked to handle the first click event, `g()` is invoked to handle the second, `h()` is invoked to handle the third, and `f()` is invoked again to handle the fourth click event. Be careful when using `toggle()`: as we'll see in §19.5.1, this method can also be used to show or hide (i.e., toggle the visibility of) the selected elements.

We'll learn about other more general ways to register event handlers in §19.4.4, and we'll end this section with one more simple and convenient way to register handlers.

Recall that you can pass a string of HTML to `$()` to create the elements described by that string, and that you can pass (as a second argument) an object of attributes to be set on the newly created elements. This second argument can be any object that you would pass to the `attr()` method. But in addition, if any of the properties have the same name as the event registration methods listed above, the property value is taken as a handler function and is registered as a handler for the named event type. For example:

```
$("<img/>", {
    src: image_url,
    alt: image_description,
    className: "translucent_image",
    click: function() { $(this).css("opacity", "50%"); }
});
```

19.4.2 jQuery Event Handlers

The event handler functions in the examples above expect no arguments and return no values. It is quite normal to write event handlers like that, but jQuery does invoke every event handler with one or more arguments, and it does pay attention to the return value of your handlers. The most important thing you should know is that every event handler is passed a jQuery event object as its first argument. The fields of this object provide details (like mouse pointer coordinates) about the event. The properties of the standard Event object were described in Chapter 17. jQuery simulates that standard Event object, even in browsers (like IE8 and before) that do not support it, and jQuery event objects have the same set of fields in all browsers. This is explained in detail in §19.4.3.

Normally, event handlers are invoked with only the single event object argument. But if you explicitly trigger an event with `trigger()` (see §19.4.6), you can pass an array of extra arguments. If you do this, those arguments will be passed to the event handler after the first event object argument.

Regardless of how they are registered, the return value of a jQuery event handler function is always significant. If a handler returns `false`, both the default action associated with the event and any future propagation of the event are canceled. That is, returning `false` is the same as calling the `preventDefault()` and `stopPropagation()` methods of the Event object. Also, when an event handler returns a value (other than `undefined`), jQuery stores that value in the `result` property of the Event object where it can be accessed by subsequently invoked event handlers.

19.4.3 The jQuery Event Object

jQuery hides implementation differences between browsers by defining its own Event object. When a jQuery event handler is invoked, it is always passed a jQuery Event object as its first argument. The jQuery Event object is based heavily on W3C standards, but it also codifies some de facto event standards. jQuery copies all of the following fields from the native Event object into every jQuery Event object (though some of them will be `undefined` for certain event types):

```
altKey          ctrlKey         newValue        screenX
attrChange      currentTarget   offsetX         screenY
attrName        detail          offsetY         shiftKey
bubbles         eventPhase      originalTarget  srcElement
button          fromElement     pageX           target
cancelable      keyCode         pageY           toElement
charCode        layerX          prevValue       view
clientX         layerY          relatedNode     wheelDelta
clientY         metaKey         relatedTarget   which
```

In addition to these properties, the Event object also defines the following methods:

```
preventDefault()                isDefaultPrevented()
stopPropagation()               isPropagationStopped()
stopImmediatePropagation()      isImmediatePropagationStopped()
```

Most of these event properties and methods were introduced in Chapter 17 and are documented in Part IV under Event. Some of these fields are specially handled by jQuery to give them a uniform cross-browser behavior and are worth noting here:

`metaKey`
: If the native event object does not have a `metaKey` property, jQuery sets this to the same value as the `ctrlKey` property. In MacOS, the Command key sets the `metaKey` property.

`pageX, pageY`
: If the native event object does not define these properties, but does define the viewport coordinates of the mouse pointer in `clientX` and `clientY`, jQuery computes the document coordinates of the mouse pointer and stores them in `pageX` and `pageY`.

`target, currentTarget, relatedTarget`
: The `target` property is the document element on which the event occurred. If the native event object has a text node as the target, jQuery reports the containing Element instead. `currentTarget` is the element on which the current executing event handler was registered. This should always be the same as `this`.

  If `currentTarget` is not the same as `target`, you're handling an event that has bubbled up from the element on which it occurred and it may be useful to test the `target` element with the `is()` method (§19.1.2):

```
if ($(event.target).is("a")) return; // Ignore events that start on links
```

  `relatedTarget` is the other element involved in transition events such as mouseover and mouseout. For mouseover events, for example, the `relatedTarget` property specifies the element that the mouse pointer exited as it moved over the `target`. If the native event object does not define `relatedTarget` but does define `toElement` and `fromElement`, `relatedTarget` is set from those properties.

`timeStamp`
: The time at which the event occurred, in the millisecond representation returned by the `Date.getTime()` method. jQuery sets the field itself to work around a long-standing bug in Firefox.

`which`
: jQuery normalizes this nonstandard event property so that it specifies which mouse button or keyboard key was pressed during the event. For keyboard events, if the native event does not define `which`, but defines `charCode` or `keyCode`, `which` will be set to whichever of those properties is defined. For mouse events, if `which` is not defined but the `button` property is defined, `which` is set based on the `button` value. 0 means no buttons are pressed. 1 means the left button is pressed, 2 means the middle button is pressed, and 3 means the right button is pressed. (Note that some browsers don't generate mouse events for right-button clicks.)

In addition, the following fields of the jQuery Event object are jQuery-specific additions that you may sometimes find useful:

`data`
: If additional data was specified when the event handler was registered (see §19.4.4), it is made available to the handler as the value of this field

`handler`
: A reference to the event handler function currently being invoked

`result`
: The return value of the most recently invoked handler for this event, ignoring handlers that do not return a value

`originalEvent`
: A reference to the native Event object generated by the browser

19.4.4 Advanced Event Handler Registration

We've seen that jQuery defines quite a few simple methods for registering event handlers. Each of these simply invoke the single, more complex method `bind()` to bind a handler for a named event type to each of the elements in the jQuery object. Using `bind()` directly allows you to use advanced event registration features that are not available through the simpler methods.[4]

In its simplest form, `bind()` expects an event type string as its first argument and an event handler function as its second. The simple event registration methods use this form of `bind()`. The call `$('p').click(f)`, for example, is equivalent to:

```
$('p').bind('click', f);
```

`bind()` can also be invoked with three arguments. In this form, the event type is the first argument and the handler function is the third. You can pass any value between those two and jQuery will set the `data` property of the Event object to the value you specify

4. jQuery uses the term "bind" for event handler registration. ECMAScript 5, and a number of JavaScript frameworks, define a `bind()` method on functions (§8.7.4), and use the term for the association of functions with objects on which they are to be invoked. jQuery's version of the `Function.bind()` method is a utility function named `jQuery.proxy()`, and you can read about it in §19.7.

before it invokes the handler. It is sometimes useful to pass additional data to your handlers in this way without having to use closures.

There are other advanced features of `bind()` as well. If the first argument is a space-separated list of event types, then the handler function will be registered for each of the named event types. The call `$('a').hover(f)` (see §19.4.1), for example, is the same as:

```
$('a').bind('mouseenter mouseleave', f);
```

Another important feature of `bind()` is that it allows you to specify a namespace (or namespaces) for your event handlers when you register them. This allows you to define groups of handlers and comes in handy if you later want to trigger or deregister the handlers in a particular namespace. Handler namespaces are particularly useful for programmers who are writing libraries or modules of reusable jQuery code. Event namespaces look like CSS class selectors. To bind an event handler in a namespace, add a period and the namespace name to the event type string:

```
// Bind f as a mouseover handler in namespace "myMod" to all <a> elements
$('a').bind('mouseover.myMod', f);
```

You can even assign a handler to multiple namespaces like this:

```
// Bind f as a mouseout handler in namespaces "myMod" and "yourMod"
$('a').bind('mouseout.myMod.yourMod', f);
```

The final feature of `bind()` is that the first argument can be an object that maps event names to handler functions. To use the `hover()` method as an example again, the call `$('a').hover(f,g)` is the same as:

```
$('a').bind({mouseenter:f, mouseleave:g});
```

When you use this form of `bind()`, the property names in the object you pass can be space-separated strings of event types and can include namespaces. If you specify a second argument after the first object argument, that value is used as the data argument for each of the event bindings.

jQuery has another event handler registration method. The `one()` method is invoked and works just like `bind()` does, except that the event handler you register will automatically deregister itself after it is invoked. This means, as the method name implies, that event handlers registered with `one()` will never be triggered more than once.

One feature that `bind()` and `one()` do not have is the ability to register capturing event handlers as you can with `addEventListener()` (§17.2.3). IE (until IE9) does not support capturing handlers, and jQuery does not attempt to simulate that feature.

19.4.5 Deregistering Event Handlers

After registering an event handler with `bind()` (or with any of the simpler event registration methods), you can deregister it with `unbind()` to prevent it from being triggered by future events. (Note that `unbind()` only deregisters event handlers registered with `bind()` and related jQuery methods. It does not deregister handlers passed to `addEventListener()` or the IE method `attachEvent()`, and it does not remove handlers defined

by element attributes such as onclick and onmouseover.) With no arguments, unbind() deregisters all event handlers (for all event types) for all elements in the jQuery object:

```
$('*').unbind();  // Remove all jQuery event handlers from all elements!
```

With one string argument, all handlers for the named event type (or types, if the string names more than one) are unbound from all elements in the jQuery object:

```
// Unbind all mouseover and mouseout handlers on all <a> elements
$('a').unbind("mouseover mouseout");
```

This is a heavy-handed approach and should not be used in modular code because the user of your module might be using other modules that register their own handlers for the same event types on the same elements. If your module registered event handlers using namespaces, however, you can use this one-argument version of unbind() to deregister only the handlers in your namespace or namespaces:

```
// Unbind all mouseover and mouseout handlers in the "myMod" namespace
$('a').unbind("mouseover.myMod mouseout.myMod");
// Unbind handlers for any kind of event in the myMod namespace
$('a').unbind(".myMod");
// Unbind click handlers that are in both namespaces "ns1" and "ns2"
$('a').unbind("click.ns1.ns2");
```

If you want to be careful to unbind only event handlers you registered yourself and you did not use namespaces, you must retain a reference to the event handler functions and use the two-argument version of unbind(). In this form, the first argument is an event type string (without namespaces) and the second argument is a handler function:

```
$('#mybutton').unbind('click', myClickHandler);
```

When invoked this way, unbind() deregisters the specified event handler function for events of the specified type (or types) from all elements in the jQuery object. Note that event handlers can be unbound using this two-argument version of unbind() even when they were registered with an extra data value using the three-argument version of bind().

You can also pass a single object argument to unbind(). In this case, unbind() is invoked recursively for each property of the object. The property name is used as the event type string and the property value is used as the handler function:

```
$('a').unbind({  // Remove specific mouseover and mouseout handlers
  mouseover: mouseoverHandler,
  mouseout: mouseoutHandler
});
```

Finally, there is one more way that unbind() can be invoked. If you pass a jQuery Event object to it, it unbinds the event handler that that event was passed to. Calling unbind(ev) is equivalent to unbind(ev.type, ev.handler).

19.4.6 Triggering Events

The event handlers you register are automatically invoked when the user uses the mouse or keyboard or when other kinds of events occur. Sometimes, however, it is useful to be able to manually trigger events. The simple way to do this is to invoke one of the simple event registration methods (like `click()` or `mouseover()`) with no argument. Just as many jQuery methods serve as both getters and setters, these event methods register an event handler when invoked with an argument and trigger event handlers when invoked with no arguments. For example:

```
$("#my_form").submit();    // Act as if the user clicked the Submit button
```

The `submit()` method in the line above synthesizes an Event object and triggers any event handlers that have been registered for the submit event. If none of those event handlers return `false` or call the `preventDefault()` method of the Event object, the form will actually be submitted. Note that events that bubble will bubble even when triggered manually like this. This means that triggering an event on a selected set of elements may also trigger handlers on the ancestors of those elements.

It is important to note that jQuery's event triggering methods will trigger any handlers registered with jQuery's event registration methods, and they will also trigger handlers defined on HTML attributes or Element properties such as `onsubmit`. But you cannot manually trigger event handlers registered with `addEventListener()` or `attachEvent()` (those handlers will still be invoked when a real event occurs, however).

Also note that jQuery's event triggering mechanism is synchronous—there is no event queue involved. When you trigger an event, event handlers are invoked immediately, before the triggering method you called returns. If you trigger a click event and one of the triggered handlers triggers a submit event, all of the matching submit handlers are invoked before the next "click" handler is invoked.

Methods like `submit()` are convenient for binding and triggering events, but just as jQuery defines a more general `bind()` method, so too it defines a more general `trigger()` method. Normally you invoke `trigger()` with an event type string as the first argument and it triggers the handlers registered for events of that type on all elements in the jQuery object. So the `submit()` call above is equivalent to:

```
$("#my_form").trigger("submit");
```

Unlike the `bind()` and `unbind()` methods, you cannot specify more than one event type in this string. Like `bind()` and `unbind()`, however, you can specify event namespaces to trigger only the handlers defined in that namespace. If you want to trigger only event handlers that have *no* namespace, append an exclamation mark to the event type. Handlers registered through properties like `onclick` are considered to have no namespace:

```
$("button").trigger("click.ns1");  // Trigger click handlers in a namespace
$("button").trigger("click!");     // Trigger click handlers in no namespace
```

Instead of passing an event type string as the first argument to `trigger()`, you can also pass an Event object (or any object that has a `type` property). The `type` property will be used to determine what kind of handlers to trigger. If you specified a jQuery Event object, that object will be the one passed to the triggered handlers. If you specified a plain object, a new jQuery Event object will be created, and the properties of the object you passed will be added to it. This is an easy way to pass additional data to event handlers:

```
// The onclick handler of button1 triggers the same event on button2
$('#button1').click(function(e) { $('#button2').trigger(e); });

// Add an extra property to the event object when triggering an event
$('#button1').trigger({type:'click', synthetic:true});

// This handler tests that extra property to distinguish real from synthetic
$('#button1').click(function(e) { if (e.synthetic) {...}; });
```

Another way to pass additional data to event handlers when you trigger them manually is to use the second argument to `trigger()`. The value you pass as the second argument to `trigger()` will become the second argument to each of the event handlers that is triggered. If you pass an array as the second argument, each of its elements will be passed as arguments to the triggered handlers:

```
$('#button1').trigger("click", true);    // Pass a single extra argument
$('#button1').trigger("click", [x,y,z]); // Pass three extra arguments
```

Sometimes you may want to trigger all handlers for a given event type, regardless of which document element those handlers are bound to. You could select all elements with `$('*')` and then call `trigger()` on the result, but that would be very inefficient. Instead, to trigger an event globally, call the `jQuery.event.trigger()` utility function. This function takes the same arguments as the `trigger()` method and efficiently triggers event handlers for the specified event type throughout the document. Note that "global events" triggered in this way do not bubble, and only handlers registered using jQuery methods (not event handlers registered with DOM properties like `onclick`) are triggered with this technique.

After invoking event handlers, `trigger()` (and the convenience methods that call it) perform whatever default action is associated with the triggered event (assuming that the event handlers didn't return false or call `preventDefault()` on the event object). For example, if you trigger a submit event on a `<form>` element, `trigger()` will call the `submit()` method of that form, and if you trigger a focus event on an element, `trigger()` will call the `focus()` method of that element.

If you want to invoke event handlers without performing the default action, use `triggerHandler()` instead of `trigger()`. This method works just like `trigger()`, except that it first calls the `preventDefault()` and `cancelBubble()` methods of the Event object. This means that the synthetic event does not bubble or perform the default action associated with it.

19.4.7 Custom Events

jQuery's event management system is designed around the standard events like mouse clicks and key presses that web browsers generate. But it is not tied to those events, and you can use any string you want as an event type name. With `bind()` you can register handlers for this kind of "custom event" and with `trigger()` you can cause those handlers to be invoked.

This kind of indirect invocation of custom event handlers turns out to be quite useful for writing modular code and implementing a publish/subscribe model or the Observer pattern. Often when using custom events you may find it useful to trigger them globally with the `jQuery.event.trigger()` function instead of the `trigger()` method:

```
// When the user clicks the "logoff" button, broadcast a custom event
// to any interested observers that need to save their state and then
// navigate to the logoff page.
$("#logoff").click(function() {
    $.event.trigger("logoff");          // Broadcast an event
    window.location = "logoff.php";     // Navigate to a new page
});
```

We'll see in §19.6.4 that jQuery's Ajax methods broadcast custom events like this to notify interested listeners.

19.4.8 Live Events

The `bind()` method binds event handlers to specific document elements just as `addEventListener()` and `attachEvent()` (see Chapter 17) do. But web applications that use jQuery often dynamically create new elements. If we've used `bind()` to bind an event handler to all `<a>` elements in the document and then we create new document content with new `<a>` elements, those new elements will not have the same event handlers as the old ones and will not behave in the same way.

jQuery addresses this issue with "live events." To use live events, use the `delegate()` and `undelegate()` methods instead of `bind()` and `unbind()`. `delegate()` is usually invoked on `$(document)` and is passed a jQuery selector string, a jQuery event type string, and a jQuery event handler function. It registers an internal handler on the document or window (or on whatever elements are in the jQuery object). When an event of the specified type bubbles up to this internal handler, it determines whether the target of the event (the element that the event occurred on) matches the selector string. If so, it invokes the specified handler function. So to handle mouseover events on both old and newly created `<a>` elements, you might register a handler like this:

```
$(document).delegate("a", "mouseover", linkHandler);
```

Or you might use `bind()` in the static portions of your document and then use `delegate()` to handle the portions that change dynamically:

```
// Static event handlers for static links
$("a").bind("mouseover", linkHandler);
```

```
// Live event handlers for parts of the document that are dynamically updated
$(".dynamic").delegate("a", "mouseover", linkHandler);
```

Just as the `bind()` method has a three-argument version that allows you to specify the value of the `data` property of the event object, the `delegate()` method has a four-argument version that allows the same thing. To use this version, pass the data value as the third argument and the handler function as the fourth.

It is important to understand that live events depend on event bubbling. By the time an event bubbles up to the document object, it may have already been passed to a number of static event handlers. And if any of those handlers called the `cancelBubble()` method of the Event object, the live event handler will never be invoked.

jQuery defines a method named `live()` that can also be used to register live events. `live()` is a little harder to understand than `delegate()`, but it has the same two- or three-argument signature as `bind()` and is more commonly used in practice. The two calls to `delegate()` shown above could also be written as follows using `live()`:

```
$("a").live("mouseover", linkHandler);
$("a", $(".dynamic")).live("mouseover", linkHandler);
```

When the `live()` method is invoked on a jQuery object, the elements in that object are not actually used. What matters instead is the selector string and the context object (the first and second arguments to `$()`) that were used to create the jQuery object. jQuery objects make these values available through their `context` and `selector` properties (see §19.1.2). Normally, you invoke `$()` with only one argument and the context is the current document. So for a jQuery object `x`, the following two lines of code do the same thing:

```
x.live(type,handler);
$(x.context).delegate(x.selector, type, handler);
```

To deregister live event handlers, use `die()` or `undelegate()`. `die()` can be invoked with one or two arguments. With one event type argument, it removes all live event handlers that match the selector and the event type. And with an event type and handler function argument, it removes only the one specified handler. Some examples:

```
$('a').die('mouseover');  // Remove all live handlers for mouseover on <a> elts
$('a').die('mouseover', linkHandler); // Remove just one specific live handler
```

`undelegate()` is like `die()` but more explicitly separates the context (the elements on which the internal event handlers are registered) and the selector string. The calls to `die()` above could instead be written like this:

```
$(document).undelegate('a'); // Remove all live handlers for <a> elements
$(document).undelegate('a', 'mouseover'); // Remove live mouseover handlers
$(document).undelegate('a', 'mouseover', linkHandler); // One specific handler
```

Finally, `undelegate()` can also be called with no arguments at all. In this case, it deregisters all live event handlers that are delegated from the selected elements.

19.5 Animated Effects

Chapter 16 showed how to script the CSS styles of document elements. By setting the CSS `visibility` property, for example, you can make elements appear and disappear. §16.3.1 went on to demonstrate how CSS scripting can be used to produce animated visual effects. Instead of just making an element disappear, for example, we might reduce the value of its `opacity` property over the period of a half second so that it quickly fades away instead of just blinking out of existence. This kind of animated visual effect creates a more pleasing experience for users, and jQuery makes them easy.

jQuery defines simple methods such as `fadeIn()` and `fadeOut()` for basic visual effects. In addition to simple effects methods, it defines an `animate()` method for producing more complex custom animations. The subsections below explain both the simple effects methods and the more general `animate()` method. First, however, we'll describe some general features of jQuery's animation framework.

Every animation has a duration that specifies how long the effect should last for. You specify this as a number of milliseconds or by using a string. The string "fast" means 200ms. The string "slow" means 600ms. If you specify a duration string that jQuery does not recognize, you'll get a default duration of 400ms. You can define new duration names by adding new string-to-number mappings to `jQuery.fx.speeds`:

```
jQuery.fx.speeds["medium-fast"] = 300;
jQuery.fx.speeds["medium-slow"] = 500;
```

jQuery's effect methods usually take effect duration as an optional first argument. If you omit the duration argument, you usually get the default 400ms. Some methods, however, produce an instant nonanimated effect when you omit the duration:

```
$("#message").fadeIn();        // Fade an element in over 400ms
$("#message").fadeOut("fast"); // Fade it out over 200ms
```

Disabling Animations

Animated visual effects have become the norm on many websites, but not all users like them: some users find them distracting and others feel they cause motion sickness. Disabled users may find that animations interfere with assistive technology like screen readers and users on old hardware may feel that they require too much processing power. As a courtesy to your users, you should generally keep your animations simple and understated and also provide an option to disable them completely. jQuery makes it easy to globally disable all effects: simply set `jQuery.fx.off` to `true`. This has the effect of changing the duration of every animation to 0ms, making them behave as instantaneous, nonanimated changes.

To allow end users to disable effects, you might use code like this in your scripts:

```
$(".stopmoving").click(function() { jQuery.fx.off = true; });
```

Then, if the web designer includes an element with class "stopmoving" on the page, the user can click it to disable animations.

jQuery's effects are asynchronous. When you call an animation method like `fadeIn()`, it returns right away and the animation is performed "in the background." Because animation methods return before the animation is complete, the second argument (also optional) to many of jQuery's effect methods is a function that will be invoked when the effect is complete. The function is not passed any arguments, but the `this` value is set to the document element that was animated. The callback function is invoked once for each selected element:

```
// Quickly fade in an element, and when it is visible, display some text in it.
$("#message").fadeIn("fast", function() { $(this).text("Hello World"); });
```

Passing a callback function to an effect method allows you to perform actions at the end of an effect. Note, however, that this is not necessary when you simply want to perform multiple effects in sequence. By default, jQuery's animations are queued (§19.5.2.2 shows how to override this default). If you call an animation method on an element that is already being animated, the new animation does not begin right away but is deferred until the current animation ends. For example, you can make an element blink before fading in permanently:

```
$("#blinker").fadeIn(100).fadeOut(100).fadeIn(100).fadeOut(100).fadeIn();
```

jQuery's effect methods are declared to accept optional duration and callback arguments. It is also possible to invoke these methods with an object whose properties specify animation options:

```
// Pass duration and callback as object properties instead of arguments
$("#message").fadeIn({
    duration: "fast",
    complete: function() { $(this).text("Hello World"); }
});
```

Passing an object of animation objects is most commonly done with the general `animate()` method, but it is also possible for the simpler effects methods. Using an options object allows you to set other advanced options to control queuing and easing, for example. The available options are explained in §19.5.2.2.

19.5.1 Simple Effects

jQuery defines nine simple effects methods to hide and show elements. They can be divided into three groups based on the kind of effect they perform:

`fadeIn()`, `fadeOut()`, `fadeTo()`
: These are the simplest effects: `fadeIn()` and `fadeOut()` simply animate the CSS `opacity` property to show or hide an element. Both accept optional duration and callback arguments. `fadeTo()` is slightly different: it expects a target opacity argument and animates the change from the element's current opacity to this target. For the `fadeTo()` method, the duration (or options object) is required as the first argument and the target opacity is required as the second argument. The callback function is an optional third argument.

`show()`, `hide()`, `toggle()`

The `fadeOut()` method listed above makes elements invisible but retains space for them in the document layout. The `hide()` method, by contrast, removes the elements from the layout as if the CSS `display` property was set to `none`. When invoked with no arguments, `hide()` and `show()` simply hide or show the selected elements immediately. With a duration (or options object) argument, however, they animate the hiding or showing process. `hide()` shrinks an element's `width` and `height` to 0 at the same time that it reduces the element's `opacity` to 0. `show()` reverses the process.

`toggle()` changes the visibility state of the elements it is invoked on: if they are hidden, it calls `show()`, and if they are visible, it calls `hide()`. As with `show()` and `hide()`, you must pass a duration or options object to `toggle()` to get an animated effect. Passing `true` to `toggle()` is the same as calling `show()` with no arguments and passing `false` is the same as calling `hide()` with no arguments. Note also that if you pass two or more function arguments to `toggle()` it registers event handlers, as described in §19.4.1.

`slideDown()`, `slideUp()`, `slideToggle()`

`slideUp()` hides the elements in the jQuery object by animating their height to 0 and then setting the CSS `display` property to "none". `slideDown()` reverses the process to make a hidden element visible again. `slideToggle()` toggles the visibility of an item using a slide up or slide down animation. Each of the three methods accepts the optional duration and callback arguments (or the options object argument).

Here is an example that invokes methods from each of these groups. Keep in mind that jQuery's animations are queued by default, so these animations are performed one after the other:

```
// Fade all images out, then show them, then slide up, then slide down
$("img").fadeOut().show(300).slideUp().slideToggle();
```

Various jQuery plug-ins (see §19.9) add additional effect methods to the library. The jQuery UI library (§19.10) includes a particularly comprehensive set of effects.

19.5.2 Custom Animations

You can use the `animate()` method to produce more general animated effects than are available with the simple effects methods. The first argument to `animate()` specifies what to animate and the remaining arguments specify how to animate it. The first argument is required: it must be an object whose properties specify CSS attributes and their target values. `animate()` animates the CSS properties of each element from its current value to the specified target value. So, for example, the `slideUp()` effect described above can also be performed with code like this:

```
// Shrink the height of all images to 0
$("img").animate({ height: 0 });
```

As an optional second argument, you can pass an options object to `animate()`:

```
$("#sprite").animate({
    opacity: .25,                 // Animate opacity to .25
    font-size: 10                 // Animate font size to 10 pixels
}, {
    duration: 500,                // Animation lasts 1/2 second
    complete: function() {        // Call this function when done
        this.text("Goodbye");     // Change element text.
    }
});
```

Instead of passing an options object as the second argument, `animate()` also allows you to specify three of the most commonly used options as arguments. You can pass the duration (as a number or string) as the second argument. You can specify the name of an easing function as the third argument. (Easing functions will be explained shortly.) And you can specify a callback function as the fourth argument.

In the most general case, `animate()` accepts two object arguments. The first specifies what to animate and the second specifies how to animate it. To fully understand how to perform animations with jQuery, there are additional details about both objects that you must be aware of.

19.5.2.1 The animation properties object

The first argument to `animate()` must be an object. The property names for this object must be CSS attribute names, and the values of those properties must be the target values toward which the animation will move. Only numeric properties can be animated: it is not possible to animate colors, fonts, or enumerated properties such as `display`. If the value of a property is a number, pixels are assumed. If the value is a string, you may specify units. If you omit the units, pixels are again assumed. To specify relative values, prefix the value string with "+=" to increase the value or with "-=" to decrease the value. For example:

```
$("p").animate({
    "margin-left": "+=.5in",  // Increase paragraph indent
    opacity: "-=.1"           // And decrease their opacity
});
```

Note the use of the quotes around the property name "margin-left" in the object literal above. The hyphen in this property name means that it is not a legal JavaScript identifier, so it must be quoted here. jQuery also allows you to use the mixed-case alternative `marginLeft`, of course.

In addition to numeric values (with optional units and "+=" and "-=" prefixes), there are three other values that can be used in jQuery animation objects. The value "hide" will save the current state of the property and then animate that property toward 0. The value "show" will animate a CSS property toward its saved value. If an animation uses "show", jQuery will call the `show()` method when the animation completes. And if an animation uses "hide", jQuery will call `hide()` when the animation completes.

You can also use the value "toggle" to perform either a show or a hide, depending on the current setting of the attribute. You can produce a "slideRight" effect (like the `slideUp()` method, but animating element width) like this:

```
$("img").animate({
    width: "hide",
    borderLeft: "hide",
    borderRight: "hide",
    paddingLeft: "hide",
    paddingRight: "hide"
});
```

Replace the property values with "show" or "toggle" to produce sideways slide effects analogous to `slideDown()` and `slideToggle()`.

19.5.2.2 The animation options object

The second argument to `animate()` is an optional object that holds options that specify how the animation is performed. You've already seen two of the most important options. The `duration` property specifies the length of the animation in milliseconds, or as the string "fast" or "slow", or any name you've defined in `jQuery.fx.speeds`.

Another option you've already seen is the `complete` property: it specifies a function that will be called when the animation is complete. A similar property, `step`, specifies a function that is called for each step or frame of the animation. The element being animated is the `this` value, and the current value of the property being animated is passed as the first argument.

The `queue` property of the options object specifies whether the animation should be queued—whether it should be deferred until any pending animations have completed. All animations are queued by default. Set the `queue` property to `false` to disable queuing. Unqueued animations start immediately. Subsequent queued animations are not deferred for unqueued animations. Consider the following code:

```
$("img").fadeIn(500)
        .animate({"width":"+=100"}, {queue:false, duration:1000})
        .fadeOut(500);
```

The `fadeIn()` and `fadeOut()` effects are queued, but the call to `animate()` (which animates the `width` property for 1000ms) is not queued. The width animation begins at the same time the `fadeIn()` effect begins. The `fadeOut()` effect begins as soon as the `fadeIn()` effect ends: it does not wait for the width animation to complete.

Easing Functions

The straightforward but naive way to perform animations involves a linear mapping between time and the value being animated. If we are 100ms into a 400ms animation, for example, the animation is 25 percent done. If we are animating the `opacity` property from 1.0 to 0.0 (for a `fadeOut()` call, perhaps) in a linear animation, the opacity should be at 0.75 at this point of the animation. It turns out, however, that visual effects are more pleasing if they are not linear. So jQuery interposes an "easing function" that

maps from a time-based completion percentage to the desired effect percentage. jQuery calls the easing function with a time-based value between 0 and 1. It returns another value between 0 and 1 and jQuery computes the value of the CSS property based on this computed value. Generally, easing functions are expected to return 0 when passed the value 0 and 1 when passed the value 1, of course, but they can be nonlinear between those two values and this nonlinearity makes the animation appear to accelerate and decelerate.

jQuery's default easing function is a sinusoid: it starts off slow, then speeds up, then slows down again to "ease" the animation to its final value. jQuery gives its easing functions names. The default one is named "swing", and jQuery also implements a linear function named "linear". You can add your own easing functions to the `jQuery.easing` object:

```
jQuery.easing["squareroot"] = Math.sqrt;
```

The jQuery UI library and a plug-in known simply as "the jQuery Easing Plugin" define a comprehensive set of additional easing functions.

The remaining animation options involve easing functions. The `easing` property of the options object specifies the name of an easing function. By default, jQuery uses the sinusoidal function it calls "swing". If you want your animations to be linear, use an options object like this:

```
$("img").animate({"width":"+=100"}, {duration: 500, easing:"linear"});
```

Recall that the `duration`, `easing`, and `complete` options can also be specified by arguments to `animate()` instead of passing an options object. So the animation above could also be written like this:

```
$("img").animate({"width":"+=100"}, 500, "linear");
```

Finally, jQuery's animation framework even allows you to specify different easing functions for the different CSS properties you want to animate. There are two different ways to achieve this, demonstrated by the code below:

```
// Hide images, as with the hide() method, but animate the image size linearly
// while the opacity is being animated with the default "swing" easing function

// One way to do it:
// Use the specialEasing option to specify custom easing functions
$("img").animate({ width:"hide", height:"hide", opacity:"hide" },
                 { specialEasing: {  width: "linear", height: "linear" }});

// Another way to do it:
// Pass [target value, easing function] arrays in the first object argument.
$("img").animate({
    width: ["hide", "linear"], height: ["hide", "linear"], opacity:"hide"
});
```

19.5.3 Canceling, Delaying, and Queuing Effects

jQuery defines a few more animation and queue-related methods that you should know about. The `stop()` method is first: it stops any currently executing animations on the selected elements. `stop()` accepts two optional boolean arguments. If the first argument is `true`, the animation queue will be cleared for the selected elements: this will cancel any pending animations as well as stopping the current one. The default is `false`: if this argument is omitted, queued animations are not canceled. The second argument specifies whether the CSS properties being animated should be left as they are currently or whether they should be set to their final target values. Passing `true` sets them to their final values. Passing false (or omitting the argument) leaves them at whatever their current value is.

