# Chapter 10

# Document Object Model and Dynamic HTML

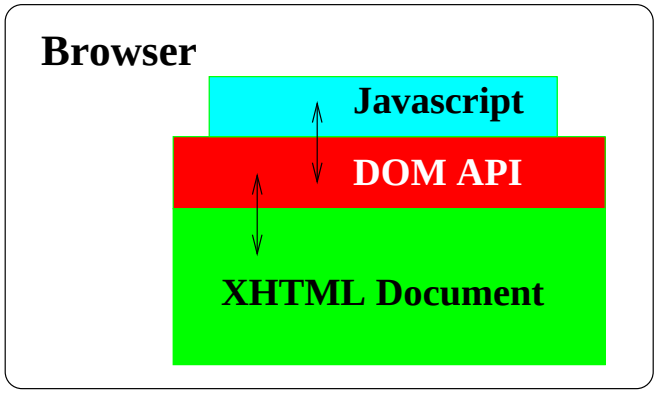
The term *Dynamic HTML*, often abbreviated as DHTML, refers to the technique of making Web pages dynamic by client-side scripting to manipulate the document content and presentation. Web pages can be made more lively, dynamic, or interactive by DHTML techniques.

With DHTML you can prescribe actions triggered by browser events to make the page more lively and responsive. Such actions may alter the content and appearance of any parts of the page. The changes are fast and efficient because they are made by the browser without having to network with any servers. Typically the client-side scripting is written in Javascript which is being standardized. Chapter 9 already introduced Javascript and basic techniques for making Web pages dynamic.

Contrary to what the name may suggest, DHTML is not a markup language or a software tool. It is a technique to make dynamic Web pages via client-side programming. In the past, DHTML relies on browser/vendor specific features to work. Making such pages work for all browsers requires much effort, testing, and unnecessarily long programs.

Standardization efforts at W3C and elsewhere are making it possible to write *standard-based DHTML* that work for all compliant browsers. Standard-based DHTML involves three aspects:

Figure 10.1: DOM Compliant Browser




1. Javascript—for cross-browser scripting (Chapter 9)

2. Cascading Style Sheets (CSS)—for style and presentation control (Chapter 6)

3. *Document Object Model* (DOM)—for a uniform programming interface to access and manipulate the Web page as a document

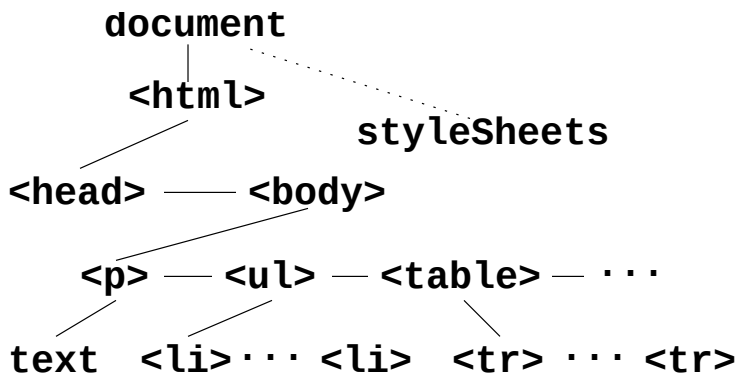
When these three aspects are combined, you get the ability to program changes in Web pages in reaction to user or browser generated events, and therefore to make HTML pages more dynamic.

Popular with Web developers, supported by all major browsers, and standardized, Javascript provides the ability to program browser actions in response to events. To have true cross-platform DHTML, we still need a uniform way for Javascript to access and manipulate Web documents. This brings us to the DOM.

## 10.1 What Is DOM?

With cooperation from major browser vendors, the W3C is establishing the *Document Object Model* (DOM) as a standard *application programming interface* (API) for scripts to access and manipulate HTML and XML documents. Compliant clients, including browsers and other user agents, provide the DOM specified API to access and modify the document being processed (Figure 10.1). The DOM API gives a logical view of the document where objects

Figure 10.2: DOM Tree Structure




represent different parts: windows, documents, elements, attributes, texts, events, style sheets, style rules, etc. These *DOM objects* are organized into a tree structure (the DOM tree) to reflect the natural organization of a document. HTML elements are represented by *tree nodes* and organized into a hierarchy. Each Web page has a `document` node at the root of the tree. The `head` and `body` nodes become *child nodes* of the `document` node (Figure 10.2).

From a node on the DOM tree, you can go down to any child node or go up to the parent node. With DOM, a script can add, modify, or delete elements and content by navigating the document structure, modifying or deleting existing nodes, and inserting dynamically built new nodes. Also attached to the document are its style sheets. Each element node on the DOM tree also contains a *style object* representing the display style for that element. Thus, through the DOM tree, style sheets and individual element styles can also be accessed and manipulated. Therefore, any parts of a page can be accessed, changed, deleted, or added and the script will work for any DOM compliant client.

DOM also specifies events available for page elements. As a result, most events that used to be reserved for links now work for all types of elements, giving the designer many more ways to make pages dynamic and responsive.

## 10.2 A Demonstration

Let's look at a simple example (Ex: **DomHello**) to illustrate DHTML. Figure 10.3 shows a very simple page with the phrase *Hello World Wide Web* on it. And Figure 10.4 shows that phrase becoming blue in a larger font on mouseover. The phrase goes back to normal again on mouseout. This is not an image rollover.

The HTML source for the page is

```
<head><title>Hello WWW with DOM</title>
<script type="text/javascript" src="hwww.js">
</script></head><body>
<p>Move the mouse over the phrase:</p>
<p><span id="hello" onmouseover="over()"
         onmouseout="out()">Hello World Wide Web</span>
---and see what happens.</p>
</body></html>
```

Note we have attached the `onmouseover` and `onmouseout` event handlers to part of a paragraph identified by the `span` with `id="hello"`.

The Javascript defined event handling functions are in the file `hwww.js`:

```
function over()
{   el = document.getElementById("hello");    // (1)
    el.style.color = "blue";                  // (2)
    el.style.fontSize = "18pt";               // (3)
    el.style.fontWeight = "bold";             // (4)
}
```

Figure 10.3: Hello WWW

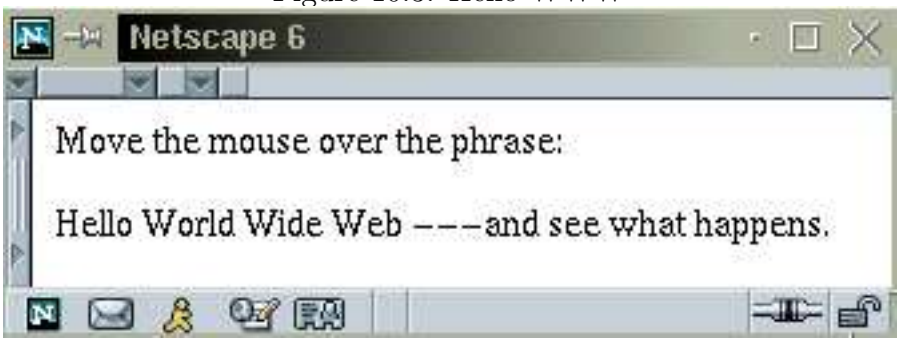


Figure 10.4: Mouse over Phrase


```
function out()
{   el = document.getElementById("hello");
    el.style.color = "";       // sets to default value
    el.style.fontSize = "";
    el.style.fontWeight = "";
}
```

The `over` function obtains a reference to the target `<span>` by using the `getElementById` method of the `document` element (line `1`). The `document` is the root node of the DOM tree and offers many useful properties and methods. The convenient call

`document.getElementById(`*str*`)`

gives you the element with *str* as `id` (or `name`). It returns a reference to the node on the DOM tree that represents the desired element.

Once you have the node for a particular element, you can go to work on that node accessing information from it or making changes to it. Here the `over` function sets style properties for the element (lines `2-4`) causing the `<span>` to display in blue with 18-point boldface font (Figure 10.4). The `out` function sets these style properties to the empty string to go back to the default display (Figure 10.3).

## 10.3 DOM History and Architecture

Early browsers such as NN 3.0 and IE 3.0 have their own object models for representing documents. Starting as early as 1997, the W3C began to organize the DOM working group for establishing a cross-platform and language-neutral standard for access, traversal, and manipulation of document objects. The first W3C recommendation was DOM Level 1 completed in October 1998. DOM Level 1 specifies a *standard object-oriented interface* to HTML and XML documents. The *Level 1 Core* specifies the most central interfaces for the DOM tree. The DOM Level 1 HTML and XML specifications inherit from the Core and specialize in HTML and XML documents, respectively. The DOM specification for HTML/XHTML is most important for website development. The very first DOM specification, informally referred to as DOM Level 0, was built on existing conventions and practices supported by NN 3.0 and IE 3.0. A second edition of DOM Level 1 is being finalized.

In November 2000, DOM Level 2 was completed and it extended Level 1 by adding support for XML 1.0 namespaces, CSS, events and event handling for user interfaces and for tree manipulation, and tree traversal. The Level 2 HTML specification was becoming a W3C recommendation in 2002. DOM Level 3, still being developed, will add more sophisticated XML support, the ability to load and save documents, etc.

As DOM evolves through levels of enhancements, its basic architecture remains stable. The DOM architecture consists of *modules* covering different domains of the document object model:

- DOM Core—specifies the DOM tree, tree nodes, its access, traversal, and manipulation. The DOM Range and DOM Traversal modules provide higher-level methods for manipulating the DOM tree defined by the Core.

- DOM HTML—inherits from the Core and provides specialized and convenient ways to access and manipulate HTML/XHTML documents.

- DOM XML—inherits from the Core and provides support for XML specific needs.

- DOM Events—specifies events and event handling for user interfaces and the DOM tree. With DOM Events, drag and drop programs, for example, can be standardized.

- DOM CSS—defines easy to use ways to manipulate Cascading Style Sheets for the formatting and presentation of documents.

There are other modules and they can be found at the W3C site: `www.w3.org/DOM/`.

When using Javascript to write DOM related code, it is important to realize that not everything has been standardized. In particular, the `window` object is very browser dependent. Also certain fields such as `element.innerHTML` and `document.location`, are not part of the DOM specification.

## 10.4 Browser Support of DOM

Major vendors realize the importance of DOM and have begun to make their Web browsers DOM compliant. NN 7 and IE 6 already have good DOM support. In particular NN led the way in supporting DOM Level 1 and Level 2. Most examples in this chapter will work under both NN 6, IE 6 and later versions.

To detect the extent of DOM support that a user agent (browser) provides, the following type of Javascript code can be used:

```
var imp = document.implementation;
if ( typeof imp != "undefined" &&
     imp.hasFeature("HTML", "1.0") &&
     imp.hasFeature("Events", "2.0") &&
     imp.hasFeature("CSS", "2.0")
   )
{
     . . .
}
```

A browser is DOM compliant if it supports the interfaces specified by DOM. But it can also add interfaces not specified or add fields and methods to the required interfaces. For

example NN and IE both add `innerHTML` to the `HTMLElement` interface. It is easy to test if a field or method is available in a browser. For example,

```
if ( document.getElementById )
  . . .
```

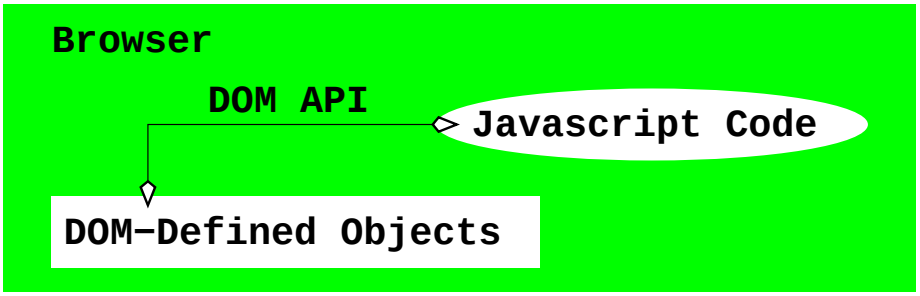
tests if the `getElementById` method is available in the `document` object.

