CHAPTER 4

HTML and JavaScript

In this chapter, you finally get your hands dirty in real JavaScript code. You'll learn how JavaScript interacts with the page structure—defined in HTML—and how to receive data and give back information to your visitors. I start with an explanation of what an HTML document is and how it is structured and explain several ways to create page content via JavaScript. You'll then hear about the JavaScript developer's Swiss Army knife—the Document Object Model (DOM)—and how to separate JavaScript and HTML to create now-seamless effects that developers used to create in an obtrusive manner with DHTML.

The Anatomy of an HTML Document

Documents displayed in user agents are normally HTML documents. Even if you use a server-side language like ASP.NET, PHP, ColdFusion, or Perl, the outcome is HTML if you want to use browsers to their full potential. Modern browsers like Mozilla or Safari do also support XML, SVG, and other formats, but for 99% of your day-to-day web work, you'll go the HTML or XHTML route.

An HTML document is a text document that starts with a `DOCTYPE` that tells the user agent what the document is and how it should be dealt with. It then uses an `HTML` element that encompasses all other elements and text content. The `HTML` element should have a `lang` attribute that defines the language in use (human language, not programming language) and a `dir` attribute that defines the reading order of the text (left to right [`ltr`] for [Indo]European/American languages or right to left [`rtl`] for Semitic ones). Inside the `HTML` element needs to be a `HEAD` element with a `TITLE` element. You can add an optional `META` element determining what encoding should be used to display the text—if you haven't set the encoding on the server. On the same level as the `HEAD` element, but after the closing `<head>` tag, is the `BODY`—the element that contains all the page content.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
    <title>Our HTML Page</title>
  </head>
```

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

A markup document like this consists of **tags** (words or letters enclosed in tag brackets, like `<p>`) and text content. Documents should be well formed (meaning that every opening tag like `<p>` must be matched by a closing tag like `</p>`) and validate against the DTD provided. You can validate documents on the W3C web site (`http://validator.w3.org/`).

`HTML` elements are everything in the brackets, `<>`, with a starting tag like `<h1>` followed by content and a closing tag of the same name—like `</h1>`. Each element can have content in between the opening and the closing tag. Each element may have several **attributes**. The Document Type Definition, or DTD, linked in the `DOCTYPE` determines the set of tags that are permitted, how they may be nested, and which attributes each tag may have. The following example is a `P` element with an attribute whose name is `class`. The attribute has the value `intro`. The `P` contains the text "Lorem Ipsum".

```
<p class="intro">Lorem Ipsum</p>
```

A browser checks the `DOCTYPE` and compares the elements it encounters with the DTD. The HTML4.01 DTD definition tells it that `P` is a paragraph and that the `class` attribute is valid for this element. It also realizes that the `class` attribute should check the linked CSS style sheet, get the definitions for a `P` with that `class`, and render it accordingly. Now, what happens if you use an `attribute` that is not defined in the DTD—like `myattribute`?

```
<p class="intro" myattribute="left">Lorem Ipsum</p>
```

Nothing—although you technically made a mistake. Browsers are very forgiving and will not stop rendering even if they encounter unknowns like these, but instead they make the attribute available in the DOM tree. This makes them very user and developer friendly, but it makes it hard to advocate proper HTML syntax and standards compliance. There are, however, several reasons why we should strive for standards compliance—even in HTML generated via JavaScript:

- It is easier to trace errors when we know the HTML is valid.
- It is easier to maintain documents that adhere to the rules—as you can use a validator to measure its quality.
- It is a lot more likely that user agents will render or convert your pages properly when you develop against an agreed standard.
- The final documents can be easily converted to other formats if they are valid HTML.

Now, if we add some more elements to our example HTML and open it in a browser, we get a rendered output as shown in Figure 4-1:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
```

```
    <title>DOM Example</title>
  </head>
  <body>
    <h1>Heading</h1>
    <p>Paragraph</p>
    <h2>Subheading</h2>
    <ul id="eventsList">
      <li>List 1</li>
      <li>List 2</li>
      <li><a href="http://www.google.com">Linked List Item</a></li>
      <li>List 4</li>
    </ul>
    <p>Paragraph</p>
    <p>Paragraph</p>
  </body>
</html>
```


Figure 4-1. *An HTML document rendered by a browser*

A QUICK WORD ON XHTML

You might have glanced at the earlier code examples and got annoyed that I am using HTML 4.01 STRICT instead of the more modern XHTML. The reason is that more browsers support HTML 4.01 STRICT, and it is as clean as XHTML when it comes to syntax. Real XHTML documents should be sent as application/XML+XHTML and not as text/HTML on the server—a step not many developers dare to take, as it means the pages will not display in MSIE. Once the most common browsers do support real XHTML, it won't be hard to convert the documents. For now, 4.01 STRICT does the trick and makes browsers display our documents as the W3C planned them to be displayed—well, most of them do. Also be aware that I am using uppercase for HTML element names in the text, like `H1`, while all elements in the code examples are lowercase—as this is necessary for XHTML. XML and XHTML are case sensitive, whereas older and lesser strict HTML versions aren't.

The user agent "sees" the document a bit differently. The DOM models a document as a set of nodes, including element nodes, text nodes, and attribute nodes. Both elements and their text content are separate *nodes*. Attribute nodes are the *attributes* of the elements. The DOM includes other sorts of nodes for other parts of a markup document, but these three—element nodes, text nodes, and attribute nodes—are the important ones if you keep your JavaScript and HTML hat on. If you want to see the document through the eyes of the browser, you can use a tool like the DOM Inspector of Firefox, which shows the document as a tree structure, as depicted in Figure 4-2.

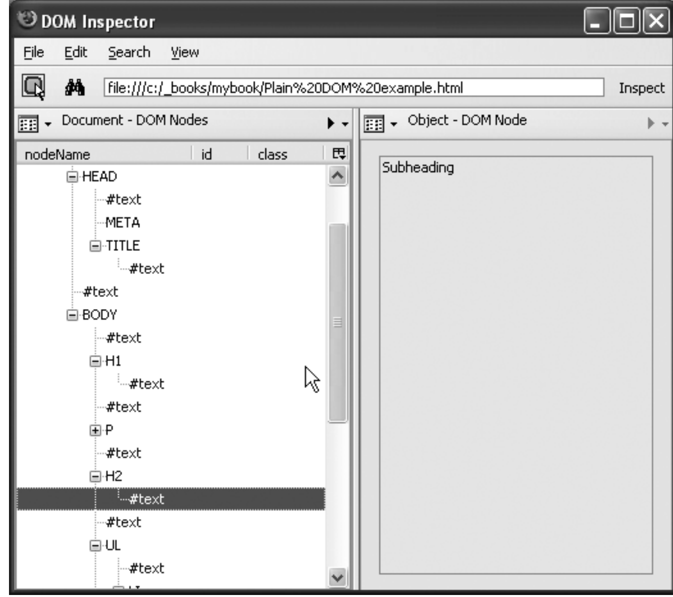

Figure 4-2. *The DOM representation of a document illustrated in the Mozilla DOM Inspector*

■ Tip You can access the DOM Inspector via Tools ➤ DOM Inspector in the Firefox toolbar or by pressing Ctrl+Shift+I. It allows you to check each part of the document in detail, and even remove elements, which is very handy when printing out pages. You can read more about the DOM Inspector in the Appendix, which covers validation and debugging.