When animations are triggered by user events, you may want to cancel any current or queued animations before beginning a new one. For example:

```
// Images become opaque when the mouse moves over them.
// Be careful that we don't keep queueing up animations on mouse events!
$("img").bind({
    mouseover: function() { $(this).stop().fadeTo(300, 1.0); },
    mouseout: function() { $(this).stop().fadeTo(300, 0.5); }
});
```

The second animation-related method we'll cover here is `delay()`. This simply adds a timed delay to the animation queue: pass a duration in milliseconds (or a duration string) as the first argument and a queue name as the optional second argument (the second argument is not normally needed: we'll talk about queue names below). You can use `delay()` in compound animations like this one:

```
// Quickly fade out halfway, wait, then slide up
$("img").fadeTo(100, 0.5).delay(200).slideUp();
```

In the `stop()` method example above, we used mouseover and mouseout events to animate the opacity of images. We can refine that example by adding a short delay before beginning the animation. That way, if the mouse quickly moves through an image without stopping, no distracting animation occurs:

```
$("img").bind({
    mouseover: function() { $(this).stop(true).delay(100).fadeTo(300, 1.0); },
    mouseout: function() { $(this).stop(true).fadeTo(300, 0.5); }
});
```

The final animation-related methods are ones that give low-level access to the jQuery queuing mechanism. jQuery queues are lists of functions to be executed in sequence. Each queue is associated with a document element (or the Document or Window objects) and each element's queues are independent of the queues of other elements. You can add a new function to the queue with the `queue()` method. When your function reaches the head of the queue, it will be automatically dequeued and invoked. When your function is invoked, the `this` value is the element with which it is associated. Your function will be passed a function as its single argument. When your function has completed its operation, it must invoke the function that was passed to it. This runs

the next operation in the queue, and if you don't call the function, the queue will stall and queued functions will never get invoked.

We've seen that you can pass a callback function to jQuery's effect methods in order to perform some kind of action after the effect completes. You can achieve the same effect by queuing up your function:

```
// Fade an element in, wait, set some text in it, and animate its border
$("#message").fadeIn().delay(200).queue(function(next) {
    $(this).text("Hello  World");      // Display some text
    next();                            // Run the next item on the queue
}).animate({borderWidth: "+=10px;"}); // Grow its border
```

The function argument to queued functions is a new feature in jQuery 1.4. In code written for earlier versions of the library, queued functions dequeue the next function "manually" by calling the `dequeue()` method:

```
$(this).dequeue();  // Instead of next()
```

If there is nothing in the queue, calling `dequeue()` does nothing. Otherwise, it removes a function from the head of the queue and invokes it, setting the `this` value and passing the function described above.

There are a few more heavy-handed ways to manipulate the queue as well. `clearQueue()` clears the queue. Passing an array of functions to `queue()` instead of a single function replaces the queue with the new array of functions. And calling `queue()` with neither a function nor an array of functions returns the current queue as an array. Also, jQuery defines versions of the `queue()` and `dequeue()` methods as utility functions. If you want to add the function `f` to the queue for an element `e`, you can use either the method or the function:

```
$(e).queue(f);      // Create a jQuery object holding e, and call queue method
jQuery.queue(e,f);  // Just call the jQuery.queue() utility function
```

Finally, note that `queue()`, `dequeue()`, and `clearQueue()` all take an optional queue name as their first argument. jQuery's effects and animation methods use a queue named "fx", and this is the queue that is used if you do not specify a queue name. jQuery's queue mechanism is useful whenever you need to perform asynchronous operations sequentially: instead of passing a callback function to each asynchronous operation so that it can trigger the next function in the sequence, you can use a queue to manage the sequence instead. Simply pass a queue name other than "fx", and remember that queued functions do not execute automatically. You must explicitly call `dequeue()` to run the first one, and each operation must dequeue the next one when it finishes.

19.6 Ajax with jQuery

Ajax is the popular name for web application programming techniques that use HTTP scripting (see Chapter 18) to load data as needed, without causing page refreshes. Because Ajax techniques are so useful in modern web apps, jQuery includes Ajax utilities to simplify them. jQuery defines one high-level utility method and four high-level utility

functions. These high-level utilities are all based on a single powerful low-level function, `jQuery.ajax()`. The subsections that follow describe the high-level utilities first, and then cover the `jQuery.ajax()` function in detail. In order to fully understand the operation of the high-level utilities, you'll need to understand `jQuery.ajax()`, even if you never need to use it explicitly.

19.6.1 The load() Method

The `load()` method is the simplest of all jQuery utilities: pass it a URL and it will asynchronously load the content of that URL and then insert that content into each of the selected elements, replacing any content that is already there. For example:

```
// Load and display the latest status report every 60 seconds
setInterval(function() { $("#stats").load("status_report.html"); }, 60000);
```

We also saw the `load()` method in §19.4.1, where it was used to register a handler for load events. If the first argument to this method is a function instead of a string, it behaves as an event handler registration method instead of as an Ajax method.

If you only want to display a portion of the loaded document, add a space to the URL and follow it with a jQuery selector. When the URL has loaded, the selector you specified will be used to select the portions of the loaded HTML to be displayed:

```
// Load and display the temperature section of the weather report
$('#temp').load("weather_report.html #temperature");
```

Note that the selector at the end of this URL looks very much like a fragment identifier (the hash portion of the URL described in §14.2). The space is required, however, if you want jQuery to insert only the selected portion (or portions) of the loaded document.

The `load()` method accepts two optional arguments in addition to the required URL. The first is data to append to the URL or to send along with the request. If you pass a string, it is appended to the URL (after a ? or & as needed). If you pass an object, it is converted to a string of ampersand-separated name=value pairs and sent along with the request. (The details of object-to-string conversion for Ajax are in the sidebar of §19.6.2.2). The `load()` method normally makes an HTTP GET request, but if you pass a data object, it makes a POST request instead. Here are two examples:

```
// Load the weather report for a specified zipcode
$('#temp').load("us_weather_report.html", "zipcode=02134");

// Here we use an object as data instead and specify degrees Fahrenheit
$('#temp').load("us_weather_report.html", { zipcode:02134, units:'F' });
```

Another optional argument to `load()` is a callback function that will be invoked when the Ajax request completes successfully or unsuccessfully and (in the case of success) after the URL has been loaded and inserted into the selected elements. If you don't specify any data, you can pass this callback function as the second argument. Otherwise, it should be the third argument. The callback you specify will be invoked once as a method of each of the elements in the jQuery object and will be passed three

arguments to each invocation: the complete text of the loaded URL, a status code string, and the XMLHttpRequest object that was used to load the URL. The status argument is a jQuery status code, not an HTTP status code, and it will be a string like "success", "error", or "timeout".

jQuery's Ajax Status Codes

All of jQuery's Ajax utilities, including the `load()` method, invoke callback functions to provide asynchronous notification of the success or failure of the request. The second argument to these callbacks is a string with one of the following values:

"success"
: Indicates that the request completed successfully.

"notmodified"
: This code indicates that the request completed normally but that the server sent an HTTP 304 "Not Modified" response, indicating that the requested URL has not changed since it was last requested. This status code only occurs if you set the `ifModified` option to true. (See §19.6.3.1.) jQuery 1.4 considers a "notmodified" status code a success, but earlier versions consider it an error.

"error"
: Indicates that the request did not complete successfully, because of an HTTP error of some sort. For more details, you can check the HTTP status code in the XMLHttpRequest object, which is also passed to each callback.

"timeout"
: If an Ajax request does not complete within the timeout interval that you select, the error callback is invoked with this status code. By default, jQuery Ajax requests do not time out; you'll only see this status code if you set the `timeout` option (§19.6.3.1).

"parsererror"
: This status code indicates that the HTTP request completed successfully, but that jQuery could not parse it in the way it expected to. This status code occurs if the server sends a malformed XML document or malformed JSON text, for example. Note that this status code is "parsererror", not "parseerror".

19.6.2 Ajax Utility Functions

The other high-level jQuery Ajax utilities are functions, not methods, and are invoked directly through `jQuery` or `$`, not on a jQuery object. `jQuery.getScript()` loads and executes files of JavaScript code. `jQuery.getJSON()` loads a URL, parses it as JSON, and passes the resulting object to the callback you specify. Both of these functions call `jQuery.get()`, which is a more general-purpose URL fetching function. Finally, `jQuery.post()` works just like `jQuery.get()`, but it performs an HTTP POST request instead of a GET. Like the `load()` method, all of these functions are asynchronous: they return to their caller before anything is loaded and notify you of the results by invoking a callback function you specify.

19.6.2.1 jQuery.getScript()

The `jQuery.getScript()` function takes the URL of a file of JavaScript code as its first argument. It asynchronously loads and then executes that code in the global scope. It can work for both same-origin and cross-origin scripts:

```
// Dynamically load a script from some other server
jQuery.getScript("http://example.com/js/widget.js");
```

You can pass a callback function as the second argument, and if you do, jQuery will invoke that function once after the code has been loaded and executed.

```
// Load a library and use it once it loads
jQuery.getScript("js/jquery.my_plugin.js", function() {
    $('div').my_plugin();  // Use the library we loaded
});
```

`jQuery.getScript()` normally uses an XMLHttpRequest object to fetch the text of the script to be executed. But for cross-domain requests (when the script is served by a server other than the one that served the current document), jQuery loads the script with a `<script>` element (see §18.2). In the same-origin case, the first argument to your callback is the text of the script, the second argument is the status code "success", and the third argument is the XMLHttpRequest object used to fetch the text of the script. The return value of `jQuery.getScript()` is also the XMLHttpRequest object in this case. For cross-origin requests, there is no XMLHttpRequest object, and the text of the script is not captured. In this case, the callback function is called with its first and third arguments `undefined`, and the return value of `jQuery.getScript()` is also `undefined`.

The callback function you pass to `jQuery.getScript()` is invoked only if the request completes successfully. If you need to be notified of errors as well as success, you'll need to use the lower-level `jQuery.ajax()` function. The same is true of the three other utility functions described in this section.

19.6.2.2 jQuery.getJSON()

`jQuery.getJSON()` is like `jQuery.getScript()`: it fetches text and then processes it specially before invoking the callback you specify. Instead of executing the text as a script, `jQuery.getJSON()` parses it as JSON (using the `jQuery.parseJSON()` function: see §19.7). `jQuery.getJSON()` is only useful when passed a callback argument. If the URL is loaded successfully and if its content is successfully parsed as JSON, the resulting object will be passed as the first argument to the callback function. As with `jQuery.getScript()`, the second and third arguments to the callback are the status code "success" and the XMLHttpRequest object:

```
// Suppose data.json contains the text: '{"x":1,"y":2}'
jQuery.getJSON("data.json", function(data) {
    // Now data is the object {x:1, y:2}
});
```

Unlike `jQuery.getScript()`, `jQuery.getJSON()` accepts an optional data argument like the one passed to the `load()` method. If you pass data to `jQuery.getJSON()`, it must be

the second argument and the callback must be the third. If you do not pass any data, the callback may be the second argument. If the data is a string, it is appended to the URL, following a ? or &. If the data is an object, it is converted to a string (see the sidebar) and then appended to the URL.

Passing Data to jQuery's Ajax Utilities

Most of jQuery's Ajax methods accept an argument (or an option) that specifies data to send to the server along with the URL. Usually this data takes the form of URL-encoded `name=value` pairs separated from each other by ampersands. (This data format is known by the MIME type "application/x-www-form-urlencoded". You can think of it as an analog of JSON: a format for converting simple JavaScript objects to and from strings.) For HTTP GET requests, this string of data is appended to the request URL. For POST requests, it is sent as the request body, after all the HTTP headers are sent.

One way to obtain a string of data in this format is to call the `serialize()` method of a jQuery object that contains forms or form elements. To submit an HTML form using the `load()` method, for example, you might use code like this:

```
$("#submit_button").click(function(event) {
    $(this.form).load(              // Replace the form by loading...
        this.form.action,           // the form url
        $(this.form).serialize());  // with the form data appended to it
    event.preventDefault();         // Don't do the default form submission
    this.disabled = "disabled";     // Prevent multiple submissions
});
```

If you set the data argument (or option) of a jQuery Ajax function to an object rather than a string, jQuery will normally (with an exception described below) convert that object to a string for you by calling `jQuery.param()`. This utility function treats object properties as name=value pairs and converts the object `{x:1,y:"hello"}`, for example, to the string `"x=1&y=hello"`.

In jQuery 1.4, `jQuery.param()` handles more complicated JavaScript objects. If the value of an object property is an array, each element of that array will have its own name/value pair in the resulting string and the property name will have square brackets appended. And if the value of a property is an object, the property names of that nested object are placed in square brackets and appended to the outer property name. For example:

```
$.param({a:[1,2,3]})        // Returns "a[]=1&a[]=2&a[]=3"
$.param({o:{x:1,y:true}})   // Returns "o[x]=1&o[y]=true"
$.param({o:{x:{y:[1,2]}}}) // Returns "o[x][y][]=1&o[x][y][]=2"
```

For backward compatibility with jQuery 1.3 and before, you can pass `true` as the second argument to `jQuery.param()` or set the `traditional` option to `true`. This will prevent the advanced serialization of properties whose values are arrays or objects.

Occasionally, you may want to pass a Document (or some other object that should not be automatically converted) as the body of a POST request. In this case you can set the `contentType` option to specify the type of your data and set the `processData` option to `false`, to prevent jQuery from passing your data object to `jQuery.param()`.

If either the URL or data string passed to `jQuery.getJSON()` contains the string "=?" at the end of the string or before an ampersand, it is taken to specify a JSONP request. (See §18.2 for an explanation of JSONP.) jQuery will replace the question mark with the name of a callback function it creates, and `jQuery.getJSON()` will then behave as if a script is being requested rather than a JSON object. This does not work for static JSON data files: it only works with server-side scripts that support JSONP. Because JSONP requests are handled as scripts, however, it does mean that JSON-formatted data can be requested cross-domain.

19.6.2.3 jQuery.get() and jQuery.post()

`jQuery.get()` and `jQuery.post()` fetch the content of the specified URL, passing the specified data, if any, and pass the result to the specified callback. `jQuery.get()` does this using an HTTP GET request and `jQuery.post()` uses a POST request, but otherwise these two utility functions are the same. These two methods take the same three arguments that `jQuery.getJSON()` does: a required URL, an optional data string or object, and a technically optional but almost always used callback function. The callback function is invoked with the returned data as its first argument, the string "success" as its second, and the XMLHttpRequest (if there was one) as its third:

```
// Request text from the server and display it in an alert dialog
jQuery.get("debug.txt", alert);
```

In addition to the three arguments described above, these two methods accept a fourth optional argument (passed as the third argument if the data is omitted) that specifies the type of the data being requested. This fourth argument affects the way the data is processed before being passed to your callback. The `load()` method uses the type "html", `jQuery.getScript()` uses the type "script", and `jQuery.getJSON()` uses the type "json". `jQuery.get()` and `jQuery.post()` are more flexible than those special-purpose utilities, however, and you can specify any of these types. The legal values for this argument, as well as jQuery's behavior when you omit the argument, are explained in the sidebar.

jQuery's Ajax Data Types

You can pass any of the following six types as an argument to `jQuery.get()` or `jQuery.post()`. Additionally, as we'll see below, you can pass one of these types to `jQuery.ajax()` using the `dataType` option:

`"text"`
: Returns the server's response as plain text with no processing.

`"html"`
: This type works just like "text": the response is plain text. The `load()` method uses this type and inserts the returned text into the document itself.

`"xml"`
: The URL is assumed to refer to XML-formatted data, and jQuery uses the `responseXML` property of the XMLHttpRequest object instead of the `responseText` property.

The value passed to the callback is a Document object representing the XML document instead of a string holding the document text.

`"script"`
: The URL is assumed to reference a file of JavaScript, and the returned text is executed as a script before being passed to the callback. `jQuery.getScript()` uses this type. When the type is "script", jQuery can handle cross-domain requests using a `<script>` element instead of an XMLHttpRequest object.

`"json"`
: The URL is assumed to reference a file of JSON-formatted data. The value passed to the callback is the object obtained by parsing the URL contents with `jQuery.parseJSON()` (§19.7). `jQuery.getJSON()` uses this type. If the type is "json" and the URL or data string contains `"=?"`, the type is converted to "jsonp".

`"jsonp"`
: The URL is assumed to refer to a server-side script that supports the JSONP protocol for passing JSON-formatted data as an argument to a client-specified function. (See §18.2 for more on JSONP.) This type passes the parsed object to the callback function. Because JSONP requests can be made with `<script>` elements, this type can be used to make cross-domain requests, like the "script" type can. When you use this type, your URL or data string should typically include a parameter like `"&jsonp=?"` or `"&callback=?"`. jQuery will replace the question mark with the name of an automatically generated callback function. (But see the `jsonp` and `jsonpCallback` options in §19.6.3.3 for alternatives.)

If you do not specify one of these types when you invoke a `jQuery.get()`, `jQuery.post()`, or `jQuery.ajax()`, jQuery examines the Content-Type header of the HTTP response. If that header includes the string "xml", an XML document is passed to the callback. Otherwise, if the header includes the string "json", the data is parsed as JSON and the parsed object is passed to the callback. Otherwise, if the header includes the string "javascript", the data is executed as a script. Otherwise, the data is treated as plain text.

19.6.3 The jQuery.ajax() Function

All of jQuery's Ajax utilities end up invoking `jQuery.ajax()`—the most complicated function in the entire library. `jQuery.ajax()` accepts just a single argument: an options object whose properties specify many details about how the Ajax request is to be performed. A call to `jQuery.getScript(url,callback)`, for example, is equivalent to this `jQuery.ajax()` invocation:

```
jQuery.ajax({
    type: "GET",          // The HTTP request method.
    url: url,             // The URL of the data to fetch.
    data: null,           // Don't add any data to the URL.
    dataType: "script",   // Execute the response as a script once we get it.
    success: callback     // Call this function when done.
});
```

You can set these five fundamental options with `jQuery.get()` and `jQuery.post()`. `jQuery.ajax()` supports quite a few other options, however, if you invoke it directly. The options (including the basic five shown above) are explained in detail below.

Before we dive into the options, note that you can set defaults for any of these options by passing an options object to `jQuery.ajaxSetup()`:

```
jQuery.ajaxSetup({
    timeout: 2000, // Abort all Ajax requests after 2 seconds
    cache: false   // Defeat browser cache by adding a timestamp to the URL
});
```

After running the code above, the specified `timeout` and `cache` options will be used for all Ajax requests (including high-level ones like `jQuery.get()` and the `load()` method) that do not specify their own values for these options.

While reading about jQuery's many options and callbacks in the sections that follow, you may find it helpful to refer to the sidebars about jQuery's Ajax status code and data type strings in §19.6.1 and §19.6.2.3.

Ajax in jQuery 1.5

jQuery 1.5, which was released as this book was going to press, features a rewritten Ajax module, with several convenient new features. The most important is that `jQuery.ajax()` and all of the Ajax utility functions described earlier now return a jqXHR object. This object simulates the XMLHttpRequest API, even for requests (like those made with `$.getScript()`) that do not use an XMLHttpRequest object. Furthermore, the jqXHR object defines `success()`, `error()` methods that you can use to register callback functions to be invoked when the request succeeds or fails. So instead of passing a callback to `jQuery.get()`, for example, you might instead pass it to the `success()` method of the jqXHR object returned by that utility function:

```
jQuery.get("data.txt")
    .success(function(data) { console.log("Got", data); })
    .success(function(data) { process(data); });
```

19.6.3.1 Common Options

The most commonly used `jQuery.ajax()` options are the following:

`type`

Specifies the HTTP request method. The default is "GET". "POST" is another commonly used value. You can specify other HTTP request methods, such as "DELETE" and "PUT", but not all browsers support them. Note that this option is misleadingly named: it has nothing to do with the data type of the request or response, and "method" would be a better name.

`url`

The URL to be fetched. For GET requests, the `data` option will be appended to this URL. jQuery may add parameters to the URL for JSONP requests and when the `cache` option is `false`.

`data`

Data to be appended to the URL (for GET requests) or sent in the body of the request (for POST requests). This can be a string or an object. Objects are usually converted to strings as described in the sidebar of §19.6.2.2, but see the `processData` option for an exception.

`dataType`

Specifies the type of data expected in the response, and the way that that data should be processed by jQuery. Legal values are "text", "html", "script", "json", "jsonp", and "xml". The meanings of these values were explained in the sidebar in §19.6.2.3. This option has no default value. When left unspecified, jQuery examines the `Content-Type` header of the response to determine what to do with the returned data.

`contentType`

Specifies the HTTP `Content-Type` header for the request. The default is "application/x-www-form-urlencoded", which is the normal value used by HTML forms and most server-side scripts. If you have set `type` to "POST" and want to send plain text or an XML document as the request body, you also need to set this option.

`timeout`

A timeout, in milliseconds. If this option is set and the request has not completed within the specified timeout, the request will be aborted and the `error` callback will be called with status "timeout". The default timeout is 0, which means that requests continue until they complete and are never aborted.

`cache`

For GET requests, if this option is set to `false`, jQuery will add a `_=` parameter to the URL or replace an existing parameter with that name. The value of this parameter is set to the current time (in millisecond format). This defeats browser-based caching, since the URL will be different each time the request is made.

`ifModified`

When this option is set to `true`, jQuery records the values of the `Last-Modified` and `If-None-Match` response headers for each URL it requests and then sets those headers in any subsequent requests for the same URL. This instructs the server to send an HTTP 304 "Not Modified" response if the URL has not changed since the last time it was requested. By default, this option is unset and jQuery does not set or record these headers.

jQuery translates an HTTP 304 response to the status code "notmodified". The "notmodified" status is not considered an error, and this value is passed to the `success` callback instead of the normal "success" status code. Thus, if you set the `ifModified` option, you must check the status code in your callback—if the

status is "notmodified", the first argument (the response data) will be undefined. Note that in versions of jQuery before 1.4, a HTTP 304 code was considered an error and the "notmodified" status code was passed to the error callback instead of the success callback. See the sidebar in §19.6.1 for more on jQuery's Ajax status codes.

`global`
: This option specifies whether jQuery should trigger events that describe the progress of the Ajax request. The default is `true`; set this option to `false` to disable all Ajax-related events. (See §19.6.4 for full event details.) The name of this option is confusing: it is named "global" because jQuery normally triggers its events globally rather than on a specific object.

19.6.3.2 Callbacks

The following options specify functions to be invoked at various stages during the Ajax request. The `success` option is already familiar: it is the callback function that you pass to methods like `jQuery.getJSON()`. Note that jQuery also sends notification about the progress of an Ajax request as events (unless you have set the `global` option to `false`).

`context`
: This option specifies the object to be used as the context—the `this` value—for invocations of the various callback functions. This option has no default value, and if left unset, callbacks are invoked on the options object that holds them. Setting the `context` option also affects the way Ajax events are triggered (see §19.6.4). If you set it, the value should be a Window, Document, or Element on which events can be triggered.

`beforeSend`
: This option specifies a callback function that will be invoked before the Ajax request is sent to the server. The first argument is the XMLHttpRequest object and the second argument is the options object for the request. The `beforeSend` callback gives programs the opportunity to set custom HTTP headers on the `XMLHttpRequest` object. If this callback function returns `false`, the Ajax request will be aborted. Note that cross-domain "script" and "jsonp" requests do not use an XMLHttpRequest object and do not trigger the `beforeSend` callback.

`success`
: This option specifies the callback function to be invoked when an Ajax request completes successfully. The first argument is the data sent by the server. The second argument is the jQuery status code, and the third argument is the XMLHttpRequest object that was used to make the request. As explained in §19.6.2.3, the type of the first argument depends on the `dataType` option or on the `Content-Type` header of the server's response. If the type is "xml", the first argument is a Document object. If the type is "json" or "jsonp", the first argument is the object that results from parsing the server's JSON-formatted response. If the type was "script", the response is the text of the loaded script (that script will already have been executed,

however, so the response can usually be ignored in this case). For other types, the response is simply the text of the requested resource.

The second argument status code is normally the string "success", but if you have set the `ifModified` option, this argument might be "notmodified" instead. In this case, the server does not send a response and the first argument is undefined. Cross-domain requests of type "script" and "jsonp" are performed with a `<script>` element instead of an XMLHttpRequest, so for those requests, the third argument will be undefined.

`error`

This option specifies the callback function to be invoked if the Ajax request does not succeed. The first argument to this callback is the XMLHttpRequest object of the request (if it used one). The second argument is the jQuery status code. This may be "error" for an HTTP error, "timeout" for a timeout, and "parsererror" for an error that occurred while parsing the server's response. If an XML document or JSON object is not well-formed, for example, the status code will be "parsererror". In this case, the third argument to the `error` callback will be the Error object that was thrown. Note that requests with `dataType` "script" that return invalid JavaScript code do not cause errors. Any errors in the script are silently ignored, and the `success` callback is invoked instead of the `error` callback.

`complete`

This option specifies a callback function to be invoked when the Ajax request is complete. Every Ajax request either succeeds and calls the `success` callback or fails and calls the `error` callback. jQuery invokes the `complete` callback after invoking either `success` or `error`. The first argument to the `complete` callback is the XMLHttpRequest object, and the second is the status code.

19.6.3.3 Uncommon options and hooks

The following Ajax options are not commonly used. Some specify options that you are not likely to set and others provide customization hooks for those who need to modify jQuery's default handling of Ajax requests.

`async`

Scripted HTTP requests are asynchronous by their very nature. The XMLHttpRequest object provides an option to block until the response is received, however. Set this option to `false` if you want jQuery to block. Setting this option does not change the return value of `jQuery.ajax()`: the function always returns the XMLHttpRequest object, if it used one. For synchronous requests, you can extract the server's response and HTTP status code from the XMLHttpRequest object yourself, or you can specify a `complete` callback (as you would for an asynchronous request) if you want jQuery's parsed response and status code.

`dataFilter`

This option specifies a function to filter or preprocess the data returned by the server. The first argument will be the raw data from the server (either as a string or

Document object for XML requests) and the second argument will be the value of the `dataType` option. If this function is specified, it must return a value, and that value will be used in place of the server's response. Note that the `dataFilter` function is invoked before JSON parsing or script execution is performed. Also note that `dataFilter` is not invoked for cross-origin "script" and "jsonp" requests.

`jsonp`
: When you set the `dataType` option to "jsonp", your `url` or `data` option usually includes a parameter like "jsonp=?". If jQuery does not find such a parameter in the URL or data, it inserts one, using this option as the parameter name. The default value of this option is "callback". Set this option if you are using JSONP with a server that expects a different parameter name and have not already encoded that parameter into your URL or data. See §18.2 for more about JSONP.

`jsonpCallback`
: For requests with `dataType` "jsonp" (or type "json" when the URL includes a JSONP parameter like "jsonp=?"), jQuery must alter the URL to replace the question mark with the name of the wrapper function that the server will pass its data to. Normally, jQuery synthesizes a unique function name based on the current time. Set this option if you want to substitute your own function for jQuery's. If you do this, however, it will prevent jQuery from invoking the `success` and `complete` callbacks and from triggering its normal events.

`processData`
: When you set the `data` option to an object (or pass an object as the second argument to `jQuery.get()` and related methods), jQuery normally converts that object to a string in the standard HTML "application/x-www-form-urlencoded" format (see the sidebar in §19.6.2.2). If you want to avoid this step (such as when you want to pass a Document object as the body of a POST request), set this option to `false`.

`scriptCharset`
: For cross-origin "script" and "jsonp" requests that use a `<script>` element, this option specifies the value of the `charset` attribute of that element. It has no effect for regular XMLHttpRequest-based requests.

`traditional`
: jQuery 1.4 altered slightly the way that data objects were serialized to "application/x-www-form-urlencoded" strings (see the sidebar in §19.6.2.2 for details). Set this option to `true` if you need jQuery to revert to its old behavior.

`username, password`
: If a request requires password-based authentication, specify the username and password using these two options.

`xhr`
: This option specifies a factory function for obtaining an XMLHttpRequest. It is invoked with no arguments and must return an object that implements the XMLHttpRequest API. This very low-level hook allows you create your own wrapper around XMLHttpRequest, adding features or instrumentation to its methods.

19.6.4 Ajax Events

§19.6.3.2 explained that `jQuery.ajax()` has four callback options: `beforeSend`, `success`, `error`, and `complete`. In addition to invoking these individually specified callback functions, jQuery's Ajax functions also fire custom events at each of the same stages in a Ajax request. The following table shows the callback options and the corresponding events:

Callback	Event Type	Handler Registration Method
`beforeSend`	"ajaxSend"	`ajaxSend()`
`success`	"ajaxSuccess"	`ajaxSuccess()`
`error`	"ajaxError"	`ajaxError()`
`complete`	"ajaxComplete"	`ajaxComplete()`
	"ajaxStart"	`ajaxStart()`
	"ajaxStop"	`ajaxStop()`

You can register handlers for these custom Ajax events using the `bind()` method (§19.4.4) and the event type string shown in the second column or using the event registration methods shown in the third column. `ajaxSuccess()` and the other methods work just like the `click()`, `mouseover()`, and other simple event registration methods of §19.4.1.

Since the Ajax events are custom events, generated by jQuery rather than the browser, the Event object passed to the event handler does not contain much useful detail. The ajaxSend, ajaxSuccess, ajaxError, and ajaxComplete events are all triggered with additional arguments, however. Handlers for these events will all be invoked with two extra arguments after the event. The first extra argument is the XMLHttpRequest object and the second extra argument is the options object. This means, for example, that a handler for the ajaxSend event can add custom headers to an XMLHttpRequest object just like the `beforeSend` callback can. The ajaxError event is triggered with a third extra argument, in addition to the two just described. This final argument to the event handler is the Error object, if any, that was thrown when the error occurred. Surprisingly, these Ajax events are not passed jQuery's status code. If the handler for an ajaxSuccess event needs to distinguish "success" from "notmodified", for example, it will need to examine the raw HTTP status code in the XMLHttpRequest object.

The last two events listed in the table above are different from the others, most obviously because they have no corresponding callback functions, and also because they are triggered with no extra arguments. ajaxStart and ajaxStop are a pair of events that indicate the start and stop of Ajax-related network activity. When jQuery is not performing any Ajax requests and a new request is initiated, it fires an ajaxStart event. If other requests begin before this first one ends, those new requests do not cause a new ajaxStart event. The ajaxStop event is triggered when the last pending Ajax request is completed and jQuery is no longer performing any network activity. This pair of events can be useful

to show and hide some kind of "Loading..." animation or network activity icon. For example:

```
$("#loading_animation").bind({
    ajaxStart: function() { $(this).show(); },
    ajaxStop: function() { $(this).hide(); }
});
```

These ajaxStart and ajaxStop event handlers can be bound to any document element: jQuery triggers them globally (§19.4.6) rather than on any one particular element. The other four Ajax events, ajaxSend, ajaxSuccess, ajaxError, and ajaxComplete, are also normally triggered globally, so you can bind handlers to any element. If you set the `context` option in your call to `jQuery.ajax()`, however, these four events are triggered on the context element rather than globally.

Finally, remember that you can prevent jQuery from triggering any Ajax-related events by setting the `global` option to `false`. Despite its confusing name, setting `global` to `false` stops jQuery from triggering events on a `context` object as well as stopping jQuery from triggering events globally.

19.7 Utility Functions

The jQuery library defines a number of utility functions (as well as two properties) that you may find useful in your programs. As you'll see in the list below, a number of these functions now have equivalents in ECMAScript 5 (ES5). jQuery's functions predate ES5 and work in all browsers. In alphabetical order, the utility functions are:

`jQuery.browser`

The `browser` property is not a function but an object that you can use for client sniffing (§13.4.5). This object will have the property `msie` set to `true` if the browser is IE. The `mozilla` property will be true if the browser is Firefox or related. The `webkit` property will be true for Safari and Chrome, and the `opera` property will be true for Opera. In addition to this browser-specific property, the `version` property contains the browser version number. Client sniffing is best avoided whenever possible, but you can use this property to work around browser-specific bugs with code like this:

```
if ($.browser.mozilla && parseInt($.browser.version) < 4) {
    // Work around a hypothetical Firefox bug here...
}
```

`jQuery.contains()`

This function expects two document elements as its arguments. It returns `true` if the first element contains the second element and returns `false` otherwise.

`jQuery.each()`

Unlike the `each()` method which iterates only over jQuery objects, the `jQuery.each()` utility function iterates through the elements of an array or the properties of an object. The first argument is the array or object to be iterated.

The second argument is the function to be called for each array element or object property. That function will be invoked with two arguments: the index or name of the array element or object property, and the value of the array element or object property. The `this` value for the function is the same as the second argument. If the function returns `false`, `jQuery.each()` returns immediately without completing the iteration. `jQuery.each()` always returns its first argument.

`jQuery.each()` enumerates object properties with an ordinary `for/in` loop, so all enumerable properties are iterated, even inherited properties. `jQuery.each()` enumerates array elements in numerical order by index and does not skip the undefined properties of sparse arrays.

`jQuery.extend()`

This function expects objects as its arguments. It copies the properties of the second and subsequent objects into the first object, overwriting any properties with the same name in the first argument. This function skips any properties whose value is `undefined` or `null`. If only one object is passed, the properties of that object are copied into the `jQuery` object itself. The return value is the object into which properties were copied. If the first argument is the value `true`, a deep or recursive copy is performed: the second argument is extended with the properties of the third (and any subsequent) objects.

This function is useful for cloning objects and for merging options objects with sets of defaults:

```
var clone = jQuery.extend({}, original);
var options = jQuery.extend({}, default_options, user_options);
```

`jQuery.globalEval()`

This function executes a string of JavaScript code in the global context, as if it were the contents of a `<script>` element. (In fact, jQuery actually implements this function by creating a `<script>` element and temporarily inserting it into the document.)

`jQuery.grep()`

This function is like the ES5 `filter()` method of the Array object. It expects an array as its first argument and a predicate function as its second, and it invokes the predicate once for each element in the array, passing the element value and the element index. `jQuery.grep()` returns a new array that contains only those elements of the argument array for which the predicate returned `true` (or another truthy value). If you pass `true` as the third argument to `jQuery.grep()`, it inverts the sense of the predicate and returns an array of elements for which the predicate returned `false` or another falsy value.