DOM compliance test suites are available from `www.w3.org/DOM/Test/`.

## 10.5 DOM API Overview

DOM is a "platform and language neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page."

Figure 10.5: The DOM API




The DOM specifies an API (application programming interface) and provides a structural view of the document. DOM lists required interface objects and the *methods* (functions in the object) and *fields* (data entries in the object) each object must support. It is up to compliant browsers (agents) to supply concrete implementation, in a particular programming language and environment, for these objects, fields and methods. NN, IE and other browsers support DOM through standard Javascript (Figure 10.5).

As an interface, each DOM object *exposes* a set of fields and methods for Javascript to access and manipulate the underlying data structure that actually implements the document

structure. The situation is like a radio interface exposing a set of nobs and dials for the user. If the radio interface were standardized, then a robot would be able to operate any standard compliant radio.

The DOM tree represents the logical structure of a document. Each tree node is a `Node` object. There are different types of nodes that all *inherit* the basic `Node` interface. Inheritance is an important object-oriented programming (OOP) concept. In OOP, interfaces are organized into a hierarchy where *extended interfaces* inherit methods and properties required by `base interfaces`. The situation is quite like defining various upgraded car models by inheriting and adding to the features of a base model. In DOM, the `Node` object sits at the top of the interface hierarchy and many types of DOM tree nodes are directly or indirectly derived from `Node`. This means all DOM tree node types must support the properties and methods required by `Node`.

On the DOM tree, some types of nodes are *internal nodes* that may have *child nodes* of various types. *Leaf nodes*, on the other hand, have no child nodes. While DOM has many uses, our discussion focuses on *DOM HTML* which applies to HTML documents.

For any Web page, the root of the DOM tree is an `HTMLDocument` node and it is usually available directly from Javascript as `document` or `window.document`. The `document` object implements the `HTMLDocument` interface which gives you access to all the quantities associated with a Web page such as `URL`, `stylesheets`, `title`, `characterSet`, and many others (Section 10.14). The field `document.documentElement` gives you the child node, of type `HTMLElement`, that typically represents the `<html>` element (Figure 10.2). `HTMLElement` (Section 10.9) is the base interface for derived interfaces representing the many different HTML elements.

## DOM Tree Nodes

The DOM tree for a Web page consists of different types of nodes (of type `Node`) including:

`HTMLDocument` —Root of the DOM tree providing access to page-wide quantities, stylesheets, markup elements, and, in most cases, the `<html>` element as a child node.

`HTMLElement` —Internal and certain leaf nodes on the DOM tree representing an HTML markup element. The `HTMLElement` interface provides access to element attributes and child nodes that may represent text and other HTML elements. Because we focus on the use of DOM in DHTML, we will use the terms element and HTML element interchangeably. The `document.getElementById(`*id*`)` call gives you any element with the given *id*.

`Attr` —An attribute in an `HTMLElement` object providing the ability to access and set an attribute. The `name` field (a string) of an `Attr` object is read-only while the `value` field can be set to a desired string. The `attributes` field of an `HTMLElement` object gives you a `NamedNodeMap` of `Attr` objects. Use the `length` property and the `item(`*index*`)` method of the named node map to visit each attribute. All DOM indices are zero-based.

`Text` —A leaf node containing the text inside a markup element. If there is no markup inside an element's content, the text is contained in a single `Text` object that is the only child of the element. The `wholeText` (or `data`) field returns the entire text as a string. Set the `data` string or call the `replaceWholeText(`*str*`)` method to make *str* the new text.

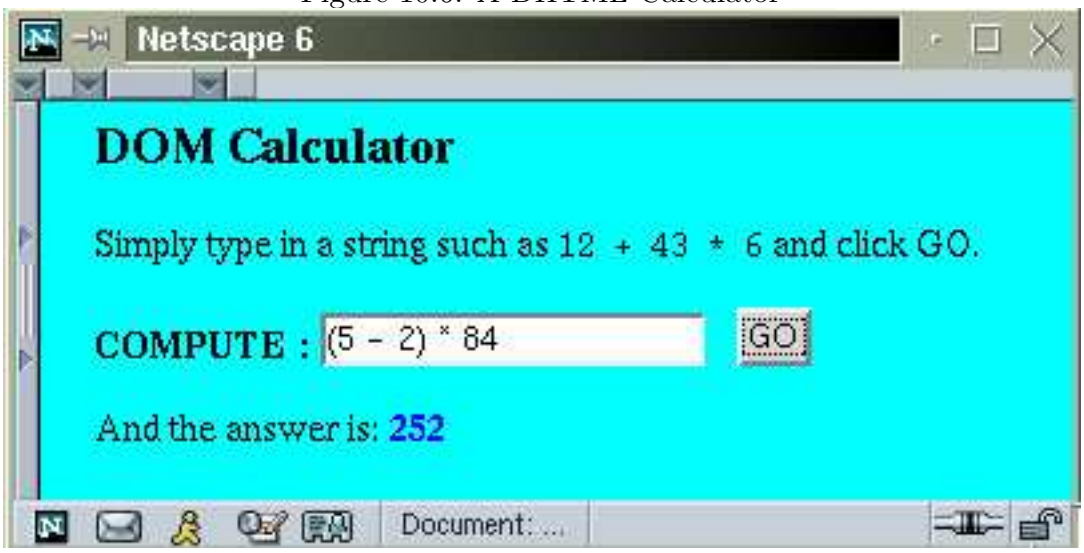
## 10.6 Getting Started with DOM

Let's create a simple calculator (Ex: **DomCalc**) to demonstrate DOM and DHTML. The user enters an arithmetic expression and clicks a button to perform the required computations. The answer is displayed in the regular running text of the page (Figure 10.6).

The HTML source shows the code for the input control (line `A`), the `GO` button (line `B`) and the `<span>` for displaying the computed result (line `C`).

```
<head><title>DOM Calculator</title>
<link rel="stylesheet" href="domcalc.css"
      type="text/css"  title="Dom Calculator" />
```

Figure 10.6: A DHTML Calculator

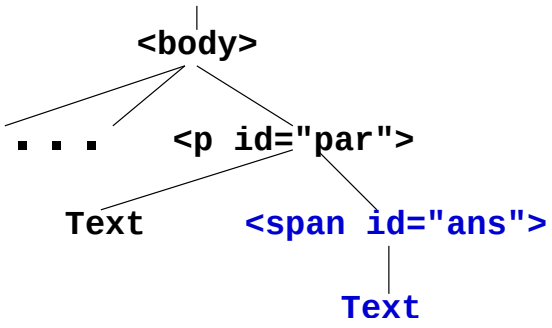



```
<script type="text/javascript" src="domcalc.js"></script>
</head>
<body onload="init()">  /* initialization onload */
<h3>DOM Calculator</h3>
<p>Simply type in a string such as
   <code>12 + 43 * 6</code> and click GO.</p>
<p><strong>COMPUTE : </strong>
  <input id="uin"
         value="(5 - 2) * 8" maxlength="30" />     (A)
    <input value="GO" type="button"
                     onclick="comp('uin')" />      (B)
</p><p id="par">And the answer is:
<span id="ans">00</span></p>                        (C)
</body>
```

The calculator is initialized immediately after page loading. The `init` and the `comp` (line `B`) event handlers are in the Javascript file `domcalc.js`:

```
var answer;

function init()
{  answer = document.getElementById("ans")
                    .firstChild;            // (D)
   comp("uin");
}
```

Figure 10.7: Partial DOM Tree for Calculator Example




```
function comp(id)
{  var el = document.getElementById(id);    // (E)
   var res = eval(el.value);                // (F)
   answer.data = res;                       // (G)
}
```

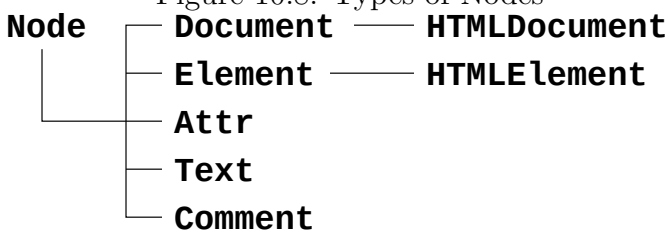
The global variable `answer` is initialized by the `init` function which is called via the `onload` event, an event that takes place immediately after page loading is complete. The variable `answer` holds the `Text` node, a lone child node in this case, of the `ans` `<span>` (line `D`).

The `comp` function is called with the `id` of the user input element. The function obtains the input text as the `value` field of the input element `el` (line `E`), evaluates the input expression (line `F`), and sets the text of the `ans` `<span>` to the result obtained by setting the `data` field of the `Text` node `answer` (line `G`).

Without DOM, Javascript computed results are usually placed in `<input>` or `<textarea>` elements (Ex: **Convert** in Section 9.14). Using the DOM interface, script computed results can be placed anywhere on a displayed page by modifying the DOM tree. Figure 10.7 shows the part of the DOM tree (in dark blue) that is used to display the results for the calculator.

`HTMLDocument` and `HTMLElement` interfaces are important and provide many methods and properties useful in practice. They inherit from the basic `Node` interface which is presented next.

Figure 10.8: Types of Nodes




## 10.7 The DOM `Node` Interface

In object-oriented programming, an interface specifies data values (called *fields*,[1]) and functions (called *methods*) that are made available to application programs.

The `Node` interface is the base of all other node types on the DOM tree and provides useful fields and methods for them.

### `Node` Fields

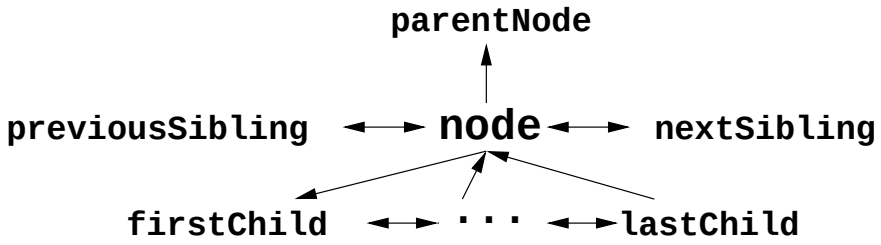
Fields provided by a `Node` are read-only and include:

- `nodeType`—A small integer representing the *derived type* of the node. Figure 10.8 shows common derived node types. The `Node` interface provides symbolic constants, such as `ELEMENT_NODE` and `TEXT_NODE`, for values of `nodeType` (See WDP site for a list). The function `whichType` demonstrates how to determine node type (Ex: **WhichType**):

```
function whichType(nd)                              // (a)
{  if ( nd.nodeType == Node.ELEMENT_NODE )          // (b)
       window.alert("Element Node");
   else if ( nd.nodeType == Node.ATTRIBUTE_NODE )
       window.alert("Attribute Node");
   else if ( nd.nodeType == Node.TEXT_NODE )
       window.alert("Text Node");
   ...
}
```

[1] In the official DOM specification, fields are called *attributes*. To distinguish them from HTML attributes, we use the commonly accepted term *fields* here.