Note Notice all the `#text` nodes between the elements? These are not text we added to the document, but the line breaks we added at the end of each line. Some browsers see those as text nodes while others don't—which can be very annoying when we try to access elements in the document via JavaScript later on.

Figure 4-3 shows another way we can visualize the document tree.

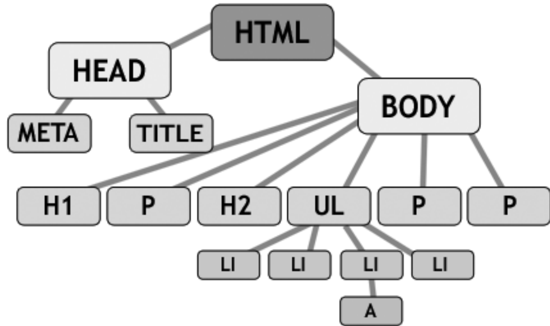

Figure 4-3. *The structure of an HTML document*

It is very important to recognize HTML for what it is: HTML is structured content and not a visual construct like an image with elements placed at different coordinates. When you have a proper, valid HTML document, the sky is the limit, and you can access and change it via JavaScript. An invalid HTML document might trip up your scripts, no matter how much testing you do. A classic mistake is using the same `id` attribute value twice in a single document, which negates the purpose of having a unique identifier (ID).

Providing Feedback in Web Pages via JavaScript: The Old School Ways

We already used one way—`document.write()`—of giving feedback to the user in an HTML document by writing out content. We also discussed the problems this method has, namely mixing the structure and the presentation layer and losing the maintenance benefits of keeping all JavaScript code in a separate file.

Using window Methods: prompt(), alert(), and confirm()

A different way of giving feedback and retrieving user-entered data is using methods the browser offers you via the window object, namely prompt(), alert(), and confirm().

The most commonly used window method is alert(), an example of which appears in Figure 4-4. What it does is display a value in a dialog (and perhaps play a sound, if the user's hardware supports it). The user then has to activate (click with the mouse or hit Enter on the keyboard) the OK button to get rid of the message.

Figure 4-4. *A JavaScript alert (on Firefox on Windows XP)*

Alerts look different from browser to browser and operating system to operating system.

As a user feedback mechanism, alert() has the merit of being supported by most user agents, but it is also a very "in-your-face" and clumsy way of passing information to the user. An alert is a message that normally bears bad news or warns someone of danger ahead—which is not necessarily your intention.

Say, for example, you want to tell the visitor to enter something in a search field before submitting a form. You could use an alert for that:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
    <title>Search example</title>
    <script type="text/javascript">
      function checkSearch()
      {
        if(!document.getElementById ||➥
           !document.createTextNode){return;}
        if(!document.getElementById('search')){return;}
        var searchValue=document.getElementById('search').value;
        if(searchValue=='')
        {
          alert('Please enter a search term');
          return false;
        }
        else
        {
          return true;
```

```
      }
    }
  </script>
</head>
<body>
  <form action="sitesearch.php" method="post"➥
        onsubmit="return checkSearch();">
    <p>
      <label for="search">Search the site:</label>
      <input type="text" id="search" name="search" />
      <input type="submit" value="Search" />
    </p>
  </form>
</body>
</html>
```

If a visitor tries to submit the form via the Submit button, he'll get the alert, and the browser will not send the form to the server after he activates the OK button. On Mozilla Firefox on Windows XP, the alert looks like what you see in Figure 4-5.

Figure 4-5. *Giving feedback on a form error via an alert*

Alerts do not return any information to the script—they simply give a message to the user and stop any further code execution until the OK button is activated.

This is different for `prompt()` and `confirm()`. The former allows visitors to enter something, and the latter asks users to confirm an action.

■ **Tip** As a debugging measure, `alert()` is simply too handy not to use. All you do is add an `alert(variableName)` in your code where you want to know what the variable value is at that time. You'll get the information and stop the rest of the code from executing until the OK button is activated—which is great for tracing back where and how your script fails. Beware of using it in loops though—there is no way to stop the loop, and you might have to press Enter a hundred times before you get back to your editing. There are other debugging tools like Mozilla's, Opera's, and Safari's JavaScript consoles and Mozilla Venkman—more on these in the Appendix.

We could extend the earlier example to ask the visitor to confirm a search for the common term *JavaScript* (see also Figure 4-6):

```
function checkSearch()
{
  if(!document.getElementById ||➥
     !document.createTextNode){return;}
  if(!document.getElementById('search')){return;}
  var searchValue=document.getElementById('search').value;
  if(searchValue=='')
  {
   alert('Please enter a search term before sending the form');
   return false;
  }
  else if(searchValue=='JavaScript')
  {
    var really=confirm('"JavaScript" is a very common term.\n' +➥
   'Do you really want to search for this?');
    return really;
  }
  else
  {
    return true;
  }
}
```

Figure 4-6. *Example of asking for user confirmation via* `confirm()`

Notice that the `confirm()` is a method that returns a Boolean value (`true` or `false`) depending on the visitor activating OK or Cancel. Confirm dialogs are an easy way to stop visitors from taking really bad steps in web applications. While they are not the prettiest way of asking a user to confirm a choice, they are really stable and offer some functionality your own confirmation functions most probably will not have—for example, playing the alert sound.

Both `alert()` and `confirm()` send information to the user, but what about retrieving information? A simple way to retrieve user input is the `prompt()` method. This one takes two parameters, the first one being a string displayed above the entry field as a label and the second a preset value for the entry field. Buttons labeled OK and Cancel (or something similar) will be displayed next to the field and the label, as shown in Figure 4-7.

```
var user=prompt('Please choose a name','User12');
```

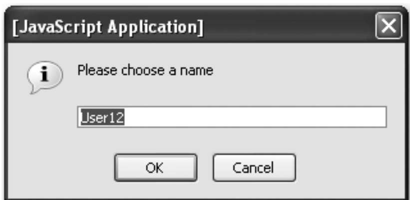

Figure 4-7. *Allowing the user to enter data in a prompt*

When the visitor activates the OK button, the value of the variable `user` will be either `User12` (when she hasn't changed the preset) or whatever she entered. When she activates the Cancel button, the value will be `null`.

We can use this functionality to allow a visitor to change a value before sending a form to the server:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
"http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
    <title>Date example</title>
    <script type="text/javascript">
      function checkDate()
      {
        if(!document.getElementById ||➥
           !document.createTextNode){return;}
        if(!document.getElementById('date')){return;}
        // Define a regular expression to check the date format
        var checkPattern=new➥
          RegExp("\\d{2}/\\d{2}/\\d{4}");
        // Get the value of the date entry field
        var dateValue=document.getElementById('date').value;
        // If there is no date entered, don't send the form
        if(dateValue=='')
```

```
        {
          alert('Please enter a date');
          return false
        }
        else
        {
          // Tell the user to change the date syntax either until
          // she presses Cancel or entered the right syntax
          while(!checkPattern.test(dateValue) && dateValue!=null)
          {
            dateValue=prompt('Your date was not in the right format. '➥
            + 'Please enter it as DD/MM/YYYY.', dateValue);
          }
          return dateValue!=null;
        }
      }
    </script>
  </head>
  <body>
    <h1>Events search</h1>
    <form action="eventssearch.php" method="post"➥
          onsubmit="return checkDate();">
      <p>
        <label for="date">Date in the format DD/MM/YYYY:</label><br />
        <input type="text" id="date" name="date" />
        <input type="submit" value="Check " />
        <br />(example 26/04/1975)
      </p>
    </form>
  </body>
</html>
```

Don't worry if the Regular Expression and the `test()` method are confusing you now; these will be covered in Chapter 9. All that is important now is that we use a `while` loop with a `prompt()` inside it. The `while` loop displays the same prompt over and over again until either the visitor presses `Cancel` (which means `dateValue` becomes `null`) or enters a date in the right format (which satisfies the test condition of the regular expression `checkPattern`).