`jQuery.inArray()`

This function is like the ES5 `indexOf()` method of the Array object. It expects an arbitrary value as its first argument and an array (or array-like object) as its second and returns the first index in the array at which the value appears, or -1 if the array does not contain the value.

`jQuery.isArray()`
: Returns `true` if the argument is a native Array object.

`jQuery.isEmptyObject`
: Returns `true` if the argument has no enumerable properties.

`jQuery.isFunction()`
: Returns `true` if the argument is a native Function object. Note that in IE8 and earlier, browser methods like `Window.alert()` and `Element.attachEvent()` are not functions in this sense.

`jQuery.isPlainObject()`
: Returns `true` if the argument is a "plain" object rather than an instance of some more specialized type or class of objects.

`jQuery.makeArray()`
: If the argument is an array-like object, this function copies the elements of that object into a new (true) array and returns that array. If the argument is not array-like, this function simply returns a new array with the argument as its single element.

`jQuery.map()`
: This function is like the ES5 `map()` method of the Array object. It expects an array or array-like object as its first argument and a function as its second. It passes each array element along with the index of that element to the function and returns a new array that collects the values returned by the function. `jQuery.map()` differs from the ES5 `map()` method in a couple of ways. If your mapping function returns `null`, that value will not be included in the result array. And if your mapping function returns an array, the elements of that array will be added to the result rather than the array itself.

`jQuery.merge()`
: This function expects two arrays or array-like objects. It appends the elements of the second to the first and returns the first. The first array is modified, the second is not. Note that you can use this function to shallowly clone an array like this:

```
var clone = jQuery.merge([], original);
```

`jQuery.parseJSON()`
: This function parses a JSON-formatted string and returns the resulting value. It throws an exception when passed malformed input. jQuery uses the standard `JSON.parse()` function in browsers that define it. Note that jQuery defines only a JSON parsing function, not a JSON serialization function.

`jQuery.proxy()`
: This function is something like the ES5 `bind()` (§8.7.4) method of the Function object. It takes a function as its first argument and an object as its second and returns a new function that invokes the function as a method of the object. It does not perform partial application of arguments like the `bind()` method does.

`jQuery.proxy()` can also be invoked with an object as its first argument and a property name as its second. The value of the named property should be a function. Invoked in this way, the function `jQuery.proxy(o,n)` returns the same thing that `jQuery.proxy(o[n],o)` does.

`jQuery.proxy()` is intended for use with jQuery's event handler binding mechanism. If you bind a proxied function, you can unbind it using the original function.

`jQuery.support`

This is a property like `jQuery.browser`, but it is intended for portable feature testing (§13.4.3) rather than more brittle browser testing. The value of `jQuery.support` is an object whose properties are all boolean values that specify the presence or absence of browser features. Most of these `jQuery.support` properties are low-level details used internally by jQuery. They may be of interest to plug-in writers, but most are not generally useful to application writers. One exception is `jQuery.support.boxModel`: this property is `true` if the browser uses the CSS standard "context-box" model and is `false` in IE6 and IE7 in quirks mode (see §16.2.3.1).

`jQuery.trim()`

This function is like the `trim()` method added to strings in ES5. It expects a string as its only argument and returns a copy of that string with leading and trailing whitespace removed.

19.8 jQuery Selectors and Selection Methods

Throughout this chapter, we've been using the jQuery selection function, `$()`, with simple CSS selectors. It is now time to study the jQuery selector grammar in depth, along with a number of methods for refining and augmenting the set of selected elements.

19.8.1 jQuery Selectors

jQuery supports a fairly complete subset of the selector grammar defined by the CSS3 Selectors draft standard, with the addition of some nonstandard but very useful pseudoclasses. Basic CSS selectors were described in §15.2.5. We repeat that material here, and add explanations for more advanced selectors as well. Bear in mind that this section documents jQuery selectors. Many, but not all, of these selectors can also be used in CSS stylesheets.

The selector grammar has three layers. You've undoubtedly seen the simplest kind of selectors before. "#test" selects an element with an `id` attribute of "test". "blockquote" selects all `<blockquote>` elements in the document, and "div.note" selects all `<div>` elements with a `class` attribute of "note". Simple selectors can be combined into "selector combinations" such as "div.note>p" and "blockquote i" by separating them with a *combinator* character. And simple selectors and selector combinations can be grouped into comma-separated lists. These selector groups are the most general kind of selector

that we pass to `$()`. Before explaining selector combinations and selector groups, we must explain the syntax of simple selectors.

19.8.1.1 Simple selectors

A simple selector begins (explicitly or implicitly) with a tag type specification. If you are only interested in <p> elements, for example, your simple selector would begin with "p". If you want to select elements without regard to their tagname, use the wildcard "*" instead. If a selector does not begin with either a tagname or a wildcard, the wildcard is implicit.

The tagname or wildcard specifies an initial set of document elements that are candidates for selection. The portion of the simple selector that follows this type specification consists of zero or more filters. The filters are applied left-to-right, in the order that they appear, and each one narrows the set of selected elements. Table 19-1 lists the filters supported by jQuery.

Table 19-1. jQuery Selector Filters

Filter	Meaning
`#id`	Matches the element with an `id` attribute of *`id`*. Valid HTML documents never have more than one element with the same ID, so this filter is usually used as a stand-alone selector.
`.class`	Matches any elements whose `class` attribute (when interpreted as a list of words separated by spaces) includes the word *`class`*.
`[attr]`	Matches any elements that have an *`attr`* attribute (regardless of its value).
`[attr=val]`	Matches any elements that have an *`attr`* attribute whose value is *`val`*.
`[attr!=val]`	Matches elements that have no *`attr`* attribute, or whose *`attr`* attribute is not equal to *`val`* (jQuery extension).
`[attr^=val]`	Matches elements whose *`attr`* attribute has a value that begins with *`val`*.
`[attr$=val]`	Matches elements whose *`attr`* attribute has a value that ends with *`val`*.
`[attr*=val]`	Matches elements whose *`attr`* attribute has a value that contains *`val`*.
`[attr~=val]`	Matches elements whose *`attr`* attribute, when interpreted as a list of words separated by spaces, includes the word *`val`*. Thus the selector "div.note" is the same as "div[class~=note]".
`[attr|=val]`	Matches elements whose *`attr`* attribute has a value that begins with *`val`* and is optionally followed by a hyphen and any other characters.
`:animated`	Matches elements that are currently being animated by jQuery.
`:button`	Matches `<button type="button">` and `<input type="button">` elements (jQuery extension).
`:checkbox`	Matches `<input type="checkbox">` elements (jQuery extension). This filter is most efficient when explicitly prefixed with the input tag: "input:checkbox".
`:checked`	Matches input elements that are checked.
`:contains(text)`	Matches elements that contain the specified *`text`* (jQuery extension). The parentheses of this filter delimit the text—no quotation marks are required. The text of the elements being filtered is

Filter	Meaning
	determined with their `textContent` or `innerText` properties—this is the raw document text, with tags and comments stripped out.
`:disabled`	Matches disabled elements.
`:empty`	Matches elements that have no children, including no text content.
`:enabled`	Matches elements that are not disabled.
`:eq(`*n*`)`	Matches only the *n*th element of the document-order zero-indexed list of matches (jQuery extension).
`:even`	Matches elements with even indexes in the list. Since the first element has an index of 0, this actually matches the first, third, and fifth (and so on) elements (jQuery extension).
`:file`	Matches `<input type="file">` elements (jQuery extension).
`:first`	Matches only the first element in the list. Same as `:eq(0)` (jQuery extension).
`:first-child`	Matches only elements that are the first child of their parent. Note that this is completely different than `:first`.
`:gt(`*n*`)`	Matches elements in the document-order list of matches whose zero-based index is greater than *n* (jQuery extension).
`:has(`*sel*`)`	Matches elements that have a descendant matching the nested selector *sel*.
`:header`	Matches any header element: `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, or `<h6>` (jQuery extension).
`:hidden`	Matches any element that is not visible on the screen: roughly those elements whose `offsetWidth` and `offsetHeight` are 0.
`:image`	Matches `<input type="image">` elements. Note that this does not match `<img>` elements (jQuery extension).
`:input`	Matches user input elements: `<input>`, `<textarea>`, `<select>`, and `<button>` (jQuery extension).
`:last`	Matches the last element in the list of matches (jQuery extension).
`:last-child`	Matches any element that is the last child of its parent. Note that this is not the same as `:last`.
`:lt(`*n*`)`	Matches all elements in the document-order list of matches whose zero-based index is less than *n* (jQuery extension).
`:not(`*sel*`)`	Matches elements that are *not* matched by the nested selector *sel*.
`:nth(`*n*`)`	A synonym for `:eq(`*n*`)` (jQuery extension).
`:nth-child(`*n*`)`	Matches elements that are the *n*th child of their parent. *n* can be a number, the word "even", the word "odd", or a formula. Use `:nth-child(even)` to select elements that are the second and fourth (and so on) in their parent's list of children. Use `:nth-child(odd)` to select elements that are first, third, and so on. Most generally, *n* can be a formula of the form *x*n or *x*n+*y* where *x* and *y* are integers and n is the literal letter n. Thus `nth-child(3n+1)` selects the first, fourth, and seventh (and so on) elements. Note that this filter uses one-based indexes, so an element that is the first child of its parent is considered odd and is matched by `3n+1`, not `3n`. Contrast this with the `:even` and `:odd` filters that filter based on an element's zero-based position in the list of matches.
`:odd`	Matches elements with odd (zero-based) indexes in the list. Note that elements 1 and 3 are the second and fourth matched element, respectively (jQuery extension).

Filter	Meaning
`:only-child`	Matches elements that are the only child of their parent.
`:parent`	Matches elements that are parents. This is the opposite of `:empty` (jQuery extension).
`:password`	Matches `<input type="password">` elements (jQuery extension).
`:radio`	Matches `<input type="radio">` elements (jQuery extension).
`:reset`	Matches `<input type="reset">` and `<button type="reset">` elements (jQuery extension).
`:selected`	Matches `<option>` elements that are selected. Use `:checked` for selected checkboxes and radio buttons (jQuery extension).
`:submit`	Matches `<input type="submit">` and `<button type="submit">` elements (jQuery extension).
`:text`	Matches `<input type="text">` elements (jQuery extension).
`:visible`	Matches all elements that are currently visible: roughly those that have nonzero `offsetWidth` and `offsetHeight`. This is the opposite of `:hidden`.

Notice that some of the filters listed in Table 19-1 accept arguments within parentheses. The following selector, for example, selects paragraphs that are the first or every third subsequent child of their parent, as long as they contain the word "JavaScript" and do not contain an `<a>` element.

```
p:nth-child(3n+1):text(JavaScript):not(:has(a))
```

Filters typically run most efficiently if prefixed with a tag type. Rather than simply using ":radio" to select radio buttons, for example, it is better to use "input:radio". The exception is ID filters, which are most efficient when they stand alone. The selector "#address" is typically more efficient than the more explicit "form#address", for example.

19.8.1.2 Selector combinations

Simple selectors can be combined using special operators or "combinators" to represent relationships between elements in the document tree. Table 19-2 lists the selector combinations supported by jQuery. These are the same selector combinations that CSS3 supports.

Table 19-2. jQuery Selector Combinations

Combination	Meaning
`A B`	Selects document elements that match selector B that are descendants of elements that match selector A. Note that the combinator character is simply whitespace for this combination.
`A > B`	Selects document elements that match selector B that are direct children of elements that match selector A.
`A + B`	Selects document elements that match selector B and immediately follow (ignoring text nodes and comments) elements that match selector A.
`A ~ B`	Selects document elements matching B that are sibling elements that come after elements that match A.

Here are some example selector combinations:

```
"blockquote i"        // Matches an <i> element within a <blockquote>
"ol > li"             // An <li> element as a direct child of an <ol>
"#output + *"         // The sibling after the element with id="output"
"div.note > h1 + p"   // A <p> following a <h1> inside a <div class="note">
```

Note that selector combinations are not limited to combinations of two selectors: three or more selectors are allowed, too. Selector combinations are processed left to right.

19.8.1.3 Selector groups

A selector group, which is the kind of selector that we pass to `$()` (or use in a stylesheet), is simply a comma-separated list of one or more simple selectors or selector combinations. A selector group matches all elements that match any of the selector combinations in the group. For our purposes here, even a simple selector can be considered a selector combination. Here are some example selector groups:

```
"h1, h2, h3"          // Matches <h1>, <h2>, and <h3> elements
"#p1, #p2, #p3"       // Matches elements with id p1, p2, and p3
"div.note, p.note"    // Matches <div> and <p> elements with class="note"
"body>p,div.note>p"   // <p> children of <body> and <div class="note">
```

Note that the CSS and jQuery selector syntax uses parentheses for some of the filters in simple selectors, but it does not allow parentheses to be used more generally for grouping. You cannot put a selector group or selector combination in parentheses and treat it like a simple selector, for example:

```
(h1, h2, h3)+p        // Not legal
h1+p, h2+p, h3+p      // Write this instead
```

19.8.2 Selection Methods

In addition to the selector grammar supported by `$()`, jQuery defines a number of selection methods. Most of the jQuery methods we've seen so far in this chapter perform some action on the selected elements. The selection methods are different: they alter the set of selected elements by refining it, augmenting it, or just using it as a starting point for a new selection.

This section describes these selection methods. You'll notice that many of the methods provide the same functionality as the selector grammar itself.

The simplest way to refine a selection is by position within the selection. `first()` returns a jQuery object that contains only the first selected element, and `last()` returns a jQuery object that contains only the last element. More generally, the `eq()` method returns a jQuery object that contains only the single selected element at the specified index. (In jQuery 1.4, negative indexes are allowed and count from the end of the selection.) Note that these methods return a jQuery object with a single element. This is different than regular array indexing, which returns a single element with no jQuery object wrapped around it:

```
var paras = $("p");
paras.first()          // Select only the first <p> element
paras.last()           // Select only the last <p>
paras.eq(1)            // Select the second <p>
paras.eq(-2)           // Select the second to last <p>
paras[1]               // The second <p> element, itself
```

The general method for refining a selection by position is `slice()`. The jQuery `slice()` method works like the `Array.slice()` method: it accepts a start and an end index (with negative indexes measured from the end of the array) and returns a jQuery object that contains elements from the start index up to, but not including, the end index. If the end index is omitted, the returned object includes all elements at or after the start index:

```
$("p").slice(2,5)      // Select the 3rd, 4th, and 5th <p> elements
$("div").slice(-3)     // The last three <div> elements
```

`filter()` is a general-purpose selection filtering method, and you can invoke it in three different ways:

- If you pass a selector string to `filter()`, it returns a jQuery object containing only those selected elements that also match that selector.
- If you pass another jQuery object to `filter()`, it returns a new jQuery object that contains the intersection of the two jQuery objects. You can also pass an array of elements, or even a single document element, to `filter()`.
- If you pass a predicate function to `filter()`, that function is called for each matched element, and `filter()` returns a jQuery object containing only those elements for which the predicate returned `true` (or any truthy value). The predicate function is called with the element as its `this` value and the element index as an argument. (See also `jQuery.grep()` in §19.7.)

```
$("div").filter(".note")                          // Same as $("div.note")
$("div").filter($(".note"))                       // Same as $("div.note")
$("div").filter(function(idx) { return idx%2==0 }) // Same as $("div:even")
```

The `not()` method is just like `filter()`, except that it inverts the sense of the filter. If you pass a selector string to `not()`, it returns a new jQuery object containing only the selected elements that do *not* match the selector. If you pass a jQuery object or an array of elements or a single element, `not()` returns all of the selected elements except for the elements you've explicitly excluded. If you pass a predicate function to `not()`, it is invoked just as it is for `filter()`, but the returned jQuery object includes only those elements for which the predicate returns `false` or a falsy value:

```
$("div").not("#header, #footer");  // All <div> elements except two special ones
```

In jQuery 1.4, the `has()` method is another way to refine a selection. If you pass a selector, it returns a new jQuery object that contains only the selected elements that have a descendant that matches the selector. If you pass a document element to `has()`, it refines the selection to match only those elements that are ancestors of the specified element:

```
$("p").has("a[href]")    // Paragraphs that include links
```

The add() method augments a selection rather than filtering or refining it. You can invoke add() with any arguments (other than a function) that you would pass to $(). add() returns the originally selected elements plus whatever elements would be selected (or created) by the arguments if those arguments were passed to $(). add() removes duplicate elements and sorts the combined selection so that the elements are in document order:

```
// Equivalent ways to select all <div> and all <p> elements
$("div, p")            // Use a selector group
$("div").add("p")      // Pass a selector to add()
$("div").add($("p"))   // Pass a jQuery object to add()
var paras = document.getElementsByTagName("p");  // An array-like object
$("div").add(paras);   // Pass an array of elements to add()
```

19.8.2.1 Using a selection as context

The filter(), add(), and not() methods described above perform set intersection, union, and subtraction operations on independent selections. jQuery defines a number of other selection methods that use the current selection as the context. For each selected element, these methods make a new selection using the selected element as the context or starting point, and then return a new jQuery object that contains the union of those selections. As with the add() method, duplicates are removed and the elements are sorted so that they are in document order.

The most general of this category of selection methods is find(). It searches the descendants of each of the currently selected elements for elements that match the specified selector string, and it returns a new jQuery object that represents that new set of matching descendants. Note that the newly selected elements are not merged with the existing selection; they are returned as a new set of elements. Note also that find() is not the same as filter(), which simply narrows the currently selected set of elements without selecting new elements:

```
$("div").find("p")  // find <p> elements inside <div>s.  Same as $("div p")
```

The other methods in this category return new jQuery objects that represent the children, siblings, or parents of each of the currently selected elements. Most accept an optional selector string as an argument. With no selector, they return all appropriate children, siblings, or parents. With the selector, they filter the list to return only those that match.

The children() method returns the immediate child elements of each selected element, filtering them with an optional selector:

```
// Find all <span> elements that are direct children of the elements with
// ids "header" and "footer". Same as $("#header>span,#footer>span")
$("#header, #footer").children("span")
```

The contents() method is similar to children(), but it returns all child nodes, including text nodes, of each element. Also, if any of the selected elements is an <iframe>,

contents() returns the document object for the content of that <iframe>. Note that contents() does not accept an optional selector string—this is because it returns document nodes that are not elements, and selector strings only describe element nodes.

The next() and prev() methods return the next and previous sibling of each selected element that has one. If a selector is specified, the sibling is selected only if it matches the selector:

```
$("h1").next("p")           // Same as $("h1+p")
$("h1").prev()              // Sibling elements before <h1> elements
```

nextAll() and prevAll() return all siblings following and all siblings preceding (if there are any) each selected element. And the siblings() method returns all siblings of each selected element (elements are not considered siblings of themselves). If a selector is passed to any of these methods, only siblings that match are returned:

```
$("#footer").nextAll("p")   // All <p> siblings following the #footer element
$("#footer").prevAll()      // All siblings before the #footer element
```

In jQuery 1.4 and later, the nextUntil() and prevUntil() methods take a selector argument and select all siblings following or preceding the selected element until a sibling is found that matches the selector. If you omit the selector, these methods work just like nextAll() and prevAll() with no selector.

The parent() method returns the parent of each selected element:

```
$("li").parent()          // Parents of list items, like <ul> and <ol> elements
```

The parents() method returns the ancestors (up to the <html> element) of each selected element. Both parent() and parents() accept an optional selector string argument:

```
$("a[href]").parents("p") // <p> elements that contain links
```

parentsUntil() returns the ancestors of each selected element until the first ancestor that matches the specified selector. The closest() method requires a selector string and returns the closest ancestor (if any) of each selected element that matches the selector. For this method, an element is considered an ancestor of itself. In jQuery 1.4, you can also pass an ancestor element as the second argument to closest(), to prevent jQuery from climbing the ancestor tree beyond the specified element:

```
$("a[href]").closest("div")           // Innermost <div>s that contain links
$("a[href]").parentsUntil(":not(div)") // All <div> wrappers directly around <a>
```

19.8.2.2 Reverting to a previous selection

To facilitate method chaining, most jQuery object methods return the object on which they are called. The methods we've covered in this section all return new jQuery objects, however. Method chaining works, but you must keep in mind that methods called later in the chain may be operating on a different set of elements than those near the start of the chain.

The situation is a little more complicated than this, however. When the selection methods described here create and return a new jQuery object, they give that object an

internal reference to the older jQuery object from which it was derived. This creates a linked list or stack of jQuery objects. The end() method pops this stack, returning the saved jQuery object. Calling end() in a method chain restores the set of matched elements to its previous state. Consider the following code:

```
// Find all <div> elements, then find the <p> elements inside them.
// Highlight the <p> elements and then give the <div> elements a border.

// First, without method chaining
var divs = $("div");
var paras = divs.find("p");
paras.addClass("highlight");
divs.css("border", "solid black 1px");

// Here's how we could do it with a method chain
$("div").find("p").addClass("highlight").end().css("border", "solid black 1px");

// Or we can reorder the operations and avoid the call to end()
$("div").css("border", "solid black 1px").find("p").addClass("highlight");
```

If you ever want to manually define the set of selected elements in a way that is compatible with the end() method, pass the new set of elements as an array or array-like object to the pushStack() method. The elements you specify become the new selected elements, and the previous set of selected elements is pushed on the stack, where they can be restored with end():

```
var sel = $("div");                                  // Select all <div> elements
sel.pushStack(document.getElementsByTagName("p")); // Modify it to all <p> elts
sel.end();                                           // Restore <div> elements
```

Now that we've covered the end() method and the selection stack that it uses, there is one final method we can cover. andSelf() returns a new jQuery object that includes all of the elements of the current selection plus all of the elements (minus duplicates) of the previous selection. andSelf() works like the add() method, and "addPrev" might be a more descriptive name for it. As an example, consider the following variant on the code above: it highlights <p> elements and the elements that hold them, and then adds a border to the elements:

```
$("div").find("p").andSelf().          // find <p>s in <div>s, and merge them
    addClass("highlight").             // Highlight them all
    end().end().                       // Pop stack twice back to $("div")
    css("border", "solid black 1px"); // Give the divs a border
```

19.9 Extending jQuery with Plug-ins

jQuery is written so that it is easy to add new functionality. Modules that add new functionality are called *plug-ins*, and you can find many of them at *http://plugins.jquery.com*. jQuery plug-ins are just ordinary files of JavaScript code, and to use them in your web pages, you just include them with a <script> element as you would any other JavaScript library (you must include plug-ins after you include jQuery itself, of course).

It is almost trivially easy to write your own jQuery extensions. The trick is to know that `jQuery.fn` is the prototype object for all jQuery objects. If you add a function to this object, that function becomes a jQuery method. Here is an example:

```
jQuery.fn.println = function() {
    // Join all the arguments into a space-separated string
    var msg = Array.prototype.join.call(arguments, " ");
    // Loop through each element in the jQuery object
    this.each(function() {
        // For each one, append the string as plain text, then append a <br/>.
        jQuery(this).append(document.createTextNode(msg)).append("<br/>");
    });
    // Return the unmodified jQuery object for method chaining
    return this;
};
```

With that `jQuery.fn.println` function defined, we can now invoke a `println()` method on any jQuery object like this:

```
$("#debug").println("x = ", x, "; y = ", y);
```

It is common practice to add new methods to `jQuery.fn`. If you find yourself using the `each()` method to "manually" iterate through the elements in a jQuery object and perform some kind of operation on them, ask yourself whether it might make sense to refactor your code so that the `each()` invocation is moved into an extension method. If you follow basic modular coding practices when writing your extension and abide by a few jQuery-specific conventions, you can call your extension a plug-in and share it with others. These are the jQuery plug-in conventions to be aware of:

- Don't rely on the `$` identifier: the including page may have called `jQuery.noConflict()` and `$()` may no longer be a synonym for the `jQuery()` function. In short plug-ins like the one shown above, you can just use `jQuery` instead of `$`. If you are writing a longer extension, you are likely to wrap it all within one anonymous function to avoid the creation of global variables. If you do so, you can use the idiom of passing the jQuery as an argument to your anonymous function, and receiving that value in a parameter named `$`:

  ```
  (function($) {  // An anonymous function with one parameter named $
      // Put your plugin code here
  }(jQuery));     // Invoke the function with the jQuery object as its argument
  ```

- If your extension method does not return a value of its own, be sure to return a jQuery object that can be used in a method chain. Usually this will just be the `this` object and you can return it unmodified. In the example above, the method ended with the line `return this;`. The method could have been made slightly shorter (and less readable) following another jQuery idiom: returning the result of the `each()` method. Then the `println()` method would have included the code `return this.each(function() {...});`
- If your extension method has more than a couple of parameters or configuration options, allow the user to pass options in the form of an object (as we saw with the `animate()` method in §19.5.2 and the `jQuery.ajax()` function in §19.6.3).

- Don't pollute the jQuery method namespace. Well-behaved jQuery plug-ins define the smallest number of methods consistent with a usable API. It is common for jQuery plug-ins to define only a single method in `jQuery.fn`. This one method takes a string as its first argument and interprets that string as the name of a function to pass its remaining arguments to. When you are able to limit your plug-in to a single method, the name of that method should be the same as the name of the plug-in. If you must define more than one method, use the plug-in name as a prefix for each of your method names.
- If your plug-in binds event handlers, put all of those handlers in an event namespace (§19.4.4). Use your plug-in name as the namespace name.
- If your plug-in uses the `data()` method to associate data with elements, place all of your data values in a single object, and store that object as a single value, giving it the same name as your plug-in.
- Save your plug-in code in a file with a name of the form "jquery.plugin.js", replacing "plugin" with the name of your plug-in.

A plug-in can add new utility functions to jQuery by adding them to the jQuery object itself. For example:

```
// This method prints its arguments (using the println() plugin method)
// to the element with id "debug". If no such element exists, it is created
// and added to the document.
jQuery.debug = function() {
    var elt = jQuery("#debug");          // Find the #debug element
    if (elt.length == 0) {               // Create it if it doesn't exist
        elt = jQuery("<div id='debug'><h1>Debugging Output</h1></div>");
        jQuery(document.body).append(elt);
    }
    elt.println.apply(elt, arguments);  // Output the arguments to it
};
```

In addition to defining new methods, it is also possible to extend other parts of the jQuery library. In §19.5, for example, we saw that it is possible to add new effect duration names (in addition to "fast" and "slow") by adding properties to `jQuery.fx.speeds` and that it is possible to add new easing functions by adding them to `jQuery.easing`. Plug-ins can even extend the jQuery CSS selector engine! You can add new pseudoclass filters (like `:first` and `:input`) by adding properties to the `jQuery.expr[':']` object. Here is an example that defines a new `:draggable` filter that returns only elements that have a `draggable=true` attribute:

```
jQuery.expr[':'].draggable = function(e) { return e.draggable === true; };
```

With this selector defined, we can select draggable images with `$("img:draggable")` instead of the more verbose `$("img[draggable=true]")`.

As you can see from the code above, a custom selector function is passed a candidate DOM element as its first argument. It should return `true` if the element matches the selector and `false` otherwise. Many custom selectors need only the one element argument, but they are actually invoked with four arguments. The second argument is an

integer index that gives the position of this element within an array of candidate elements. That array is passed as the fourth argument and your selector must not modify it. The third argument is an interesting one: it is the array result of a call to the `RegExp.exec()` method. The fourth element of this array (at index 3) is the value, if any, within parentheses after the pseudoclass filter. The parentheses and any quotes inside are stripped, leaving only the argument string. Here, for example, is how you could implement a `:data(x)` pseudoclass that returns `true` only for arguments that have a `data-x` attribute (see §15.4.3):

```
jQuery.expr[':'].data = function(element, index, match, array) {
    // Note: IE7 and before do not implement hasAttribute()
    return element.hasAttribute("data-" + match[3]);
};
```


19.10 The jQuery UI Library

jQuery limits itself to providing core DOM, CSS, event handling, and Ajax functionality. These provide an excellent foundation for building higher-level abstractions, such as user interface widgets, and the jQuery UI library does just that. Full coverage of jQuery UI is beyond the scope of this book, and all we can do here is offer a simple overview. You can find the library and its documentation at *http://jqueryui.com*.

As its name implies, jQuery UI defines a number of user interface widgets: autocompletion input fields, date pickers for entering dates, accordions and tabs for organizing information, sliders and progress bars for visually displaying numbers, and modal dialogs for urgent communication with the user. In addition to these widgets, jQuery UI implements more general "interactions", which allow any document element to be easily made draggable, droppable, resizable, selectable, or sortable. Finally, jQuery UI adds a number of new visual effects methods (including the ability to animate colors) to those offered by jQuery itself, and it defines lots of new easing functions as well.

Think of jQuery UI as a bunch of related jQuery plug-ins packed into a single JavaScript file. To use it, simply include the jQuery UI script in your web page after including the jQuery code. The Download page at *http://jqueryui.com* allows you to select the components you plan to use and will build a custom download bundle for you that may reduce your page load times compared to the full jQuery UI library.

jQuery UI is fully themeable, and its themes take the form of CSS files. So in addition to loading the jQuery UI JavaScript code into your web pages, you'll have to include the CSS file for your selected theme as well. The jQuery UI website features a number of prebuilt themes and also a "ThemeRoller" page that allows you to customize and download your own theme.

jQuery UI widgets and interactions are structured as jQuery plug-ins, and each defines a single jQuery method. Typically, when you call this method on an existing document element, it transforms that element into the widget. For example, to alter a text input

field so that it pops up a date picker widget when clicked or focused, simply call the `datepicker()` method with code like this:

```
// Make <input> elements with class="date" into date picker widgets
$("input.date").datepicker();
```

In order to make full use of a jQuery UI widget, you must be familiar with three things: its configuration options, its methods, and its events. All jQuery UI widgets are configurable, and some have many configuration options. You can customize the behavior and appearance of your widgets by passing an options object (like the animations options object passed to `animate()`) to the widget method.

jQuery UI widgets usually define at least a handful of "methods" for interacting with the widget. In order to avoid a proliferation of jQuery methods, however, jQuery UI widgets do not define their "methods" as true methods. Each widget has only a single method (like the `datepicker()` method in the example above). When you want to call a "method" of the widget, you pass the name of the desired "method" to the single true method defined by the widget. To disable a date picker widget, for example, you don't call a `disableDatepicker()` method; instead, you call `datepicker("disable")`.

jQuery UI widgets generally define custom events that they trigger in response to user interaction. You can bind event handlers for these custom events with the normal `bind()` method, and you can also usually specify event handler functions as properties in the options object you pass to the widget method. The first argument to these handler methods is an Event object as usual. Some widgets pass a second "UI" object as the second argument to the event handler. This object typically provides state information about the widget.

Note that the jQuery UI documentation sometimes describes "events" that are not truly custom events and could better be described as callback functions set through the configuration options object. The date picker widget, for example, supports a number of callback functions that it can call at various times. None of these functions have the standard event handler signature, however, and you cannot register handlers for these "events" with `bind()`. Instead, you specify appropriate callbacks when you configure the widget in your initial call to the `datepicker()` method.

CHAPTER 20
Client-Side Storage

Web applications can use browser APIs to store data locally on the user's computer. This client-side storage serves to give the web browser a memory. Web apps can store user preferences, for example, or even store their complete state, so that they can resume exactly where you left off at the end of your last visit. Client-side storage is segregated by origin, so pages from one site can't read the data stored by pages from another site. But two pages from the same site can share storage and can use it as a communication mechanism. Data input in a form on one page can be displayed in a table on another page, for example. Web applications can choose the lifetime of the data they store: data can be stored temporarily so that it is retained only until the window closes or the browser exits, or it can be saved to the hard drive and stored permanently, so that it is available months or years later.

There are a number of forms of client-side storage:

Web Storage
: Web Storage is an API that was originally defined as part of HTML5 but was spun off as a standalone specification. That specification is still in draft form, but it is partially (and interoperably) implemented in all current browsers including IE8. This API consists of the `localStorage` and `sessionStorage` objects, which are essentially persistent associative arrays that map string keys to string values. Web Storage is very easy to use, is suitable for storing large (but not huge) amounts of data, and is available on all current browsers, but it is not supported by older browsers. `localStorage` and `sessionStorage` are covered in §20.1.

Cookies
: Cookies are an old client-side storage mechanism that was designed for use by server-side scripts. An awkward JavaScript API makes cookies scriptable on the client-side, but they are hard to use and are suitable only for storing small amounts of textual data. Also, any data stored as cookies is always transmitted to the server with every HTTP request, even if the data is only of interest to the client. Cookies continue to be of interest to client-side programmers because all browsers, old and new, support them. Once Web Storage is universally available, however, cookies

will revert to their original role as a client-side storage mechanism for server-side scripts. Cookies are covered in §20.2.

IE User Data
: Microsoft implements its own proprietary client-side storage mechanism, known as "userData," in IE5 and later. userData enables the storage of medium amounts of string data and can be used as an alternative to Web Storage in versions of IE before IE8. The userData API is covered in §20.3.

Offline Web Applications
: HTML5 defines an "Offline Web Applications" API that allows the caching of web pages and their associated resources (scripts, CSS files, images, and so on). This is client-side storage for web applications themselves rather than just their data, and it allows web apps to install themselves so that they are available even when there is no connection to the Internet. Offline web apps are covered in §20.4.

Web Databases
: Developers who need to work with really huge amounts of data like to use databases, and the most recent browsers have started to integrate client-side database functionality into their browsers. Safari, Chrome, and Opera include a client-side API to a SQL database. The standardization effort for that API has failed, however, and it is unlikely to be implemented by Firefox or IE. An alternative database API is being standardized under the name "Indexed Database API." This is an API to a simple object database without a query language. Both of the client-side database APIs are asynchronous and require the use of event handlers, which makes them somewhat complicated. They are not documented in this chapter, but see §22.8 for an overview and an example of the IndexedDB API.

