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.
  </body>
</html>
```

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

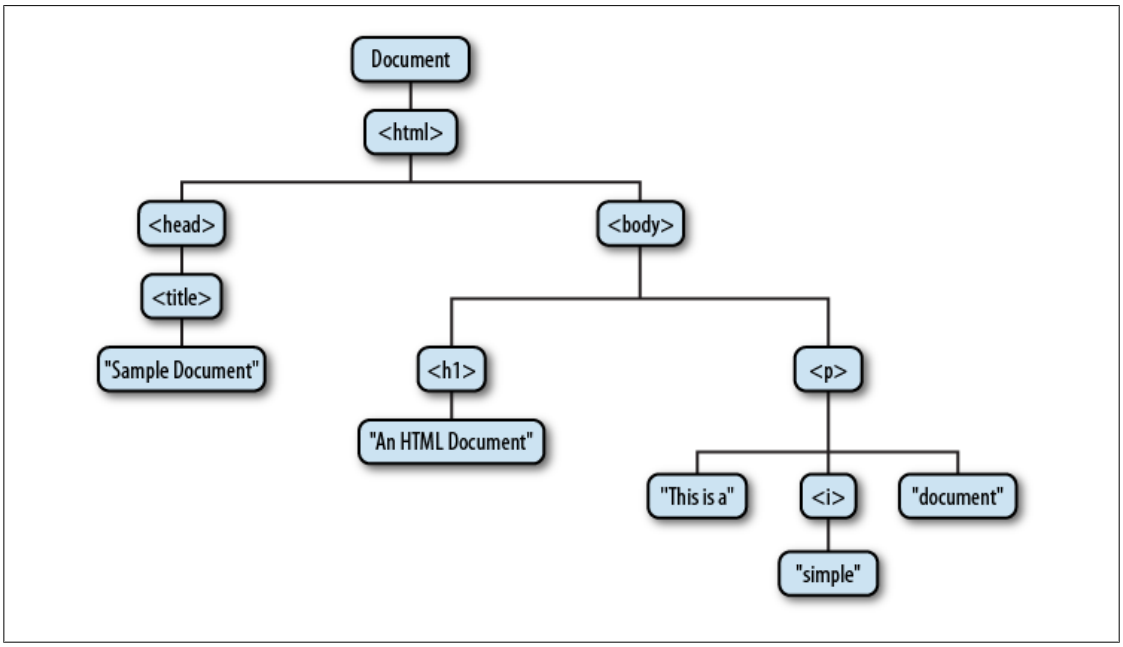

Figure 15-1. The tree representation of an HTML document

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.

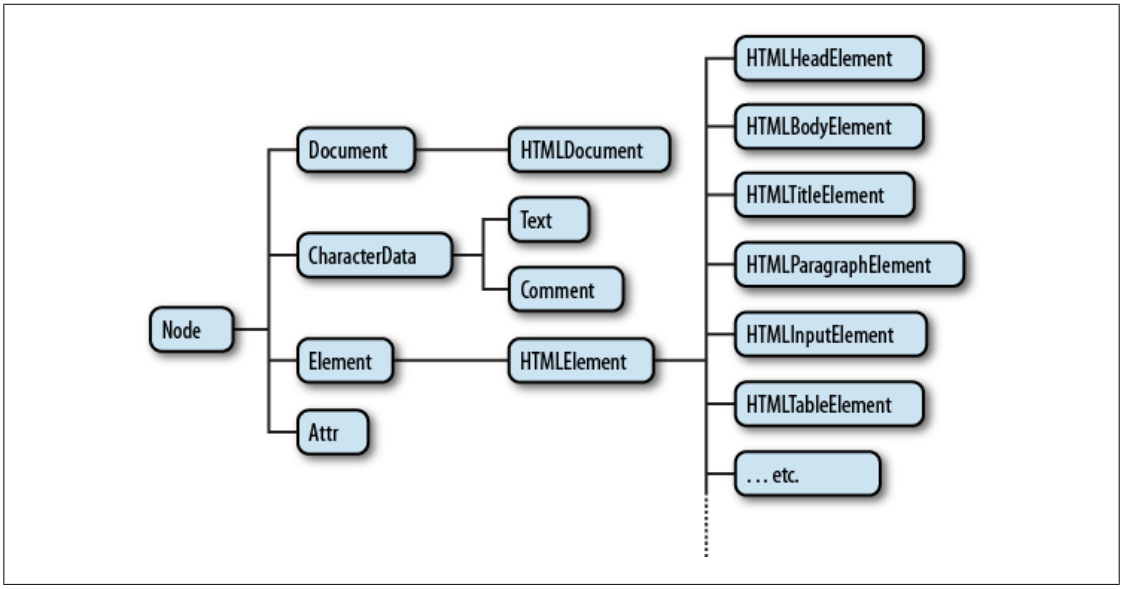

Figure 15-2. A partial class hierarchy of document nodes

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 `<span>` 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 `<span>` 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 `<span>` 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 element 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 standalone 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 HTML 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