Figure 10.9: Node Relations




The parameter `nd` is any DOM node whose type is to be determined (line `a`). The `nd.nodeType` is compared with the type constants defined by the `Node` interface to determine the node type of `nd` (line `b`).

- `parentNode`, `firstChild`, `lastChild`, `previousSibling`, and `nextSibling`—Related `Node`s of a node (Figure 10.9).

- `nodeName` and `nodeValue`—Strings representing the name and value of a `Node`. The exact meaning of these strings depends on the node type, as shown in Table 10.1. For example the `nodeValue` of any `Element` or `HTMLElement` node is `null`.

Table 10.1: Meaning of `nodeName` and `nodeValue`

| **Node Type** | `nodeName` | `nodeValue` |
|---|---|---|
| `Element` | Tag name | `null` |
| `Attribute` | Attribute name | Attribute value string |
| `Text` | `#text` | Text string |
| `Entity` | Entity name | `null` |
| `Comment` | `#comment` | Comment string |

- `childNodes`—A `NodeList` of child nodes of the node. Some nodes have children and others don't. For a `Document` or an `HTMLElement` node, child nodes represent the HTML elements and text strings contained in that element.

  The `length` field and the `item(i)` method of `NodeList` provide an easy way to visit each node on the node list. For example (Ex: **ChildNodes**), applying the function `visitChildren`:

```
function visitChildren(id)
{  var nd = document.getElementById(id);
   var ch = nd.childNodes;
   var len = ch.length;        // number of nodes
   for ( i=0; i < len; i++)
   {   nd = ch.item(i);        // node i
       window.alert( nd.nodeName + "  "
                     + nd.nodeValue );
   }
}
```

on the element with `id="par"`

```
<p id="par">Here is <img ...  /><br /> a picture.</p>
```

displays this sequence

```
#text   Here is
IMG
BR
#text   a picture.
```

- `attributes`—A `NamedNodeMap` of `Attr` nodes representing HTML attributes of the node. Attribute nodes are not child nodes of a node but are attached to the node via the `attributes` field. DOM defines a `NamedNodeMap` as a collection of nodes accessible by name. Thus, `attributes` is a list of `Attribute` objects representing the HTML attributes for a given node. Let `att = nd.attributes` be the attribute list of some node `nd`, then you can go through all listed attributes with the code (Ex: **AttrAccess**):

```
var len = att.length;
for ( i=0; i < len; i++ )
{   window.alert(att.item(i).name + " = " +
                 att.item(i).value );
}
```

The length of the attribute list `nd.attributes` can be browser dependent. NN lists only attributes set explicitly in the HTML code, whiel IE gives all possible attributes. To examine a specific attribute you can use, for example, the code

```
var b = att.getNamedItem("border");
window.alert(b.value);              // value of border
```

The value returned by `getNamedItem` is a node with the given name in the `NamedNodeMap` or `null`.

The `ownerDocument` field of a node leads you to the root of the DOM tree. It is worth emphasizing that the fields of `Node` are read-only. Assignments to them have no effect.

Also, `NodeList` and `NamedNodeMap` objects in the DOM are *live*, meaning changes to the underlying document structure are reflected in all relevant `NodeList` and `NamedNodeMap` objects. For example, if you get the `childNodes` of an `HTMLElement`, then subsequently add or remove child nodes, the changes are automatically reflected in the `childNodes` you got before. This behavior is usually supported by returning a reference to the data structure containing the actual child nodes of the `HTMLElement`.

### `Node` Methods

In addition to fields, the `Node` interface provides many methods, inherited by all node types. These fields and methods combine to provide the basis for accessing, navigating, and modifying the DOM tree. Specialized interfaces for other node types offer additional features for functionality and convenience.

Among `Node` methods, the following are more frequently used.

- *node*`.normalize()`—Adjusts the subtree rooted at *node* to remove empty nodes and to combine adjacent text nodes. The resulting *normalized DOM tree* has no empty or adjacent text nodes. Before normization, a DOM tree may contain empty and/or adjacent text nodes due to spacing and line breaks in the page source code. Such white space are often used to avoid long lines and to make the source easier to read. For example, the call

  ```
  document.documentElement.normalize();
  ```

normalizes the entire `<html>` node.

- *node*`.hasChildNodes()`—Returns true/false.
- *node*`.hasAttributes()`—Returns true/false.
- *node*`.appendChild(` *child* `)`—Adds *child* as a new child node of *node*.
- *node*`.removeChild(` *child* `)`—Removes the indicated *child* node from the *node*.
- *node* `.insertBefore(` *child* `,` *target* `)`—Adds the *child* node just before the specified *target* child of this *node*.
- *node* `.replaceChild(` *child* `,` *target* `)`—Replaces the *target* child node with the given *child*. If *child* is a `DocumentFragment` then all its child nodes are inserted in place of *target*.

Note, if *child* is already in the DOM tree, it is first removed before becoming a new child. Section 10.14 shows how to create a new node.

## 10.8 DOM Tree Depth-First Traversal

Using the DOM for DHTML basically involves accessing nodes and modifying nodes on the DOM tree. The easiest way to access a target HTML element is to use

`document.getElementById(` *id* `)`

to obtain the node for the element directly by its *id*.

But it is also possible to reach all parts of the DOM tree by following the parent, child, and sibling relationships. A systematic visit of all parts of the DOM tree, a *traversal*, may be performed *depth-first* or *breadth-first*. In depth-first traversal, you finish visiting the subtree representing the first child before visiting the second child, etc. In breadth-first traversal, you visit all the child nodes before visiting the grandchild nodes and so on. These are well-established concepts in computer science.

Let's look at a Javascript program that performs a depth-first traversal (Ex: **DomDft**) starting from any given node on the DOM tree. The example demonstrates navigating the DOM tree to access information.

```
var result="";

function traversal(node)
{   result = "";                                        //  (1)
    node.normalize();                                   //  (2)
    dft(node);                                          //  (3)
    alert(result);                                      //  (4)
}

function dft(node)
{   var children;
    if ( node.nodeType == Node.TEXT_NODE )              //  (5)
       result += node.nodeValue;
    else if ( node.nodeType == Node.ELEMENT_NODE )      //  (6)
    {  openTag(node);                                   //  (7)
       if ( node.hasChildNodes() )                      //  (8)
       {  children = node.childNodes;                   //  (9)
          for (var i=0; i < children.length; i++)       // (10)
              dft( children[i] );
          closeTag(node);                               // (11)
       }
    }
}
```

Given any `node` on the DOM tree, the `traversal` function builds the HTML source code for the node. It initializes the `result` string (line 1), normalizes the subtree rooted at `node` (line 2), calls the depth-first algorithm `dft` (line 3), and displays the result (line 4).

The `dft` function recursively visits the subtree rooted at the `node` argument. It first checks if `node` is a text node (a leaf) and, if true, adds the text to `result` (line 5). Otherwise, if `node` is an element node (representing an HTML element), it adds the HTML tag for the node to `result` by calling `openTag` (line 7), and, if there are child nodes, recursively visits them (lines 8-10) before adding the close tag (line 11). The subscript notation `children[i]` is a shorthand for `children.node(i)`.

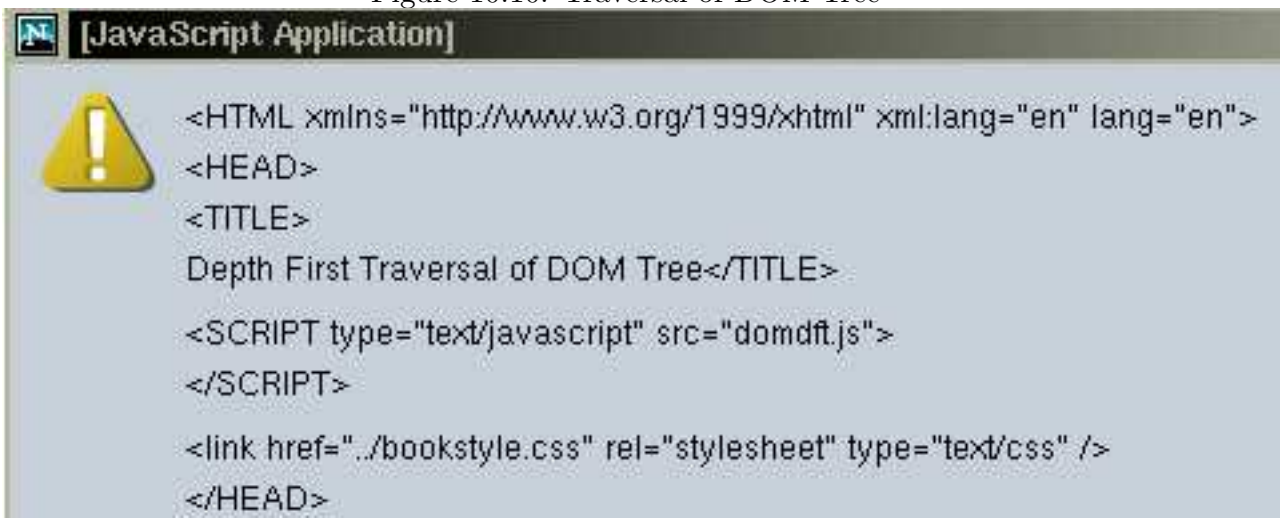
```
function closeTag(node)
{  result += "</" + node.tagName + ">\n";  }

function openTag(node)
{  result += "<" + node.tagName;
   var at;
   if ( node.hasAttributes() )                          // (12)
       tagAttributes(node.attributes);
   if ( node.hasChildNodes() )
       result += ">\n";                                 // (13)
   else
       result += " />\n";                               // (14)
}

function tagAttributes(am)
{  var attr, val;
   for (var i=0; i < am.length; i++)                    // (15)
   {   attr = am[i];  val = attr.value;
       if ( val != undefined && val != null             // (16)
            && val != "null" && val != "" )
       {  result +=  " " + attr.name + "=\"" +          // (17)
                     val + "\"";
       }
   }
}
```

The `openTag` function adds any attributes for the tag (line `12`) by calling `tagAttributes`. The open tag is terminated by either `">"` (line `13`) for non-empty elements or `" />"` for empty elements (line `14`) conforming to the XHTML convention.

The argument `am` of `tagAttributes` is a `NamedNodeMap` of `Attr` nodes. The function goes through each attribute (line `15`) and adds each defined attribute (line `16`) to the `result` string (line `17`). Note the use of the `name` and `value` fields of an `Attr` node.

Figure 10.10 shows the first part of the result of the depth-first traversal when called on the `document.documentElement` node corresponding to the `<html>` element of the page. The complete example can be tested on the WDP site.