`Filesystem API`
: We saw in Chapter 18 that modern browsers support a File object that allows user-selected files to be uploaded through an XMLHttpRequest. Related draft standards define an API for obtaining a private local filesystem and for reading and writing files from and to that filesystem. These emerging APIs are described in §22.7. When they are more widely implemented, web applications will be able to use the kind of file-based storage mechanisms that are already familiar to many programmers.

Storage, Security, and Privacy

Web browsers often offer to remember web passwords for you, and they store them safely in encrypted form on the disk. But none of the forms of client-side data storage described in this chapter involve encryption: anything you save resides on the user's hard disk in unencrypted form. Stored data is therefore accessible to curious users who share access to the computer and to malicious software (such as spyware) that exists on the computer. For this reason, no form of client-side storage should ever be used for passwords, financial account numbers, or other similarly sensitive information. Remember: just because a user types something into a form field when interacting with your website doesn't mean that he wants a copy of that value stored on disk. Consider a credit card number as an example. This is sensitive information that people keep

hidden in their wallets. If you save this information using client-side persistence, it is almost as if you wrote the credit card number on a sticky note and stuck it to the user's keyboard.

Also, bear in mind that many web users mistrust websites that use cookies or other client-side storage mechanisms to do anything that resembles "tracking." Try to use the storage mechanisms discussed in this chapter to enhance a user's experience at your site; don't use them as a privacy-invading data collection mechanism. If too many sites abuse client-side storage, users will disable it or clear it frequently, which will defeat the purpose and cripple the sites that depend on it.

20.1 localStorage and sessionStorage

Browsers that implement the "Web Storage" draft specification define two properties on the Window object: `localStorage` and `sessionStorage`. Both properties refer to a Storage object—a persistent associative array that maps string keys to string values. Storage objects work much like regular JavaScript objects: simply set a property of the object to a string, and the browser will store that string for you. The difference between `localStorage` and `sessionStorage` has to do with *lifetime* and *scope*: how long the data is saved for and who the data is accessible to.

Storage lifetime and scope are explained in more detail below. First, however, let's look at some examples. The following code uses `localStorage`, but it would also work with `sessionStorage`:

```
var name = localStorage.username;          // Query a stored value.
name = localStorage["username"];           // Array notation equivalent
if (!name) {
    name = prompt("What is your name?");   // Ask the user a question.
    localStorage.username = name;          // Store the user's response.
}

// Iterate through all stored name/value pairs
for(var name in localStorage) {            // Iterate all stored names
    var value = localStorage[name];        // Look up the value of each one
}
```

Storage objects also define methods for storing, retrieving, iterating, and deleting data. Those methods are covered in §20.1.2.

The Web Storage draft specification says that we should be able to store structured data (objects and arrays) as well as primitive values and built-in types such as dates, regular expressions, and even File objects. At the time of this writing, however, browsers only allow the storage of strings. If you want to store and retrieve other kinds of data, you'll have to encode and decode it yourself. For example:

```
// If you store a number, it is automatically converted to a string.
// Don't forget to parse it when retrieving it from storage.
localStorage.x = 10;
var x = parseInt(localStorage.x);
```

```
// Convert a Date to a string when setting, and parse it when getting
localStorage.lastRead = (new Date()).toUTCString();
var lastRead = new Date(Date.parse(localStorage.lastRead));

// JSON makes a convenient encoding for any primitive or data structure
localStorage.data = JSON.stringify(data);  // Encode and store
var data = JSON.parse(localStorage.data);  // Retrieve and decode.
```

20.1.1 Storage Lifetime and Scope

The difference between `localStorage` and `sessionStorage` involves the lifetime and scope of the storage. Data stored through `localStorage` is permanent: it does not expire and remains stored on the user's computer until a web app deletes it or the user asks the browser (through some browser-specific UI) to delete it.

`localStorage` is scoped to the document origin. As explained in §13.6.2, the origin of a document is defined by its protocol, hostname, and port, so each of the following URLs has a different origin:

```
http://www.example.com          // Protocol: http; hostname: www.example.com
https://www.example.com         // Different protocol
http://static.example.com       // Different hostname
http://www.example.com:8000     // Different port
```

All documents with the same origin share the same `localStorage` data (regardless of the origin of the scripts that actually access `localStorage`). They can read each other's data. And they can overwrite each other's data. But documents with different origins can never read or overwrite each other's data (even if they're both running a script from the same third-party server).

Note that `localStorage` is also scoped by browser vendor. If you visit a site using Firefox, and then visit again using Chrome (for example), any data stored during the first visit will not be accessible during the second visit.

Data stored through `sessionStorage` has a different lifetime than data stored through `localStorage`: it has the same lifetime as the top-level window or browser tab in which the script that stored it is running. When the window or tab is permanently closed, any data stored through `sessionStorage` is deleted. (Note, however, that modern browsers have the ability to reopen recently closed tabs and restore the last browsing session, so the lifetime of these tabs and their associated `sessionStorage` may be longer than it seems.)

Like `localStorage`, `sessionStorage` is scoped to the document origin so that documents with different origins will never share `sessionStorage`. But `sessionStorage` is also scoped on a per-window basis. If a user has two browser tabs displaying documents from the same origin, those two tabs have separate `sessionStorage` data: the scripts running in one tab cannot read or overwrite the data written by scripts in the other tab, even if both tabs are visiting exactly the same page and are running exactly the same scripts.

Note that this window-based scoping of sessionStorage is only for top-level windows. If one browser tab contains two <iframe> elements, and those frames hold two documents with the same origin, those two framed documents will share sessionStorage.

20.1.2 Storage API

localStorage and sessionStorage are often used as if they were regular JavaScript objects: set a property to store a string and query the property to retrieve it. But these objects also define a more formal method-based API. To store a value, pass the name and value to setItem(). To retrieve a value, pass the name to getItem(). To delete a value, pass the name to removeItem(). (In most browsers you can also use the delete operator to remove a value, just as you would for an ordinary object, but this technique does not work in IE8.) To delete all stored values, call clear() (with no arguments). Finally, to enumerate the names of all stored values, use the length property and pass numbers from 0 to length-1 to the key() method. Here are some examples using localStorage. The same code would work using sessionStorage instead:

```
localStorage.setItem("x", 1);        // Store an number with the name "x"
localStorage.getItem("x");           // Retrieve a value

// Enumerate all stored name/value pairs
for(var i = 0; i < localStorage.length; i++) {  // Length gives the # of pairs
    var name = localStorage.key(i);             // Get the name of pair i
    var value = localStorage.getItem(name);     // Get the value of that pair
}

localStorage.removeItem("x");        // Delete the item "x"
localStorage.clear();                // Delete any other items, too
```

Although it is usually more convenient to store and retrieve values by setting and querying properties, there are some times when you might want to use these methods. First, the clear() method has no equivalent and is the only way to delete all name/value pairs in a Storage object. Similarly, the removeItem() method is the only portable way to delete a single name/value pair, since IE8 does not allow the delete operator to be used in that way.

If browser vendors fully implement the specification and allow objects and arrays to be stored in a Storage object, there will be another reason to use methods like setItem() and getItem(). Objects and array values are normally mutable, so a Storage object is required to make a copy when you store a value, so that any subsequent changes to the original value have no effect on the stored value. The Storage object is also required to make a copy when you retrieve a value so that any changes you make to the retrieved value have no effect on the stored value. When this kind of copying is being done, using the property-based API can be confusing. Consider this (hypothetical, until browsers support structured values) code:

```
localStorage.o = {x:1};   // Store an object that has a property x
localStorage.o.x = 2;     // Attempt to set the property of the stored object
localStorage.o.x          // => 1: x is unchanged
```

The second line of code above wants to set a property of the stored object, but instead it retrieves a copy of the stored object, sets a property in that copied object, and then discards the copy. The stored object remains unchanged. There would be less chance of confusion if we used `getItem()` here:

```
localStorage.getItem("o").x = 2; // We don't expect this to store the value 2
```

Finally, another reason to use the explicit method-based Storage API is that we can emulate that API on top of other storage mechanisms in browsers that do not yet support the Web Storage specification. The sections that follow will implement the Storage API using cookies and IE userData. If you use the method-based API, you can write code that makes use of `localStorage` when available and falls back on one of the other storage mechanisms in other browsers. Your code might start like this:

```
// Figure out what memory I'm using
var memory = window.localStorage ||
             (window.UserDataStorage && new UserDataStorage()) ||
             new CookieStorage();
// Then search my memory
var username = memory.getItem("username");
```

20.1.3 Storage Events

Whenever the data stored in `localStorage` or `sessionStorage` changes, the browser triggers a storage event on any other Window objects to which that data is visible (but not on the window that made the change). If a browser has two tabs open to pages with the same origin, and one of those pages stores a value in `localStorage`, the other tab will receive a storage event. Remember that `sessionStorage` is scoped to the top-level window, so storage events are only triggered for `sessionStorage` changes when there are frames involved. Also note that storage events are only triggered when storage actually changes. Setting an existing stored item to its current value does not trigger an event, nor does removing an item that does not exist in storage.

Register a handler for storage events with `addEventListener()` (or `attachEvent()` in IE). In most browsers, you can also set the `onstorage` property of the Window object, but at the time of this writing, Firefox does not support that property.

The event object associated with a storage event has five important properties (they are not supported by IE8, unfortunately):

`key`
: The name or key of the item that was set or removed. If the `clear()` method was called, this property will be `null`.

`newValue`
: Holds the new value of the item, or `null` if `removeItem()` was called.

`oldValue`
: Holds the old value of an existing item that changed or was deleted, or `null` if a new item was inserted.

`storageArea`
: This property will equal either the `localStorage` or the `sessionStorage` property of the target Window object.

`url`
: The URL (as a string) of the document whose script made this storage change.

Finally, note that `localStorage` and the storage event can serve as a broadcast mechanism by which a browser sends a message to all windows that are currently visiting the same website. If a user requests that a website stop performing animations, for example, the site might store that preference in `localStorage` so that it can honor it in future visits. And by storing the preference, it generates an event that allows other windows displaying the same site to honor the request as well. As another example, imagine a web-based image editing application that allows the user to display tool palettes in separate windows. When the user selects a tool, the application uses `localStorage` to save the current state and to generate a notification to other windows that a new tool has been selected.

20.2 Cookies

A *cookie* is a small amount of named data stored by the web browser and associated with a particular web page or website. Cookies were originally designed for server-side programming, and at the lowest level, they are implemented as an extension to the HTTP protocol. Cookie data is automatically transmitted between the web browser and web server, so server-side scripts can read and write cookie values that are stored on the client. This section demonstrates how client-side scripts can also manipulate cookies using the `cookie` property of the Document object.

Why "Cookie?"

The name "cookie" does not have a lot of significance, but it is not used without precedent. In the annals of computing history, the term "cookie" or "magic cookie" has been used to refer to a small chunk of data, particularly a chunk of privileged or secret data, akin to a password, that proves identity or permits access. In JavaScript, cookies are used to save state and can establish a kind of identity for a web browser. Cookies in JavaScript do not use any kind of cryptography, however, and are not secure in any way (although transmitting them across an `https:` connection helps).

The API for manipulating cookies is a very old one, which means that it is universally supported. Unfortunately, the API is also quite cryptic. There are no methods involved: cookies are queried, set, and deleted by reading and writing the `cookie` property of the Document object using specially formatted strings. The lifetime and scope of each cookie can be individually specified with cookie attributes. These attributes are also specified with specially formatted strings set on the same `cookie` property.

The subsections that follow explain the cookie attributes that specify lifetime and scope, and then demonstrate how to set and query cookie values in JavaScript. The section concludes with an example that implements the Storage API on top of cookies.

Determining Whether Cookies Are Enabled

Cookies have gotten a bad reputation for many web users because of the unscrupulous use of third-party cookies—cookies associated with the images on a web page rather than the web page itself. Third-party cookies enable an ad-hosting company to track a user from one client site to another client site, for example, and the privacy implications of this practice cause some users to disable cookies in their web browsers. Before using cookies in your JavaScript code, you may want to first check that they are enabled. In most browsers, you can do this by checking the `navigator.cookieEnabled` property. If `true`, cookies are enabled, and if `false`, cookies are disabled (although nonpersistent cookies that last for only the current browsing session may still be enabled). This is not a standard property, and if you find that it is undefined in the browser your code is running in, you must test for cookie support by trying to write, read, and delete a test cookie using the techniques explained below.

20.2.1 Cookie Attributes: Lifetime and Scope

In addition to a name and a value, each cookie has optional attributes that control its lifetime and scope. Cookies are transient by default; the values they store last for the duration of the web browser session but are lost when the user exits the browser. Note that this is a subtly different lifetime than `sessionStorage`: cookies are not scoped to a single window, and their default lifetime is the same as the entire browser process, not the lifetime of any one window. If you want a cookie to last beyond a single browsing session, you must tell the browser how long (in seconds) you would like it to retain the cookie by specifying a *max-age* attribute. If you specify a lifetime, the browser will store cookies in a file and delete them only once they expire.

Cookie visibility is scoped by document origin as `localStorage` and `sessionStorage` are, and also by document path. This scope is configurable through cookie attributes *path* and *domain*. By default, a cookie is associated with, and accessible to, the web page that created it and any other web pages in the same directory or any subdirectories of that directory. If the web page *http://www.example.com/catalog/index.html* creates a cookie, for example, that cookie is also visible to *http://www.example.com/catalog/order.html* and *http://www.example.com/catalog/widgets/index.html*, but it is not visible to *http://www.example.com/about.html*.

This default visibility behavior is often exactly what you want. Sometimes, though, you'll want to use cookie values throughout a website, regardless of which page creates the cookie. For instance, if the user enters his mailing address in a form on one page, you may want to save that address to use as the default the next time he returns to the page and also as the default in an entirely unrelated form on another page where he is asked to enter a billing address. To allow this usage, you specify a *path* for the cookie.

Then, any web page from the same web server whose URL begins with the path prefix you specified can share the cookie. For example, if a cookie set by *http://www.example.com/catalog/widgets/index.html* has its path set to "/catalog", that cookie is also visible to *http://www.example.com/catalog/order.html*. Or, if the path is set to "/", the cookie is visible to any page on the *http://www.example.com* web server.

Setting the *path* of a cookie to "/" gives scoping like that of `localStorage` and also specifies that the browser must transmit the cookie name and value to the server whenever it requests any web page on the site. Note that cookie *path* attribute should not be treated as any kind of access-control mechanism. If a web page wants to read the cookies from some other page of the same website, it can simply load that other page into a hidden `<iframe>` and read the cookies from the framed document. The same-origin policy (§13.6.2) prevents this kind of cookie-snooping from happening across sites, but it is perfectly legal for documents from the same site.

By default, cookies are scoped by document origin. Large websites may want cookies to be shared across subdomains, however. For example, the server at *order.example.com* may need to read cookie values set from *catalog.example.com*. This is where the *domain* attribute comes in. If a cookie created by a page on *catalog.example.com* sets its *path* attribute to "/" and its *domain* attribute to ".example.com", that cookie is available to all web pages on *catalog.example.com*, *orders.example.com*, and any other server in the *example.com* domain. If the *domain* attribute is not set for a cookie, the default is the hostname of the web server that serves the page. Note that you cannot set the domain of a cookie to a domain other than the domain of your server.

The final cookie attribute is a boolean attribute named *secure* that specifies how cookie values are transmitted over the network. By default, cookies are insecure, which means that they are transmitted over a normal, insecure HTTP connection. If a cookie is marked secure, however, it is transmitted only when the browser and server are connected via HTTPS or another secure protocol.

20.2.2 Storing Cookies

To associate a transient cookie value with the current document, simply set the `cookie` property to a string of the form:

```
name=value
```

For example:

```
document.cookie = "version=" + encodeURIComponent(document.lastModified);
```

The next time you read the `cookie` property, the name/value pair you stored is included in the list of cookies for the document. Cookie values cannot include semicolons, commas, or whitespace. For this reason, you may want to use the core JavaScript global function `encodeURIComponent()` to encode the value before storing it in the cookie. If you do this, you'll have to use the corresponding `decodeURIComponent()` function when you read the cookie value.

A cookie written with a simple name/value pair lasts for the current web-browsing session but is lost when the user exits the browser. To create a cookie that can last across browser sessions, specify its lifetime (in seconds) with a `max-age` attribute. You can do this by setting the `cookie` property to a string of the form:

```
name=value; max-age=seconds
```

The following function sets a cookie with an optional `max-age` attribute:

```
// Store the name/value pair as a cookie, encoding the value with
// encodeURIComponent() in order to escape semicolons, commas, and spaces.
// If daysToLive is a number, set the max-age attribute so that the cookie
// expires after the specified number of days. Pass 0 to delete a cookie.
function setCookie(name, value, daysToLive) {
    var cookie = name + "=" + encodeURIComponent(value);
    if (typeof daysToLive === "number")
        cookie += "; max-age=" + (daysToLive*60*60*24);
    document.cookie = cookie;
}
```

Similarly, you can set the `path`, `domain`, and `secure` attributes of a cookie by appending strings of the following format to the cookie value before that value is written to the `cookie` property:

```
; path=path
; domain=domain
; secure
```

To change the value of a cookie, set its value again using the same name, path, and domain along with the new value. You can change the lifetime of a cookie when you change its value by specifying a new `max-age` attribute.

To delete a cookie, set it again using the same name, path, and domain, specifying an arbitrary (or empty) value, and a `max-age` attribute of 0.

20.2.3 Reading Cookies

When you use the `cookie` property in a JavaScript expression, the value it returns is a string that contains all the cookies that apply to the current document. The string is a list of *name* = *value* pairs separated from each other by a semicolon and a space. The cookie *value* does not include any of the attributes that may have been set for the cookie. In order to make use of the `document.cookie` property, you must typically call the `split()` method to break it into individual name=value pairs.

Once you have extracted the value of a cookie from the `cookie` property, you must interpret that value based on whatever format or encoding was used by the cookie's creator. You might, for example, pass the cookie value to `decodeURIComponent()` and then to `JSON.parse()`.

Example 20-1 defines a `getCookie()` function that parses the `document.cookie` property and returns an object whose properties specify the name and values of the document's cookies.

Example 20-1. Parsing the document.cookies property

```
// Return the document's cookies as an object of name/value pairs.
// Assume that cookie values are encoded with encodeURIComponent().
function getCookies() {
    var cookies = {};           // The object we will return
    var all = document.cookie;  // Get all cookies in one big string
    if (all === "")             // If the property is the empty string
        return cookies;         // return an empty object
    var list = all.split("; "); // Split into individual name=value pairs
    for(var i = 0; i < list.length; i++) {  // For each cookie
        var cookie = list[i];
        var p = cookie.indexOf("=");        // Find the first = sign
        var name = cookie.substring(0,p);   // Get cookie name
        var value = cookie.substring(p+1);  // Get cookie value
        value = decodeURIComponent(value);  // Decode the value
        cookies[name] = value;              // Store name and value in object
    }
    return cookies;
}
```

20.2.4 Cookie Limitations

Cookies are intended for storage of small amounts of data by server-side scripts, and that data is transferred to the server each time a relevant URL is requested. The standard that defines cookies encourages browser manufacturers to allow unlimited numbers of cookies of unrestricted size but does not require browsers to retain more than 300 cookies total, 20 cookies per web server, or 4 KB of data per cookie (both name and value count toward this 4 KB limit). In practice, browsers allow many more than 300 cookies total, but the 4 KB size limit may still be enforced by some.

20.2.5 Storage with Cookies

Example 20-2 demonstrates how to implement the methods of the Storage API on top of cookies. Pass the desired *max-age* and *path* attributes to the `CookieStorage()` constructor, and then use the resulting object as you would use `localStorage` or `sessionStorage`. Note, though, that the example does not implement the storage event and it does not automatically store and retrieve values when you set and query properties of the CookieStorage object.

Example 20-2. Implementing the Storage API using cookies

```
/*
 * CookieStorage.js
 * This class implements the Storage API that localStorage and sessionStorage
 * do, but implements it on top of HTTP Cookies.
 */
function CookieStorage(maxage, path) {  // Arguments specify lifetime and scope

    // Get an object that holds all cookies
    var cookies = (function() { // The getCookies() function shown earlier
        var cookies = {};           // The object we will return
        var all = document.cookie;  // Get all cookies in one big string
```

```
    if (all === "")             // If the property is the empty string
        return cookies;         // return an empty object
    var list = all.split("; "); // Split into individual name=value pairs
    for(var i = 0; i < list.length; i++) {  // For each cookie
        var cookie = list[i];
        var p = cookie.indexOf("=");        // Find the first = sign
        var name = cookie.substring(0,p);   // Get cookie name
        var value = cookie.substring(p+1);  // Get cookie value
        value = decodeURIComponent(value);  // Decode the value
        cookies[name] = value;              // Store name and value
    }
    return cookies;
}());

// Collect the cookie names in an array
var keys = [];
for(var key in cookies) keys.push(key);

// Now define the public properties and methods of the Storage API

// The number of stored cookies
this.length = keys.length;

// Return the name of the nth cookie, or null if n is out of range
this.key = function(n) {
    if (n < 0 || n >= keys.length) return null;
    return keys[n];
};

// Return the value of the named cookie, or null.
this.getItem = function(name) { return cookies[name] || null; };

// Store a value
this.setItem = function(key, value) {
    if (!(key in cookies)) { // If no existing cookie with this name
        keys.push(key);      // Add key to the array of keys
        this.length++;       // And increment the length
    }

    // Store this name/value pair in the set of cookies.
    cookies[key] = value;

    // Now actually set the cookie.
    // First encode value and create a name=encoded-value string
    var cookie = key + "=" + encodeURIComponent(value);

    // Add cookie attributes to that string
    if (maxage) cookie += "; max-age=" + maxage;
    if (path) cookie += "; path=" + path;

    // Set the cookie through the magic document.cookie property
    document.cookie = cookie;
};

// Remove the specified cookie
this.removeItem = function(key) {
```

```
        if (!(key in cookies)) return;  // If it doesn't exist, do nothing

        // Delete the cookie from our internal set of cookies
        delete cookies[key];

        // And remove the key from the array of names, too.
        // This would be easier with the ES5 array indexOf() method.
        for(var i = 0; i < keys.length; i++) {  // Loop through all keys
            if (keys[i] === key) {              // When we find the one we want
                keys.splice(i,1);               // Remove it from the array.
                break;
            }
        }
        this.length--;                          // Decrement cookie length

        // Finally actually delete the cookie by giving it an empty value
        // and an immediate expiration date.
        document.cookie = key + "=; max-age=0";
    };

    // Remove all cookies
    this.clear = function() {
        // Loop through the keys, removing the cookies
        for(var i = 0; i < keys.length; i++)
            document.cookie = keys[i] + "=; max-age=0";
        // Reset our internal state
        cookies = {};
        keys = [];
        this.length = 0;
    };
}
```

20.3 IE userData Persistence

IE5 and later enable client-side storage by attaching a proprietary "DHTML behavior" to a document element. You can do that with code like this:

```
var memory = document.createElement("div");          // Create an element
memory.id = "_memory";                               // Give it a name
memory.style.display = "none";                       // Never display it
memory.style.behavior = "url('#default#userData')"; // Attach a magic behavior
document.body.appendChild(memory);                   // Add it to the document
```

Once you add the "userData" behavior to an element, that element gets new `load()` and `save()` methods. Call `load()` to load stored data. You must pass a string to this method—it is like a file name, identifying a particular batch of stored data. When data is loaded, the name/value pairs become available as attributes of the element, and you can query them with `getAttribute()`. To save new data, set attributes with `setAttribute()` and then call the `save()` method. To delete a value, use `removeAttribute()` and `save()`. Here is an example, using the `memory` element initialized above:

```
memory.load("myStoredData");                // Load a named batch of saved data
var name = memory.getAttribute("username"); // Get one piece of stored data
if (!name) {                                // If it wasn't defined
```

```
    name = prompt("What is your name?");     // Get user input
    memory.setAtttribute("username", name); // Set it as an attribute
    memory.save("myStoredData");            // And save it for next time
}
```

By default, data saved with userData has an indefinite lifetime and lasts until you delete it. But you can specify an expiration date by setting the `expires` property. For example, you might add the following lines to the previous code to specify an expiration date 100 days in the future:

```
var now = (new Date()).getTime();              // Now, in milliseconds
var expires = now + 100 * 24 * 60 * 60 * 1000; // 100 days from now in ms
expires = new Date(expires).toUTCString();     // Convert it to a string
memory.expires = expires;                      // Set userData expiration
```

IE userData is scoped to documents in the same directory as the document that set it. This is a narrower scope than cookies, which also make cookies available to documents in subdirectories of the original directory. The userData mechanism does not have any equivalent of the cookie *path* and *domain* attributes to widen the scope of the data.

userData allows much more data to be stored than cookies do, but much less than `localStorage` and `sessionStorage` do.

Example 20-3 implements the `getItem()`, `setItem()`, and `removeItem()` methods of the Storage API on top of IE's userData. (It does not implement `key()` or `clear()`, because userData does not define a way to iterate through all stored items.)

Example 20-3. A partial Storage API based on IE's userData

```
function UserDataStorage(maxage) {
    // Create a document element and install the special userData
    // behavior on it so it gets save() and load() methods.
    var memory = document.createElement("div");          // Create an element
    memory.style.display = "none";                       // Never display it
    memory.style.behavior = "url('#default#userData')"; // Attach magic behavior
    document.body.appendChild(memory);                   // Add to the document

    // If maxage is specified, expire the userData in maxage seconds
    if (maxage) {
        var now = new Date().getTime();     // The current time
        var expires = now + maxage * 1000;  // maxage seconds from now
        memory.expires = new Date(expires).toUTCString();
    }

    // Initialize memory by loading saved values.
    // The argument is arbitrary, but must also be passed to save()
    memory.load("UserDataStorage");                      // Load any stored data

    this.getItem = function(key) {     // Retrieve saved values from attributes
        return memory.getAttribute(key) || null;
    };
    this.setItem = function(key, value) {
        memory.setAttribute(key,value); // Store values as attributes
        memory.save("UserDataStorage"); // Save state after any change
    };
```

```
    this.removeItem = function(key) {
        memory.removeAttribute(key);    // Remove stored value attribute
        memory.save("UserDataStorage"); // Save new state
    };
}
```

Because the code in Example 20-3 can only work in IE, you might use IE conditional comments to prevent browsers other than IE from loading it:

```
<!--[if IE]>
<script src="UserDataStorage.js"></script>
<![endif]-->
```


20.4 Application Storage and Offline Webapps

HTML5 adds an "application cache" that web applications can use to store themselves locally in a user's browser. `localStorage` and `sessionStorage` store data for a web application, but the application cache stores the application itself—all the files (HTML, CSS, JavaScript, images, and so on) the application needs to run. The application cache is different from a web browser's regular cache: it isn't cleared when the user clears the regular cache. And cached applications aren't cleared out on a LRU (least-recently used) basis as they might be in an ordinary fixed-size cache. Applications aren't temporarily stored in the cache: they're installed there, and they remain there until they uninstall themselves or the user deletes them. The application cache is really not a cache at all: a better name would be "application storage."

The reason to allow web applications to be locally installed is to guarantee their availability when offline (such as when on an airplane or when a cellphone isn't receiving a signal). Web applications that work while offline install themselves in the application cache, use localStorage to store their data, and have a synchronization mechanism to transfer stored data to the server when they go back online. We'll see an example offline webapp in §20.4.3. First, however, we'll see how an application can install itself in the application cache.

20.4.1 The Application Cache Manifest

In order to install an application in the application cache, you must create a *manifest*: a file that lists all of the URLs the application requires. Then, simply link the main HTML page of your application to the manifest by setting the `manifest` attribute of the `<html>` tag:

```
<!DOCTYPE HTML>
<html manifest="myapp.appcache">
<head>...</head>
<body>...</body>
</html>
```

Manifest files must begin with the string "CACHE MANIFEST" as their first line. The lines that follow should list URLs to be cached, one URL per line. Relative URLs are

relative to the URL of the manifest file. Blank lines are ignored. Lines that begin with # are comments and are ignored. Comments may have space before them, but they cannot follow any nonspace characters on the same line. Here is a simple manifest file:

```
CACHE MANIFEST
# The line above identifies the file type. This line is a comment

# The lines below specify the resources the application needs to run
myapp.html
myapp.js
myapp.css
images/background.png
```

Cache Manifest MIME Type

By convention, application cache manifest files are given the filename extension *.appcache*. This is a convention only, however, and to actually identify the file type, the web server *must* serve a manifest with MIME type "text/cache-manifest". If the server sets the manifest's Content-Type header to any other MIME type, your application will not be cached. You may have to configure your web server to use this MIME type by, for example, creating an Apache *.htaccess* file in the web app directory.

It is the manifest file that serves as the identity of the cached application. If a web app has more than one web page (more than one HTML file that the user can link to), each one of these pages should use the `<html manifest=>` attribute to link to the manifest file. The fact that these pages all link to the same manifest file makes it clear that they are all to be cached together as part of the same web app. If there are only a few HTML pages in the application, it is conventional to list those pages explicitly in the manifest file. But this is not required: any file that links to the manifest file will be considered part of the web app and will be cached along with it.

A simple manifest like the one shown above must list *all* of the resources required by the web application. Once a web app has been downloaded the first time and cached, any subsequent loads will be from the cache. When an application is loaded from the cache, any resources it requires must be listed in the manifest. Resources that are not listed will not be loaded. This policy simulates being offline. If a simple cached application can run from the cache, it will also be able to run while the browser is offline. More complicated web apps cannot, in general, cache every single resource they require. They can still use the application cache if they have a more complex manifest.

20.4.1.1 Complex manifests

When an application is loaded from the application cache, only resources listed in its manifest file will be loaded. The example manifest file shown previously lists resources one URL at a time. Manifest files actually have a more complicated syntax than that example shows, and there are two other ways to list resources in a manifest file. Special section header lines are used to identify the type of manifest entry that follows the

header. Simple cache entries like those shown above go in a "CACHE:" section, which is the default section. The other two sections begin with the headers "NETWORK:" and "FALLBACK:". (A manifest can have any number of sections and can switch back and forth between sections as needed.)

The "NETWORK:" section specifies URLs that must never be cached and should always be retrieved from the network. You might list server-side scripts here, for example. The URLs in a "NETWORK:" section are actually URL prefixes. A resource whose URL begins with any one of these prefixes will be loaded from the network. If the browser is offline, that attempt will fail, of course. The "NETWORK:" section allows a wildcard URL "*". If you use this wildcard, the browser will attempt to load any resource not mentioned in the manifest from the network. This effectively defeats the rule that says that cached applications must list all their resources in the manifest.

The manifest entries in the "FALLBACK:" section include two URLs on each line. The second URL is loaded and is stored in the cache. The first URL is a prefix. URLs matching this prefix will not be cached, but they will be loaded from the network when possible. If the attempt to load such a URL fails, the cached resource specified by the second URL will be used instead. Imagine a web application that includes a number of video tutorials. Because these videos are very large, they are not appropriate to cache locally. For offline use, a manifest file could fall back on a text-based help file instead.

Here is a more complicated cache manifest:

```
CACHE MANIFEST

CACHE:
myapp.html
myapp.css
myapp.js

FALLBACK:
videos/ offline_help.html

NETWORK:
cgi/
```

20.4.2 Cache Updates

When you load a web application that has been cached, all of its files come directly from the cache. If the browser is online, it will also asynchronously check to see if the manifest file has changed. If it has changed, the new manifest file, and all the files it references, are downloaded and reinstalled in the application cache. Note that the browser does not check to see whether any of the cached files have changed: only the manifest. If you modify a cached JavaScript file, for example, and want sites that have cached your web app to update their cache, you must update the manifest. Since the list of files required by your app has not changed, the easiest way to do this is by updating a version number:

```
CACHE MANIFEST
# MyApp version 1 (change this number to make browsers redownload the files)
MyApp.html
MyApp.js
```

Similarly, if you want a web app to uninstall itself from the application cache, you should delete the manifest file on the server, so that requests for it return an HTTP 404 Not Found error, and modify your HTML file or files so that they no longer link to the manifest.

Note that the browser checks the manifest and updates the cache asynchronously, after (or while) loading the cached copy of an application. For simple web apps, this means that after you update the manifest, the user must load the application twice before he sees the new version: the first load loads the old version from the cache and then updates the cache. Then the second load loads the new version from the cache.

The browser fires a number of events during the cache update process, and you can register handlers to track the process and provide feedback to the user. For example:

```
applicationCache.onupdateready = function() {
    var reload = confirm("A new version of this application is available\n" +
                         "and will be used the next time you reload.\n" +
                         "Do you want to reload now?");
    if (reload) location.reload();
}
```

Note that this event handler is registered on the ApplicationCache object that is the value of the `applicationCache` property of the Window. Browsers that support an application cache will define this property. In addition to the updateready event shown above, there are seven other application cache events you can monitor. Example 20-4 shows simple handlers that display messages to the user informing them of the progress of the cache update and of the current cache status.