Summary

You can create pretty nifty JavaScripts using the `prompt()`, `alert()`, and `confirm()` methods, and they have some points in their favor:

- They are easy to grasp, as the methods use the functionality and look and feel of the browser and offer a richer interface than HTML (specifically, the alert sound, if and when present, can help a lot of users).
- They are dead easy to implement and only need JavaScript instead of the appropriate HTML elements—that is also why they are used in bookmarklets/favelets (small scripts you can call via a bookmark or a favorite that alter the current document—in essence browser extensions that are available in all JavaScript capable browsers; see `http://www.bookmarklets.com`).
- They appear outside and above the current document—which gives them utmost importance.

However, there are some points that speak against the use of these methods to retrieve data and give feedback:

- You cannot style the messages, and they obstruct the web page, which does give them more importance, but also makes them appear clumsy from a design point of view. As they are part of the user's operating system or browser UI, they are easy to recognize for the user, but they break design conventions and guidelines the product may have to adhere to.
- Feedback does not happen in the web site's look and feel—which renders the site less important and stops dead the user's journey through our usability enhancing design elements.
- They are dependent on JavaScript—feedback should also be available when JavaScript is turned off.
- Unless you use Mozilla or older Netscape browsers, there is no way of telling whether the alert() or the prompt() is from this site or a different one. This is a security problem, as you could make a prompt() appear on a third-party site and read and transfer user input to yours. This is one technique of the phenomenon called **phishing**, and it is a big security and privacy threat. As some security companies publish information on the subject, visitors might not trust your site if it uses these feedback and input mechanisms (`http://secunia.com/advisories/15489/`).

Accessing the Document via the DOM

In addition to the window methods you now know about, you can access a web document via the DOM. In a manner of speaking, we have done that already with the `document.write()` examples. The `document` object is what we want to alter and add to, and `write()` is one method to do that. However, `document.write()` adds a string to the document and not a set of nodes and attributes, and you cannot separate the JavaScript out into a separate file—`document.write()` only works where you put it in the HTML. What we need is a way to reach where we want to change or add content, and this is exactly what the DOM and its methods provide us with. Earlier on you found out that user agents read a document as a set of nodes and attributes, and the DOM gives us tools to grab these. We can reach elements of the document via two methods:

- `document.getElementsByTagName('p')`
- `document.getElementById('id')`

The `getElementsByTagName('p')` method returns a list of all the elements with the name `p` as objects (where `p` can be any HTML element) and `getElementById('id')` returns us the element with the ID as an object. If you are already familiar with CSS, you could compare these two methods with the CSS selectors for elements and for IDs—`tag{}` and `#id{}`.

■ **Note** Similarities with CSS end here, as there is no DOM equivalent of the `class` selector `.class{}`. However, as this might be a handy method to have, some developers have come up with their own solutions to that problem and created `getElementsByClassName()` functions.

If you go back to the HTML example we used earlier, you can write a small JavaScript that shows how to use these two methods:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
    <title>DOM Example</title>
    <script type="text/JavaScript"  src="exampleFindElements.js">
    </script>
  </head>
  <body>
    <h1>Heading</h1>
    <p>Paragraph</p>
    <h2>Subheading</h2>
    <ul id="eventsList">
      <li>List 1</li>
      <li>List 2</li>
```

```
    <li><a href="http://www.google.com">Linked List Item</a></li>
    <li>List 4</li>
  </ul>
  <p>Paragraph</p>
  <p>Paragraph</p>
 </body>
</html>
```

Our script could now read the number of list items and paragraphs in the document by calling the `getElementsByTagName()` method and assigning the return value to variables—one time with the tag name `li` and the other time with `p`.

```
var listElements=document.getElementsByTagName('li');
var paragraphs=document.getElementsByTagName('p');
var msg='This document contains '+listElements.length+' list items\n';
msg+='and '+paragraphs.length+' paragraphs.';
alert(msg);
```

As Figure 4-8 shows, if you open the HTML document in a browser, you will find that both values are zero!

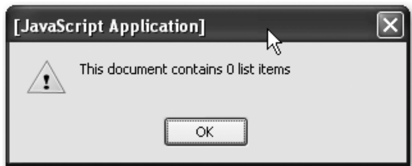

Figure 4-8. *Unwanted result when trying to reach elements before the page was rendered*

There aren't any list elements because the document has not yet been rendered by the browser when we try to read its content. We need to delay that reading until the document has been fully loaded and rendered.

You can achieve this by calling a function when the window has finished loading. The document has finished loading when the `onload` event of the `window` object gets triggered. You'll hear more about events in the next chapter; for now, let's just use the `onload` event handler of the `window` object to trigger the function:

```
function findElements()
{
  var listElements = document.getElementsByTagName('li');
  var paragraphs = document.getElementsByTagName('p');
  var msg = 'This document contains ' + listElements.length +➥
    ' list items\n';
  msg += 'and ' + paragraphs.length + ' paragraphs.';
  alert(msg);
}
window.onload = findElements;
```

If you open the HTML document in a browser now, you'll see an alert like the one in Figure 4-9 with the right number of list elements and paragraphs.

Figure 4-9. *Output alert indicating the number of found elements*

You can access each of the elements of a certain name like you access an array—once again bearing in mind that the counter of an array starts at `0` and not at `1`:

```
// Get the first paragraph
var firstpara = document.getElementsByTagName('p')[0];
// Get the second list item
var secondListItem = document.getElementsByTagName('p')[1];
```

You can combine several `getElementsByTagName()` method calls to read child elements directly. For example, to reach the first link item inside the third list item, you use

```
var➥ targetLink=document.getElementsByTagName('li')[2].getElementsByTagName➥
('a')[0];
```

This can get pretty messy though, and there are more clever ways to reach child elements—we'll get to these in a second. If you wanted to reach the last element, you can use the `length` property of the array:

```
var lastListElement = listElements[listElements.length - 1];
```

The `length` property also allows you to loop through elements and change all of them one after the other:

```
var linkItems = document.getElementsByTagName('li');
for(var i = 0; i < linkItems.length; i++)
{
  // Do something...
}
```

Element IDs need to be unique to the document; therefore the return value of `getElementById()` is a single object rather than an array of objects.