Figure 10.10: Traversal of DOM Tree



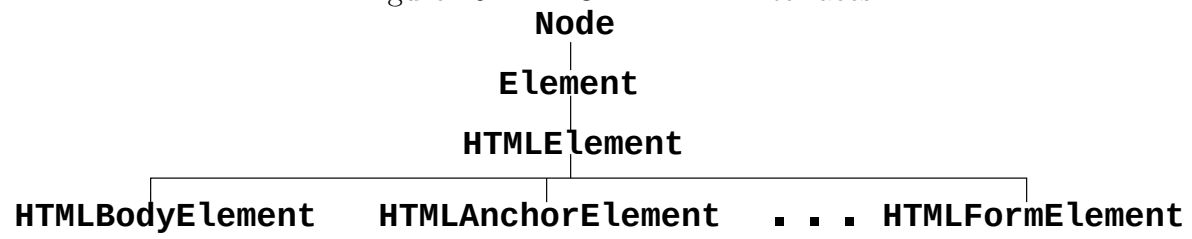

## 10.9 The DOM `HTMLElement` Interface

Derived node types (interfaces extending `Node`) add fields and methods specialized to a particular node type and may provide alternative ways to access some of the same features provided by `Node`. HTML markup elements in a page are represented by nodes extending the base `HTMLElement` which itself extends `Node`. For each element in HTML, DOM HTML provides an interface

`HTML`*FullTagName*`Element`

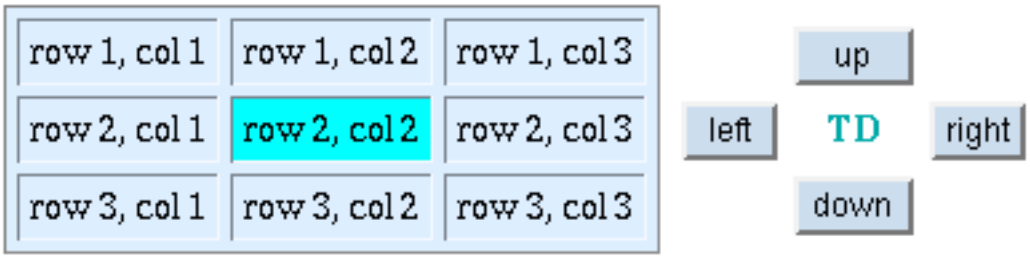
that derives from `HTMLElement` (Figure 10.11). The complete list of all the HTML element interfaces can be found in the DOM HTML specification.

Figure 10.11: DOM HTML Interfaces




The `HTMLElement` interface is rather central for DHTML. Before we systematically discuss the fields and methods of `HTMLElement`, let's see it in action in an example (Ex: **DomNav**)

Figure 10.12: DOM Tree Visual Navigation




where we combine navigation and modification of the DOM tree to achieve the kind of visual effects attributable to DHTML.

We can illustrate DOM tree navigation visually by visiting a subtree representing a `<table>` element, for instance. As you traverse the subtree, the part of the table corresponding to the node being visited will be highlighted. A control panel enables you to go *up* (to the parent node), *down* (to the first child), *left* (to the previous sibling) or *right* (to the next sibling) within the table. The control panel also displays the tag name associated with the current node (Figure 10.12).

The HTML code for the table that we will be traversing is shown here in an easy-to-read form

```
<table id="tbl" border="1"
       style="background-color: #def"
       cellspacing="4" cellpadding="4" >
 <tr><td>row 1, col 1</td>
     <td>row 1, col 2</td>
     <td>row 1, col 3</td></tr>
 <tr><td>row 2, col 1</td>
     <td id="center">row 2, col 2</td>
     <td>row 2, col 3</td></tr>
 <tr><td>row 3, col 1</td>
     <td>row 3, col 2</td>
     <td>row 3, col 3</td></tr>
</table>
```

In the actual file, we eliminate all line breaks and whitespaces between elements to avoid potential extraneous nodes on the DOM tree.

The `init()` function is executed `onload` and sets the stage for the visual navigation:

```
var currentNode, tableNode, nameNode, normal, highlight;

function init()
{  tableNode=document.getElementById("tbl");
   tableNode.normalize();
   highlight="#0ff";
   normal=tableNode.style.backgroundColor;                // (A)
   currentNode=document.getElementById("center");         // (B)
   currentNode.style.backgroundColor  = highlight;        // (C)
   nameNode=document.getElementById("tname").firstChild;
   nameNode.data=currentNode.tagName;                     // (D)
}
```

The Javascript global variables used are:

- `tableNode`—the node for `<table>` that is to be traversed
- `currentNode`—the node for the current traversal position on the `tableNode` subtree
- `nameNode`—the node to display the `tagName` of `currentNode`
- `normal` and `highlight`—the background colors used to indicate visually the part of the table being visited

The `init()` function assigns initial values to these variables. The normal background color is set to the background of the table (line `A`). The center cell of the $3 \times 3$ table is chosen as the starting point of the traversal and `currentNode` is set (line `B`) and highlighted (line `C`). The text of `nameNode`, at the center of the control panel, is set using the `tagName` field of an `HTMLElement` (line `D`). The `init()` function is called `onload`:

```
<body onload="init()">
```

The control panel (Figure 10.12) for interactive traversal is another table

```
<table cellspacing="2" cellpadding="2">
<tr align="center">
   <td></td>
```

```
   <td><input type="button"
              value="  up  " onclick="up()" /></td>
   <td></td></tr>
<tr align="center">
   <td><input type="button" value=" left "
              onclick="left()" /></td>
   <td id="tname" style="color: #0aa;                    (E)
           font-weight: bold">tag name</td>
   <td><input type="button" value="right"
              onclick="right()" /></td></tr>
<tr align="center">
   <td></td>
   <td><input type="button" value="down"
              onclick="down()" /></td>
   <td></td></tr></table>
```

The data cell `id=tname` (line `E`) is used to display the tag name of the current traversal position. The four buttons each triggers a corresponding function that does the obvious. The `up()` function keeps the traversal from leaving the subtree (line `F`).

```
function up()
{  if ( currentNode == tableNode ) return;          // (F)
   toNode(currentNode.parentNode);
}

function down()
{  toNode(currentNode.firstChild); }

function left()
{  toNode(currentNode.previousSibling); }

function right()
{  toNode(currentNode.nextSibling); }
```

Each of these four functions calls `toNode` to visit the new node passed as the argument.

The `toNode` function does the actual work of walking from the current node to the new node given as `nd` (line `G`). If `nd` is `null` or a leaf node (type `TEXT_NODE`), then nothing is done (line`H`). If we are leaving an internal node on the subtree the highlight is removed by calling the `removeAttribute` method of the `HTMLElement` interface (line `I`). If we are leaving the

root `tableNode`, the original background color of the table is restored (line `J`). The arrival node is then highlighted and set as the current node (lines `K-L`). Finally, the tag name of the current node is displayed as the text content of `nameNode` (line `M`)

```
function toNode(nd)                                    // (G)
{  if ( nd == null ||
        nd.nodeType == 3 )  // Node.TEXT_NODE         // (H)
      return false;
   if ( currentNode != tableNode )
      currentNode.style.backgroundColor="";            // (I)
   else
      currentNode.style.backgroundColor = normal;      // (J)
   nd.style.backgroundColor = highlight;               // (K)
   currentNode=nd;                                     // (L)
   nameNode.data=currentNode.tagName;                  // (M)
   return true;
}
```

The example , further illustrates the DOM tree structure, use of the `style` property of HTML elements, and the `tagName` field. It also shows how DHTML can help the delivery of information, enable in-page user interactions, and enhance understanding.

You can find the complete, ready-to-run, version in the example package. You may want to experiment with it and see what it can show about the DOM tree and DHTML.

Assignment 5 suggests adding a display of the table subtree to show the current node position on the subtree as the user performs the traversal.

## 10.10 `HTMLElement` Fields and Methods

Every HTML element is represented on the DOM tree by a node of type `HTMLElement`. The `HTMLELement` interface extends the `Element` interface which, in turn, extends the basic `Node` interface. We list often-used fields and methods available in any node object of type `HTMLELement`.

- `tagName`—is a read-only field representing the HTML tag name as a string.

- `style`—is a field to a style object representing the style declarations associated with an element. For example, use *element*`.style.backgroundColor` to access or set the `background-color` style. Setting a style property to the empty string indicates an inherited or default style. If you change the style of an element by setting its `style` attribute instead, the new `style` attribute replaces all existing style properties on that element and, normally, that is not what you want to do. It is advisable to use the `style` field to set individual style properties you wish to change.

- `innerHTML`—is a read-write field representing the HTML source code contained inside this element as a string. By setting the `innerHTML` field of an element, you replace the content of an element. This useful field is not part of the DOM specification but supported by all major browsers.

- `getAttribute(`*attrName*`)`—returns the value of the given attribute *attrName*. The returned value is a string, an integer, or a `boolean` depending on the attribute. Specifically, a `CDATA` (character data) value is returned as a string; a `NUMBER` value is returned as an integer; an on-or-off attribute value is returned as a `boolean`. A value from an allowable list of values (e.g. `left|right|center`) is returned as a string. For an attribute that is unspecified and does not have a default value, the return value is an empty string, zero, or `false` as appropriate.

- `setAttribute(`*attrName*`,` *value*`)`—sets the given attribute to the specified string *value*.

- `removeAttribute(`*attrName*`)`—removes any specified value for *attrName* causing any default value to take effect.

- `hasAttribute(`*attrName*`)`—returns `true` if *attrName* is specified for this element, `false` otherwise.

When setting values, use lower-case strings for attribute names and most attribute values. When checking strings obtained by `tagName` or `getAttribute()`, be sure to make case-insensitive comparisons to guard against non-uniformity in case conventions. For example,

```
var nd = node1.firstChild;
var re = /table/i;
if ( re.test(nd.tagName) )
{  ...  }
```

tests a `tagName` with the case-insensitive pattern `/table/i`.

HTML input control elements have these additional fields and methods

- `name` and `value`—are the name and value strings of an element to be submitted with a form.
- `focus()`—causes the input element to get *input focus* so it will receive keyboard input.
- `blur()`—causes the input element to lose input focus.
- `select()`—selects the current textual content in the input element for user editing or copying.
- `click()`—causes a click event on the element.

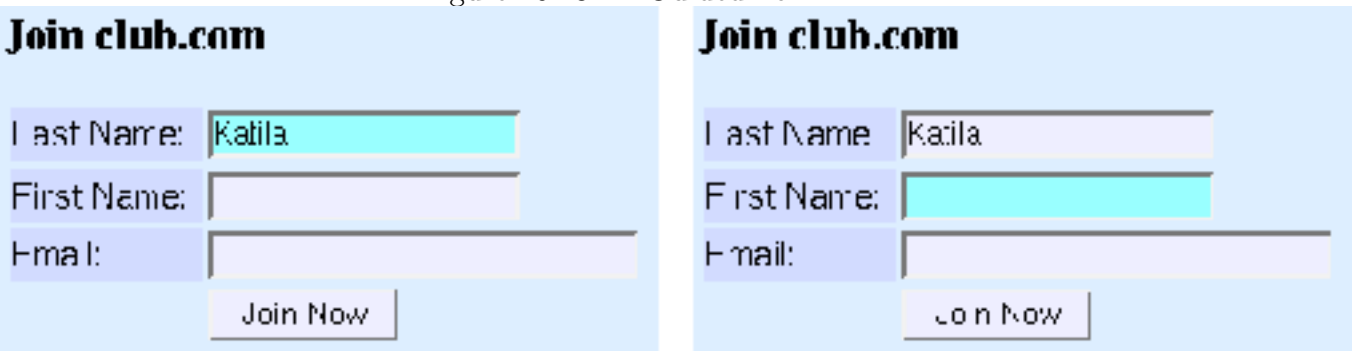
## 10.11 A Guided Form

Let's look at a practical example of DHTML where we use a combination of style, DOM, and Javascript to implement a *guided form* (Ex: **GuidedForm**). The idea is simple, we want to guide the end user visually through the form. This can be done by highlighting the input field that has keyboard focus (Figure 10.13). With a regular form, it is hard to spot which field has input focus. Thus, a guided form can be more user friendly and less confusing.