Example 20-4. Handling application cache events

```
// The event handlers below all use this function to display status messages.
// Since the handlers all display status messages this way, they return false
// to cancel the event and prevent the browser from displaying its own status.
function status(msg) {
    // Display the message in the document element with id "statusline"
    document.getElementById("statusline").innerHTML = msg;
    console.log(msg);  // And also in the console for debugging
}

// Each time the application is loaded, it checks its manifest file.
// The checking event is always fired first when this process begins.
window.applicationCache.onchecking = function() {
    status("Checking for a new version.");
    return false;
};

// If the manifest file has not changed, and the app is already cached,
// the noupdate event is fired and the process ends.
window.applicationCache.onnoupdate = function() {
```

```
    status("This version is up-to-date.")
    return false;
};

// If the application is not already cached, or if the manifest has changed,
// the browser downloads and caches everything listed in the manifest.
// The downloading event signals the start of this download process.
window.applicationCache.ondownloading = function() {
    status("Downloading new version");
    window.progresscount = 0;  // Used in the progress handler below
    return false;
};

// progress events are fired periodically during the downloading process,
// typically once for each file downloaded.
window.applicationCache.onprogress = function(e) {
    // The event object should be a progress event (like those used by XHR2)
    // that allows us to compute a completion percentage, but if not,
    // we keep count of how many times we've been called.
    var progress = "";
    if (e && e.lengthComputable) // Progress event: compute percentage
        progress = " " + Math.round(100*e.loaded/e.total) + "%"
    else                         // Otherwise report # of times called
        progress = " (" + ++progresscount + ")"

    status("Downloading new version" + progress);
    return false;
};

// The first time an application is downloaded into the cache, the browser
// fires the cached event when the download is complete.
window.applicationCache.oncached = function() {
    status("This application is now cached locally");
    return false;
};

// When an already-cached application is updated, and the download is complete
// the browser fires "updateready". Note that the user will still be seeing
// the old version of the application when this event arrives.
window.applicationCache.onupdateready = function() {
    status("A new version has been downloaded.  Reload to run it");
    return false;
};

// If the browser is offline and the manifest cannot be checked, an "error"
// event is fired. This also happens if an uncached application references
// a manifest file that does not exist
window.applicationCache.onerror = function() {
    status("Couldn't load manifest or cache application");
    return false;
};

// If a cached application references a manifest file that does not exist,
// an obsolete event is fired and the application is removed from the cache.
// Subsequent loads are done from the network rather than from the cache.
window.applicationCache.onobsolete = function() {
```

```
    status("This application is no longer cached. " +
           "Reload to get the latest version from the network.");
    return false;
};
```

Each time an HTML file with a `manifest` attribute is loaded, the browser fires a checking event and loads the manifest file from the network. The events that follow the checking event are different in different situations:

No update available
: If the application is already in the cache, and the manifest file has not changed, the browser fires a noupdate event.

Update available
: If an application is cached, and its manifest file has changed, the browser fires a downloading event and begins downloading and caching all the files listed in the manifest. As this download occurs, it fires progress events. And when the download is complete, it fires an updateready event.

First load of new application
: If the application is not yet in the cache, downloading and progress events are fired as they are for the cache update case above. Once this initial download is complete, however, the browser fires a cached event rather than an updateready event.

Browser is offline
: If the browser is offline, it cannot check the manifest and it fires an error event. This also happens when an application that has not yet been cached refers to a manifest file that does not exist.

Manifest not found
: If the browser is online and the application is already cached, but the manifest file returns a 404 Not Found error, it fires an obsolete event and removes the application from the cache.

Note that all of these events are cancelable. The handlers in Example 20-4 returns `false` to cancel the default action associated with the events. This prevents browsers from displaying their own cache status messages. (At the time of this writing, browsers do not display any such messages.)

As an alternative to the event handlers, an application can also use the `applicationCache.status` property to determine the cache status. There are six possible values for this property:

`ApplicationCache.UNCACHED (0)`
: This application does not have a `manifest` attribute: it is not cached.

`ApplicationCache.IDLE (1)`
: The manifest has been checked and this application is cached and up to date.

`ApplicationCache.CHECKING (2)`
: The browser is checking the manifest file.

`ApplicationCache.DOWNLOADING (3)`
The browser is downloading and caching files listed in the manifest.

`ApplicationCache.UPDATEREADY (4)`
A new version of the application has been downloaded and cached.

`ApplicationCache.OBSOLETE (5)`
The manifest no longer exists and the cache will be deleted.

The ApplicationCache object also defines two methods. `update()` explicitly invokes the cache update algorithm to check for a new version of the application. This causes the browser to run through the same manifest check (and fire the same events) as it does when an application is first loaded.

The `swapCache()` method is trickier. Recall that when the browser downloads and caches an updated version of an application, the user is still running the out-of-date version. If the user reloads the app, she'll see the new version. But if the user does not reload, the old version must still run correctly. And notice that the old version may still be loading resources from the cache: it might be using XMLHttpRequest to request files, for example, and these requests must be satisfied by files in the old version of the cache. Therefore, the browser must generally keep the old version of the cache until the user reloads the application.

The `swapCache()` method tells the browser that it can discard the old cache and satisfy any future requests from the new cache. Note that this does not reload the application: HTML files, images, scripts, and so on that have already been loaded are not changed. But any future requests will come from the new version of the cache. This can cause version-skew issues and is not generally a good idea unless your app is carefully designed to allow it. Imagine, for example, an application that does nothing but display a splash screen of some sort while the browser is checking the manifest. When it sees the noupdate event, it goes ahead and loads the application's start page. If it sees a downloading event, it displays appropriate progress feedback while the cache is updated. And when it gets an updateready event, it calls `swapCache()` and then loads the updated start page from the latest version of the cache.

Note that it only makes sense to call `swapCache()` when the `status` property has the value `ApplicationCache.UPDATEREADY` or `ApplicationCache.OBSOLETE`. (Calling `swapCache()` when the `status` is `OBSOLETE` discards the obsolete cache immediately, and satisfies all future requests via the network.) If you call `swapCache()` when `status` has any other value, it will throw an exception.

20.4.3 Offline Web Applications

An offline web application is one that installs itself in the application cache so that it is always available, even when the browser is offline. For the simplest cases—things like clocks and fractal generators—this is all a web app needs to do to become an offline web app. But most nontrivial web apps need to upload data to the server as well: even simple game apps might want to upload a user's high score to the server. Apps that

need to upload data to a server can be offline web apps if they use `localStorage` to store application data, and then upload it when an Internet connection is available. Synchronizing data between local storage and the server can be the trickiest part of converting a web app for offline use, especially when the user might be accessing the data from more than one device.

In order to work offline, a web application needs to be able to tell whether it is offline or online and needs to know when the state of the Internet connection changes. To check whether the browser is online, a web app can use the `navigator.onLine` property. And to detect changes in the connection state, it can register handlers for online and offline events on the Window object.

This chapter concludes with a simple offline web app that demonstrates these techniques. The app is called PermaNote—it is a simple note application that saves the user's text to `localStorage` and by uploading it to the server whenever an Internet connection is available.[1] PermaNote only allows the user to edit a single note, and it ignores authorization and authentication issues—it assumes the server has some way of distinguishing one user from another, but does not include any kind of login screen. The PermaNote implementation consists of three files. Example 20-5 is the cache manifest. It lists the other two files and specifies that the URL "note" should not be cached: that is the URL we use to read and write the note to the server.

Example 20-5. permanote.appcache

```
CACHE MANIFEST
# PermaNote v8
permanote.html
permanote.js
NETWORK:
note
```

Example 20-6 is the second PermaNote file: it is an HTML file that defines a very simple editor UI. It displays a `<textarea>` element with a row of buttons across the top and a status line for messages along the bottom. Notice that the `<html>` tag has a `manifest` attribute.

Example 20-6. permanote.html

```
<!DOCTYPE HTML>
<html manifest="permanote.appcache">
  <head>
    <title>PermaNote Editor</title>
    <script src="permanote.js"></script>
    <style>
    #editor { width: 100%; height: 250px; }
    #statusline { width: 100%; }
    </style>
```

1. This example was loosely inspired by Halfnote, by Aaron Boodman. Halfnote was one of the first offline web apps.

```
  </head>
  <body>
    <div id="toolbar">
      <button id="savebutton" onclick="save()">Save</button>
      <button onclick="sync()">Sync Note</button>
      <button onclick="applicationCache.update()">Update Application</button>
    </div>
    <textarea id="editor"></textarea>
    <div id="statusline"></div>
  </body>
</html>
```


Finally, Example 20-7 lists the JavaScript code that makes the PermaNote web application work. It defines a `status()` function for displaying messages on the status line, a `save()` function for saving the current version of the note to the server, and a `sync()` function for making sure that the server's copy and the local copy are in sync. The `save()` and `sync()` functions use scripted HTTP techniques from Chapter 18. (Interestingly, the `save()` function uses the HTTP "PUT" method instead of the much more common POST method.)

In addition to these three basic functions, Example 20-7 defines event handlers. In order to keep the local copy and the server's copy of the note synchronized, the app requires quite a few event handlers:

`onload`
: Try to sync with the server, in case there is a newer version of the note there, and once synchronization is complete, enable the editor window.

  The `save()` and `sync()` functions make HTTP requests, and they register an onload handler on the XMLHttpRequest object to be notified when the upload or download is complete.

`onbeforeunload`
: Save the current version of the note to the server if it has not been uploaded.

`oninput`
: Whenever the text in the `<textarea>` changes, save it in `localStorage`, and start a timer. If the user stops editing for 5 seconds, save the note to the server.

`onoffline`
: When the browser goes offline, display a message in the status line.

`ononline`
: When the browser comes back online, synchronize with the server, checking for a newer version, and saving the current version.

`onupdateready`
: If a new version of the cached application is ready, display a message in the status line to let the user know.

`onnoupdate`
: If the application cache has not changed, let the user know that he or she is running the current version.

With that overview of PermaNote's event-driven logic, here is Example 20-7.

Example 20-7. permanote.js

```
// Some variables we need throughout
var editor, statusline, savebutton, idletimer;

// The first time the application loads
window.onload = function() {
    // Initialize local storage if this is the first time
    if (localStorage.note == null) localStorage.note = "";
    if (localStorage.lastModified == null) localStorage.lastModified = 0;
    if (localStorage.lastSaved == null) localStorage.lastSaved = 0;

    // Find the elements that are the editor UI. Initialize global variables.
    editor = document.getElementById("editor");
    statusline = document.getElementById("statusline");
    savebutton = document.getElementById("savebutton");

    editor.value = localStorage.note; // Initialize editor with saved note
    editor.disabled = true;           // But don't allow editing until we sync

    // Whenever there is input in the textarea
    editor.addEventListener("input",
                            function (e) {
                                // Save the new value in localStorage
                                localStorage.note = editor.value;
                                localStorage.lastModified = Date.now();
                                // Reset the idle timer
                                if (idletimer) clearTimeout(idletimer);
                                idletimer = setTimeout(save, 5000);
                                // Enable the save button
                                savebutton.disabled = false;
                            },
                            false);

    // Each time the application loads, try to sync up with the server
    sync();
};

// Save to the server before navigating away from the page
window.onbeforeunload = function() {
    if (localStorage.lastModified > localStorage.lastSaved)
        save();
};

// If we go offline, let the user know
window.onoffline = function() { status("Offline"); }

// When we come online again, sync up.
window.ononline = function() { sync(); };

// Notify the user if there is a new version of this application available.
// We could also force a reload here with location.reload()
window.applicationCache.onupdateready = function() {
    status("A new version of this application is available. Reload to run it");
```

```
};

// Also let the user know if there is not a new version available.
window.applicationCache.onnoupdate = function() {
    status("You are running the latest version of the application.");
};

// A function to display a status message in the status line
function status(msg) { statusline.innerHTML = msg; }

// Upload the note text to the server (if we're online).
// Will be automatically called after 5 seconds of inactivity whenever
// the note has been modified.
function save() {
    if (idletimer) clearTimeout(idletimer);
    idletimer = null;

    if (navigator.onLine) {
        var xhr = new XMLHttpRequest();
        xhr.open("PUT", "/note");
        xhr.send(editor.value);
        xhr.onload = function() {
            localStorage.lastSaved = Date.now();
            savebutton.disabled = true;
        };
    }
}

// Check for a new version of the note on the server. If a newer
// version is not found, save the current version to the server.
function sync() {
   if (navigator.onLine) {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/note");
        xhr.send();
        xhr.onload = function() {
            var remoteModTime = 0;
            if (xhr.status == 200) {
                var remoteModTime = xhr.getResponseHeader("Last-Modified");
                remoteModTime = new Date(remoteModTime).getTime();
            }

            if (remoteModTime > localStorage.lastModified) {
                status("Newer note found on server.");
                var useit =
                    confirm("There is a newer version of the note\n" +
                            "on the server. Click Ok to use that version\n"+
                            "or click Cancel to continue editing this\n"+
                            "version and overwrite the server");
                var now = Date.now();
                if (useit) {
                    editor.value = localStorage.note = xhr.responseText;
                    localStorage.lastSaved = now;
                    status("Newest version downloaded.");
                }
                else
```

```
                status("Ignoring newer version of the note.");
            localStorage.lastModified = now;
        }
        else
            status("You are editing the current version of the note.");

        if (localStorage.lastModified > localStorage.lastSaved) {
            save();
        }

        editor.disabled = false;  // Re-enable the editor
        editor.focus();           // And put cursor in it
    }
}
else { // If we are currently offline, we can't sync
    status("Can't sync while offline");
    editor.disabled = false;
    editor.focus();
}
}
```

CHAPTER 21

Scripted Media and Graphics

This chapter describes how to use JavaScript to manipulate images, control audio and video streams, and draw graphics. §21.1 explains traditional JavaScript techniques for visual effects such as image rollovers in which one static image is replaced by another when the mouse pointer moves over it. §21.2 covers the HTML5 `<audio>` and `<video>` elements and their JavaScript APIs.

After these first two sections on images, audio and video, the chapter moves on to cover two powerful technologies for drawing client-side graphics. The ability to dynamically generate sophisticated graphics in the browser is important for several reasons:

- The code used to produce graphics on the client side is typically much smaller than the images themselves, creating a substantial bandwidth savings.
- Dynamically generating graphics from real-time data uses a lot of CPU cycles. Offloading this task to the client reduces the load on the server, potentially saving on hardware costs.
- Generating graphics on the client is consistent with modern web application architecture in which servers provide data and clients manage the presentation of that data.

§21.3 explains Scalable Vector Graphics, or SVG. SVG is an XML-based language for describing graphics, and SVG drawings can be created and scripted using JavaScript and the DOM. Finally, §21.4 covers the HTML5 `<canvas>` element and its full-featured JavaScript API for client-side drawing. The `<canvas>` element is a revolutionary technology, and this chapter covers it in detail.

21.1 Scripting Images

Web pages include images using the HTML `<img>` element. Like all HTML elements, an `<img>` element can be scripted: setting the `src` property to a new URL causes the browser to load (if necessary) and display a new image. (You can also script the width and height of an image, which will make the browser shrink or enlarge the image, but that technique is not demonstrated here.)

The ability to dynamically replace one image with another in an HTML document opens the door to a number of special effects. One common use for image replacement is to implement image rollovers, in which an image changes when the mouse pointer moves over it. When you make images clickable by placing them inside your hyperlinks, rollover effects are a powerful way to invite the user to click on the image. (Similar effects can be achieved without scripting using the CSS `:hover` pseudoclass to alter the background image of an element.) The following HTML fragment is a simple example: it creates an image that changes when the mouse moves over it:

```
<img src="images/help.gif"
     onmouseover="this.src='images/help_rollover.gif'"
     onmouseout="this.src='images/help.gif'">
```

The event handlers of the `<img>` element set the `src` property when the mouse moves over or out of the image. Image rollovers are strongly associated with clickability, so this `<img>` element should still be enclosed in an `<a>` element or given an `onclick` event handler.

In order to be useful, image rollovers (and similar effects) need to be responsive. This means that you need some way to ensure that the necessary images are "prefetched" into the browser's cache. Client-side JavaScript defines a special-purpose API for this purpose: to force an image to be cached, create an offscreen Image object using the `Image()` constructor. Next, load an image into it by setting the `src` property of this object to the desired URL. This image is not added to the document, so it does not become visible, but the browser nevertheless loads and caches the image data. Later, when the same URL is used for an onscreen image, it can be quickly loaded from the browser's cache, rather than slowly loaded over the network.

The image-rollover code fragment shown in the previous section did not prefetch the rollover image it used, so the user might notice a delay in the rollover effect the first time she moves the mouse over the image. To fix this problem, modify the code as follows:

```
<script>(new Image()).src = "images/help_rollover.gif";</script>
<img src="images/help.gif"
     onmouseover="this.src='images/help_rollover.gif'"
     onmouseout="this.src='images/help.gif'">
```

21.1.1 Unobtrusive Image Rollovers

The image rollover code just shown requires one `<script>` element and two JavaScript event-handler attributes to implement a single rollover effect. This is a perfect example of *obtrusive* JavaScript: the amount of JavaScript code is so large that it effectively obscures the HTML. Example 21-1 shows an unobtrusive alternative that allows you to create image rollovers by simply specifying a `data-rollover` attribute (see §15.4.3) on any `<img>` element. Note that this example uses the `onLoad()` function of Example 13-5. It also uses the `document.images[]` array (see §15.2.3) to find all `<img>` elements in the document.

Example 21-1. Unobtrusive Image Rollovers

```
/**
 * rollover.js: unobtrusive image rollovers.
 *
 * To create image rollovers, include this module in your HTML file and
 * use the data-rollover attribute on any <img> element to specify the URL of
 * the rollover image. For example:
 *
 *   <img src="normal_image.png" data-rollover="rollover_image.png">
 *
 * Note that this module requires onLoad.js
 */
onLoad(function() { // Everything in one anonymous function: no symbols defined
    // Loop through all images, looking for the data-rollover attribute
    for(var i = 0; i < document.images.length; i++) {
        var img = document.images[i];
        var rollover = img.getAttribute("data-rollover");
        if (!rollover) continue;  // Skip images without data-rollover

        // Ensure that the rollover image is in the cache
        (new Image()).src = rollover;

        // Define an attribute to remember the default image URL
        img.setAttribute("data-rollout", img.src);

        // Register the event handlers that create the rollover effect
        img.onmouseover = function() {
            this.src = this.getAttribute("data-rollover");
        };
        img.onmouseout = function() {
            this.src = this.getAttribute("data-rollout");
        };
    }
});
```

21.2 Scripting Audio and Video

HTML5 introduces <audio> and <video> elements that are, in theory, as easy to use as the <img> element. In HTML5-enabled browsers, you no longer need to use plug-ins (like Flash) to embed sounds and movies in your HTML documents:

```
<audio src="background_music.mp3"/>
<video src="news.mov" width=320 height=240/>
```

In practice, the use of these elements is trickier than this, since browser vendors have not been able to agree on a standard audio and video codec that all will support, so you typically end up using <source> elements to specify multiple media sources in different formats:

```
<audio id="music">
<source src="music.mp3" type="audio/mpeg">
<source src="music.ogg" type='audio/ogg; codec="vorbis"'>
</audio>
```

Note that <source> elements have no content: there is no closing </source> tag, and you do not need to end them with />

Browsers that support <audio> and <video> elements will not render these element's content. But browsers that do not support them do render their content, so you can put fallback content (such as an <object> element that invokes the Flash plug-in) inside:

```
<video id="news" width=640 height=480 controls preload>
  <!-- WebM format for Firefox and Chrome -->
  <source src="news.webm" type='video/webm; codecs="vp8, vorbis"'>
  <!-- H.264 format for IE and Safari -->
  <source src="news.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
  <!-- Fall back on the Flash plugin -->
  <object width=640 height=480 type="application/x-shockwave-flash"
          data="flash_movie_player.swf">
    <!-- Param elements here configure the Flash movie player you're using -->
    <!-- Text is the ultimate fallback content -->
    <div>video element not supported and Flash plugin not installed.</div>
  </object>
</video>
```

<audio> and <video> elements support a controls attribute. If present (or if the corresponding JavaScript property is set to true), they display a set of playback controls that includes play and pause buttons, a volume control, and so on. In addition, however, the <audio> and <video> elements expose an API that gives scripts the power to control media playback, and you can use this API to add simple sound effects to your web application or to create your own custom control panels for sound and video. Although their visual appearance is very different, the <audio> and <video> elements share essentially the same API (the only real difference between them is that the <video> element has width and height properties) and almost everything that follows in this section applies to both elements.

The Audio() constructor

<audio> elements don't have any visual appearance in the document unless you set the controls attribute. And just as you can create an offscreen image with the Image() constructor, the HTML5 media API allows you to create an offscreen audio element with the Audio() constructor, passing a source URL as the argument:

```
new Audio("chime.wav").play();  // Load and play a sound effect
```

The return value of the Audio() constructor is the same kind of object you'd get when querying an <audio> element from the document, or creating a new one with document.createElement("audio"). Note that this is an audio-only feature of the media API: there is no corresponding Video() constructor.

Despite the frustrating requirement to define media in multiple file formats, the ability to play audio and video natively in the browser, without the use of plug-ins, is a powerful new feature in HTML5. Note that the problem of media codecs and browser

compatibility is beyond the scope of this book. The subsections that follow focus only on the JavaScript API for working with audio and video streams.

21.2.1 Type Selection and Loading

If you want to test whether a media element can play a particular type of media, pass the media MIME type (possibly including a `codec` parameter) to the `canPlayType()` method. The element returns the empty string (a falsy value) if it cannot play that media type. Otherwise, it returns the string "maybe" or "probably". Because of the complicated nature of audio and video codecs, a player cannot in general be more certain than "probably" that it can play a particular media type without actually downloading the media and trying:

```
var a = new Audio();
if (a.canPlayType("audio/wav")) {
    a.src = "soundeffect.wav";
    a.play();
}
```

When you set the `src` property of a media element, it begins the process of loading that media. (That process won't go very far unless `preload` is "auto".) Setting `src` when some other media is loading or playing will abort the loading or playing of the old media. If you add `<source>` elements to a media element instead of setting the `src` attribute, the element cannot know when you have inserted a complete set of elements, and it will not begin choosing among the `<source>` elements and loading data until you explicitly call the `load()` method.

21.2.2 Controlling Media Playback

The most important methods of the `<audio>` and `<video>` elements are `play()` and `pause()`, which start and stop playback of the media:

```
// When the document has loaded, start playing some background music
window.addEventListener("load", function() {
                            document.getElementById("music").play();
                        }, false);
```

In addition to starting and stopping sound and video, you can skip (or "seek") to a desired location within the media by setting the `currentTime` property. This property specifies the time in seconds to which the player should skip, and it can be set while the media is playing or while it is paused. (The `initialTime` and `duration` properties give the legal range of values for `currentTime`; more about those properties follows.)

The `volume` property specifies playback volume as a number between 0 (silent) and 1 (maximum volume). The `muted` property can be set to `true` to mute playback or set to `false` to resume playing sound at the specified `volume` level.

The `playbackRate` property specifies the speed at which the media is played. A value of 1.0 is the normal speed. Values greater than 1 are "fast forward" and values between 0 and 1 are "slow motion." Negative values are supposed to play the sound or video

backward, but browsers do not support that feature at the time of this writing. <audio> and <video> elements also have a defaultPlaybackRate property. Whenever the play() method is invoked, the playbackRate property is set to defaultPlaybackRate.

Note that the currentTime, volume, muted, and playbackRate properties are not only for controlling media playback. If an <audio> or <video> element has the controls attribute, it displays player controls, giving the user control over playback. In that case, a script might query properties like muted and currentTime to discover how the media is being played.

The HTML attributes controls, loop, preload, and autoplay affect audio and video playback and can also be set and queried as JavaScript properties. controls specifies whether playback controls are displayed in the browser. Set this property to true to display controls or false to hide controls. The loop property is a boolean that specifies whether the media should play in a loop (true) or stop when it reaches the end (false). The preload property specifies whether (or how much) media content should be preloaded before the user starts playing the media. The value "none" means no data should be preloaded. The value "metadata" means that metadata such as duration, bitrate, and frame size should be loaded, but not media data itself. Browsers typically load metadata if you do not specify a preload attribute. The preload value "auto" means that the browser should preload as much of the media as it deems appropriate. Finally, the autoplay property specifies whether the media should begin to play automatically when a sufficient amount has been buffered. Setting autoplay to true obviously implies that the browser should preload media data.

21.2.3 Querying Media Status

<audio> and <video> elements have a number of read-only properties that describe the current state of the media and of the player. paused is true if the player is paused. seeking is true if the player is skipping to a new playback position. ended is true if the player has reached the end of the media and has stopped. (ended never becomes true if loop is true.)

The duration property specifies the duration of the media in seconds. If you query this property before the media metadata has been loaded, it returns NaN. For streaming media, such as Internet radio, with an indefinite duration, this property returns Infinity.

The initialTime property specifies the start time of the media, in seconds. For media clips of fixed duration, this is usually 0. For streaming media, this property gives the earliest time for which data is still buffered and which it is possible to seek back to. currentTime can never be set to less than initialTime.

Three other properties provide a finer-grained view of the media timeline and its playback and buffering status. The played property returns the time range or ranges that have been played. The buffered property returns the time range or ranges that are currently buffered, and the seekable property returns the time range or ranges that the

player can currently seek to. (You might use these properties to implement a progress bar that illustrates the currentTime and duration along with how much of the media has played and how much is buffered.)

played, buffered, and seekable are TimeRanges objects. Each object has a length property that specifies the number of ranges it represents and start() and end() methods that return the start and end times (in seconds) of a numbered range. In the most common case of a single contiguous range of times, you'd use start(0) and end(0). Assuming that no seeking has happened and media is buffered from the beginning, for example, you might use code like this to determine what percentage of a resource was buffered:

```
var percent_loaded = Math.floor(song.buffered.end(0) / song.duration * 100);
```

Finally, three more properties, readyState, networkState, and error, give low-level status details about <audio> and <video> elements. Each of the properties has a numeric value, and constants are defined for each of the legal values. Note that these constants are defined on the media object (or the error object) itself. You might use one in code like this:

```
if (song.readyState === song.HAVE_ENOUGH_DATA) song.play();
```

readyState specifies how much media data has been loaded, and therefore, how ready the element is to begin playing that data. The values for this property and their meanings are as follows:

Constant	Value	Description
HAVE_NOTHING	0	No media data or metadata has been loaded.
HAVE_METADATA	1	The media metadata has been loaded, but no data for the current playback position has been loaded. This means that you can query the duration of the media or the dimensions of a video and you can seek by setting currentTime, but the browser cannot currently play the media at currentTime.
HAVE_CURRENT_DATA	2	Media data for currentTime has been loaded, but not enough data has been loaded to allow the media to play. For video, this typically means that the current frame has loaded, but the next one has not. This state occurs at the end of a sound or movie.
HAVE_FUTURE_DATA	3	Enough media data has been loaded to begin playing, but it is likely not enough to play to the end of the media without pausing to download more data.
HAVE_ENOUGH_DATA	4	Enough media data has been loaded that the browser is likely to be able to play to the end without pausing.

The networkState property specifies whether (or why not) a media element is using the network:

Constant	Value	Description
NETWORK_EMPTY	0	The element has not started using the network. This would be the state before the src attribute was set, for example.
NETWORK_IDLE	1	The element is not currently loading data from the network. It might have loaded the complete resource, or might have buffered all the data it currently needs. Or it might have preload set to "none" and not yet have been asked to load or play the media.
NETWORK_LOADING	2	The element is currently using the network to load media data.
NETWORK_NO_SOURCE	3	The element was not able to find a media source that it is able to play.

When an error occurs in media loading or playback, the browser sets the error property of the <audio> or <video> element. If no error has occurred, error is null. Otherwise, it is an object with a numeric code property that describes the error. The error object also defines constants that describe the possible error codes:

Constant	Value	Description
MEDIA_ERR_ABORTED	1	The user asked the browser to stop loading the media
MEDIA_ERR_NETWORK	2	The media is of the right type, but a network error prevented it from being loaded.
MEDIA_ERR_DECODE	3	The media is of the right type, but an encoding error prevented it from being decoded and played.
MEDIA_ERR_SRC_NOT_SUPPORTED	4	The media specified by the src attribute is not a type that the browser can play.

You might use the error property with code like this:

```
if (song.error.code == song.error.MEDIA_ERR_DECODE)
    alert("Can't play song: corrupt audio data.");
```

21.2.4 Media Events

<audio> and <video> are fairly complex elements—they must respond to user interaction with their playback controls, to network activity, and even, during playback, to the simple passage of time—and we've just seen that these elements have quite a few properties that define their current state. Like most HTML elements, <audio> and <video> fire events whenever their state changes. Because these elements have such a complicated state, they can fire quite a few events.

The table below summarizes the 22 media events, loosely in the order in which they are likely to occur. There are no event registration properties for these events. Use the addEventListener() method of the <audio> or <video> element to register handler functions.

Event Type	Description
loadstart	Triggered when the element begins requesting media data. networkState is NETWORK_LOADING.
progress	Network activity is continuing to load media data. networkState is NETWORK_LOADING. Typically fired between 2 and 8 times per second.
loadedmetadata	The media metadata has been loaded, and the duration and dimensions of the media are ready. readyState has changed to HAVE_METADATA for the first time.
loadeddata	Data for the current playback position has loaded for the first time, and readyState has changed to HAVE_CURRENT_DATA.
canplay	Enough media data has loaded that playback can begin, but additional buffering is likely to be required. readyState is HAVE_FUTURE_DATA.
canplaythrough	Enough media data has loaded that the media can probably be played all the way through without pausing to buffer more data. readyState is HAVE_ENOUGH_DATA.
suspend	The element has buffered enough data and has temporarily stopped downloading. networkState has changed to NETWORK_IDLE.
stalled	The element is trying to load data, but no data is arriving. networkState remains at NETWORK_LOADING.
play	The play() method has been invoked, or the autoplay attribute has caused the equivalent. If sufficient data has loaded, this event will be followed by a playing event. Otherwise a waiting event will follow.
waiting	Playback cannot begin, or playback has stopped, because there is not enough data buffered. A playing event will follow when enough data is ready.
playing	The media has begun to play.
timeupdate	The currentTime property has changed. During normal playback, this event is fired between 4 and 60 times per second, possibly depending on system load and how long the event handlers are taking to complete.
pause	The pause() method was called and playback has been paused.
seeking	The script or user has requested that playback skip to an unbuffered portion of the media and playback has stopped while data loads. The seeking property is true.
seeked	The seeking property has changed back to false.
ended	Playback has stopped because the end of the media has been reached.
durationchange	The duration property has changed
volumechange	The volume or muted property has changed.
ratechange	The playbackRate or defaultPlaybackRate has changed.
abort	The element has stopped loading data, typically at the user's request. error.code is MEDIA_ERR_ABORTED.
error	A network or other error prevented media data from being loaded. error.code is a value other than MEDIA_ERR_ABORTED.
emptied	An error or abort has caused the networkState to return to NETWORK_EMPTY.

21.3 SVG: Scalable Vector Graphics

SVG is an XML grammar for graphics. The word "vector" in its name indicates that it is fundamentally different from raster image formats, such as GIF, JPEG, and PNG, that specify a matrix of pixel values. Instead, an SVG "image" is a precise, resolution-independent (hence "scalable") description of the steps necessary to draw the desired graphic. Here is what a simple SVG file looks like:

```
<!-- Begin an SVG figure and declare our namespace -->
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 1000 1000">  <!-- Coordinate system for figure -->
  <defs>                       <!-- Set up some definitions we'll use -->
    <linearGradient id="fade"> <!-- a color gradient named "fade" -->
      <stop offset="0%" stop-color="#008"/>    <!-- Start a dark blue -->
      <stop offset="100%" stop-color="#ccf"/>  <!-- Fade to light blue -->
    </linearGradient>
  </defs>
  <!-- Draw a rectangle with a thick black border and fill it with the fade -->
  <rect x="100" y="200" width="800" height="600"
     stroke="black" stroke-width="25" fill="url(#fade)"/>
</svg>
```

Figure 21-1 shows what this SVG file looks like when rendered graphically.

SVG is a large and moderately complex grammar. In addition to simple shape-drawing primitives, it includes support for arbitrary curves, text, and animation. SVG graphics can even incorporate JavaScript scripts and CSS stylesheets to add behavior and presentation information. This section shows how client-side JavaScript code (embedded in HTML, not in SVG) can dynamically draw graphics using SVG. It includes examples of SVG drawing but can only scratch the surface of what is possible with SVG. Full details about SVG are available in the comprehensive, but quite readable, specification. The specification is maintained by the W3C at *http://www.w3.org/TR/SVG/*. Note that the specification includes a complete Document Object Model for SVG documents. This section manipulates SVG graphics using the standard XML DOM and does not use the SVG DOM at all.