```
var events = document.getElementById('eventsList');
```

You can mix both methods to cut down on the number of elements to loop through. While the earlier `for` loop accesses all `LI` elements in the document, this one will only go through

those that are inside the element with the ID `eventsList` (the name of the object with the ID replaces the `document` object):

```
var events = document.getElementById('eventsList');
var eventlinkItems = events.getElementsByTagName('li');
for(var i = 0; i < eventLinkItems.length; i++)
{
  // Do something...
}
```

With the help of `getElementsByTagName()` and `getElementById()`, you can reach every element of the document or specifically target one single element. As mentioned earlier, `getElementById()` is a method of `document` and `getElementsByTagName()` is a method of any element. Now it is time to look at ways how to navigate around the document once you reached the element.

Of Children, Parents, Siblings, and Values

You know already that you can reach elements inside other elements by concatenating `getElementsByTagName` methods. However, this is rather cumbersome, and it means that you need to know the HTML you are altering. Sometimes that is not possible, and you have to find a more generic way to travel through the HTML document. The DOM already planned for this, via **children**, **parents**, and **siblings**.

These relationships describe where the current element is in the tree and whether it contains other elements or not. Let's take a look at our simple HTML example once more, concentrating on the body of the document:

```
<body>
  <h1>Heading</h1>
  <p>Paragraph</p>
  <h2>Subheading</h2>
  <ul id="eventsList">
    <li>List 1</li>
    <li>List 2</li>
    <li><a href="http://www.google.com">Linked List Item</a></li>
    <li>List 4</li>
  </ul>
  <p>Paragraph</p>
  <p>Paragraph</p>
</body>
```

All the indented elements are children of the `BODY`. `H1`, `H2`, `UL`, and `P` are siblings, and the `LI` elements are children of the `UL` element—and siblings to another. The link is a child of the third `LI` element. All in all, they are one big happy family.

However, there are even more children. The text inside the paragraphs, headings, list elements, and links also consists of nodes, as you may recall from Figure 4-2 earlier, and while they are not elements, they still follow the same relationship rules.

Every node in the document has several valuable properties.

- The most important is `nodeType`, which describes what the node is—an `element`, an `attribute`, a `comment`, `text`, and several more types (12 in all). For our HTML examples, only the `nodeType` values `1` and `3` are important, where `1` is an element node and `3` is a text node.
- Another important property is `nodeName`, which is the name of the element or `#text` if it is a text node. Depending on the document type and the user agent, `nodeName` can be either upper- or lowercase, which is why it is a good idea to convert it to lowercase before testing for a certain name. You can use the `toLowerCase()` method of the `string` object for that: `if(obj.nodeName.toLowerCase()=='li'){};`. For element nodes, you can use the `tagName` property.
- `nodeValue` is the value of the node: `null` if it is an element, and the text content if it is a text node.

In the case of text nodes, `nodeValue` can be read and set, which allows you to alter the text content of the element. If, for example, you wanted to change the text of the first paragraph, you might think it is enough to set its `nodeValue`:

```
document.getElementsByTagName('p')[0].nodeValue='Hello World';
```

However, this doesn't work (although—strangely enough—it does not cause an error), as the first paragraph is an `element` node. If you wanted to change the text inside the paragraph, you need to access the text node inside it or, in other words, the first child node of the paragraph:

```
document.getElementsByTagName('p')[0].firstChild.nodeValue='Hello World';
```

From the Parents to the Children

The `firstChild` property is a shortcut. Every element can have any number of children, listed in a property called `childNodes`.

- `childNodes` is a list of all the first-level child nodes of the element—it does not cascade down into deeper levels.
- You can access a child element of the current element via the array counter or the `item()` method.
- The shortcut properties `yourElement.firstChild` and `yourElement.lastChild` are easier versions of `yourElement.childNodes[0]` and `yourElement.childNodes[yourElement.childNodes.length-1]` and make it quicker to reach them.
- You can check whether an element has any children by calling the method `hasChildNodes()`, which returns a Boolean value.

Returning to the earlier example, you can access the `UL` element and get information about its children as shown here:

HTML

```
<ul id="eventsList">
  <li>List 1</li>
  <li>List 2</li>
  <li><a href="http://www.google.com">Linked List Item</a></li>
  <li>List 4</li>
</ul>
```

JavaScript:

```
function myDOMinspector()
{
  var DOMstring='';
  if(!document.getElementById || !document.createTextNode){return;}
  var demoList=document.getElementById('eventsList');
  if (!demoList){return;}
  if(demoList.hasChildNodes())
  {
    var ch=demoList.childNodes;
    for(var i=0;i<ch.length;i++)
    {
      DOMstring+=ch[i].nodeName+'\n';
    }
    alert(DOMstring);
  }
}
```

We create an empty string called `DOMstring` and check for DOM support and whether the `UL` element with the right `id` attribute is defined. Then we test whether the element has child nodes and, if it does, store them in a variable named `ch`. We loop through the variable (which automatically becomes an array) and add the `nodeName` of each child node to `DOMString`, followed by a line break (`\n`). We then look at the outcome using the `alert()` method.

If you run this script in a browser, you will see one of the main differences between MSIE and more modern browsers. While MSIE only shows four `LI` elements, other browsers like Firefox also count the line breaks in between the elements as text nodes, as you see in Figure 4-10.

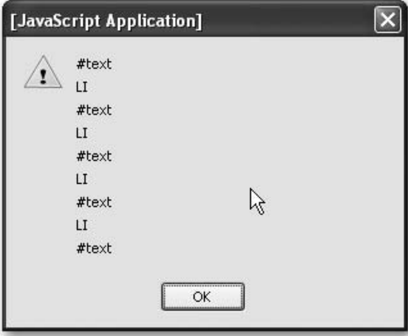

Figure 4-10. *Nodes found by our script, including text nodes that in reality are line breaks*

From the Children to the Parents

You can also navigate from child elements back to their parents, via the `parentNode` property. First let's make it easier for us to reach the link—by adding an ID to it:

```
<ul id="eventsList">
  <li>List</li>
  <li>List</li>
  <li>
    <a id="linkedItem" href="http://www.google.com">
      Linked List Item
    </a>
  </li>
  <li>List</li>
 </ul>
```

Now we assign a variable to the link object and read the parent node's name:

```
var myLinkItem=document.getElementById('linkedItem');
alert(myLinkItem.parentNode.nodeName);
```

The result is `LI`, and if we add another `parentNode` to the object reference, we'll get `UL`, which is the link's grandparent element:

```
alert(myLinkItem.parentNode.parentNode.nodeName);
```

You can add as many of those as you want—that is, if there are parent elements left in the document tree and you haven't reached the top level yet. If you use `parentNode` in a loop, it is important that you also test the `nodeName` and end the loop when it reaches the `BODY`. Say for example you want to check whether an object is inside an element with a class called `dynamic`. You can do that with a `while` loop:

```
var myLinkItem = document.getElementById('linkedItem');
var parentElm = myLinkItem.parentNode;
while(parentElm.className != 'dynamic')
```

```
{
  parentElm = parentElm.parentNode;
}
```

However, this loop will end in an "object required" error when there is no element with the right class. If you tell the loop to stop at the body, you can avoid that error:

```
var myLinkItem = document.getElementById('linkedItem');
var parentElm = myLinkItem.parentNode;
while(!parentElm.className != 'dynamic' && parentElm != 'document.body')
{
  parentElm=parentElm.parentNode;
}
alert(parentElm);
```

Among Siblings

The family analogy continues with siblings, which are elements on the same level (they don't come in different genders like brothers or sisters though). You can reach a different child node on the same level via the `previousSibling` and `nextSibling` properties of a node. Going back to our list example:

```
<ul id="eventsList">
  <li>List Item 1</li>
  <li>List Item 2</li>
  <li>
    <a id="linkedItem" href="http://www.google.com/">
      Linked List Item
    </a>
  </li>
  <li>List Item 4</li>
 </ul>
```

You can reach the link via `getElementById()` and the `LI` containing the link via `parentNode`. The properties `previousSibling` and `nextSibling` allow you to get `List Item 2` and `List Item 3` respectively:

```
var myLinkItem = document.getElementById('linkedItem');
var listItem = myLinkItem.parentNode;
var nextListItem = myLinkItem.nextSibling;
var prevListItem = myLinkItem.previousSibling;
```

■ **Note** This is a simplified example working only in MSIE; because of the difference of browser implementations, the next and previous siblings are not the `LI` elements on modern browsers, but text nodes with the line break as content.

If the current object is the last child of the parent element, `nextSibling` will be `undefined` and cause an error if you don't test for it properly. Unlike with `childNodes`, there are no shortcut properties for the first and last siblings, but you could write utility methods to find them; say, for example, you want to find the first and the last `LI` in our demo HTML:

```
window.onload=function()
{
  var myLinkItem=document.getElementById('linkedItem');
  var first=firstSibling(myLinkItem.parentNode);
  var last=lastSibling(myLinkItem.parentNode);
  alert(getTextContent(first));
  alert(getTextContent(last));
}
function lastSibling(node){
  var tempObj=node.parentNode.lastChild;
  while(tempObj.nodeType!=1 && tempObj.previousSibling!=null)
  {
    tempObj=tempObj.previousSibling;
  }
  return (tempObj.nodeType==1)?tempObj:false;
}
function firstSibling(node)
{
  var tempObj=node.parentNode.firstChild;
  while(tempObj.nodeType!=1 && tempObj.nextSibling!=null)
  {
    tempObj=tempObj.nextSibling;
  }
  return (tempObj.nodeType==1)?tempObj:false;
}
function getTextContent(node)
{
  return node.firstChild.nodeValue;
}
```

Notice that you need to check for `nodeType`, as the last or first child of the `parentNode` might be a text node and not an element.

Let's make our date checking script less obtrusive by using DOM methods to provide text feedback instead of sending an alert to the user. First, you need a container to show your error message:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
```

```
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8" />
    <title>Date example</title>
    <style type="text/css">
      .error{color:#c00;font-weight:bold;}
    </style>
    <script type="text/javascript" src="checkdate.js"></script>
  </head>
  <body>
    <h1>Events search</h1>
    <form action="eventssearch.php" method="post"➥
          onsubmit="return checkDate();">
      <p>
        <label for="date">Date in the format DD/MM/YYYY:</label><br />
        <input type="text" id="date" name="date" />
        <input type="submit" value="Check " />
        <br />(example 26/04/1975) <span class="error">  </span>
      </p>
    </form>
  </body>
</html>
```

The checking script more or less stays the same as it was before—the difference is that we use the SPAN as a means of displaying the error:

```
function checkDate(){
  if(!document.getElementById || !document.createTextNode){return;}
  var dateField=document.getElementById('date');
  if(!dateField){return;}
  var errorContainer=dateField.parentNode.getElementsByTagName➥
    ('span')[0];
  if(!errorContainer){return;}
  var checkPattern=new RegExp("\\d{2}/\\d{2}/\\d{4}");
  var errorMessage='';
  errorContainer.firstChild.nodeValue=' ';
  var dateValue=dateField.value;
  if(dateValue=='')
  {
    errorMessage='Please provide a date.';
  }
  else if(!checkPattern.test(dateValue))
  {
    errorMessage='Please provide the date in the defined format.';
  }
```

```
  if(errorMessage!='')
  {
    errorContainer.firstChild.nodeValue=errorMessage;
    dateField.focus();
    return false;
  }
  else
  {
    return true;
  }
}
```

First, test whether DOM is supported and that all the needed elements are there:

```
  if(!document.getElementById || !document.createTextNode){return;}
  var dateField=document.getElementById('date');
  if(!dateField){return;}
  var errorContainer=dateField.parentNode.getElementsByTagName➥
    ('span')[0];
  if(!errorContainer){return;}
```

Then define the test pattern and an empty error message. Set the text value of the error Span to a single space. This is necessary to avoid multiple error messages to display when the visitor sends the form a second time without rectifying the error.

```
  var checkPattern=new RegExp("\\d{2}/\\d{2}/\\d{4}");
  var errorMessage='';
  errorContainer.firstChild.nodeValue=' ';
```

Next is the validation of the field. Read the value of the date field and check whether there is an entry. If there is no entry, the error message will be that the visitor should enter a date. If there is a date, but it is in the wrong format, the message will indicate so.

```
var dateValue=dateField.value;
  if(dateValue=='')
  {
    errorMessage='Please provide a date.';
  }
  else if(!checkPattern.test(dateValue))
  {
    errorMessage='Please provide the date in the defined format.';
  }
```

All that is left then is to check whether the initial empty error message was changed or not. If it wasn't changed, the script should return `true` to the form `onsubmit="return checkDate();"`—thus submitting the form and allowing the back end to take over the work. If the error message was changed, the script adds the error message to the text content (the `nodeValue` of the first child

node) of the error SPAN, and sets the focus of the document back to the date entry field without submitting the form.

```
if(errorMessage!='')
{
  errorContainer.firstChild.nodeValue+=errorMessage;
  dateField.focus();
  return false;
}
else
{
  return true;
}
```

As Figure 4-11 shows, the end result is a lot more visually appealing than the alert message, and you can style it in any way you want to.

Figure 4-11. *Displaying a dynamic error message*

Now you know how to access and to change the text value of existing elements. But what if you want to change other attributes or you need HTML that is not necessarily provided for you?

Changing Attributes of Elements

Once you have reached the element you want to change, you can read and change its attributes in two ways: an older way—which is more widely supported by user agents—and the way of using newer DOM methods.

Older and newer user agents allow you to get and set element attributes as object properties:

```
var firstLink=document.getElementsByTagName('a')[0];
if(firstLink.href=='search.html')
{
  firstLink.href='http://www.google.com';
}
var mainImage=document.getElementById('nav').getElementsByTagName➥
  ('img')[0];
```

```
mainImage.src='dynamiclogo.gif';
mainImage.alt='Generico Corporation - We do generic stuff';
mainImage.title='Go back to Home';
```

All the attributes defined in the HTML specifications are available and can be accessed. Some are read-only for security reasons, but most can be set and read. You can also come up with your own attributes—JavaScript does not mind. Sometimes storing a value in an attribute of an element can save you a lot of testing and looping.

■ **Caution** Beware of attributes that have the same name as JavaScript commands—for example, `for`. If you try to set `element.for='something'`, it will result in an error. Browser vendors came up with workarounds for this; in the case of `for`—which is a valid attribute of the `label` element—the property name is `htmlFor`. Even more bizarre is the `class` attribute—something we will need a lot in the next chapter. This is a reserved word; you need to use `className` instead.

The DOM specifications provide two methods to read and set attributes—`getAttribute()` and `setAttribute()`. The `getAttribute()` method has one parameter—the attribute name; `setAttribute()` takes two parameters—the attribute name and the new value.

The earlier example using the newer methods looks like this:

```
var firstLink=document.getElementsByTagName('a')[0];
if(firstLink.getAttribute('href') =='search.html')
{
  firstLink.setAttribute('href') ='http://www.google.com';
}
var mainImage=document.getElementById('nav').getElementsByTagName➥
  ('img')[0];
mainImage.setAttribute('src') ='dynamiclogo.gif';
mainImage.getAttribute('alt') ='Generico Corporation - We do generic stuff';
mainImage.getAttribute('title') ='Go back to Home';
```

This may seem a bit superfluous and bloated, but the benefit is that it is a lot more consistent with other—higher—programming languages. It is also more likely to be supported by future user agents than the property way of assigning attributes to elements, and they work easily with arbitrary attribute names.