Here is the HTML code for this example:

```
<head><title>DOM Example: Guided Form</title>
<link rel="stylesheet" href="guidedform.css"
      type="text/css" title="guided form" />
<script type="text/javascript" src="guidedform.js">
</script></head>
```

Figure 10.13: A Guided Form




```
<body onload="init()" style="background-color: #def">
<form method="post" action="/cgi-bin/wb/join.cgi">
<p style="font-weight: bold; font-size: larger">
Join club.com</p>
<table width="280">
<tr><td class="fla">Last Name:</td>
    <td><input onfocus="highlight(this)"                (1)
               onblur="normal(this)"                    (2)
               name="lastname" size="18" /></td></tr>
<tr><td class="fla">First Name:</td>
    <td><input onfocus="highlight(this)"
               onblur="normal(this)"
               name="firstname" size="18" /></td></tr>
<tr><td class="fla">Email:</td>
    <td><input onfocus="highlight(this)"
               onblur="normal(this)"
               name="email" size="25" /></td></tr>
<tr><td></td>
    <td><input onfocus="highlight(this)"
             onblur="normal(this)"
             type="submit" value="Join Now" /></td></tr>
</table></form></body>
```

The four input controls, last name, first name, email, and `submit`, call `hightlight` onfocus (line 1) and `normal` onblur (line 2). The style of the form is defined in the `guidedform.css` file.

```
td.fla
{  background-color: #d2dbff;
   font-family: Arial, Helvetica, sans-serif
```

```
}

form input, select, textarea
{  background-color: #eef  }                /* (3) */
```

The background color of input controls have been softened a bit from pure white (line 3).

The actual highlighting is done by these Javascript functions:

```
var base="", high;

function init() {  high = "#9ff"; }         // (4)

function highlight(nd)
{  base = nd.style.backgroundColor;         // (5)
   nd.style.backgroundColor=high;           // (6)
}

function normal(nd)
{  nd.style.backgroundColor=base; }         // (7)
```

The `init()` function, called onload, defines the highlight color to use. Before highlighting an input field (line 6), its background color is saved in the global variable `base` (line 5) for later restoration. The `onblur` event handler `normal` restores the original background color (line 7).

When experimenting with this example, you can move the input focus with the mouse or forward with TAB and backward with SHIFT TAB.

## 10.12 Fade-in Headlines

Another useful DHTML effect provides an easy and efficient way to include page headlines that move into place as they fade in (Ex: **FadeIn**). The effect calls attention to the headline and gives the page a touch of animation.

The HTML code for such a centered headline involves

```
<body id="bd" onload="centerNow('ct', 60, 50, 40, 25)">
```

```
<p id="ct" class="headline"
   onclick="centerNow('ct', 60, 50, 40, 25)">Super ABC Company</p>
```

The style of the headline is given by a style rule such as

```
p.headline
{   text-align: center;
    font-family: verdana, arial, helvetica;
    font-size: x-large;
    font-weight: bold;
}
```

The Javascript function `centerNow` performs the centering while fading-in animation. You specify the target color of the headline and the number of animation steps, it does the rest. The function call

`centerNow(id,` *r*`,` *g*`,` *b*`,` *steps*`)`

gives the *id* of the headline element (`ct` in our example), the red, green, and blue components of the target color `rgb(`*r*`%,` *g*`%,`*b*`%)`, and an integer *steps*, the number of steps for the animation.

To move the text from left to center, we use increasingly smaller right margins for the centered text. Typically, we can begin with a `60%` right margin and decrease it down to `0%` in the given number of steps. The code to set the right margin is

*element*`.style.marginRight =` *setting* `+ "%";`

To achieve fade-in, we can repeatedly set the `color` style property

*element*`.style.color = rgb(`*red*`,` *green*`,` *blue*`)`

The animation can begin with the most-faded color and gradually change to the least-faded color (the target color) in the given number of steps. To fade a color we increase the rgb components while keeping their ratios. All colors used must keep the ratios among the rgb components as closely as possible to the original ratios in the target color. See Figure 10.14 for a sample set of headline fade-in colors.

Here is a procedure to compute the most-faded color:

Figure 10.14: Color Fade In


1. Let the target color be `rgb(`*r0*`%, `*g0*`%, `*b0*`%)`.

2. Let *high* be the maximum of *r0*, *g0* and *b0*.

3. Let *m* be a multiplier such that *m* `*` *high* `= 100`.

4. The most faded color is `rgb(`*m*`*`*r0*`%, `*m*`*`*g0*`%, `*m*`*`*b0*`%)`.

The multiplier $m$ is $\geq 1$. The fade-in can then be done by multiplying the target color by a sequence of numbers from $m$ to 1 in the given number of steps.

Now, let's see how this is achieved in Javascript.

```
var steps, mar, m, r, g, b;
var sty=null;

function centerNow(nd, rr, gg, bb, st)
{   mar = 60;                                      // (A)
    steps = st;                                    // (B)
    r =rr;  g =gg;  b=bb;                          // (C)
    if ( sty == null )
        sty = document.getElementById(nd).style; // (D)
    margin_d = mar/steps;                          // (E)
    m = 100/Math.max(Math.max(r, g), b);           // (F)
    color_d = (m-1.0)/steps;                       // (G)
    centering();
}
```

The `centerNow` function initializes the starting right margin, the total number of animation steps, and the target color values (lines `A-C`). The `sty` (line `D`), the style of the headline element, will be used repeatedly. The percentage setting of the right margin will decrease by `margin_d` (line `E`) after each step. The multiplier `m` begins with the maximum value (line `F`) and decreases by `color_d` (line `G`) after each step.

With these values set, `centering()` is called to perform the actual animation.

```
function centering()
{  if (steps > 1)
   {  sty.marginRight = mar+"%";                // margin
      sty.color="rgb(" +r*m+ "%," +g*m+ "%,"
                  +b*m+ "%)";                   // color
      mar -= margin_d;                          // decrements
      m -= color_d;
      steps--;
      setTimeout("centering()", 18);            // (H)
   }
   else   // final position and color           // (I)
   {  sty.marginRight="0%";
      sty.color="rgb("+r+"%,"+g+"%,"+b+"%)";
   }
}
```

The `centering` function performs one step of the animation. It sets the right margin and color and decrements the quantities that will be used in the next step. The Javascript built-in function `setTimeout` (Section 9.17) schedules the call to `centering` after 18 milliseconds (line `H`). A smooth animation requires some 30 frames per second, making the delay between each step about 33 milliseconds. The last step of the animation (line `I`) makes sure we have the correct centered position and the true target color, without floating-point errors.

You can easily modify this example for similar visual effects. For example, the in-place fade-in (Figure 10.15) of a centered headline (Ex: **InPlace**) can be done with the same color technique plus changes in the letter spacing. Recall the style property `letter-spacing` controls the spacing between characters in a piece of text (Section 6.14).

With the variable `sp` set to `1` at first. The Javascript statements

Figure 10.15: In-place Fade-in


```
sty.letterSpacing = sp + "px";
sp++;
```

can be included in a fade-in function that is called for a given number of steps to fade in any target headline. The full example (Ex: **FadeIn**) can be found at the WDP site and in the example package.

## 10.13 Mouse Tracking

DOM also specifies an `Event` interface to provide standards for an event system, event names, event registration, and event objects.

For example a `MouseEvent` object has the `clientX` and `clientY` fields giving, respectively, the x and y coordinates of the mouse event position in the document display area. Using these coordinates associated with the `mousemove` event, we can *drag* an element by moving the mouse.

The following HTML code displays an image, a crystal ball, that when clicked will follow the mouse until the mouse is clicked again (Ex: **DragDrop**).

```
<head><title>Drag and Drop</title>
<script type="text/javascript"  src="dragdrop.js"></script>
</head> <body onload="init()">
<div id="ball"  onclick="drag()"                              (1)
     style="position: absolute;
            top: 20px; left: 20px; z-index: 1">
<img src="crystalball_stand.gif" width="110" height="142"
     alt="Crystal ball on a stand">
</div></body>
```

The technique is straight forward. A mouse click calls the Javascript function `drag()` (line 1) that sets up the `trackMouse` event handler for the `mousemove` event (line 3). Mouse tracking changes the absolute position of the crystal ball. The `left` and `top` style properties are set to the event coordinates plus any scrolling that may have taken place for the browser window (lines 5-6).

```
// file: dragdrop.js
var ball, ballstyle;

function init()
{  ball = document.getElementById('ball');
   ballstyle = ball.style;
}

function drag()
{  if ( document.onmousemove )
      document.onmousemove = null;                    // (2)
   else
      document.onmousemove = trackMouse;              // (3)
}

function trackMouse(e)                                // (4)
{  var x = e.clientX + window.scrollX;                // (5)
   var y = e.clientY + window.scrollY;                // (6)
   ballstyle.left = x + "px"; // left style property
   ballstyle.top = y + "px";  // top style property
}
```

A second mouse click calls `drag()` and cancels the mouse tracking (line 2).

## 10.14 The DOM `HTMLDocument` Interface

Browsers display Web pages in windows. Each window has a unique `document` object that represents the entire Web page displayed in the window. The `document` object contains all other elements in the page.

The `document` object, implementing the DOM `HTMLDocument` interface which inherits from the `Document` interface, offers fields and methods useful for page-wide operations.

### HTMLDocument Fields

A select set of fields available from the `document` object is listed here.

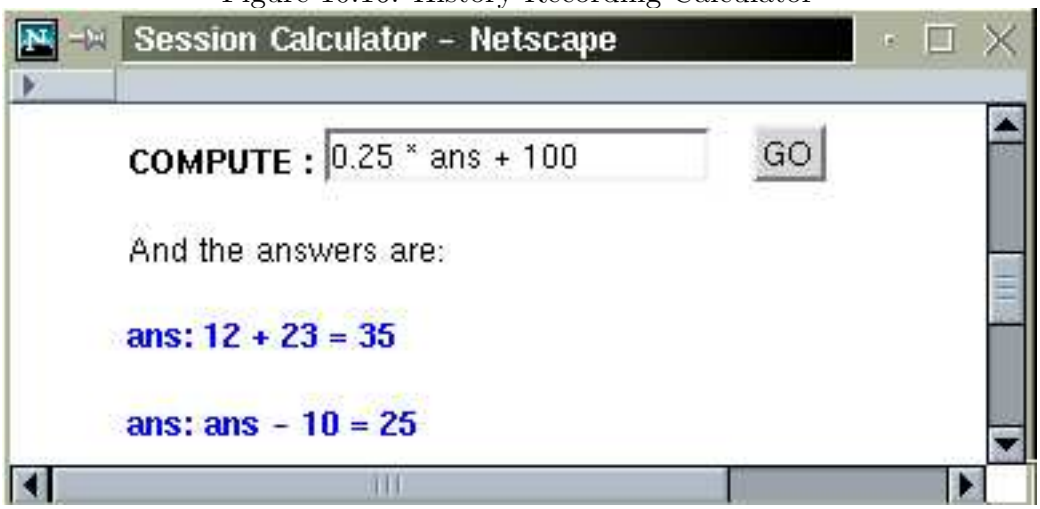
- `documentElement`—the `<html>` element of the page.
- `body`—the `<body>` element of the page.
- `URL`—a read-only string for the complete URL of the page.
- `title`—the title string specified by `<title>`.
- `referrer`—the read-only URL of the page leading to this page (empty string if no referrer).
- `domain`—the read-only domain name of the Web server that supplied the page.
- `cookie`—a SEMICOLON-separated string of *name*=*value* pairs (the cookies) associated with
- `anchors`, `applets`, `forms`, `images`, `links`—read-only lists of different elements in the page: `<a>` elements as named anchors, `<applet>` elements, `<form>` elements, `<img>` elements, and `<a>` and `<area>` elements as `href` links, respectively. Such a list has a `length` field, an `item(`*n*`)` method, and a `namedItem(`*name*`)` method which returns an element with *name* as `id` or, failing that, as `name`.