At the time of this writing, all current browsers except IE support SVG (and IE9 will support it). In the latest browsers, you can display SVG images using an ordinary `<img>` element. Some slightly older browsers (such as Firefox 3.6) do not support this and require the use of an `<object>` element:

```
<object data="sample.svg" type="image/svg+xml" width="100" height="100"/>
```

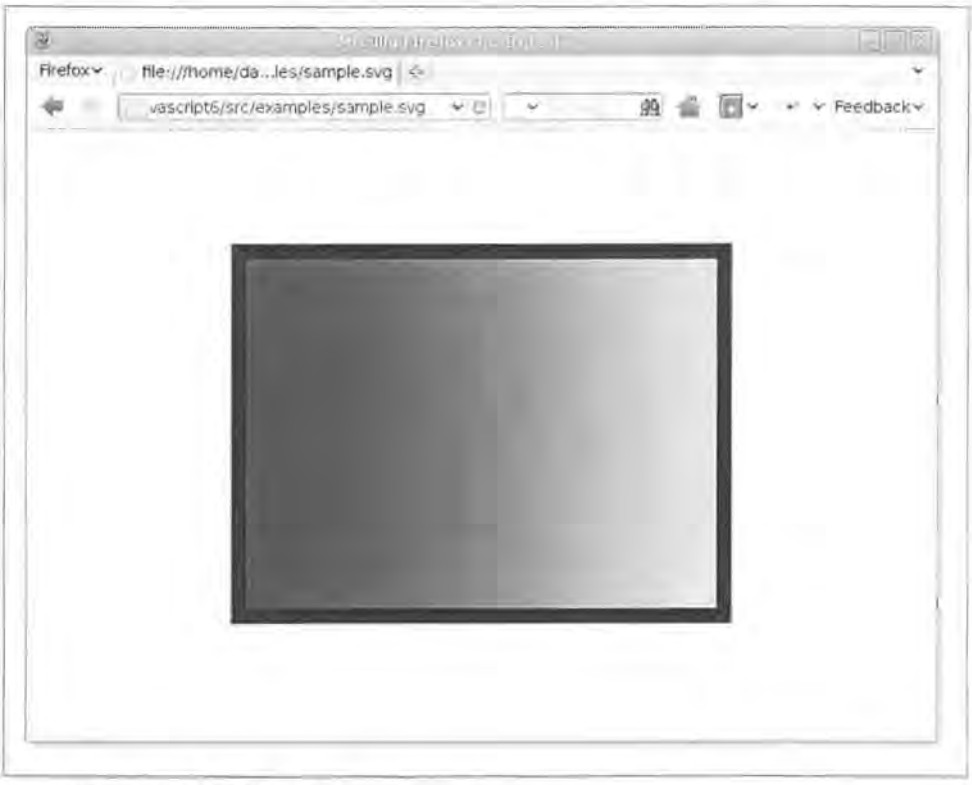


Figure 21-1. A simple SVG graphic

When used with an <img> or <object> element, SVG is just another image format, and it is not particularly interesting to JavaScript programmers. It is more useful to embed SVG images directly within your documents, so they can be scripted. Since SVG is an XML grammar, you can embed it within XHTML documents like this:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
<!-- declare HTML as default namespace and SVG with "svg:" prefix -->
<body>
This is a red square: <svg:svg width="10" height="10">
    <svg:rect x="0" y="0" width="10" height="10" fill="red"/>
</svg:svg>
This is a blue circle: <svg:svg width="10" height="10">
    <svg:circle cx="5" cy="5" r="5" fill="blue"/>
</svg:svg>
</body>
</html>
```

This technique works in all current browsers except IE. Figure 21-2 shows how Firefox renders this XHTML document.



Figure 21-2. SVG graphics in an XHTML document

HTML5 minimizes the distinction between XML and HTML and allows SVG (and MathML) markup to appear directly in HTML files, without namespace declarations or tag prefixes:

```
<!DOCTYPE html>
<html>
<body>
This is a red square: <svg width="10" height="10">
  <rect x="0" y="0" width="10" height="10" fill="red"/>
</svg>
This is a blue circle: <svg width="10" height="10">
  <circle cx="5" cy="5" r="5" fill="blue"/>
</svg>
</body>
</html>
```

At the time of this writing, direct embedding of SVG into HTML works only in the very newest browsers.

Since SVG is an XML grammar, drawing SVG graphics is simply a matter of using the DOM to create appropriate XML elements. Example 21-2 is a listing of a `pieChart()` function that creates the SVG elements to produce the pie chart shown in Figure 21-3.

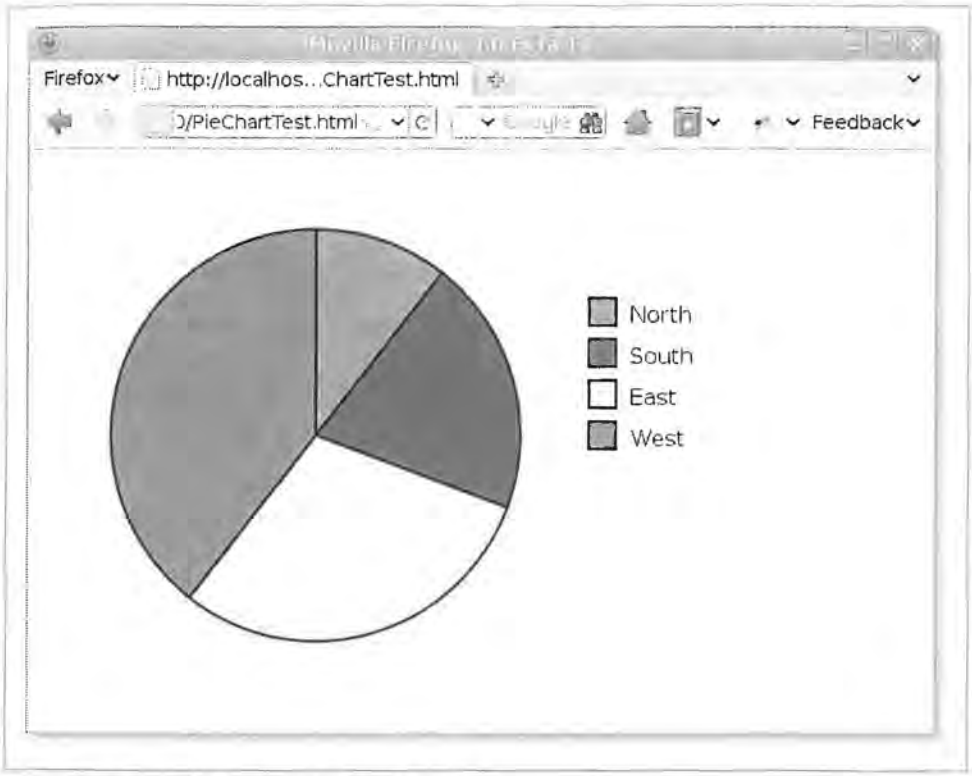


Figure 21-3. An SVG pie chart built with JavaScript

Example 21-2. Drawing a pie chart with JavaScript and SVG

```
/**
 * Create an <svg> element and draw a pie chart into it.
 * Arguments:
 *   data: an array of numbers to chart, one for each wedge of the pie.
 *   width,height: the size of the SVG graphic, in pixels
 *   cx, cy, r: the center and radius of the pie
 *   colors: an array of HTML color strings, one for each wedge
 *   labels: an array of labels to appear in the legend, one for each wedge
 *   lx, ly: the upper-left corner of the chart legend
 * Returns:
 *    An <svg> element that holds the pie chart.
 *    The caller must insert the returned element into the document.
 */
function pieChart(data, width, height, cx, cy, r, colors, labels, lx, ly) {
    // This is the XML namespace for svg elements
    var svgns = "http://www.w3.org/2000/svg";
```

```
// Create the <svg> element, and specify pixel size and user coordinates
var chart = document.createElementNS(svgns, "svg:svg");
chart.setAttribute("width", width);
chart.setAttribute("height", height);
chart.setAttribute("viewBox", "0 0 " + width + " " + height);

// Add up the data values so we know how big the pie is
var total = 0;
for(var i = 0; i < data.length; i++) total += data[i];

// Now figure out how big each slice of pie is. Angles in radians.
var angles = []
for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

// Loop through each slice of pie.
startangle = 0;
for(var i = 0; i < data.length; i++) {
    // This is where the wedge ends
    var endangle = startangle + angles[i];

    // Compute the two points where our wedge intersects the circle
    // These formulas are chosen so that an angle of 0 is at 12 o'clock
    // and positive angles increase clockwise.
    var x1 = cx + r * Math.sin(startangle);
    var y1 = cy - r * Math.cos(startangle);
    var x2 = cx + r * Math.sin(endangle);
    var y2 = cy - r * Math.cos(endangle);

    // This is a flag for angles larger than than a half circle
    // It is required by the SVG arc drawing component
    var big = 0;
    if (endangle - startangle > Math.PI) big = 1;

    // We describe a wedge with an <svg:path> element
    // Notice that we create this with createElementNS()
    var path = document.createElementNS(svgns, "path");

    // This string holds the path details
    var d = "M " + cx + "," + cy +  // Start at circle center
        " L " + x1 + "," + y1 +     // Draw line to (x1,y1)
        " A " + r + "," + r +       // Draw an arc of radius r
        " 0 " + big + " 1 " +       // Arc details...
        x2 + "," + y2 +             // Arc goes to to (x2,y2)
        " Z";                       // Close path back to (cx,cy)

    // Now set attributes on the <svg:path> element
    path.setAttribute("d", d);             // Set this path
    path.setAttribute("fill", colors[i]);  // Set wedge color
    path.setAttribute("stroke", "black");  // Outline wedge in black
    path.setAttribute("stroke-width", "2"); // 2 units thick
    chart.appendChild(path);               // Add wedge to chart

    // The next wedge begins where this one ends
    startangle = endangle;

    // Now draw a little matching square for the key
```

```
        var icon = document.createElementNS(svgns, "rect");
        icon.setAttribute("x", lx);              // Position the square
        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("width", 20);          // Size the square
        icon.setAttribute("height", 20);
        icon.setAttribute("fill", colors[i]);    // Same fill color as wedge
        icon.setAttribute("stroke", "black");    // Same outline, too.
        icon.setAttribute("stroke-width", "2");
        chart.appendChild(icon);                 // Add to the chart

        // And add a label to the right of the rectangle
        var label = document.createElementNS(svgns, "text");
        label.setAttribute("x", lx + 30);        // Position the text
        label.setAttribute("y", ly + 30*i + 18);
        // Text style attributes could also be set via CSS
        label.setAttribute("font-family", "sans-serif");
        label.setAttribute("font-size", "16");
        // Add a DOM text node to the <svg:text> element
        label.appendChild(document.createTextNode(labels[i]));
        chart.appendChild(label);                // Add text to the chart
    }

    return chart;
}
```

The code in Example 21-2 is relatively straightforward. There is a little math to convert the data being charted into pie-wedge angles. The bulk of the example, however, is DOM code that creates SVG elements and sets attributes on those elements. In order to work with browsers that do not fully support HTML5, this example treats SVG as an XML grammar and uses the SVG namespace and the DOM method `createElementNS()` instead of `createElement()`.

The most opaque part of this example is the code that draws the actual pie wedges. The element used to display each wedge is `<svg:path>`. This SVG element describes arbitrary shapes comprised of lines and curves. The shape description is specified by the `d` attribute of the `<svg:path>` element. The value of this attribute uses a compact grammar of letter codes and numbers that specify coordinates, angles, and other values. The letter M, for example, means "move to" and is followed by X and Y coordinates. The letter L means "line to" and draws a line from the current point to the coordinates that follow it. This example also uses the letter A to draw an arc. This letter is followed by seven numbers describing the arc. The precise details are not important here, but you can look them up in the specification at *http://www.w3.org/TR/SVG/*.

Note that the `pieChart()` function returns an `<svg>` element that contains a description of the pie chart, but it does not insert that element into the document. The caller is expected to do that. The pie chart in Figure 21-3 was created using a file like this:

```
<html>
<head>
<script src="PieChart.js"></script>
</head>
<body onload="document.body.appendChild(
```

```
                pieChart([12, 23, 34, 45], 640, 400, 200, 200, 150,
                         ['red','blue','yellow','green'],
                         ['North','South', 'East', 'West'], 400, 100));
            ">
    </body>
    </html>
```

Example 21-3 is another scripted SVG example: it uses SVG to display an analog clock. (See Figure 21-4.) Rather than dynamically building the tree of SVG elements from scratch, however, it starts with a static SVG image of a clock embedded in the HTML page. This static graphic includes two SVG `<line>` elements that represent the hour hand and the minute hand. Both lines point straight up, and the static image displays the time 12:00. To turn this image into a functioning clock, we use JavaScript to set a `transform` attribute on each of `<line>` elements, rotating them by the appropriate angles so that the clock displays the current time.



Figure 21-4. An SVG clock

Note that Example 21-3 embeds SVG markup directly into an HTML5 file and does not use XML namespaces within an XHTML file. This means that as shown here it will work only with browsers that support direct embedding of SVG. By converting the HTML file to XHTML, however, this same technique works in older SVG-capable browsers.

Example 21-3. Displaying the time by manipulating an SVG image

```
<!DOCTYPE HTML>
<html>
<head>
<title>Analog Clock</title>
<script>
function updateTime() { // Update the SVG clock graphic to show current time
    var now = new Date();                       // Current time
    var min = now.getMinutes();                 // Minutes
    var hour = (now.getHours() % 12) + min/60;  // Fractional hours
    var minangle = min*6;                       // 6 degrees per minute
    var hourangle = hour*30;                    // 30 degrees per hour
```

```
    // Get SVG elements for the hands of the clock
    var minhand = document.getElementById("minutehand");
    var hourhand = document.getElementById("hourhand");

    // Set an SVG attribute on them to move them around the clock face
    minhand.setAttribute("transform", "rotate(" + minangle + ",50,50)");
    hourhand.setAttribute("transform", "rotate(" + hourangle + ",50,50)");

    // Update the clock again in 1 minute
    setTimeout(updateTime, 60000);
}
</script>
<style>
/* These CSS styles all apply to the SVG elements defined below */
#clock {                          /* styles for everything in the clock */
   stroke: black;                 /* black lines */
   stroke-linecap: round;         /* with rounded ends */
   fill: #eef;                    /* on a light blue gray background */
}
#face { stroke-width: 3px;}       /* clock face outline */
#ticks { stroke-width: 2; }       /* lines that mark each hour */
#hourhand {stroke-width: 5px;}    /* wide hour hand */
#minutehand {stroke-width: 3px;}  /* narrow minute hand */
#numbers {                        /* how to draw the numbers */
    font-family: sans-serif; font-size: 7pt; font-weight: bold;
    text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body onload="updateTime()">
  <!-- viewBox is coordinate system, width and height are on-screen size -->
  <svg id="clock" viewBox="0 0 100 100" width="500" height="500">
    <defs>   <!-- Define a filter for drop-shadows -->
     <filter id="shadow" x="-50%" y="-50%" width="200%" height="200%">
        <feGaussianBlur in="SourceAlpha" stdDeviation="1" result="blur" />
        <feOffset in="blur" dx="1" dy="1" result="shadow" />
        <feMerge>
          <feMergeNode in="SourceGraphic"/><feMergeNode in="shadow"/>
        </feMerge>
      </filter>
    </defs>
    <circle id="face" cx="50" cy="50" r="45"/>  <!-- the clock face -->
    <g id="ticks">                              <!-- 12 hour tick marks -->
      <line x1='50' y1='5.000' x2='50.00' y2='10.00'/>
      <line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>
      <line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>
      <line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>
      <line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>
      <line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>
      <line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>
      <line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>
      <line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>
      <line x1='5.000' y1='50.00' x2='10.00' y2='50.00'/>
      <line x1='11.03' y1='27.50' x2='15.36' y2='30.00'/>
      <line x1='27.50' y1='11.03' x2='30.00' y2='15.36'/>
    </g>
```

```
    <g id="numbers">                    <!-- Number the cardinal directions-->
      <text x="50" y="18">12</text><text x="85" y="53">3</text>
      <text x="50" y="88">6</text><text x="15" y="53">9</text>
    </g>
    <!-- Draw hands pointing straight up. We rotate them in the code. -->
    <g id="hands" filter="url(#shadow)"> <!-- Add shadows to the hands -->
      <line id="hourhand" x1="50" y1="50" x2="50" y2="24"/>
      <line id="minutehand" x1="50" y1="50" x2="50" y2="20"/>
    </g>
  </svg>
</body>
</html>
```

21.4 Graphics in a <canvas>

The <canvas> element has no appearance of its own but creates a drawing surface within the document and exposes a powerful drawing API to client-side JavaScript. The canvas element is standardized by HTML5 but has been around for longer than that. It was introduced by Apple in Safari 1.3, and it has been supported by Firefox since version 1.5 and Opera since version 9. It is also supported in all versions of Chrome. The <canvas> element is not supported by IE before IE9, but it can be reasonably well emulated in IE6, 7, and 8 using the open source ExplorerCanvas project at *http://code.google.com/p/explorercanvas/*.

An important difference between the <canvas> element and SVG is that with the canvas you create drawings by calling methods and with SVG you create drawings by building a tree of XML elements. These two approaches are equivalently powerful: either one can be simulated with the other. On the surface, they are quite different, however, and each has its strengths and weaknesses. An SVG drawing, for example, is easily edited by removing elements from its description. To remove an element from the same graphic in a <canvas>, it is often necessary to erase the drawing and redraw it from scratch. Since the Canvas drawing API is JavaScript-based and relatively compact (unlike the SVG grammar), it is documented completely in this book. See Canvas, CanvasRenderingContext2D, and related entries in the client-side reference section.

Most of the Canvas drawing API is defined not on the <canvas> element itself, but instead on a "drawing context" object obtained with the getContext() method of the canvas. Call getContext() with the argument "2d" to obtain a CanvasRenderingContext2D object that you can use to draw two-dimensional graphics into the canvas. It is important to understand that the canvas element and its context object are two very different objects. Because it has such a long class name, I do not often refer to the CanvasRenderingContext2D object by name, instead simply calling it the "context object". Similarly, when I write about the "Canvas API," I usually mean "the methods of the CanvasRenderingContext2D object."

3D Graphics in a Canvas

At the time of this writing, browser vendors are starting to implement a 3D graphics API for the <canvas> element. The API is known as WebGL, and it is a JavaScript binding to the OpenGL standard API. To obtain a context object for 3D graphics, pass the string "webgl" to the `getContext()` method of the canvas. WebGL is a large, complicated, and low-level API that is not documented in this book: web developers are more likely to use utility libraries built on top of WebGL than to use the WebGL API directly.

As a simple example of the Canvas API, the following code draws a red square and blue circle into <canvas> elements to produce output like the SVG graphics shown in Figure 21-2 :

```
<body>
This is a red square: <canvas id="square" width=10 height=10></canvas>.
This is a blue circle: <canvas id="circle" width=10 height=10></canvas>.
<script>
var canvas = document.getElementById("square");  // Get first canvas element
var context = canvas.getContext("2d");           // Get 2D drawing context
context.fillStyle = "#f00";                      // Set fill color to red
context.fillRect(0,0,10,10);                     // Fill a square

canvas = document.getElementById("circle");      // Second canvas element
context = canvas.getContext("2d");               // Get its context
context.beginPath();                             // Begin a new "path"
context.arc(5, 5, 5, 0, 2*Math.PI, true);        // Add a circle to the path
context.fillStyle = "#00f";                      // Set blue fill color
context.fill();                                  // Fill the path
</script>
</body>
```

We've seen that SVG describes complex shapes as a "path" of lines and curves that can be drawn or filled. The Canvas API also uses the notion of a path. Instead of describing a path as a string of letters and numbers, a path is defined by a series of method calls, such as the `beginPath()` and `arc()` invocations in the code above. Once a path is defined, other methods, such as `fill()`, operate on that path. Various properties of the context object, such as `fillStyle`, specify how these operations are performed. The subsections that follow explain:

- How to define paths, how to draw or "stroke" the outline of a path, and how to fill the interior of a path.
- How to set and query the graphics attributes of the canvas context object, and how to save and restore the current state of those attributes.
- Canvas dimensions, the default canvas coordinate system, and how to transform that coordinate system.
- The various curve-drawing methods defined by the Canvas API.
- Some special-purpose utility methods for drawing rectangles.

- How to specify colors, work with transparency, and draw with color gradients and repeating image patterns.
- The attributes that control line width and the appearance of line endpoints and vertexes.
- How to draw text in a <canvas>.
- How to "clip" graphics so that no drawing is done outside of a region you specify.
- How to add drop shadows to your graphics.
- How to draw (and optionally scale) images into a canvas, and how to extract the contents of a canvas as an image.
- How to control the compositing process by which newly drawn (translucent) pixels are combined with the existing pixels in the canvas.
- How to query and set the raw red, green, blue, and alpha (transparency) values of the pixels in the canvas.
- How to determine whether a mouse event occurred above something you've drawn in a canvas.

The section ends with a practical example that uses <canvas> elements to render small inline charts known as *sparklines*.

Much of the <canvas> example code that follows operates on a variable c. This variable holds the CanvasRenderingContext2D object of the canvas, but the code to initialize that variable is not typically shown. In order to make these examples run, you would need to add HTML markup to define a canvas with appropriate `width` and `height` attributes, and then add code like this to initialize the variable c:

```
var canvas = document.getElementById("my_canvas_id");
var c = canvas.getContext('2d');
```

The figures that follow were all generated by JavaScript code drawing into a <canvas> element—typically into a large offscreen canvas to produce high-resolution print-quality graphics.

21.4.1 Drawing Lines and Filling Polygons

To draw lines on a canvas and to fill the areas enclosed by those lines, you begin by defining a *path*. A path is a sequence of one or more subpaths. A subpath is a sequence of two or more points connected by line segments (or, as we'll see later, by curve segments). Begin a new path with the `beginPath()` method. Begin a new subpath with the `moveTo()` method. Once you have established the starting point of a subpath with `moveTo()`, you can connect that point to a new point with a straight line by calling `lineTo()`. The following code defines a path that includes two line segments:

```
c.beginPath();       // Start a new path
c.moveTo(100, 100);  // Begin a subpath at (100,100)
c.lineTo(200, 200);  // Add a line from (100,100) to (200,200)
c.lineTo(100, 200);  // Add a line from (200,200) to (100,200)
```

The code above simply defines a path; it does not draw anything on the canvas. To draw (or "stroke") the two line segments in the path, call the `stroke()` method, and to fill the area defined by those line segments, call `fill()`:

```
c.fill();           // Fill a triangular area
c.stroke();         // Stroke two sides of the triangle
```

The code above (along with some additional code to set line widths and fill colors) produced the drawing shown in Figure 21-5.

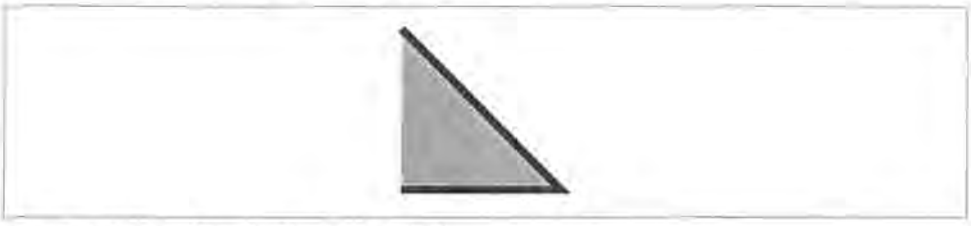


Figure 21-5. A simple path, filled and stroked

Notice that the subpath defined above is "open". It consists of just two line segments and the end point is not connected back to the starting point. This means that it does not enclose a region. The `fill()` method fills open subpaths by acting as if a straight line connected the last point in the subpath to the first point in the subpath. That is why the code above fills a triangle, but strokes only two sides of the triangle.

If you wanted to stroke all three sides of the triangle above, you would call the `closePath()` method to connect the end point of the subpath to the start point. (You could also call `lineTo(100,100)`, but then you end up with three line segments that share a start and end point but are not truly closed. When drawing with wide lines, the visual results are better if you use `closePath()`.)

There are two other important points to notice about `stroke()` and `fill()`. First, both methods operate on all subpaths in the current path. Suppose we had added another subpath in the code above:

```
c.moveTo(300,100);    // Begin a new subpath at (300,100);
c.lineTo(300,200);    // Draw a vertical line down to (300,200);
```

If we then called `stroke()`, we would draw two connected edges of a triangle and a disconnected vertical line.

The second point to note about `stroke()` and `fill()` is that neither one alters the current path: you can call `fill()` and the path will still be there when you call `stroke()`. When you are done with a path and want to begin another, you must remember to call `beginPath()`. If you don't, you'll end up adding new subpaths to the existing path and you may end up drawing those old subpaths over and over again.

Example 21-4 defines a function for drawing regular polygons and demonstrates the use of `moveTo()`, `lineTo()`, and `closePath()` for defining subpaths and of `fill()` and `stroke()` for drawing those paths. It produces the drawing shown in Figure 21-6.

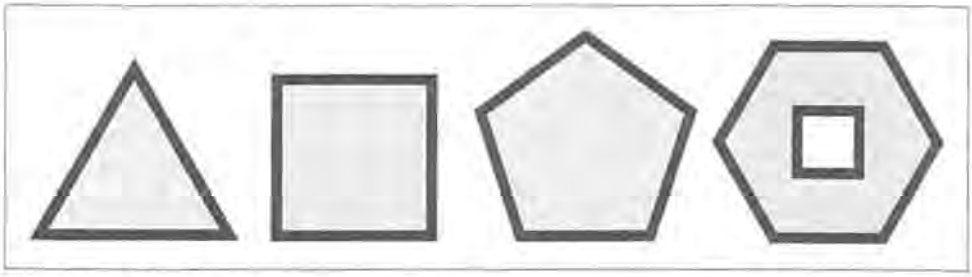


Figure 21-6. Regular polygons

Example 21-4. Regular polygons with moveTo(), lineTo() and closePath()

```
// Define a regular polygon with n sides, centered at (x,y) with radius r.
// The vertices are equally spaced along the circumference of a circle.
// Put the first vertex straight up or at the specified angle.
// Rotate clockwise, unless the last argument is true.
function polygon(c,n,x,y,r,angle,counterclockwise) {
    angle = angle || 0;
    counterclockwise = counterclockwise || false;
    c.moveTo(x + r*Math.sin(angle),  // Begin a new subpath at the first vertex
             y - r*Math.cos(angle)); // Use trigonometry to compute position
    var delta = 2*Math.PI/n;         // Angular distance between vertices
    for(var i = 1; i < n; i++) {     // For each of the remaining vertices
        angle += counterclockwise?-delta:delta; // Adjust angle
        c.lineTo(x + r*Math.sin(angle),         // Add line to next vertex
                 y - r*Math.cos(angle));
    }
    c.closePath();                   // Connect last vertex back to the first
}

// Start a new path and add polygon subpaths
c.beginPath();
polygon(c, 3, 50, 70, 50);                  // Triangle
polygon(c, 4, 150, 60, 50, Math.PI/4);      // Square
polygon(c, 5, 255, 55, 50);                 // Pentagon
polygon(c, 6, 365, 53, 50, Math.PI/6);      // Hexagon
polygon(c, 4, 365, 53, 20, Math.PI/4, true); // Small square inside the hexagon

// Set some properties that control how the graphics will look
c.fillStyle = "#ccc";    // Light gray interiors
c.strokeStyle = "#008"; // outlined with dark blue lines
c.lineWidth = 5;         // five pixels wide.

// Now draw all the polygons (each in its own subpath) with these calls
c.fill();                // Fill the shapes
c.stroke();              // And stroke their outlines
```

Notice that this example draws a hexagon with a square inside it. The square and the hexagon are separate subpaths, but they overlap. When this happens (or when a single subpath intersects itself), the canvas needs to be able to determine which regions are inside the path and which are outside. The canvas uses a test known as the "nonzero winding rule" to achieve this. In this case, the interior of the square is not filled because the square and the hexagon were drawn in the opposite directions: the vertexes of the hexagon were connected with line segments moving clockwise around the circle. The vertexes of the square were connected counterclockwise. Had the square been drawn clockwise as well, the call to `fill()` would have filled the interior of the square as well.

The Nonzero Winding Rule

To test whether a point P is inside a path, using the nonzero winding rule, imagine a ray drawn from P, in any direction, off to infinity (or, more practically, to some point outside of the path's bounding box). Now initialize a counter to zero and enumerate all places where the path crosses the ray. Each time the path crosses the ray in a clockwise direction, add one to the count. Each time the path crosses the ray counterclockwise, subtract one from the count. If, after all crossings have been enumerated, the count is nonzero, the point P is inside the path. If, on the other hand, the count is zero, P is outside the path.

21.4.2 Graphics Attributes

Example 21-4 set the properties `fillStyle`, `strokeStyle`, and `lineWidth` on the context object of the canvas. These properties are graphics attributes that specify the color to be used by `fill()`, the color to be used by `stroke()`, and the width of the lines to be drawn by `stroke()`. Notice that these parameters are not passed to the `fill()` and `stroke()` methods, but are instead part of the general *graphics state* of the canvas. If you define a method that draws a shape and do not set these properties yourself, the caller of your method can define the color of the shape by setting the `strokeStyle` and `fillStyle` properties before calling your method. This separation of graphics state from drawing commands is fundamental to the Canvas API and is akin to the separation of presentation from content achieved by applying CSS stylesheets to HTML documents.

The Canvas API defines 15 graphics attribute properties on the CanvasRenderingContext2D object. These properties are listed in Table 21-1 and explained in detail in the relevant sections below.

Table 21-1. Graphics attributes of the Canvas API

Property	Meaning
`fillStyle`	the color, gradient, or pattern for fills
`font`	the CSS font for text-drawing commands
`globalAlpha`	transparency to be added to all pixels drawn
`globalCompositeOperation`	how to combine new pixels with the ones underneath
`lineCap`	how the ends of lines are rendered
`lineJoin`	how vertexes are rendered
`lineWidth`	the width of stroked lines
`miterLimit`	maximum length of acute mitered vertexes
`textAlign`	horizontal alignment of text
`textBaseline`	vertical alignment of text
`shadowBlur`	how crisp or fuzzy shadows are
`shadowColor`	the color of drop shadows
`shadowOffsetX`	the horizontal offset of shadows
`shadowOffsetY`	the vertical offset of shadows
`strokeStyle`	the color, gradient, or pattern for lines

Since the Canvas API defines graphics attributes on the context object, you might be tempted to call `getContext()` multiple times to obtain multiple context objects. If you could do this, you could define different attributes on each context: each context would then be like a different brush and would paint with a different color or draw lines of different widths. Unfortunately, you cannot use the canvas in this way. Each `<canvas>` element has only a single context object, and every call to `getContext()` returns the same CanvasRenderingContext2D object.

Although the Canvas API only allows you to define a single set of graphics attributes at a time, it does allow you to save the current graphics state so that you can alter it and then easily restore it later. The `save()` method pushes the current graphics state onto a stack of saved states. The `restore()` method pops the stack and restores the most recently saved state. All of the properties listed in Table 21-1 are part of the saved state, as are the current transformation and clipping region (both are explained below). Importantly, the currently defined path and the current point are not part of the graphics state and cannot be saved and restored.

If you need more flexibility than a simple stack of graphics states allows, you may find it helpful to define utility methods like the ones shown in Example 21-5.

Example 21-5. Graphics state management utilities

```
// Revert to the last saved graphics state, but don't pop the stack.
CanvasRenderingContext2D.prototype.revert = function() {
    this.restore();  // Restore the old graphics state.
    this.save();     // Save it again so we can go back to it.
    return this;     // Allow method chaining.
};