Creating, Removing, and Replacing Elements

DOM also provides methods for changing the structure of the document you can use in an HTML/JavaScript environment (there are more if you do XML conversion via JavaScript). You can not only change existing elements, but also create new ones and replace or remove old ones as well. These methods are as follows:

- `document.createElement('element')`: Creates a new element node with the tag name `element`.
- `document.createTextNode('string')`: Creates a new text node with the node value of `string`.
- `node.appendChild(newNode)`: Adds `newNode` as a new child node to `node`, following any existing children of `node`.
- `newNode=node.cloneNode(bool)`: Creates `newNode` as a copy (clone) of `node`. If `bool` is `true`, the clone includes clones of all the child nodes and attributes of the original.
- `node.insertBefore(newNode,oldNode)`: Inserts `newNode` as a new child node of `node` before `oldNode`.
- `node.removeChild(oldNode)`: Removes the child `oldNode` from `node`.
- `node.replaceChild(newNode, oldNode)`: Replaces the child node `oldNode` of `node` with `newNode`.

■ **Note** Notice that both `createElement()` and `createTextNode()` are methods of `document`; all the others are methods of any node.

All of these are indispensable when you want to create web products that are enhanced by JavaScript but don't rely exclusively on it. Unless you give all your user feedback via alert, confirm, and prompt pop-ups, you will have to rely on HTML elements to be provided to you—like the error message `SPAN` in the earlier example. However, as the HTML you need for your JavaScript to work only makes sense when JavaScript is enabled, it should not be available when there is no scripting support. An extra `SPAN` does not hurt anybody—however, form controls that offer the user great functionality (like date picker tools) but don't work are a problem.

Let's use these methods to counteract a problem that is quite common in web design: replacing Submit buttons with links. There are two ways to submit form data: a Submit button or an image button (actually, there are three—if you count hitting the Enter key).

Submit buttons are a thorn in the side of a lot of designers, as you cannot style them consistently across operating systems and browsers. Image buttons are a problem, as they require a lot of work on sites with localization, and they do not resize when a visitor has a larger font setting (unless you define their size in ems, which results in unsightly pixelation artifacts).

Links are much nicer, as you can style them with CSS, they resize, and they can be easily populated from a localized dataset. The problem with links though is that you need JavaScript to submit a form with them. That is why the lazy or too busy developer's answer to this dilemma is a link that simply submits the form using the `javascript:` protocol (a lot of code generators or frameworks will come up with the same):

```
<a href="javascript:document.forms[0].submit()">Submit</a>
```

A lot earlier in this book I established that this is just not an option—as without JavaScript this will be a dead link, and there won't be any way to submit the form. If you want to have the best of both worlds—a simple Submit button for non-JavaScript users and a link for the ones with scripting enabled—you'll need to do the following:

1. Loop through all `INPUT` elements of the document.
2. Test whether the `type` is `submit`.
3. If that is not the case, `continue` the loop, skipping the rest of it.
4. If it is the case, create a new link with a text node.
5. Set the node value of the text node to the value of the `INPUT` element.
6. Set the `href` of the link to the invalid `javascript:document.forms[0].submit()`.
7. Replace the `input` element with the link.

■ **Note** Setting the `href` attribute to the `javascript:` construct is not the cleanest way of doing this—in the next chapter, you'll learn about event handlers—a much better way to achieve this solution.

In code, this could be

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8">
    <title>Example: Submit buttons to links</title>
    <style type="text/css"></style>
    <script type="text/javascript" src="submitToLinks.js"></script>
  </head>
  <body>
    <form action="nogo.php" method="post">
      <label for="Name">Name:</label>
      <input type="text" id="Name" name="Name" />
      <input type="submit" value="send" />
    </form>
  </body>
</html>
function submitToLinks()
{
  if(!document.getElementById || !document.createTextNode){return;}
  var inputs,i,newLink,newText;
  inputs=document.getElementsByTagName('input');
  for (i=0;i<inputs.length;i++)
  {
    if(inputs[i].getAttribute('type').toLowerCase()!='submit')➥
      {continue;i++}
    newLink=document.createElement('a');
    newText=document.createTextNode(inputs[i].getAttribute('value'));
    newLink.appendChild(newText);
    newLink.setAttribute('href','javascript:document.forms[0]➥
      .submit()');
    inputs[i].parentNode.replaceChild(newLink,inputs[i]);
  }
}
window.onload=submitToLinks;
```

When JavaScript is available, the visitor gets a link to submit the form; otherwise, he gets a Submit button, as shown in Figure 4-12.

Figure 4-12. *Depending on JavaScript availability, the user gets a link or a button to send the form.*

However, the function has one major flaw: it will fail when there are more input elements following the Submit button. Change the HTML to have another input after the Submit button:

```
<form action="nogo.php" method="post">
  <p>
    <label for="Name">Name:</label>
    <input type="text" id="Name" name="Name" />
    <input type="submit" value="check" />
    <input type="submit" value="send" />
  </p>
  <p>
    <label for="Email">email:</label>
    <input type="text" id="Email" name="Email" />
  </p>
</form>
```

You will see that the "send" Submit button does not get replaced with a link. This happens because you removed an input element, which changes the size of the array, and the loop gets out of sync with the elements it should reach. A fix for this problem is to decrease the loop counter every time you remove an item. However, you need to check whether the loop is already in the final iteration by comparing the loop counter with the length of the array (simply decreasing the counter would cause the script to fail, as it tries to access an element that is not there):

```
function submitToLinks()
{
  if(!document.getElementById || !document.createTextNode){return;}
  var inputs,i,newLink,newText;
  inputs=document.getElementsByTagName('input');
  for (i=0;i<inputs.length;i++)
  {
    if(inputs[i].getAttribute('type').toLowerCase()!='submit')➥
      {continue;i++}
    newLink=document.createElement('a');
    newText=document.createTextNode(inputs[i].getAttribute('value'));
```

```
    newLink.appendChild(newText);
    newLink.setAttribute('href','javascript:document.forms[0]➥
      .submit()');
    inputs[i].parentNode.replaceChild(newLink,inputs[i]);
    if(i<inputs.length){i--};
  }
}
window.onload=submitToLinks;
```

This version of the script will not fail, and replaces both buttons with links.

■ **Note** There is one usability aspect of forms that this script will break: you can submit forms by hitting the Enter button when there is a Submit button in them. When we remove all Submit buttons, this will not be possible any longer. A workaround is to add a blank image button or hide the Submit buttons instead of removing them. We'll get back to that option in the next chapter. The other usability concern is whether you should change the look and feel of forms at all—as you lose the instant recognizability of form elements. Visitors are used to how forms look on their browser and operating system—if you change that, they'll have to look for the interactive elements and might expect other functionality. People trust forms with their personal data and money transactions—anything that might confuse them is easily perceived as a security issue.

Avoiding NOSCRIPT

The SCRIPT element has a counterpart in NOSCRIPT. This element was originally intended to provide visitors alternative content when JavaScript was not available. Semantic HTML discourages the use of script blocks inside the document (the body should only contain elements that contribute to the document's structure, and SCRIPT does not do that); NOSCRIPT became deprecated. However, you will find a lot of accessibility tutorials on the Web that advocate the use of NOSCRIPT as a safety measure. It is tempting to simply add a message inside a <noscript> tag to the page that explains that you'll need JavaScript to use the site to its full extent, and it seems to be a really easy approach to the problem. This is why a lot of developers frowned upon the W3C deprecating NOSCRIPT or simply sacrificed HTML validity for the cause. However, by using DOM methods, you can work around the issue.

In a perfect world, there wouldn't be any web site that needed JavaScript to work—only web sites that work faster and are sometimes easier to use when scripting is available. In the real world, however, you will sometimes have to use out-of-the-box products or frameworks that simply generate code that is dependent on scripting. When reengineering or replacing those with less obtrusive systems is not an option, you might want to tell visitors that they need scripting for the site to work.

With NOSCRIPT, this was done quite simply:

```
<script type="text/javascript">myGreatApplication();</script>
<noscript>
  Sorry but you need to have scripting enabled to use this site.
</noscript>
```

A message like that is not very helpful—the least you should do is to allow visitors who cannot have scripting enabled (for example, workers in banks and financial companies that turn off scripting as it poses a security threat) to contact you.

Modern scripting tackles this problem from the other side: we give some information and replace it when scripting is available. In the case of an application dependent on scripting, this can be

```
<p id="noscripting">
  We are sorry, but this application needs JavaScript
  to be enabled to work. Please <a href="contact.html">contact us</a>
  If you cannot enable scripting and we will try to help you in other
  ways.
</p>
```

You then write a script that simply removes it and even use this opportunity to test for DOM support at the same time.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Example: Replacing noscript</title>
    <script type="text/javascript">
      function noscript()
      {
        if(!document.getElementById || ➥
          !document.createTextNode){return;}
        // Add more tests as needed (cookies, objects...)
        var noJSmsg=document.getElementById('noscripting');
        if(!noJSmsg){return;}
        var headline='Browser test succeeded';
        replaceMessage='We tested if your browser is capable of ';
        replaceMessage+='supporting the application, and all checked➥
          out fine. ';
        replaceMessage+='Please proceed by activating the following➥
          link.';
        var linkMessage='Proceed to application.';
        var head=document.createElement('h1');
        head.appendChild(document.createTextNode(headline));
```

```
        noJSmsg.parentNode.insertBefore(head,noJSmsg);
        var infoPara=document.createElement('p');
        infoPara.appendChild(document.createTextNode(replaceMessage));
        noJSmsg.parentNode.insertBefore(infoPara,noJSmsg);
        var linkPara=document.createElement('p');
        var appLink=document.createElement('a');
        appLink.setAttribute('href','application.aspx');
        appLink.appendChild(document.createTextNode(linkMessage));
        linkPara.appendChild(appLink);
        noJSmsg.parentNode.replaceChild(linkPara,noJSmsg);
      }
      window.onload=noscript;
    </script>
  </head>
  <body>
    <p id="noscripting">
      We are sorry, but this application needs JavaScript to be
     enabled to work. Please <a href="contact.html">contact us</a>
     if you cannot enable scripting and we will try to help you in
     other ways
    </p>
  </body>
</html>
```

You can see that generating a lot of content via the DOM can be rather cumbersome, which is why for situations like these—where you really don't need every node you generate as a variable—a lot of developers use innerHTML instead.

Shortening Your Scripts via InnerHTML

Microsoft implemented the nonstandard property innerHTML quite early in the development of Internet Explorer. It is now supported by most browsers; there has even been talk of adding it to the DOM standard. What it allows you to do is to define a string containing HTML and assign it to an object. The user agent then does the rest for you—all the node generation and adding of the child nodes. The NOSCRIPT example using innerHTML is a lot shorter:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"➥
  "http://www.w3.org/TR/html4/strict.dtd">
<html dir="ltr" lang="en">
  <head>
    <meta http-equiv="Content-Type"➥
          content="text/html; charset=utf-8">
    <title>Example: Replacing noscript</title>
    <script type="text/javascript">
      function noscript()
      {
        if(!document.getElementById || ➥
          !document.createTextNode){return;}
```

```
        // Add more tests as needed (cookies, objects...)
        var noJSmsg=document.getElementById('noscripting');
        if(!noJSmsg){return;}
        var replaceMessage='<h1>Browser test succeeded<h1>';
        replaceMessage='<p>We tested if your browser is capable of ';
        replaceMessage+='supporting the application, and all checked➥
          out fine. ';
        replaceMessage+='Please proceed by activating the following➥
          link.</p>';
        replaceMessage+='<p><a href="application.aspx">➥
          Proceed to application</a></p>';
        noJSmsg.innerHTML=replaceMessage;
      }
    </script>
  </head>
  <body>
    <p id="noscripting">
      We are sorry, but this application needs JavaScript to be
      enabled to work. Please <a href="contact.html">contact us</a>
      if you cannot enable scripting and we will try to help you in
      other ways.
    </p>
  </body>
</html>
```

It is also possible to read out the `innerHTML` property of an element, which can be extremely handy when debugging your code—as not all browsers have a "view generated source" feature. It is also very easy to replace whole parts of HTML with other HTML, which is something we do a lot when we display content retrieved via Ajax from the back end.

DOM Summary: Your Cheat Sheet

That was a lot to take in, and it might be good to have all the DOM features you need in one place to copy and have on hand, so here you go.

Reaching Elements in a Document

- `document.getElementById('id')`: Retrieves the element with the given `id` as an object
- `document.getElementsByTagName('tagname')`: Retrieves all elements with the tag name `tagname` and stores them in an array-like list

Reading Element Attributes, Node Values, and Other Node Data

- `node.getAttribute('attribute')`: Retrieves the value of the attribute with the name `attribute`
- `node.setAttribute('attribute', 'value')`: Sets the value of the attribute with the name `attribute` to `value`
- `node.nodeType`: Reads the type of the node (`1` = element, `3` = text node)
- `node.nodeName`: Reads the name of the node (either element name or `#textNode`)
- `node.nodeValue`: Reads or sets the value of the node (the text content in the case of text nodes)

Navigating Between Nodes

- `node.previousSibling`: Retrieves the previous sibling node and stores it as an object.
- `node.nextSibling`: Retrieves the next sibling node and stores it as an object.
- `node.childNodes`: Retrieves all child nodes of the object and stores them in an list. There are shortcuts for the first and last child node, named `node.firstChild` and `node.lastChild`.
- `node.parentNode`: Retrieves the node containing `node`.

Creating New Nodes

- `document.createElement(element)`: Creates a new element node with the name `element`. You provide the element name as a string.
- `document.createTextNode(string)`: Creates a new text node with the node value of `string`.
- `newNode =node.cloneNode(bool)`: Creates `newNode` as a copy (clone) of `node`. If `bool` is `true`, the clone includes clones of all the child nodes of the original.
- `node.appendChild(newNode)`: Adds `newNode` as a new (last) child node to `node`.
- `node.insertBefore(newNode,oldNode)`: Inserts `newNode` as a new child node of `node` before `oldNode`.
- `node.removeChild(oldNode)`: Removes the child `oldNode` from `node`.
- `node.replaceChild(newNode, oldNode)`: Replaces the child node `oldNode` of `node` with `newNode`.
- `element.innerHTML`: Reads or writes the HTML content of the given `element` as a string—including all child nodes with their attributes and text content.

DOMhelp: Our Own Helper Library

The most annoying thing when working with the `DOM` is browser inconsistencies—especially when this means you have to test the `nodeType` every time you want to access a `nextSibling`, as the user agent might or might not read the line break as its own text node.

It is therefore a good idea to have a set of tool functions handy to work around these issues and allow you to concentrate on the logic of the main script instead. There are many JavaScript frameworks and libraries available on the Web—the biggest and most up to date is probably prototype (`http://prototype.conio.net/`).

Let's start our own helper method library right here and now to illustrate the issues you'd have to face without it otherwise.

■ **Note** You'll find the `DOMhelp.js` file and a test HTML file in the code demo zip file accompanying this book—the version in the zip has more methods, which will be discussed in the next chapter, so don't get confused.

The library will consist of one object called `DOMhelp` with several utility methods. The following is the skeleton for the utility that we will flesh out over the course of this and the next chapter.