### HTMLDocument Methods

Frequently used methods of the `document` object include:

- `createElement(`*tagName*`)`—returns a newly created element object for the `<`*tagName*`>` element. By setting attributes and adding child nodes to this element, you can build a DOM structure for any desired HTML element.
- `createTextNode(`*textString*`)`— returns a node of type `TEXT_NODE` containing the given `textString`.

Figure 10.16: History Recording Calculator




- `getElementById(`*id*`)`— returns the unique HTML element with the given *id* string. We have seen this method used often.

- `getElementsByTagName(`*tag*`)`—returns a list of all elements with the given *tag* name in the document.

## 10.15 Generating New Content

Applying the features discussed in the previous section, let's do more with DHTML by adding new content to a displayed HTML page. The content is computed by Javascript, built into element nodes, and placed on the DOM tree.

### A Session-Recording Calculator

To get started, we can take the interactive calculator example (Ex: **DomCalc**) shown in Figure 10.6 and make it more useful by recording the current answer and displaying a history of computation steps. The answer from the previous step can be used in the next step (Figure 10.16).

The HTML code for the session calculator (Ex: **DomSession**) is revised slightly from

that for the basic calculator of Ex: **DomCalc**:

```
<body onload="init()">
<h3>DOM Calculator Session</h3>
<p>Simply type in a string such as
   <code>12 + 43 * 6</code> and click GO.</p>
<p><strong>COMPUTE : </strong>
  <input id="uin" name="uin" maxlength="30" />
    
  <input value="GO" type="button"
         onclick="comp('uin')" /></p>
<p>And the answers are:</p>
<div id="session"> </div></body>                    (A)
```

The empty `<div>` is where the computation session will be displayed (line `A`).

The `init()` function (line `B`) is called onload to obtain the `<div>` element and store it in the global variable `session`.

```
var session, ans = 0;  // global variables

function init()
{   session=document.getElementById("session");    // (B)
}

function comp(id)
{   var input = document.getElementById(id);
    var str = input.value;                         // (C)
    ans = eval(str);                               // (D)
    var ansNode = document.createTextNode("ans: "
                            + str + " = " + ans); // (E)
    var parNode = document.createElement("p");     // (F)
    parNode.appendChild(ansNode);                  // (G)
    session.appendChild(parNode);                  // (H)
    input.value="";                                // (I)
}
```

The `comp` function, triggered by the `GO` button, obtains the user input (line `C`) and evaluates it. The Javascript function `evel` (line `D`) takes any string and executes it as code. The result obtained is stored in the global variable `ans` (line `D`) which can be used in the subsequent step. To record the computation step, it creates a new text node (line `E`), wraps a `<p>`

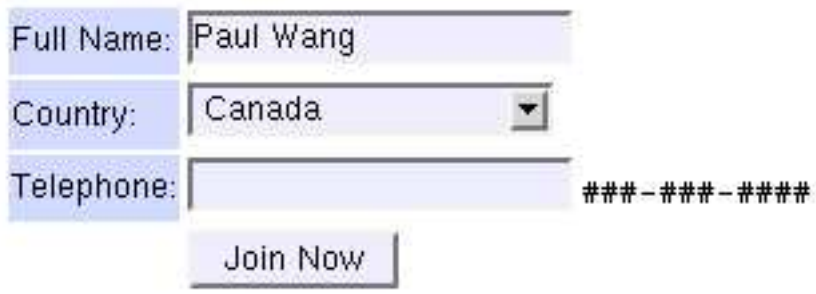
element around it (lines F-G), and appends the element as a new (last) child of the session  (line H). Finally, the input field is cleared (line I), ready for the next step. Users may use ans in the next computation step to perform a session of stpes (Figure 10.16). Further, users may store values for use in subsequent steps by creating their own variables with input strings such as:

```
taxRate = 0.08
total = subtotal + subtotal * taxRate
```

## 10.16 A Smart Form

As another example of dynamically adding and removing page content, let's add some smarts to the guided form discussed in Section 10.11 (Figure 10.13).

Figure 10.17: Smart Form I




A website in North America may collect customer address and telephone information without asking for an *international telephone country code*. But, if the customer selects a country outside of North America, it may be a good idea to require this information as well. In may situations, the information to collect on a form can depend on data already entered on the form. It would be nice to have the form dynamically adjusts itself as the user fills out the form. This can be done with DHTML.

As an example, let's design a smart form (Ex: **SmartForm**) that examines the country setting in the address part of the form and adds/removes (Figures 10.17 and 10.18) an input field for the international telephone code.

Our strategy is straight-forward:

1. When the country name is selected, the `onchange` event triggers a call to check the country name.

2. Any country outside North America causes an input field to be added to obtain the telephone country code.

3. If the country is inside North America, then any telephone country code input field is removed.

Figure 10.18: Smart Form II


The HTML code is as follows:

```
<head><title>DOM Example: Smart Form</title>
<link rel="stylesheet" href="guidedform.css"
      type="text/css" title="dynamic guided form" />
<script type="text/javascript" src="smartdform.js">
</script>
</head>
<body onload="init()" style="background-color: #def">
<form method="post" action="http:/cgi-bin/join.pl">
<p style="font-weight: bold; font-size: larger">
Join club.com</p>
```

```
<table><tbody id="tb">                                          (A)
<tr><td class="fla">Full Name:</td>
  <td><input onfocus="highlight(this)"
         onblur="normal(this)"
         name="fullname" size="20" /></td></tr>
<tr><td class="fla">Country:</td>
  <td><select  id="country" name="country"
         size="1"
         onfocus="highlight(this);"
         onchange="countryCode(this);"                          (B)
         onblur="normal(this);" >
       <option value="US">USA</option>
       <option value="CA">Canada</option>
       <option value="MX">Mexico</option>
       <option value="CN">China</option>
       <option value="RU">Russian Federation
       </option>
  </select></td></tr>
<tr><td class="fla" >Telephone:</td>
  <td><input onfocus="highlight(this)"
       onblur="normal(this)"
       name="phone" size="20" /><span id="pinst">               (C)
       ###-###-####</span></td></tr>
<tr id="bt"><td></td>                                           (D)
  <td><input onfocus="highlight(this)"
         onblur="normal(this)"
         type="submit" value="Join Now" /></td>
</tr></tbody></table></form></body>
```

The `onchange` event of `<select>` triggers the function `countryCode` (line `B`) which can add/remove a form entry for the telephone country code. The new form entry will be a new table row element, a child `<tr>` of `<tbody>` (line `A`) inserted just before the row (line `D`) for the submit button.

The `init()` function, executed onload, sets the telephone instruction node (`inode`) to the `pinst span` (lines `C` and `1`). The text child of `inode` (line `2`) can be replaced later by a generic instruction (`oph`) for other countries (line `3`).

```
var oph, iph, inode;
```

```
function init()
{  inode = document.getElementById("pinst");   // (1)
   iph = inode.firstChild;                     // (2)
   oph = document.createTextNode(
         " AreaCode-Phone Number");            // (3)
}
```

Two additional global variables (line 4) are used: `crow` (the table row to be created) and `cc` (the `<input>` element for the telephone country code). The `isLocal` function checks to see if a country is local to North America (line 5).

```
var crow = null, cc=null;                      // (4)

function isLocal(ct)                           // (5)
{  return (ct == "US" || ct == "CA"
                      || ct == "MX" );
}

function countryCode(country)
{  var d1, d2, t1, t2, button;
   var tbody = document.getElementById("tb");
   if ( isLocal(country.value) )               // (6)
   {  if ( crow != null )
      {  tbody.removeChild(crow);              // (7)
         inode.replaceChild(iph, oph);
         cc = crow = null;                     // (8)
      }
      return;
   }
// country outside North America
   if ( crow != null )                         // (9)
   {  cc.value = ""; return;  }                // (10)

   crow = document.createElement("tr");        // (11)
   crow.appendChild(makeLabel());              // (12)
   crow.appendChild(makeCC());                 // (13)

   button = document.getElementById("bt");     // (14)
   tbody.insertBefore(crow, button);           // (15)
   inode.replaceChild(oph, iph);               // (16)
}
```

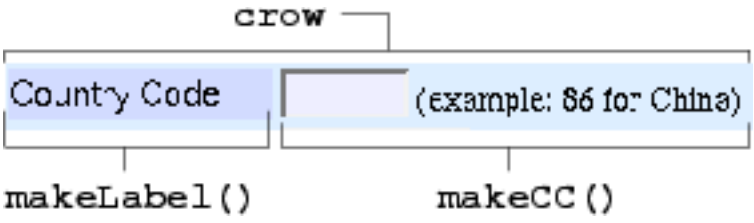
A call to `countryCode` is triggered by the `onchange` event on the `<select>` element for the country part of an address. If the given `country` is in North America, it removes any telephone country code entry from the form, restores the phone instructions, resets the global variables, and then returns (line 6-8).

The function continues to process `country` which is outside of North America. If the telephone country code entry is already displayed (line 9), it simply makes sure any previously entered code is removed and returns (line 10). Otherwise, a new `<tr>` element is created (line 11), filled with two table cells (lines 12-13), and inserted into the table body just before the submit button (lines 14-15). The generic phone instruction is also put in place (line 16)

Each of the following functions makes a `<td>` element for the table row needed (Figure 10.19). The `input` element is made to match the style and dynamic behavior of other input controls in the form (lines 17-18). A text label is created by `makeLabel()` and the actual input element for the telephone country code is created by `makeCC()` which also sets the global variable `cc`.

```
function makeLabel()
{  var t, d;
   d = document.createElement("td");
   d.setAttribute("class", "fla");
   t = document.createTextNode("Country Code:");
   d.appendChild(t);
   return d;
}

function makeCC()
{  var t, d;
   d = document.createElement("td");
   cc = document.createElement("input");            // (17)
   cc.setAttribute("onfocus", "highlight(this)");
   cc.setAttribute("onblur", "normal(this)");
   cc.setAttribute("name", "cc");
   cc.setAttribute("id", "cc");
   cc.setAttribute("size", "7");                    // (18)
   d.appendChild(cc);
   t = document.createTextNode("(example: 86 for China)");
   d.appendChild(t);
```

Figure 10.19: Creating A Form Entry




```
   return d;
}
```

Note these element creation functions use `setAttribute` to set up many attributes so the newly created form entry fits in with the style and dynamic behavior on this smart form.

Experiment with Ex: **SmartForm** and see for yourself.

One small problem for this form shows up when you click the `back` button after submitting the form, perhaps because of an error in filling the form. The form contains the data you entered, but the telephone code entry disappears. It is possible to add code to the `init()` function called onload to fix this problem and the solution is left as an exercise (Assignment 8).