// Set the graphics attributes specified by the properties of the object o.
// Or, if no argument is passed, return the current attributes as an object.
// Note that this does not handle the transformation or clipping region.
CanvasRenderingContext2D.prototype.attrs = function(o) {
    if (o) {
        for(var a in o)       // For each property in o
            this[a] = o[a];   // Set it as a graphics attribute
        return this;          // Enable method chaining
    }
    else return {
        fillStyle: this.fillStyle, font: this.font,
        globalAlpha: this.globalAlpha,
        globalCompositeOperation: this.globalCompositeOperation,
        lineCap: this.lineCap, lineJoin: this.lineJoin,
        lineWidth: this.lineWidth, miterLimit: this.miterLimit,
        textAlign: this.textAlign, textBaseline: this.textBaseline,
        shadowBlur: this.shadowBlur, shadowColor: this.shadowColor,
        shadowOffsetX: this.shadowOffsetX, shadowOffsetY: this.shadowOffsetY,
        strokeStyle: this.strokeStyle
    };
};
```

21.4.3 Canvas Dimensions and Coordinates

The `width` and `height` attributes of the `<canvas>` element and the corresponding `width` and `height` properties of the Canvas object specify the dimensions of the canvas. The default canvas coordinate system places the origin (0,0) at the upper left corner of the canvas. X coordinates increase to the right and Y coordinates increase as you go down the screen. Points on the canvas can be specified using floating-point values, and these are not automatically rounded to integers—the Canvas uses anti-aliasing techniques to simulate partially filled pixels.

The dimensions of a canvas are so fundamental that they cannot be altered without completely resetting the canvas. Setting either the `width` or `height` properties of a Canvas (even setting them to their current value) clears the canvas, erases the current path, and resets all graphics attributes (including current transformation and clipping region) to their original state.

Despite this fundamental importance of canvas dimensions, they do not necessarily match either the on-screen size of the canvas or the number of pixels that make up the canvas drawing surface. Canvas dimensions (and also the default coordinate system) are measured in CSS pixels. CSS pixels are usually the same thing as regular pixels. On high-resolution displays, however, implementations are allowed to map multiple

device pixels to single CSS pixels. This means that the rectangle of pixels that the canvas draws into may be larger than the canvas's nominal dimensions. You need to be aware of this when working with the pixel-manipulation features (see §21.4.14) of the canvas, but the distinction between virtual CSS pixels and actual hardware pixels does not otherwise have any effect on the canvas code you write.

By default, a `<canvas>` is displayed on-screen at the size (in CSS pixels) specified by its HTML `width` and `height` attributes. Like any HTML element, however, a `<canvas>` element can have its on-screen size specified by CSS `width` and `height` style attributes. If you specify an on-screen size that is different than the actual dimensions of the canvas, the pixels of the canvas are automatically scaled as needed to fit the screen dimensions specified by the CSS attributes. The on-screen size of the canvas does not affect the number of CSS or hardware pixels reserved in the canvas bitmap and the scaling that is done is an image scaling operation. If the on-screen dimensions are substantially larger than the actual dimensions of the canvas, this results in pixelated graphics. This is an issue for graphic designers and does not affect canvas programming.

21.4.4 Coordinate System Transforms

As noted above, the default coordinate system of a canvas places the origin in the upper left corner, has X coordinates increasing to the right, and has Y coordinates increasing downward. In this default system, the coordinates of a point map directly to a CSS pixel (which then maps directly to one or more device pixels). Certain canvas operations and attributes (such as extracting raw pixel values and setting shadow offsets) always use this default coordinate system. In addition to the default coordinate system, however, every canvas has a "current transformation matrix" as part of its graphics state. This matrix defines the current coordinate system of the canvas. In most canvas operations, when you specify the coordinates of a point, it is taken to be a point in the current coordinate system, not in the default coordinate system. The current transformation matrix is used to convert the coordinates you specified to the equivalent coordinates in the default coordinate system.

The `setTransform()` method allows you to set a canvas's transformation matrix directly, but coordinate system transformations are usually easier to specify as a sequence of translations, rotations, and scaling operations. Figure 21-7 illustrates these operations and their effect on the canvas coordinate system. The program that produced the figure drew the same set of axes seven times in a row. The only thing that changed each time was the current transform. Notice that the transforms affect the text as well as the lines that are drawn.

The `translate()` method simply moves the origin of the coordinate system left, right, up, or down. The `rotate()` method rotates the axes clockwise by the specified angle. (The Canvas API always specifies angles in radians. To convert degrees to radians, divide by 180 and multiply by `Math.PI`.) The `scale()` method stretches or contracts distances along the X or Y axes.

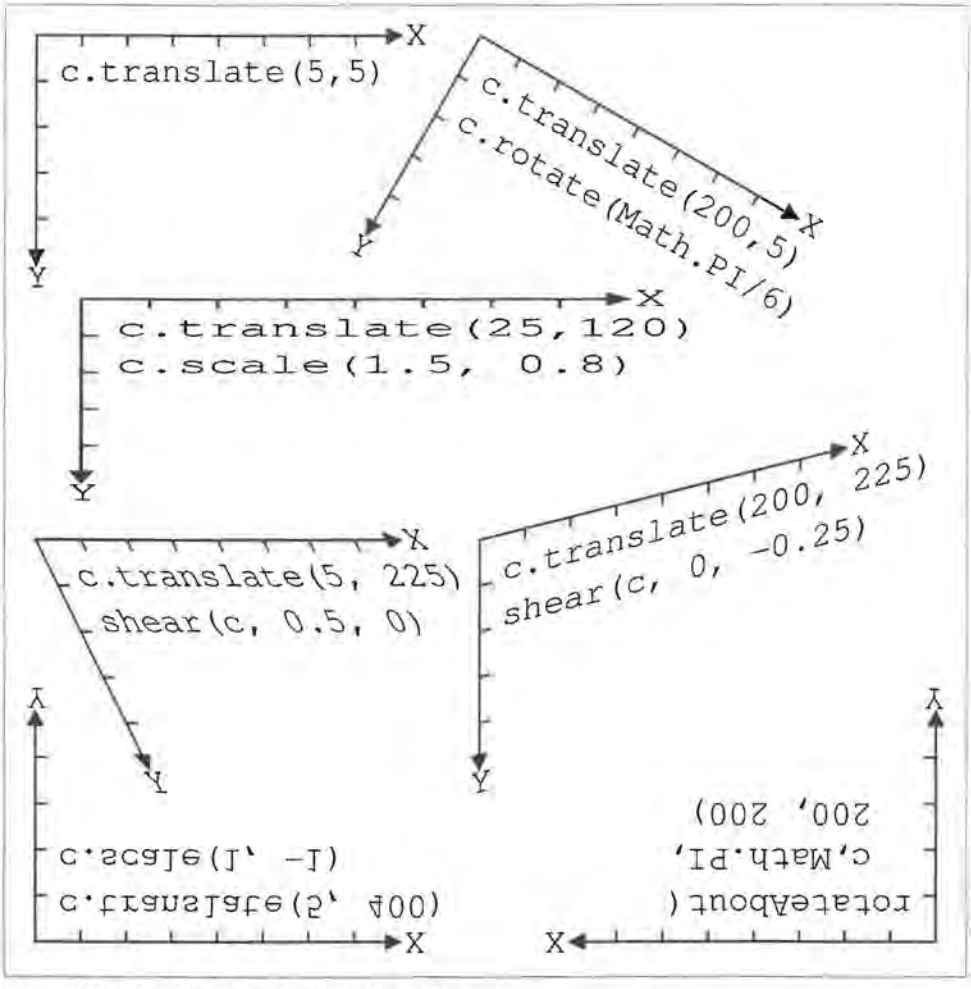


Figure 21-7. Coordinate system transformations

Passing a negative scale factor to the `scale()` method flips that axis across the origin, as if it were reflected in a mirror. This is what was done in the lower left of Figure 21-7: `translate()` was used to move the origin to the bottom left corner of the canvas, and then `scale()` was used to flip the Y axis around so that Y coordinates increase as we go up the page. A flipped coordinate system like this is familiar from algebra class and may be useful for plotting data points on charts. Note, however, that it makes text difficult to read!

21.4.4.1 Understanding transformations mathematically

I find it easiest to understand transforms geometrically, thinking about `translate()`, `rotate()`, and `scale()` as transforming the axes of the coordinate system as illustrated in Figure 21-7. It is also possible to understand transforms algebraically, as equations

that map the coordinates of a point (x,y) in the transformed coordinate system back to the coordinates of the same point (x',y') in the previous coordinate system.

The method call c.translate(dx,dy) can be described with these equations:

```
x' = x + dx;  // An X coordinate of 0 in the new system is dx in the old
y' = y + dy;
```

Scaling operations have similarly simple equations. A call c.scale(sx,sy) can be described like this:

```
x' = sx * x;
y' = sy * y;
```

Rotations are more complicated. The call c.rotate(a) is described by these trigonometric equations:

```
x' = x * cos(a) - y * sin(a);
y' = y * cos(a) + x * sin(a);
```

Notice that the order of transformations matters. Suppose we start with the default coordinate system of a canvas, and then translate it, and then scale it. In order to map the point (x,y) in the current coordinate system back to the point (x'',y'') in the default coordinate system, we must first apply the scaling equations to map the point to an intermediate point (x', y') in the translated but unscaled coordinate system, and then use the translation equations to map from this intermediate point to (x'',y''). The result is this:

```
x'' = sx*x + dx;
y'' = sy*y + dy;
```

If, on the other hand, we'd called scale() before calling translate(), the resulting equations would be different:

```
x'' = sx*(x + dx);
y'' = sy*(y + dy);
```

The key thing to remember when thinking algebraically about sequences of transformations is that you must work backward from the last (most recent) transformation to the first. When thinking geometrically about transformed axes, however, you work forward from first transformation to last.

The transformations supported by the canvas are known as *affine transforms*. Affine transforms may modify the distances between points and the angles between lines, but parallel lines always remain parallel after an affine transformation—it is not possible, for example, to specify a fish-eye lens distortion with an affine transform. An arbitrary affine transform can be described by the six parameters a through f in these equations:

```
x' = ax + cy + e
y' = bx + dy + f
```

You can apply an arbitrary transformation to the current coordinate system by passing those six parameters to the transform() method. Figure 21-7 illustrates two types of

transformations—shears and rotations about a specified point—that you can implement with the `transform()` method like this:

```
// Shear transform:
//   x' = x + kx*y;
//   y' = y + ky*x;
function shear(c,kx,ky) { c.transform(1, ky, kx, 1, 0, 0); }

// Rotate theta radians clockwise around the point (x,y)
// This can also be accomplished with a translate,rotate,translate sequence
function rotateAbout(c,theta,x,y) {
    var ct = Math.cos(theta), st = Math.sin(theta);
    c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);
}
```

The `setTransform()` method takes the same arguments as `transform()`, but instead of transforming the current coordinate system, it ignores the current system, transforms the default coordinate system, and makes the result the new current coordinate system. `setTransform()` is useful to temporarily reset the canvas to its default coordinate system:

```
c.save();                       // Save current coordinate system
c.setTransform(1,0,0,1,0,0);    // Revert to the default coordinate system
// Perform operations using default CSS pixel coordinates
c.restore();                    // Restore the saved coordinate system
```

21.4.4.2 Transformation example

Example 21-6 demonstrates the power of coordinate system transformations by using the `translate()`, `rotate()`, and `scale()` methods recursively to draw a Koch snowflake fractal. The output of this example appears in Figure 21-8, which shows Koch snowflakes with 0, 1, 2, 3, and 4 levels of recursion.

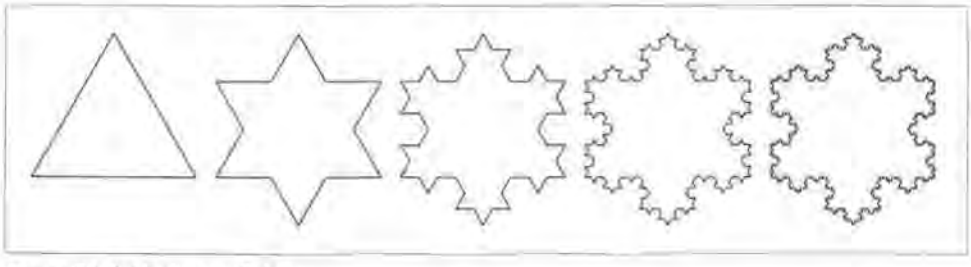


Figure 21-8. Koch snowflakes

The code that produces these figures is elegant, but its use of recursive coordinate system transformations makes it somewhat difficult to understand. Even if you don't follow all the nuances, note that the code includes only a single invocation of the `lineTo()` method. Every single line segment in Figure 21-8 is drawn like this:

```
c.lineTo(len, 0);
```

The value of the variable len does not change during the execution of the program, so the position, orientation, and length of each of the line segments is determined by translations, rotations, and scaling operations.

Example 21-6. A Koch snowflake with transformations

```
var deg = Math.PI/180;  // For converting degrees to radians

// Draw a level-n Koch Snowflake fractal on the canvas context c,
// with lower-left corner at (x,y) and side length len.
function snowflake(c, n, x, y, len) {
    c.save();           // Save current transformation
    c.translate(x,y);   // Translate origin to starting point
    c.moveTo(0,0);      // Begin a new subpath at the new origin
    leg(n);             // Draw the first leg of the snowflake
    c.rotate(-120*deg); // Now rotate 120 degrees counterclockwise
    leg(n);             // Draw the second leg
    c.rotate(-120*deg); // Rotate again
    leg(n);             // Draw the final leg
    c.closePath();      // Close the subpath
    c.restore();        // And restore original transformation

    // Draw a single leg of a level-n Koch snowflake.
    // This function leaves the current point at the end of the leg it has
    // drawn and translates the coordinate system so the current point is (0,0).
    // This means you can easily call rotate() after drawing a leg.
    function leg(n) {
        c.save();               // Save the current transformation
        if (n == 0) {           // Nonrecursive case:
            c.lineTo(len, 0);   //   Just draw a horizontal line
        }                       //
        else {                  // Recursive case: draw 4 sub-legs like: ¯\/¯
            c.scale(1/3,1/3);   // Sub-legs are 1/3rd the size of this leg
            leg(n-1);           // Recurse for the first sub-leg
            c.rotate(60*deg);   // Turn 60 degrees clockwise
            leg(n-1);           // Second sub-leg
            c.rotate(-120*deg); // Rotate 120 degrees back
            leg(n-1);           // Third sub-leg
            c.rotate(60*deg);   // Rotate back to our original heading
            leg(n-1);           // Final sub-leg
        }
        c.restore();            // Restore the transformation
        c.translate(len, 0);    // But translate to make end of leg (0,0)
    }
}

snowflake(c,0,5,115,125);    // A level-0 snowflake is an equilateral triangle
snowflake(c,1,145,115,125);  // A level-1 snowflake is a 6-sided star
snowflake(c,2,285,115,125);  // etc.
snowflake(c,3,425,115,125);
snowflake(c,4,565,115,125);  // A level-4 snowflake looks like a snowflake!
c.stroke();                  // Stroke this very complicated path
```

21.4.5 Drawing and Filling Curves

A path is a sequence of subpaths, and a subpath is a sequence of connected points. In the paths we defined in §21.4.1, those points were connected with straight line segments, but that need not always be the case. The CanvasRenderingContext2D object defines a number of methods that add a new point to the subpath and connect the current point to that new point with a curve:

`arc()`
: This method adds an arc to the current subpath. It connects the current point to the beginning of the arc with a straight line, and then connects the beginning of the arc to the end of the arc with a portion of a circle, leaving the end of the arc as the new current point. The arc to be drawn is specified with six parameters: the X and Y coordinates of the center of a circle, the radius of the circle, the start and end angles of the arc, and the direction (clockwise or counterclockwise) of the arc between those two angles.

`arcTo()`
: This method draws a straight line and a circular arc just like the `arc()` method does, but it specifies the arc to be drawn using different parameters. The arguments to `arcTo()` specify points P1 and P2 and a radius. The arc that is added to the path has the specified radius and is tangent to the line between the current point and P1 and also the line between P1 and P2. This unusual-seeming method of specifying arcs is actually quite useful for drawing shapes with rounded corners. If you specify a radius of 0, this method just draws a straight line from the current point to P1. With a nonzero radius, however, it draws a straight line from the current point in the direction of P1 then curves that line around in a circle until it is heading in the direction of P2.

`bezierCurveTo()`
: This method adds a new point P to the subpath and connects it to the current point with a cubic Bezier curve. The shape of the curve is specified by two "control points" C1 and C2. At the start of the curve (at the current point), the curve heads in the direction of C1. At the end of the curve (at point P), the curve arrives from the direction of C2. In between these points the direction of the curve varies smoothly. The point P becomes the new current point for the subpath.

`quadraticCurveTo()`
: This method is like `bezierCurveTo()`, but it uses a quadratic Bezier curve instead of a cubic Bezier curve and has only a single control point.

You can use these methods to draw paths like those in Figure 21-9.



Figure 21-9. Curved paths in a canvas

Example 21-7 shows the code used to create Figure 21-9. The methods demonstrated in this code are some of the most complicated in the Canvas API; see the reference section for complete details on the methods and their arguments.

Example 21-7. Adding curves to a path

```
// A utility function to convert angles from degrees to radians
function rads(x) { return Math.PI*x/180; }

// Draw a circle. Scale and rotate if you want an ellipse instead.
// There is no current point, so draw just the circle with no straight
// line from the current point to the start of the circle.
c.beginPath();
c.arc(75,100,50,          // Center at (75,100), radius 50
      0,rads(360),false); // Go clockwise from 0 to 360 degrees

// Draw a wedge. Angles are measured clockwise from the positive x axis.
// Note that arc() adds a line from the current point to the arc start.
c.moveTo(200, 100);       // Start at the center of the circle
c.arc(200, 100, 50,       // Circle center and radius
      rads(-60), rads(0), // start at angle -60 and go to angle 0
      false);             // false means clockwise
c.closePath();            // Add radius back to the center of the circle

// Same wedge, opposite direction
c.moveTo(325, 100);
c.arc(325, 100, 50, rads(-60), rads(0), true); // counterclockwise
c.closePath();

// Use arcTo() for rounded corners. Here we draw a square with
// upper left corner at (400,50) and corners of varying radii.
c.moveTo(450, 50);          // Begin in the middle of the top edge.
c.arcTo(500,50,500,150,30); // Add part of top edge and upper right corner.
```

```
c.arcTo(500,150,400,150,20); // Add right edge and lower right corner.
c.arcTo(400,150,400,50,10);  // Add bottom edge and lower left corner.
c.arcTo(400,50,500,50,0);    // Add left edge and upper left corner.
c.closePath();               // Close path to add the rest of the top edge.

// Quadratic Bezier curve: one control point
c.moveTo(75, 250);                      // Begin at (75,250)
c.quadraticCurveTo(100,200, 175, 250);  // Curve to (175,250)
c.fillRect(100-3,200-3,6,6);            // Mark the control point (100,200)

// Cubic Bezier curve
c.moveTo(200, 250);                       // Start at (200,250)
c.bezierCurveTo(220,220,280,280,300,250); // Curve to (300,250)
c.fillRect(220-3,220-3,6,6);              // Mark control points
c.fillRect(280-3,280-3,6,6);

// Define some graphics attributes and draw the curves
c.fillStyle = "#aaa";  // Gray fills
c.lineWidth = 5;       // 5-pixel black (by default) lines
c.fill();              // Fill the curves
c.stroke();            // Stroke their outlines
```

21.4.6 Rectangles

CanvasRenderingContext2D defines four methods for drawing rectangles. Example 21-7 used one of them, `fillRect()`, to mark the control points of the Bezier curves. All four of these rectangle methods expect two arguments that specify one corner of the rectangle followed by the rectangle width and height. Normally, you specify the upper left corner and then pass a positive width and positive height, but you may also specify other corners and pass negative dimensions.

`fillRect()` fills the specified rectangle with the current `fillStyle`. `strokeRect()` strokes the outline of the specified rectangle using the current `strokeStyle` and other line attributes. `clearRect()` is like `fillRect()`, but it ignores the current fill style and fills the rectangle with transparent black pixels (the default color of all blank canvases). The important thing about these three methods is that they do not affect the current path or the current point within that path.

The final rectangle method is named `rect()`, and it does affect the current path: it adds the specified rectangle, in a subpath of its own, to the path. Like other path-definition methods, it does not fill or stroke anything itself.

21.4.7 Colors, Transparency, Gradients, and Patterns

The `strokeStyle` and `fillStyle` attributes specify how lines are stroked and regions are filled. Most often, these attributes are used to specify opaque or translucent colors, but you can also set them to CanvasPattern or CanvasGradient objects to stroke or fill with a repeated background image or with a linear or radial color gradient. In addition, you can set the `globalAlpha` property to make everything you draw translucent.

To specify a solid color, use one of the color names defined by the HTML4 standard,[1] or use a CSS color string:

```
context.strokeStyle = "blue";   // Stroke lines in blue
context.fillStyle = "#aaa";     // Fill areas with light gray
```

The default value for both `strokeStyle` and `fillStyle` is "#000000": opaque black.

Current browsers support CSS3 colors and allow the use of the RGB, RGBA, HSL, and HSLA color spaces in addition to basic hexadecimal RGB colors. Here are some examples:

```
var colors = [
    "#f44",                     // Hexadecimal RGB value: red
    "#44ff44",                  // Hexadecimal RRGGBB value: green
    "rgb(60, 60, 255)",         // RGB as integers 0-255: blue
    "rgb(100%, 25%, 100%)",     // RGB as percentages: purple
    "rgba(100%, 25%, 100%, 0.5)", // RGB plus alpha 0-1: translucent purple
    "rgba(0,0,0,0)",            // Completely transparent black
    "transparent",              // Synonym for the above
    "hsl(60, 100%, 50%)",       // Fully saturated yellow
    "hsl(60, 75%, 50%)",        // Less saturated yellow
    "hsl(60, 100%, 75%)",       // Fully saturated yellow, lighter
    "hsl(60, 100%, 25%)",       // Fully saturated yellow, darker
    "hsla(60, 100%, 50%, 0.5)", // Fully saturated yellow, 50% opaque
];
```

The HSL color space defines a color with three numbers that specify its hue, saturation, and lightness. Hue is an angle in degrees around a color wheel. A hue of 0 is red, 60 is yellow, 120 is green, 180 is cyan, 240 is blue, 300 is magenta, and 360 is back to red again. Saturation describes the intensity of the color, and it is specified as a percentage. Colors with 0 percent saturation are shades of gray. Lightness describes how light or dark a color is and is also specified as a percentage. Any HSL color with 100 percent lightness is pure white, and any color with 0 percent lightness is pure black. The HSLA color space is just like HSL, but it adds an alpha value that ranges from 0.0 (transparent) to 1.0 (opaque).

If you want to work with translucent colors, but do not want to explicitly specify an alpha channel for each color, or if you want to add translucency to opaque images or patterns (for example), you can set the `globalAlpha` property. Every pixel you draw will have its alpha value multiplied by `globalAlpha`. The default is 1, which adds no transparency. If you set `globalAlpha` to 0, everything you draw will be fully transparent and nothing will appear in the canvas. If you set this property to 0.5, pixels that would otherwise have been opaque will be 50 percent opaque. And pixels that would have been 50 percent opaque will be 25 percent opaque instead. If you set `globalAlpha`, all your pixels will be translucent and you may have to consider how those pixels are combined (or "composited") with the pixels they are drawn over—see §21.4.13 for details about Canvas compositing modes.

1. Aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow.

Instead of drawing with solid (but possibly translucent) colors, you can also use color gradients and repeating images when filling and stroking paths. Figure 21-10 shows a rectangle stroked with wide lines and a patterned stroke style on top of a linear gradient fill and underneath a translucent radial gradient fill. The code fragments below show how the pattern and gradients were created.

To fill or stroke using a background image pattern instead of a color, set `fillStyle` or `strokeStyle` to the CanvasPattern object returned by the `createPattern()` method of the context object:

```
var image = document.getElementById("myimage");
c.fillStyle = c.createPattern(image, "repeat");
```

The first argument to `createPattern()` specifies the image to use as the pattern. It must be an `<img>`, `<canvas>`, or `<video>` element from the document (or an image object created with the `Image()` constructor). The second argument is typically "repeat" for a repeating image fill that is independent of the size of the image, but you can also use "repeat-x", "repeat-y", or "no-repeat".

Note that you can use a `<canvas>` element (even one that has never been added to the document and is not visible) as the pattern source for another `<canvas>`:

```
var offscreen = document.createElement("canvas");  // Create an offscreen canvas
offscreen.width = offscreen.height = 10;           // Set its size
offscreen.getContext("2d").strokeRect(0,0,6,6);    // Get its context and draw
var pattern = c.createPattern(offscreen,"repeat"); // And use it as a pattern
```

To fill (or stroke) with a color gradient, set `fillStyle` (or `strokeStyle`) to a CanvasGradient object returned by the `createLinearGradient()` or `createRadialGradient()` methods of the context. Creating gradients is a multistep process, and using them is trickier than using patterns.

The first step is to create the CanvasGradient object. The arguments to `createLinearGradient()` are the coordinates of two points that define a line (it does not need to be horizontal or vertical) along which the colors will vary. The arguments to `createRadialGradient()` specify the centers and radii of two circles. (They need not be concentric, but the first circle typically lies entirely inside the second.) Areas inside the smaller circle or outside the larger will be filled with solid colors: areas between the two will be filled with a color gradient.

After creating the CanvasGradient object and defining the regions of the canvas that will be filled, you must define the gradient colors by calling the `addColorStop()` method of the CanvasGradient. The first argument to this method is a number between 0.0 and 1.0. The second argument is a CSS color specification. You must call this method at least twice to define a simple color gradient, but you may call it more than that. The color at 0.0 will appear at the start of the gradient, and the color at 1.0 will appear at the end. If you specify additional colors, they will appear at the specified fractional position within the gradient. Elsewhere, colors will be smoothly interpolated. Here are some examples:

```
// A linear gradient, diagonally across the canvas (assuming no transforms)
var bgfade = c.createLinearGradient(0,0,canvas.width,canvas.height);
bgfade.addColorStop(0.0, "#88f");  // Start with light blue in upper left
bgfade.addColorStop(1.0, "#fff");  // Fade to white in lower right

// A gradient between two concentric circles. Transparent in the middle
// fading to translucent gray and then back to transparent.
var peekhole = c.createRadialGradient(300,300,100, 300,300,300);
peekhole.addColorStop(0.0, "transparent");           // Transparent
peekhole.addColorStop(0.7, "rgba(100,100,100,.9)"); // Translucent gray
peekhole.addColorStop(1.0, "rgba(0,0,0,0)");         // Transparent again
```

An important point to understand about gradients is that they are not position-independent. When you create a gradient, you specify bounds for the gradient. If you then attempt to fill an area outside of those bounds, you'll get the solid color defined at one end or the other of the gradient. If you define a gradient along the line between (0,0) and (100, 100), for example, you should only use that gradient to fill objects located within the rectangle (0,0,100,100).

The graphic shown in Figure 21-10 was created with the `pattern` pattern and the `bgfade` and `peekhole` gradients defined above using this code:

```
c.fillStyle = bgfade;           // Start with the linear gradient
c.fillRect(0,0,600,600);        // Fill the entire canvas
c.strokeStyle = pattern;        // Use the pattern for stroking lines
c.lineWidth = 100;              // Use really wide lines
c.strokeRect(100,100,400,400); // Draw a big square
c.fillStyle = peekhole;         // Switch to the radial gradient
c.fillRect(0,0,600,600);        // Cover canvas with this translucent fill
```

21.4.8 Line Drawing Attributes

You've already seen the `lineWidth` property, which specifies the width of the lines drawn by `stroke()` and `strokeRect()`. In addition to `lineWidth` (and `strokeStyle`, of course), there are three other graphics attributes that affect line drawing.

The default value of the `lineWidth` property is 1, and you can set it to any positive number, even fractional values less than 1. (Lines that are less than one pixel wide are drawn with translucent colors, so they look less dark than 1-pixel-wide lines). To fully understand the `lineWidth` property, it is important to visualize paths as infinitely thin one-dimensional lines. The lines and curves drawn by the `stroke()` method are centered over the path, with half of the `lineWidth` on either side. If you're stroking a closed path and only want the line to appear outside the path, stroke the path first, and then fill with an opaque color to hide the portion of the stroke that appears inside the path. Or if you only want the line to appear inside a closed path, call the `save()` and `clip()` methods (§21.4.10) first, and then call `stroke()` and `restore()`.

Line widths are affected by the current transformation, as you may be able to make out in the scaled axes of Figure 21-7. If you call `scale(2,1)` to scale the X dimension and leave Y unaffected, vertical lines will be twice as wide as horizontal lines drawn with the same `lineWidth` setting. It is important to understand that line width is determined

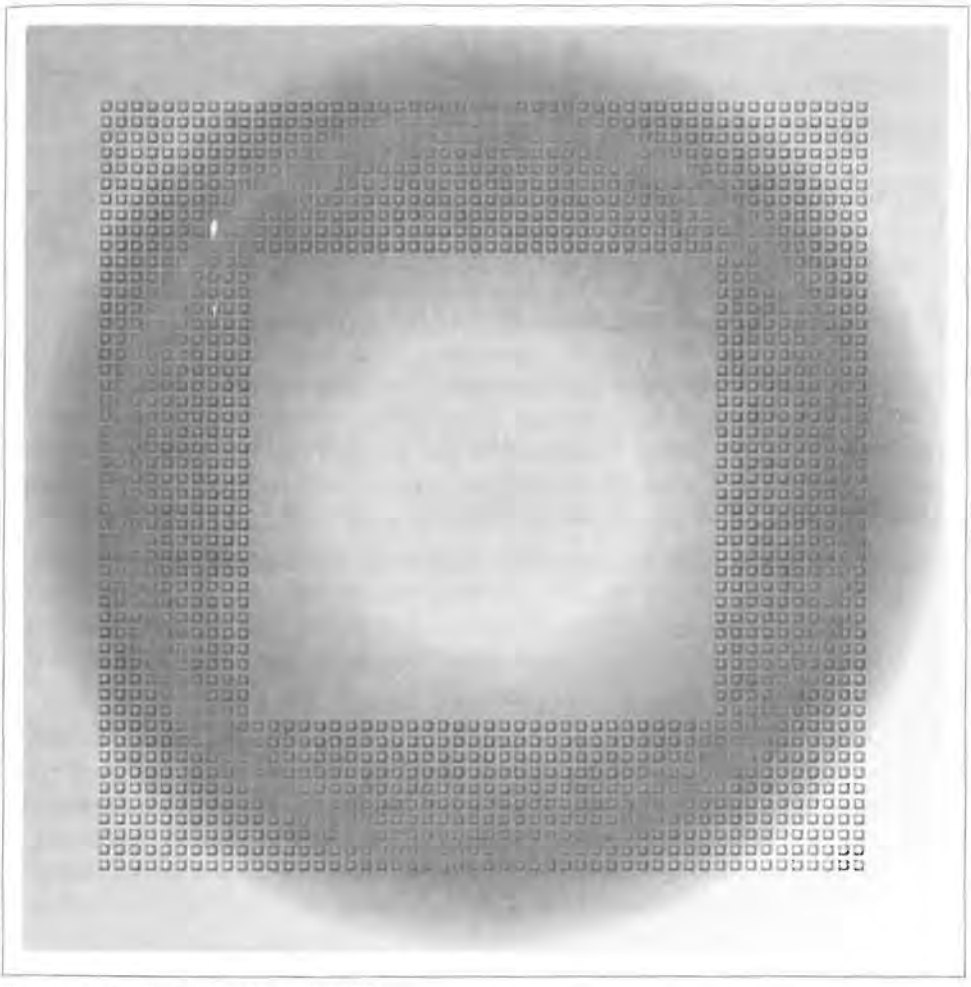


Figure 21-10. Pattern and gradient fills

by the `lineWidth` and the current transform at the time `stroke()` is called, not at the time that `lineTo()` or another path-building method is called.

The other three line drawing attributes affect the appearance of the unconnected ends of paths and the vertexes where two path segments meet. They have very little visual impact for narrow lines, but they make a big difference when you are drawing with wide lines. Two of these properties are illustrated in Figure 21-11. The figure shows the path as a thin black line and the stroke as the gray area that surrounds it.

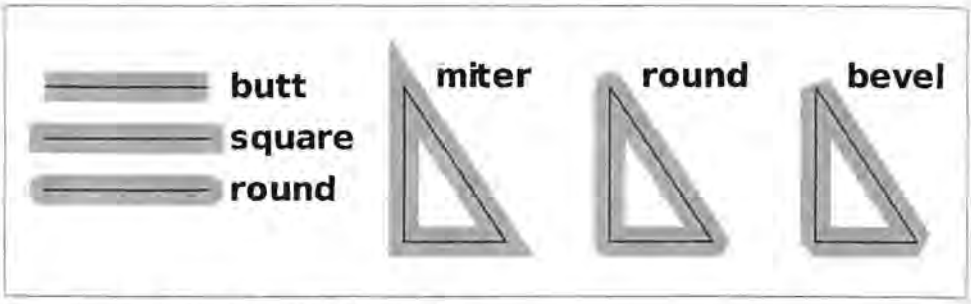


Figure 21-11. The lineCap and lineJoin attributes

The `lineCap` property specifies how the ends of an open subpath are "capped." The value "butt" (the default) means that the line terminates abruptly at the end point. The value "square" means that the line extends, by half of the line width, beyond the endpoint. And the value "round" means that the line is extended with a half circle (of radius one-half the line width) beyond the endpoint.

The `lineJoin` property specifies how the vertexes between subpath segments are connected. The default value is "miter", which means that the outside edges of the two path segments are extended until they meet at a point. The value "round" means that the vertex is rounded off, and the value "bevel" means that the vertex is cut off with a straight line.

The final line drawing property is `miterLimit`, which only applies when `lineJoin` is "miter". When two lines meet at a sharp angle, the miter between them can become quite long, and these long jagged miters are visually distracting. The `miterLimit` property places an upper bound on miter length. If the miter at a given vertex would be longer than half of the line width times `miterLimit`, that vertex will be drawn with a beveled join instead of a mitered join.

21.4.9 Text

To draw text in a canvas, you normally use the `fillText()` method, which draws text using the color (or gradient or pattern) specified by the `fillStyle` property. For special effects at large text sizes, you can use `strokeText()` to draw the outline of the individual text glyphs (an example of outlined text appears in Figure 21-13). Both methods take the text to be drawn as their first argument and take the X and Y coordinates of the text as the second and third arguments. Neither method affects the current path or the current point. As you can see in Figure 21-7, text is affected by the current transformation.

The `font` property specifies the font to be used for text drawing. The value should be a string in the same syntax as the CSS `font` attribute. Some examples:

```
"48pt sans-serif"
"bold 18px Times Roman"
"italic 12pt monospaced"
"bolder smaller serif"   // bolder and smaller than the font of the <canvas>
```

The textAlign property specifies how the text should be horizontally aligned with respect to the X coordinate passed to fillText() or strokeText(). The textBaseline property specifies how the text should be vertically aligned with respect to the Y coordinate. Figure 21-12 illustrates the allowed values for these properties. The thin line near each string of text is the baseline, and the small square marks the point (x,y) that was passed to fillText().

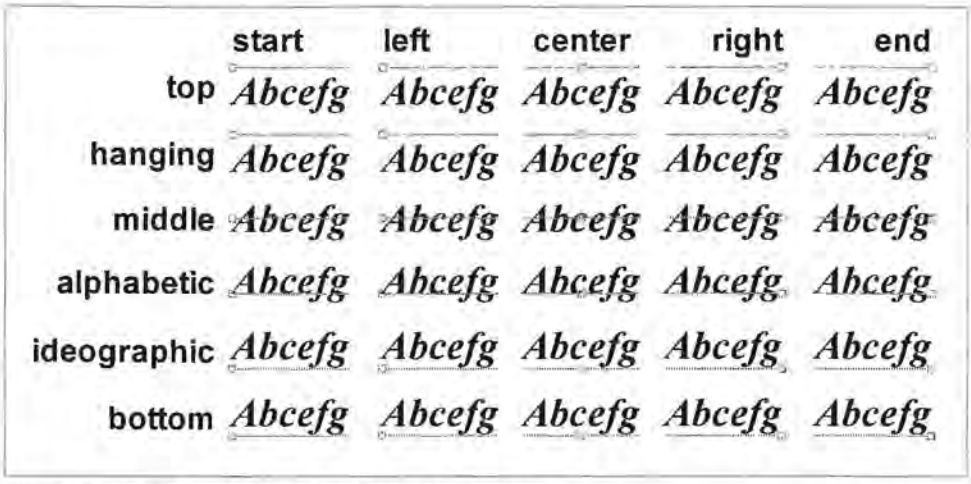


Figure 21-12. The textAlign and textBaseline properties

The default textAlign is "start". Note that for left-to-right text, an alignment of "start" is the same as "left" and an alignment of "end" is the same as "right". If you set the dir attribute of the <canvas> element to "rtl" (right-to-left), however, "start" alignment is the same and "right" alignment and "end" is the same as "left".

The default textBaseline is "alphabetic", and it is appropriate for Latin and similar scripts. The value "ideographic" is used with ideographic scripts such as Chinese and Japanese. The value "hanging" is intended for use with Devangari and similar scripts (which are used for many of the languages of India). The "top", "middle", and "bottom" baselines are purely geometric baselines, based on the "em square" of the font.

fillText() and strokeText() take an optional fourth argument. If given, this argument specifies the maximum width of the text to be displayed. If the text would be wider than the specified value when drawn using the font property, the canvas will make it fit by scaling it or by using a narrower or smaller font.

If you need to measure text yourself before drawing it, pass it to the measureText() method. This method returns a TextMetrics object that specifies the measurements of the text when drawn with the current font. At the time of this writing, the only "metric" contained in the TextMetrics object is the width. Query the on-screen width of a string like this:

```
var width = c.measureText(text).width;
```

21.4.10 Clipping

After defining a path, you usually call `stroke()` or `fill()` (or both). You can also call the `clip()` method to define a clipping region. Once a clipping region is defined, nothing will be drawn outside of it. Figure 21-13 shows a complex drawing produced using clipping regions. The vertical stripe running down the middle and the text along the bottom of the figure were stroked with no clipping region and then filled after the triangular clipping region was defined.

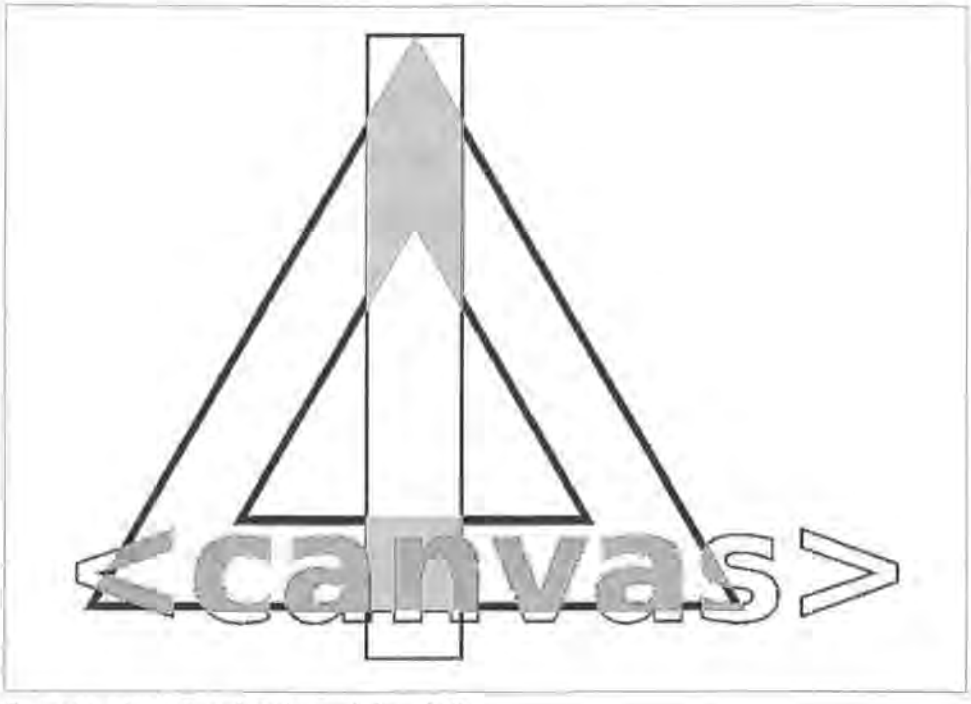


Figure 21-13. Unclipped strokes and clipped fills

Figure 21-13 was generated using the `polygon()` method of Example 21-4 and the following code:

```
// Define some drawing attributes
c.font = "bold 60pt sans-serif";    // Big font
c.lineWidth = 2;                    // Narrow lines
c.strokeStyle = "#000";             // Black lines

// Outline a rectangle and some text
c.strokeRect(175, 25, 50, 325);     // A vertical stripe down the middle
c.strokeText("<canvas>", 15, 330);  // Note strokeText() instead of fillText()

// Define a complex path with an interior that is outside.
polygon(c,3,200,225,200);           // Large triangle
polygon(c,3,200,225,100,0,true);    // Smaller reverse triangle inside
```

```
// Make that path the clipping region.
c.clip();

// Stroke the path with a 5 pixel line, entirely inside the clipping region.
c.lineWidth = 10;       // Half of this 10 pixel line will be clipped away
c.stroke();

// Fill the parts of the rectangle and text that are inside the clipping region
c.fillStyle = "#aaa"                // Light gray
c.fillRect(175, 25, 50, 325);       // Fill the vertical stripe
c.fillStyle = "#888"                // Darker gray
c.fillText("<canvas>", 15, 330);  // Fill the text
```


It is important to note that when you call `clip()`, the current path is itself clipped to the current clipping region, and then that clipped path becomes the new clipping region. This means that the `clip()` method can shrink the clipping region but can never enlarge it. There is no method to reset the clipping region, so before calling `clip()` you should typically call `save()`, so that you can later `restore()` the unclipped region.

21.4.11 Shadows

Four graphics attribute properties of the CanvasRenderingContext2D object control the drawing of drop shadows. If you set these properties appropriately, any line, area, text, or image you draw will be given a drop shadow, which will make it appear as if it is floating above the canvas surface. Figure 21-14 shows shadows beneath a filled rectangle, a stroked rectangle, and filled text.

The `shadowColor` property specifies the color of the shadow. The default is fully transparent black, and shadows will never appear unless you set this property to a translucent or opaque color. This property can only be set to a color string: patterns and gradients are not allowed for shadows. Using a translucent shadow color produces the most realistic shadow effects because it allows the background to show through.

The `shadowOffsetX` and `shadowOffsetY` properties specify the X and Y offsets of the shadow. The default for both properties is 0, which places the shadow directly beneath your drawing, where it is not visible. If you set both properties to a positive value, shadows will appear below and to the right of what you draw, as if there were a light source above and to the left, shining onto the canvas from outside the computer screen. Larger offsets produce larger shadows and make drawn objects appear as if they are floating "higher" above the canvas.

The `shadowBlur` property specifies how blurred the edges of the shadow are. The default value is 0, which produces crisp, unblurred shadows. Larger values produce more blur, up to an implementation-defined upper bound. This property is a parameter to a Gaussian blur function and is not a size or length in pixels.

Example 21-8 shows the code used to produce Figure 21-14 and demonstrates each of these four shadow properties.



Figure 21-14. Automatically generated shadows

Example 21-8. Setting shadow attributes

```
// Define a subtle shadow
c.shadowColor = "rgba(100,100,100,.4)"; // Translucent gray
c.shadowOffsetX = c.shadowOffsetY = 3;  // Shadow offset to lower right
c.shadowBlur = 5;                       // Soften shadow edges

// Draw some text in a blue box using that shadow
c.lineWidth = 10;
c.strokeStyle = "blue";
c.strokeRect(100, 100, 300, 200);       // Draw a rectangle
c.font = "Bold 36pt Helvetica";
c.fillText("Hello World", 115, 225);    // Draw some text

// Define a less subtle shadow. A larger offset makes items "float" higher.
// Note how the transparent shadow overlaps the blue box.
c.shadowOffsetX = c.shadowOffsetY = 20;
c.shadowBlur = 10;
c.fillStyle = "red";                    // Draw a solid red rectangle
c.fillRect(50,25,200,65);               // that floats above the blue box
```

The `shadowOffsetX` and `shadowOffsetY` properties are always measured in the default coordinate space and are not affected by the `rotate()` or `scale()` methods. Suppose,

for example, that you rotate the coordinate system by 90 degrees to draw some vertical text and then restore the old coordinate system to draw horizontal text. Both the vertical and horizontal text will have shadows oriented in the same direction, which is what you probably want. Similarly, shapes drawn with different scaling transforms will still have shadows of the same "height".[2]

21.4.12 Images

In addition to vector graphics (paths, lines, etc.), the Canvas API also supports bitmap images. The `drawImage()` method copies the pixels of a source image (or of a rectangle within the source image) onto the canvas, scaling and rotating the pixels of the image as necessary.

`drawImage()` can be invoked with three, five, or nine arguments. In all cases, the first argument is the source image from which pixels are to be copied. This image argument is often an `<img>` element or an off-screen image created with the `Image()` constructor, but it can also be another `<canvas>` element or even a `<video>` element. If you specify an `<img>` or `<video>` element that is still loading its data, the `drawImage()` call will do nothing.

In the three-argument version of `drawImage()`, the second and third arguments specify the X and Y coordinates at which the upper left corner of the image is to be drawn. In this version of the method, the entire source image is copied to the canvas. The X and Y coordinates are interpreted in the current coordinate system and the image is scaled and rotated if necessary.

The five-argument version of `drawImage()` adds `width` and `height` arguments to the `x` and `y` arguments described above. These four arguments define a destination rectangle within the canvas. The upper left corner of the source image goes at `(x,y)` and the lower right corner goes at `(x+width, y+height)`. Again, the entire source image is copied. The destination rectangle is measured in the current coordinate system. With this version of the method, the source image will be scaled to fit the destination rectangle, even if no scaling transform has ever been specified.

The nine-argument version of `drawImage()` specifies both a source rectangle and a destination rectangle and copies only the pixels within the source rectangle. Arguments two through five specify the source rectangle. They are measured in CSS pixels. If the source image is another canvas the source rectangle uses the default coordinate system for that canvas and ignores any transformations that have been specified. Arguments six through nine specify the destination rectangle into which the image is drawn and are in the current coordinate system of the canvas, not in the default coordinate system.

Example 21-9 is a simple demonstration of `drawImage()`. It uses the nine-argument version to copy pixels from a portion of a canvas and draw them, enlarged and rotated

2. At the time of this writing, Google's Chrome browser, version 5, gets this wrong and transforms the shadow offsets.

back onto the same canvas. As you can see in Figure 21-15, the image is enlarged enough to be pixelated, and you can see the translucent pixels used to smooth the edges of the line.



Figure 21-15. Pixels enlarged with drawImage()

Example 21-9. Using drawImage()

```
// Draw a line in the upper left
c.moveTo(5,5);
c.lineTo(45,45);
c.lineWidth = 8;
c.lineCap = "round";
c.stroke();

// Define a transformation
c.translate(50,100);
c.rotate(-45*Math.PI/180); // Straighten out the line
c.scale(10,10);            // Enlarge it so we can see the individual pixels

// Use draw image to copy the line
c.drawImage(c.canvas,
            0, 0, 50, 50,  // source rectangle: untransformed
            0, 0, 50, 50); // destination rectangle: transformed
```

In addition to drawing images into a canvas, we can also extract the content of a canvas as an image using the `toDataURL()` method. Unlike all the other methods described here, `toDataURL()` is a method of the Canvas element itself, not of the CanvasRenderingContext2D object. You normally invoke `toDataURL()` with no arguments, and it returns the content of the canvas as a PNG image, encoded as a string using a `data:` URL. The returned URL is suitable for use with an `<img>` element, and you can make a static snapshot of a canvas with code like this:

```
var img = document.createElement("img");  // Create an <img> element
img.src = canvas.toDataURL();             // Set its src attribute
document.body.appendChild(img);           // Append it to the document
```

All browsers are required to support the PNG image format. Some implementations may support other formats as well, and you can specify the desired MIME type with the optional first argument to `toDataURL()`. See the reference page for details.

There is one important security restriction you must be aware of when using `toDataURL()`. To prevent cross-origin information leaks, `toDataURL()` does not work on `<can`

vas> elements that are not "origin-clean." A canvas is not origin-clean if it has ever had an image drawn in it (directly by `drawImage()` or indirectly through a CanvasPattern) that has a different origin than the document that contains the canvas.

21.4.13 Compositing

When you stroke lines, fill regions, draw text, or copy images, you expect the new pixels to be drawn on top of the pixels that are already in the canvas. If you are drawing opaque pixels, they simply replace the pixels that are already there. If you are drawing with translucent pixels, the new ("source") pixel is combined with the old ("destination") pixel so that the old pixel shows through the new pixel based on how transparent that pixel is.

This process of combining new translucent source pixels with existing destination pixels is called *compositing*, and the compositing process described above is the default way that the Canvas API combines pixels. You don't always want compositing to happen, however. Suppose you've drawn into a canvas using translucent pixels and now want to make a temporary alteration to the canvas and then restore it to its original state. An easy way to do this is to copy the canvas (or a region of it) to an offscreen canvas using `drawImage()`. Then, when it is time to restore the canvas, you can copy your pixels from the offscreen canvas in which you saved them back to the on-screen canvas. Remember, though, that the pixels you saved were translucent. If compositing is on, they won't fully obscure and erase the temporary drawing you've done. In this scenario, you need a way to turn compositing off: to draw the source pixels and ignore the destination pixels regardless of the transparency of the source.

To specify the kind of compositing to be done, set the `globalCompositeOperation` property. The default value is "source-over", which means that source pixels are drawn "over" the destination pixels and are combined with them if the source is translucent. If you set this property to "copy", compositing is turned off: source pixels are copied to the canvas unchanged and destination pixels are ignored. Another `globalCompositeOperation` value that is sometimes useful is "destination-over". This kind of compositing combines pixels as if the new source pixels were drawn beneath the existing destination pixels. If the destination is translucent or transparent, some or all of the source pixel color is visible in the resulting color.

"source-over", "destination-over", and "copy" are three of the most commonly used types of compositing, but the Canvas API supports 11 values for the `globalCompositeOperation` attribute. The names of these compositing operations are suggestive of what they do, and you can go a long way toward understanding compositing by combining the operation names with visual examples of how they work. Figure 21-16 illustrates all 11 operations using "hard" transparency: all the pixels involved are fully opaque or fully transparent. In each of the 11 boxes, the square is drawn first and serves as the destination. Next `globalCompositeOperation` is set, and the circle is drawn as the source.

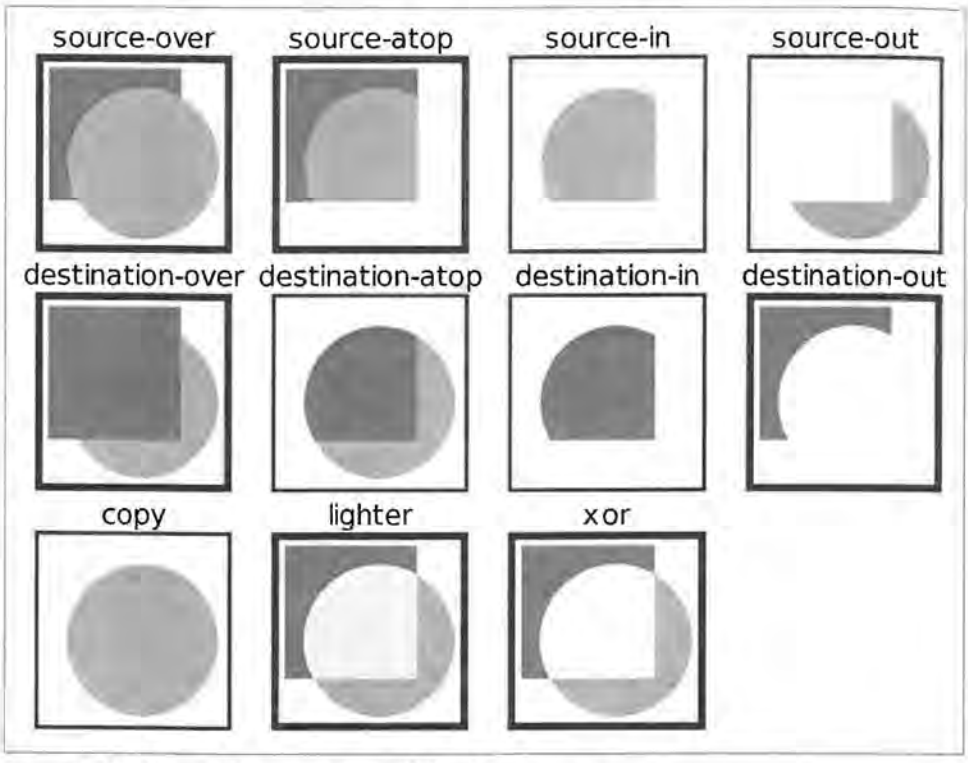


Figure 21-16. Compositing operations with hard transparency

Figure 21-17 is a similar example that uses "soft" transparency. In this version, the source circle and destination square are drawn using color gradients so that the pixels have a range of transparencies.

You may find that it is not so easy to understand the compositing operations when used with translucent pixels like these. If you are interested in a deeper understanding, the reference page for `CanvasRenderingContext2D` includes the equations that specify how individual pixel values are computed from source and destination pixels for each of the 11 compositing operations.

At the time of this writing, browser vendors disagree on the implementation of 5 of the 11 compositing modes: "copy", "source-in", "source-out", "destination-atop", and "destination-in" behave differently in different browsers and cannot be used portably. A detailed explanation follows, but you can skip to the next section if you don't plan on using any of these compositing operations.

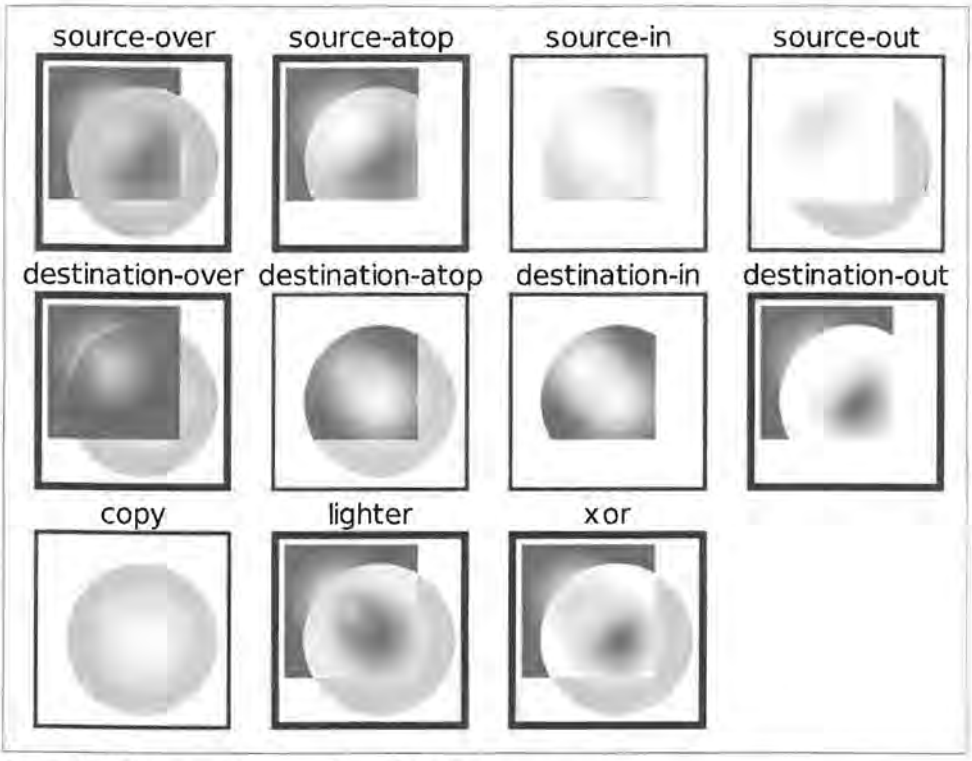


Figure 21-17. Compositing operations with soft transparency

The five compositing modes listed above either ignore the destination pixel values in the computation of result pixels or make the result transparent anywhere the source is transparent. The difference in implementation has to do with the definition of the source pixels. Safari and Chrome perform compositing "locally": only the pixels actually drawn by the `fill()`, `stroke()`, or other drawing operation count as part of the source. IE9 is likely to follow suit. Firefox and Opera perform compositing "globally": every pixel within the current clipping region is composited for every drawing operation. If the source does not set that pixel, it is treated as transparent black. In Firefox and Opera, this means that the five compositing modes listed above actually erase destination pixels outside of the source and inside the clipping region. Figures 21-16 and 21-17 were generated in Firefox, and this is why the boxes around "copy", "source-in", "source-out", "destination-atop", and "destination-in" are thinner than the other boxes: the rectangle around each sample is the clipping region and these four compositing operations erase the portion of the stroke (half of the `lineWidth`) that falls inside the path. For comparison, Figure 21-18 shows the same figure as Figure 21-17, but generated in Chrome.

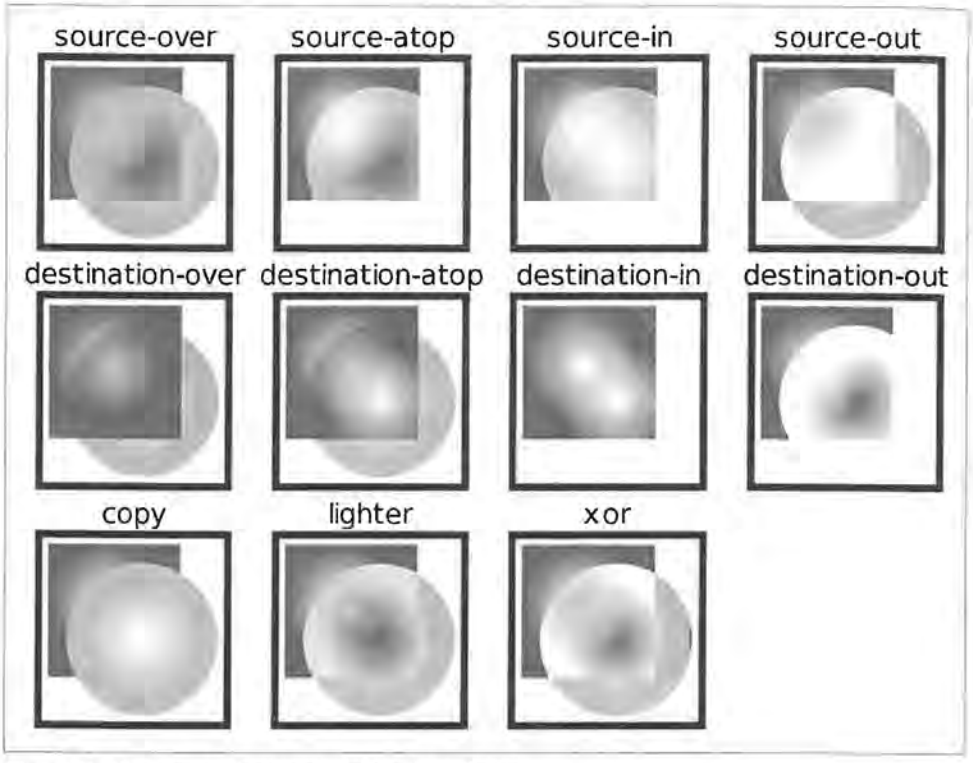


Figure 21-18. Compositing locally rather than globally

The HTML5 draft current at the time of this writing specifies the global compositing approach implemented by Firefox and Opera. Browser vendors are aware of the incompatibility and are not satisfied with the current state of the specification. There is a distinct possibility that the specification will be altered to require local compositing instead of global compositing.

Finally, note that it is possible to perform global compositing in browsers like Safari and Chrome that implement local compositing. First, create a blank offscreen canvas of the same dimensions as the on-screen canvas. Then draw your source pixels into the offscreen canvas and use `drawImage()` to copy the offscreen pixels to the on-screen canvas and composite them globally within the clipping region. There is not a general technique for performing local compositing in browsers like Firefox that implement global compositing, but you can often come close by defining an appropriate clipping region before performing the drawing operation that is to be locally composited.

21.4.14 Pixel Manipulation

The `getImageData()` method returns an ImageData object that represents the raw (non-premultiplied) pixels (as R, G, B, and A components) from a rectangular region of your canvas. You can create empty blank `ImageData` objects with `createImageData()`. The pixels in an ImageData object are writable, so you can set them any way you want, and then copy those pixels back onto the canvas with `putImageData()`.

These pixel manipulation methods provide very low-level access to the canvas. The rectangle you pass to `getImageData()` is in the default coordinate system: its dimensions are measured in CSS pixels and it is not affected by the current transformation. When you call `putImageData()`, the position you specify is also measured in the default coordinate system. Furthermore, `putImageData()` ignores all graphics attributes. It does not perform any compositing, it does not multiply pixels by `globalAlpha`, and it does not draw shadows.

Pixel manipulation methods are useful for implementing image processing. Example 21-10 shows how to create a simple motion blur or "smear" effect on the graphics in a canvas. The example demonstrates `getImageData()` and `putImageData()` and shows how to iterate through and modify the pixel values in an ImageData object, but it does not explain these things in any detail. See the CanvasRenderingContext2D reference pages for complete details on `getImageData()` and `putImageData()` and see the ImageData reference page for details on that object.

Example 21-10. Motion blur with ImageData

```
// Smear the pixels of the rectangle to the right, producing a
// sort of motion blur as if objects are moving from right to left.
// n must be 2 or larger. Larger values produce bigger smears.
// The rectangle is specified in the default coordinate system.
function smear(c, n, x, y, w, h) {
    // Get the ImageData object that represents the rectangle of pixels to smear
    var pixels = c.getImageData(x,y,w,h);

    // This smear is done in-place and requires only the source ImageData.
    // Some image processing algorithms require an additional ImageData to
    // store transformed pixel values. If we needed an output buffer, we could
    // create a new ImageData with the same dimensions like this:
    //   var output_pixels = c.createImageData(pixels);

    // These dimensions may be different than w and h arguments: there may be
    // more than one device pixel per CSS pixel.
    var width = pixels.width, height = pixels.height;

    // This is the byte array that holds the raw pixel data, left-to-right and
    // top-to-bottom. Each pixel occupies 4 consecutive bytes in R,G,B,A order.
    var data = pixels.data;

    // Each pixel after the first in each row is smeared by replacing it with
    // 1/nth of its own value plus m/nths of the previous pixel's value
    var m = n-1;
```

```
    for(var row = 0; row < height; row++) {  // For each row
        var i = row*width*4 + 4;  // The offset of the second pixel of the row
        for(var col = 1; col < width; col++, i += 4) { // For each column
            data[i] =   (data[i] + data[i-4]*m)/n;     // Red pixel component
            data[i+1] = (data[i+1] + data[i-3]*m)/n;   // Green
            data[i+2] = (data[i+2] + data[i-2]*m)/n;   // Blue
            data[i+3] = (data[i+3] + data[i-1]*m)/n;   // Alpha component
        }
    }

    // Now copy the smeared image data back to the same position on the canvas
    c.putImageData(pixels, x, y);
}
```

Note that `getImageData()` is subject to the same cross-origin security restriction that the `toDataURL()` is: it does not work on any canvas that has ever had an image drawn in it (directly by `drawImage()` or indirectly through a CanvasPattern) that has a different origin than the document that contains the canvas.

21.4.15 Hit Detection

The method `isPointInPath()` determines whether a specified point falls within (or on the boundary of) the current path and returns `true` if so, or `false` otherwise. The point you pass to the method is in the default coordinate system and is not transformed. This makes this method useful for *hit detection*: determining whether a mouse click occurred over a particular shape.

You can't pass the `clientX` and `clientY` fields of a MouseEvent object directly to `isPointInPath()`, however. First, the mouse event coordinates must be translated to be relative to the canvas element rather than the Window object. Second, if the onscreen size of the canvas is different than its actual dimensions, the mouse event coordinates must be scaled appropriately. Example 21-11 shows a utility function you can use to determine whether a give MouseEvent was over the current path.

Example 21-11. Testing whether a mouse event is over the current path

```
// Returns true if the specified mouse event is over the current path
// in the specified CanvasRenderingContext2D object.
function hitpath(context, event) {
    // Get <canvas> element from the context object
    var canvas = context.canvas;

    // Get canvas size and position
    var bb = canvas.getBoundingClientRect();

    // Translate and scale mouse event coordinates to canvas coordinates
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Call isPointInPath with these transformed coordinates
    return context.isPointInPath(x,y);
}
```

You might use this hitpath() function in an event handler like this:

```
canvas.onclick = function(event) {
    if (hitpath(this.getContext("2d"), event) {
        alert("Hit!");  // Click over current path
    }
};
```

Instead of doing path-based hit detection, you can use getImageData() to test whether the pixel under the mouse point has been painted. If the returned pixel (or pixels) are fully transparent, nothing has been drawn into that pixel and the mouse event is a miss. Example 21-12 shows how you can do this kind of hit detection.

Example 21-12. Testing whether a mouse event is over a painted pixel

```
// Returns true if the specified mouse event is over a nontransparent pixel.
function hitpaint(context, event) {
    // Translate and scale mouse event coordinates to canvas coordinates
    var canvas = context.canvas;
    var bb = canvas.getBoundingClientRect();
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Get the pixel (or pixels if multiple device pixels map 1 CSS pixel)
    var pixels = c.getImageData(x,y,1,1);

    // If any pixels have a nonzero alpha, return true (hit)
    for(var i = 3; i < pixels.data.length; i+=4) {
        if (pixels.data[i] !== 0) return true;
    }

    // Otherwise it was a miss.
    return false;
}
```

21.4.16 Canvas Example: Sparklines

We'll end this chapter with a practical example for drawing sparklines. A *sparkline* is a small data-display graphic intended to be included within the flow of text, like this one: Server load:  8. The term "sparkline" was coined by author Edward Tufte, who describes them as "small, high-resolution graphics embedded in a context of words, numbers, images. Sparklines are data-intense, design-simple, word-sized graphics." (Learn more about sparklines in Tufte's book *Beautiful Evidence* [Graphics Press].)

Example 21-13 is a relatively simple module of unobtrusive JavaScript code for enabling sparklines in your web pages. The comments explain how it works. Note that it uses the onLoad() function of Example 13-5.

Example 21-13. Sparklines with the <canvas> element

```
/*
 * Find all elements of CSS class "sparkline", parse their content as
 * a series of numbers, and replace it with a graphical representation.
 *
```

```
 * Define sparklines with markup like this:
 *   <span class="sparkline">3 5 7 6 6 9 11 15</span>
 *
 * Style sparklines with CSS like this:
 *   .sparkline { background-color: #ddd; color: red; }
 *
 * - Sparkline color is from the computed style of the CSS color property.
 * - Sparklines are transparent, so the normal background color shows through.
 * - Sparkline height is from the data-height attribute if defined or from
 *   the computed style for the font-size otherwise.
 * - Sparkline width is from the data-width attribute if it is defined
 *   or the number of data points times data-dx if that is defined or
 *   the number of data points times the height divided by 6
 * - The minimum and maximum values of the y axis are taken from the data-ymin
 *   and data-ymax attributes if they are defined, and otherwise come from
 *   the minimum and maximum values of the data.
 */
onLoad(function() {   // When the document firsts loads
    // Find all elements of class "sparkline"
    var elts = document.getElementsByClassName("sparkline");
    main: for(var e = 0; e < elts.length; e++) { // For each element
        var elt = elts[e];

        // Get content of the element and convert to an array of numbers.
        // If the conversion fails, skip this element.
        var content = elt.textContent || elt.innerText;  // Element content
        var content = content.replace(/^\s+|\s+$/g, ""); // Strip spaces
        var text = content.replace(/#.*$/gm, "");  // Strip comments
        text = text.replace(/[\n\r\t\v\f]/g, " "); // Convert \n etc, to space
        var data = text.split(/\s+|\s*,\s*/);      // Split on space or comma
        for(var i = 0; i < data.length; i++) {     // For each chunk
            data[i] = Number(data[i]);             // Convert to a number
            if (isNaN(data[i])) continue main;     // and abort on failure
        }

        // Now compute the color, width, height, and y axis bounds of the
        // sparkline from the data, from data- attributes of the element,
        // and from the computed style of the element.
        var style = getComputedStyle(elt, null);
        var color = style.color;
        var height = parseInt(elt.getAttribute("data-height")) ||
            parseInt(style.fontSize) || 20;
        var width = parseInt(elt.getAttribute("data-width")) ||
            data.length * (parseInt(elt.getAttribute("data-dx")) || height/6);
        var ymin = parseInt(elt.getAttribute("data-ymin")) ||
            Math.min.apply(Math, data);
        var ymax = parseInt(elt.getAttribute("data-ymax")) ||
            Math.max.apply(Math, data);
        if (ymin >= ymax) ymax = ymin + 1;

        // Create the canvas element.
        var canvas = document.createElement("canvas");
        canvas.width = width;     // Set canvas dimensions
        canvas.height = height;
        canvas.title = content;   // Use the element content as a tooltip
        elt.innerHTML = "";       // Erase existing element content
```

```
        elt.appendChild(canvas);  // Insert the canvas into the element

        // Now plot the points (i,data[i]), transforming to canvas coordinates.
        var context = canvas.getContext('2d');
        for(var i = 0; i < data.length; i++) {          // For each data point
            var x = width*i/data.length;                // Scale i
            var y = (ymax-data[i])*height/(ymax-ymin);  // Scale data[i]
            context.lineTo(x,y); // First lineTo() does a moveTo() instead
        }
        context.strokeStyle = color;   // Specify the color of the sparkline
        context.stroke();              // and draw it
    }
});
```