```
DOMhelp=
{
  // Find the last sibling of the current node
  lastSibling:function(node){},

  // Find the first sibling of the current node
  firstSibling:function(node){},

  // Retrieve the content of the first text node sibling of the current node
  getText:function(node){},

  // Set the content of the first text node sibling of the current node
  setText:function(node,txt){},

  // Find the next or previous sibling that is an element
  //  and not a text node or line break
  closestSibling:function(node,direction){},

  // Create a new link containing the given text
  createLink:function(to,txt){},

  // Create a new element containing the given text
  createTextElm:function(elm,txt){},
```

```
  // Simulate a debugging console to avoid the need for alerts
  initDebug:function(){},
  setDebug:function(bug){},
  stopDebug:function(){}
}
```

You've already encountered the last and first sibling functions earlier in this chapter; the only thing that was missing in those was a test for whether there really is a previous or next sibling to check before trying to assign it to the temporary object. Each of these two methods checks for the existence of the sibling in question, and returns `false` if it isn't available:

```
lastSibling:function(node)
{
  var tempObj=node.parentNode.lastChild;
  while(tempObj.nodeType!=1 && tempObj.previousSibling!=null)
  {
    tempObj=tempObj.previousSibling;
  }
  return (tempObj.nodeType==1)?tempObj:false;
},
firstSibling:function(node)
{
  var tempObj=node.parentNode.firstChild;
  while(tempObj.nodeType!=1 && tempObj.nextSibling!=null)
  {
    tempObj=tempObj.nextSibling;
  }
  return (tempObj.nodeType==1)?tempObj:false;
},
```

Next up is the `getText` method, which reads out the text value of the first text node of an element:

```
getText:function(node)
{
  if(!node.hasChildNodes()){return false;}
  var reg=/^\s+$/;
  var tempObj=node.firstChild;
  while(tempObj.nodeType!=3 && tempObj.nextSibling!=null ||➥
    reg.test(tempObj.nodeValue))
  {
    tempObj=tempObj.nextSibling;
  }
  return tempObj.nodeType==3?tempObj.nodeValue:false;
},
```

The first problem we might encounter is that the node has no children whatsoever, therefore we check for `hasChildNodes`. The other problems are embedded elements in the node and whitespace such as line breaks and tabs being read as nodes by browsers other than MSIE. Therefore, we test the first child node and jump to the next sibling until the `nodeType` is `text (3)`, and the node does not consist solely of whitespace characters (this is what the regular expression checks). We also test whether there is a next sibling node to go to before we try to assign it to `tempObj`. If all works out fine, the method returns the `nodeValue` of the first text node; otherwise it returns `false`.

The same testing pattern works for `setText`, which replaces the first real text child of the node with new text and avoids any line breaks or tabs:

```
setText:function(node,txt)
{
  if(!node.hasChildNodes()){return false;}
  var reg=/^\s+$/;
  var tempObj=node.firstChild;
  while(tempObj.nodeType!=3 && tempObj.nextSibling!=null ||➥
    reg.test(tempObj.nodeValue))
  {
    tempObj=tempObj.nextSibling;
  }
  if(tempObj.nodeType==3){tempObj.nodeValue=txt}else{return false;}
},
```

The next two helper methods help us with the common tasks of creating a link with a target and text inside it and creating an element with text inside.

```
createLink:function(to,txt)
{
  var tempObj=document.createElement('a');
  tempObj.appendChild(document.createTextNode(txt));
  tempObj.setAttribute('href',to);
  return tempObj;
},
createTextElm:function(elm,txt)
{
  var tempObj=document.createElement(elm);
  tempObj.appendChild(document.createTextNode(txt));
  return tempObj;
},
```

They contain nothing you haven't already seen here earlier, but they are very handy to have in one place.

The fact that some browsers read line breaks as text nodes and others don't means that you cannot trust `nextSibling` or `previousSibling` to return the next element—for example, in an unordered list. The utility method `closestSibling()` works around that problem. It needs `node` and `direction` (`1` for the next sibling, `-1` for the previous sibling) as parameters.

```
closestSibling:function(node,direction)
{
  var tempObj;
  if(direction==-1 && node.previousSibling!=null)
  {
    tempObj=node.previousSibling;
    while(tempObj.nodeType!=1 && tempObj.previousSibling!=null)
    {
       tempObj=tempObj.previousSibling;
    }
  }
  else if(direction==1 && node.nextSibling!=null)
  {
    tempObj=node.nextSibling;
    while(tempObj.nodeType!=1 && tempObj.nextSibling!=null)
    {
      tempObj=tempObj.nextSibling;
    }
  }
  return tempObj.nodeType==1?tempObj:false;
},
```

The final set of methods is there to simulate a programmable JavaScript debug console. Using `alert()` as a means to display values is handy, but it can become a real pain when you want to watch changes inside a large loop—who wants to press Enter 200 times? Instead of using `alert()`, we add a new `DIV` to the document and output any data we want to check as new child nodes of that one. With a proper style sheet, we could float that `DIV` above the content. We start with an initialization method that checks whether the console already exists and removes it if that is the case. This is necessary to avoid several consoles existing simultaneously. We then create a new `DIV` element, give it an `ID` for styling, and add it to the document.

```
initDebug:function()
{
  if(DOMhelp.debug){DOMhelp.stopDebug();}
  DOMhelp.debug=document.createElement('div');
  DOMhelp.debug.setAttribute('id',DOMhelp.debugWindowId);
  document.body.insertBefore(DOMhelp.debug,document.body.firstChild);
},
```

The `setDebug` method takes a string called `bug` as a parameter. It tests whether the console already exists and calls the initialization method to create the console if necessary. It then adds the bug string followed by a line break to the HTML content of the console.

```
setDebug:function(bug)
{
  if(!DOMhelp.debug){DOMhelp.initDebug();}
  DOMhelp.debug.innerHTML+=bug+'\n';
},
```

The final method is the one that removes the console from the document if it exists. Notice that we both need to remove the element and set the object property to `null`; otherwise testing for `DOMhelp.debug` would be `true` even if there is no console to write to.

```
stopDebug:function()
{
  if(DOMhelp.debug)
  {
    DOMhelp.debug.parentNode.removeChild(DOMhelp.debug);
    DOMhelp.debug=null;
  }
}
```

We will extend this helper library over the course of the next chapters.

Summary

After reading this chapter, you should be fully equipped to tackle any HTML document, get the parts you need, and change or even create markup via the DOM.

You learned about the anatomy of an HTML document and how the DOM offers you what you see as a collection of element, attribute, and text nodes. You also saw the window methods `alert()`, `confirm()`, and `prompt()`; these are quick and widely supported—albeit insecure and clumsy—ways of retrieving data and giving feedback.

You then learned about the DOM, and how to reach elements, navigate between them, and how to create new content.

In the next chapter, you'll learn how to deal with presentation issues and to track how a visitor has interacted with the document in her browser, and to react accordingly via event handling.