## 10.17 Reordering Rows in Tables

Applying DHTML, we can make tables more usable by allowing the end user to reorder rows based on the contents of cells in any given table column. Thus, tables representing invoices, shopping carts, airfares, addresses, student grades, and so on, can be sorted at will by the end user. The user may want to list the largest amount first, the least expensive item first, or names alphabetically. With DHTML, the user can do this by clicking the mouse and the sorting will be performed by client-side Javascript. Not going back to the server, the redisplay is instantaneous and very dynamic.

For example, the shopping cart in Figure 10.20 is in increasing `Amount`. The same shopping cart is shown in Figure 10.21 in increasing unit `Price`.

Figure 10.20: Shopping Cart Sorted by Amount

**Your Cart**

| Item | Code | Price | Quantity | Amount |
|---|---|---|---|---|
| Shovel | G01 | 14.99 | 2 | 29.98 |
| Power Saw | P12 | 34.99 | 1 | 34.99 |
| Hand Shovel | T01 | 4.99 | 10 | 49.90 |
| Hand Saw | H43 | 24.99 | 5 | 124.95 |
| | | | Subtotal: | 239.82 |

In this example (Ex: **TableSort**), clicking (double-clicking) on a table header cell sorts that column in increasing (decreasing) order.

## Sortable Table Organization

The HTML code for the sortable table is organized as follows.

- The first table row is placed within a `<thead>` element and contains table header (`<th>`) cells connected to event `onclick` and `ondblclick` handlers. These cells have a button look to visually indicate their active nature. Here is a typical `<th>` cell:

```
<th class="button"
    onclick="sortTable(2, 'num', '1');"
    ondblclick="sortTable(2, 'num', '-1');">Price</th>
```

  The arguments to `sortTable` are column position (a zero-based index), numerical or alphabetical ordering (`num` or `str`), and increasing or decreasing order (`1` or `-1`). Here is the header for the `Item` column

```
<th class="button"
    onclick="sortTable(0, 'str', '1');"
    ondblclick="sortTable(0, 'str', '-1');">Item</th>
```

- The sortable table rows are organized in a `<tbody>` group with a given `id`, `tb` in our example. Each sortable column must contain all numbers or all text strings. Here is a typical row

```
<tr id="cc" valign="middle" align="right">
    <th>Hand Shovel</th>
    <td align="center">T01</td>
    <td>4.99</td> <td>10</td> <td>49.90</td></tr>
```

The table layout and the button look for clickable table header cells are created with CSS rules (Section 6.12):

```
table.sort tr { background-color:#f0f0f0; }

table.sort th.button
{   background-color: #fc0; border-width: 3px;
    border: outset; border-color: #fc0
}
```

## Table Sorting

Now let's look at the Javascript code for table sorting.

As stated in the previous subsection, `onclick` and `ondblclick` events on an active table header trigger calls to the `sortTable` function with appropriate arguments: column position (`c`), numerical or alphabetical ordering (`n`), and increasing or decreasing direction(`d`).

```
var col=null, numerical=false, direction=1;

function sortTable(c, n, d)
{  if ( col==c && Number(d)==direction ) return;      // (a)
   col=c;                                             // (b)
   direction = Number(d);
```

Figure 10.21: Shopping Cart Sorted by Price

**Your Cart**

| Item | Code | Price | Quantity | Amount |
|---|---|---|---|---|
| Hand Shovel | T01 | 4.99 | 10 | 49.90 |
| Shovel | G01 | 14.99 | 2 | 29.98 |
| Hand Saw | H43 | 24.99 | 5 | 124.95 |
| Power Saw | P12 | 34.99 | 1 | 34.99 |
| | | | Subtotal: | 239.82 |

```
   numerical = (n == "num");                          // (c)
   var tbody = document.getElementById("tb");         // (d)
   var r = tbody.childNodes;                          // (e)
   n = r.length;
   var arr = new Array(n);
   for ( i=0; i < n; i++ ) arr[i]=r.item(i);          // (f)
   quicksort(arr, 0, n-1);                            // (g)
   for ( i=0; i < n; i++ ) tbody.appendChild(arr[i]); // (h)
}
```

If `c` is the same as the recorded column position `col` and `d` the same as the recorded sorting direction (line `a`), the sorting has already been done and the `sortTable` returns immediately. To prepare for sorting, the arguments are stored in the global variables (lines `b-c`). The child nodes of `<tbody>` are copied into a new array `arr` (lines `d-f`). The copying is needed because `tbody.childNodes` is a read-only list. The notation (line `f`)

```
r.item(i)
```

gets you the `i`-th item on the list of child nodes. It is possible that the notation `r[i]` will also work.

The call to `quicksort` (line `g`) sorts the array `arr` with the quicksort algorithm, one of the most efficient sorting algorithms known. The elements on the sorted `arr` are then appended in sequence as children of `<tbody>` (line `g`).

Inserting existing nodes from the DOM tree into the DOM tree is very different from inserting newly created nodes (Section 10.16). An existing node is first removed from the DOM tree automatically before it is inserted. The removal is necessary to protect the structure integrity of the DOM tree. This is why no explicit removal of child nodes from `<tbody>` is needed before appending the nodes from the sorted array `arr`.

If you accept the `quicksort` function as a magical black box that does the sorting, then we have completed the description DHTML table sorting.

For those interested, the inner workings of `quicksort` are presented next.

## Quicksort

The basic idea of the quicksort algorithm is simple. First pick any element of the array to be sorted as the *partition element* `pe`. By exchanging the elements, the array can be arranged so all elements to the right of `pe` are greater than or equal to `pe`, and all elements to the left of `pe` are less than or equal to `pe`. Now the same method is applied to sort each of the smaller arrays on either side of `pe`. The recursion is terminated when the length of the array becomes less than 2.

```
function quicksort(arr, l, h)
{   if ( l >= h || l < 0 || h < 0 ) return;     // (1)
    if ( h - l == 1 )                           // (2)
    {   if (compare(arr[l], arr[h]) > 0)        // (3)
        {  swap(arr, l, h)   }                  // (4)
        return;
    }
    var k = partition(arr, l, h);               // (5)
    quicksort(arr, l, k-1);                     // (6)
    quicksort(arr, k+1, h);                     // (7)
}
```

The `quicksort` function is called with the array to be sorted, the low index `l` and the high index `h`. It sorts all elements between `l` and `h` inclusive. If the sorting range is empty (line 1), `quicksort` returns immediately. If the range has just two elements (line 2), they are compared (line 3) and switched (line 4) if necessary; and `quicksort` returns. For a wider range, `partition` is called to obtain a partition element and the left and right parts of the array. Each of these two parts is sorted by calling `quicksort` (lines 6-7).

The call `compare(`*a*`,` *b*`)` compares the arguments and returns a positive, zero, or negative number depending for *a* `>` *b*, *a* equals *b*, or *a* `<` *b*. The signs are reversed for sorting in decreasing order.

```
function compare(r1, r2)
{   ke1 =  key(r1, col);                          // (8)
    ke2 =  key(r2, col);
    if ( numerical )                              // (9)
    {   ke1 = Number(ke1);
        ke2 = Number(ke2);
        return direction * (ke1 - ke2);
    }
    return (direction * strCompare(ke1, ke2));    // (10)
}
```

For sorting HTML tables, `compare` is called with DOM nodes `r1` and `r2` representing two different table rows. The function `key` obtains the string content in the designated table cell (line 8) and compares them either numerically as numbers (line 9) or as alphabetically as text string (line 10).

The function `key` extracts the textual content (line 12) of the table cell from the given row `r` at the column position `c`.

```
function key(r, c)
{   var cell = r.firstChild;
    while ( c > 0 )
    {  cell = cell.nextSibling;
       c--;
    }
    return cell.firstChild.nodeValue;             // (12)
}
```

The `strCompare` function compares two text strings `a` and `b` by comparing corresponding characters.

```
function strCompare(a, b)
{   var m = a.length;
    var n = b.length;
    var i = 0;
    if ( m==0 && n==0 ) return 0;
    if ( m==0 ) return -1;
    if ( n==0 ) return 1;
    for ( i=0; i < m && i < n; i++ )
    {  if ( a.charAt(i) < b.charAt(i) ) return -1;
       if ( a.charAt(i) > b.charAt(i) ) return 1;
    }
    return (m - n);
}
```

And swapping two elements on the array is simple.

```
function swap(arr, i, j)
{   var tmp = arr[i];
    arr[i]=arr[j];
    arr[j]=tmp;
}
```

Now we can turn our attention to `partition`, the work horse in the `quicksort` algorithm. The function is called with an array `arr`, and a sorting range, defined by the low index `l` and the high index `h`, which has at least 3 elements. The function picks the middle element as the `pe` (line `13`), and partitions the given range into two parts separated by the `pe`. All elements to the left of `pe` are less then or equal to `pe` and all elements to the right of `pe` are greater than or equal to `pe`. The index of the `pe` is returned (line `18`).

```
function partition(arr, l, h)   // h > l+1
{   var i=l, j=h;
    swap(arr, ((i+j)+(i+j)%2)/2, h);             // (13)
    var pe = arr[h];
    while (i < j)
    {  while (i < j && compare(arr[i], pe) < 1)  // (14)
       {  i++; }   // from left side
```

```
        while (i < j && compare(arr[j], pe) > -1) // (15)
        {  j--; }   // from right side
        if (i < j) {  swap(arr, i++, j); }        // (16)
    }
    if (i != h) swap(arr, i, h);                  // (17)
    return i;                                     // (18)
}
```

Searching from the left (line 14) and right (line 15) end of the range, it looks for a pair of out-of-order pair of elements and swaps them (line 16). When done, it moves the `pe` back into position (line 17) and returns.

The complete `quicksort` and the table sorting example can be found in the example package.

## 10.18 A TicTacToe Game

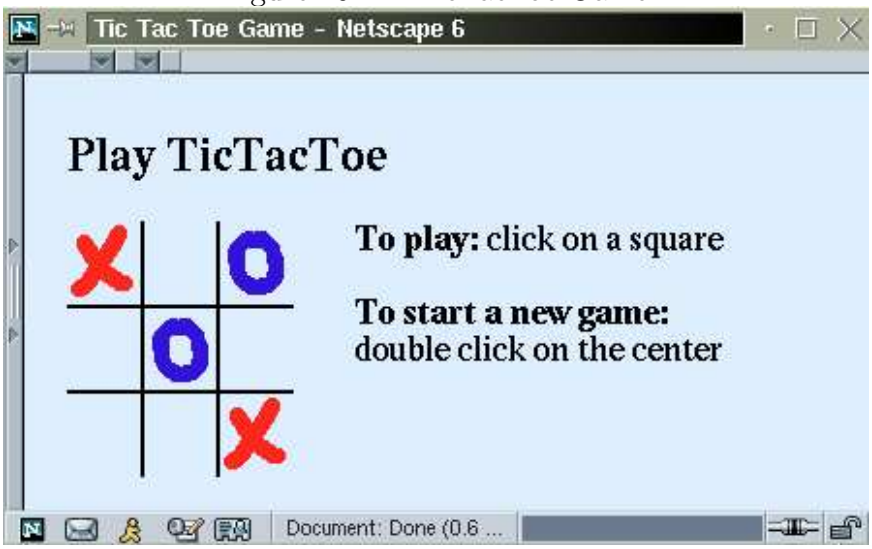
With DHTML many kinds of interactive games can be implemented. Let's look at TicTacToe as an example (Ex: **TicTacToe**). A CSS controlled `<table>` can server as the playing board. Moves are made by clicking on the game board squares. Two files `x.gif` and `o.gif` provide the graphical images for the game tokens (Figure 10.22).

The HTML code for the game board is a 3 by 3 `<table>`:

```
<table class="tic" cellspacing="0" border="0">
<tr><td id="tl" onclick="play('tl')"           (a)
        width="37" height="44"> </td>
    <td id="tc" onclick="play('tc')"           (b)
        width="37" height="44"> </td>
    <td id="tr" onclick="play('tr')"           (c)
        width="37" height="44"> </td>
</tr>

   ...

</table>
```

The first row is shown here. The other rows are entirely similar. Each table cell (`<td>`)

Figure 10.22: TicTacToe Game




has an `id` and a `class` attribute set to a string such as `tl` (top-left line `a`) and `tr` (top-right line `c`). We thus simply identify the nine game squares. Each `<td>` has a `width` and `height` setting to accommodate the game token image that, when played, will replace the non-breaking space ` ` place holder as the content of `<td>`. Onclick, each `<td>` calls `play` with its own `id`.

The game board is drawn with CSS `border` settings. For example, the style for the top left square is

```
td#tl     // id selector
{  border-right: thin #000 solid;
   border-bottom: thin #000 solid;
}
```

And the style for the center square is

```
td#cc
{  border-top: thin #000 solid;
   border-left: thin #000 solid;
   border-right: thin #000 solid;
   border-bottom: thin #000 solid;
}
```

These table cell styles when combined with the `cellspacing` and `border` attributes for `<table>`, draws the game board.

In the Javascript code, nine variables (line 1) are used to indicate whether a game square is open (zero) or taken (nonzero). The function `play` plays a token on the given (`id`) position. It returns immediately if the square is not open (line 2). Otherwise, it proceeds to place a token in the target square. It sets `cell` to the target `<td>` element (line 3), marks the position as taken by player one or player two (line 4), obtains an element object representing the game token (line 5), saves a copy of the content of `<td>` in the global variable `sp` (line 6), and replaces the content of `cell` with the token (lines 7-8).

```
var tl=0, tc=0, tr=0, cl=0, cc=0,
    cr=0, bl=0, bc=0, br=0;                      // (1)
var which=false, sp=null;

function play(id)
{   if ( eval(id) > 0 ) return;                  // (2)
    var cell = document.getElementById(id);      // (3)
    eval(id + (which ? "= 1;" : "=2;"));         // (4)
    tnode = token();                             // (5)
    if ( sp == null )
        sp = cell.firstChild.cloneNode(true);    // (6)
    cell.removeChild(cell.firstChild);           // (7)
    cell.appendChild(tnode);                     // (8)
}
```

The saved `sp` is used when restoring the game board for another game. Note, the call `cloneNode(true)` performs a *deep copy* of a node, copying all nodes in the subtree rooted at the node. If the argument `false` is given, only the children of `node` will be copied.

The `token` function creates a new `<img>` element for `x.gif` or `o.gif` depending on the setting of the Boolean variable `which` (line 9) whose value alternates every time `token` is called (line 10).

```
function token()
{   var t = document.createElement("img");
    if ( which )                                 // (9)
       t.setAttribute("src", "o.gif");
```

```
    else
       t.setAttribute("src", "x.gif");
    which = ! which;                          // (10)
    t.setAttribute("width", "35");
    t.setAttribute("height", "40");
    t.style.display = "block";                // (11)
    return t;
}
```

The token image displays as a block element (line 11) so it fits exactly on the board. The center square (id=cc), calls newgame() on double click which restores the game board (line 12) and resets the game variables (line 13) for another game.

```
function newgame()
{   blank(tl, "tl"); blank(tc, "tc");         // (12)
    blank(tr, "tr"); blank(cl, "cl");
    blank(cc, "cc"); blank(cr, "cr");
    blank(bl, "bl"); blank(bc, "bc");
    blank(br, "br");
    tl=tc=tr=cl=cc=cr=bl=bc=br=0;             // (13)
}

function blank(n, id)
{   if ( n == 0 ) return;   // no token
    var cell = document.getElementById(id);
    cell.removeChild(cell.firstChild);
    if ( sp != null )
       cell.appendChild(sp.cloneNode(true));
}
```

The function blank replaces a board position with a token by a copy of the saved blank node sp. The function blank can be simplified to

```
function blank(n, id)
{   if ( n == 0 ) return;   // no token
    var cell = document.getElementById(id);
    cell.innerHTML = " ";
}
```

for browsers that support the `innerHTML` feature. Under NN and IE all HTML elements have the `innerHTML` field that gives you the HTML code contained inside the element. You can also set this field to modify the content of any element. It is very convenient for programming.

This simple implementation is functional enough to be used by two players. Chapter-end exercises suggest improvements to this program. isectionWindows and Frames

Before the introduction of DOM, browsers such as NN and IE already had their own objects for windows, frames, and various HTML elements. DOM makes the document model more complete, systematic, and uniform across all browsers. The DOM addresses only the object structure of the document. The window object that represents the on-screen window containing the document is not part of the DOM specification. Many important features of the `window` object do work consistently across major browsers (Section 9.12). In particular `window.document` gives you the root of the DOM tree.

With the introduction of DOM, browser vendors are implementing the window object in the same DOM spirit leading to a more functional, and better defined interface to the `window` object. Since version 6, NN is leading the way in this regard. Materials in this section show how the `window` object works with frames and the DOM tree in NN 6 and later versions.

## The `window` Object

Useful fields of `window` include

- `window.document`—a reference to the root of the DOM tree conforming to the DOM `Document` interface.
- `window.frames`—an array of frames in this window.
- `window.innerHeight`, `window.innerWidth`—the height and width in pixels for content display in the window.
- `window.navigator`—a reference to the `navigator` object (Section 9.11).
- `window.parent`—the parent window containing this window or `null`.

- `window.top`—the top-level window.
- `window.screen`—an object representing the computer display screen.

Many `window` methods have been described in Section 9.12. We list a few more here.

- `window.dump(`*str*`)`—outputs *str* to the Javascript console.
- `window.print()`—prints `window.document`

## Vertical Page Positioning

Sometimes, a page layout calls for the positioning of an HTML element at a certain vertical position in the display window. For example a site entry may be a graphics image or a Flash animation that is vertically centered in the window or starts 1/3 of the way down. Vertical centering is not easy with HTML alone. But can be done rather simply with DHTML.

Here is a simple example (Ex: **VCenter**).

```
<body onload="vcenter()" style="margin: 0">
<table width="100%" cellpadding="0" cellspacing="0">           (1)
<tr><td><img id="padding" src="img/clear.gif"
         alt="padding" width="100" height="150" /></td></tr>   (2)
<tr align="left">
<td style="width:100%; height:300px"><a href="main.html">
   <img src="img/entry.gif" width="1200"
        height="300" style="border-style:none" /></a></td>
</tr></table></body>
```

The entry graphic `entry.gif` is 300 pixels high. It needs to be centered vertically in the window. We put the graphic in the second row of a table that covers the whole window (line 1). The first row is a padding provided by a transparent image (line 2) whose height is set dynamically by the Javascript function `vcenter()`:

```
function vcenter()
{   var ht = window.innerHeight-300;                          // (3)
    ht = (ht - ht%2)/2;                                       // (4)
    var cell = document.getElementById("padding");            // (5)
```

```
    cell.setAttribute("height", ht);                          // (6)
}

window.onresize=vcenter;                                      // (7)
```

To center vertically, `vcenter` computes the difference of the window inner height and the height of the image to be centered (line 3). Dividing that by 2 gives the height of the desired padding which is set as the height attribute of the padding image (lines 4-6). The function is called on load and also on window resizing (line 3) so the element stays centered. This idea is easily generalized to perform other dynamically computed vertical positioning.

The WDP website provides the complete working version of this example.

## 10.19 A Code Experimenter

For people learning or using HTML, style sheets, and Javascript, it is often beneficial to experiment with code fragments to see their effects. We can build a page with two frames, side-by-side (Ex: **TryCode**). In one frame, the user can enter and edit code in a textarea. With DHTML, the other frame can show the effect of the code (Figure 10.23).

The `frameset` page has the following code.

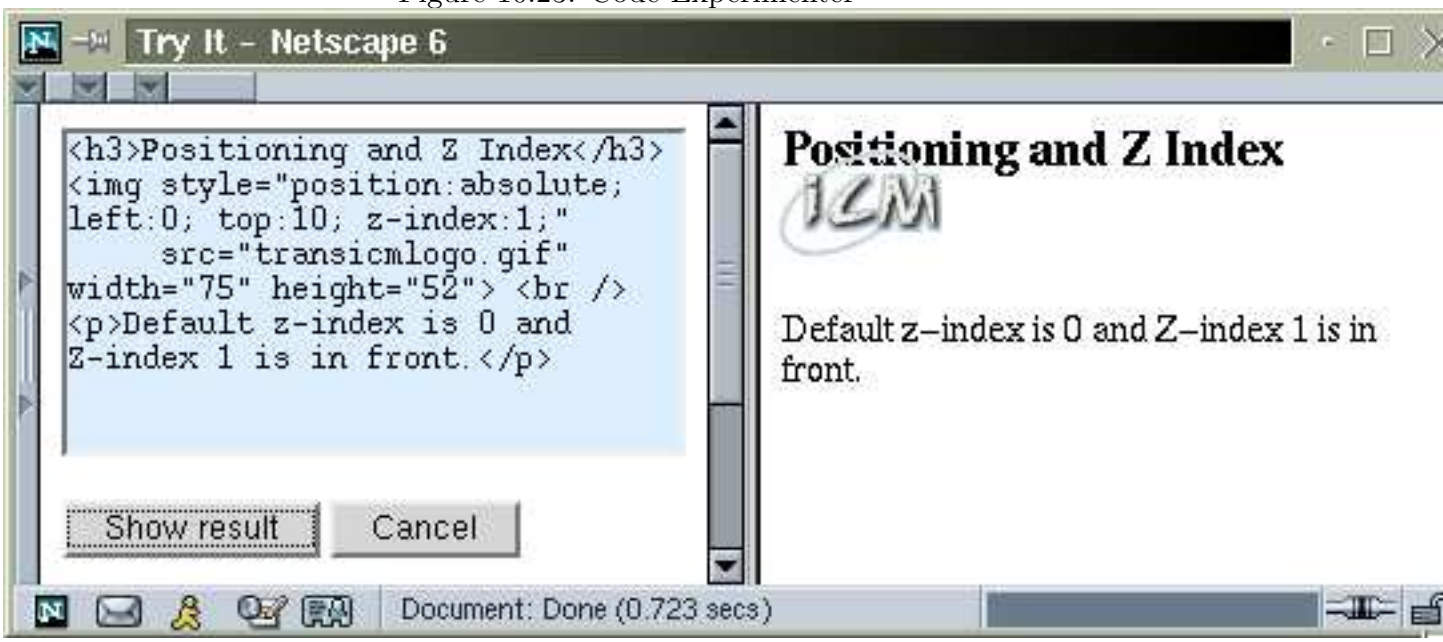
```
<head>
<title>Code Experimenter</title></head>
<frameset cols="40%,60%">
   <frame frameborder="1" id="codeframe" src="code.html"
          scrolling="auto" />
   <frame id="resultframe" src="result.html" />
</frameset>
</html>
```

The `result.html` page is very simple.

```
<head><title>Results</title></head>
<body><p>The result is shown here</p></body>
</html>
```

And the `code.html` page contains the `textarea` for entering and editing experimental code.

Figure 10.23: Code Experimenter




```
<textarea style="width:100%; background-color: #def"
          id="code"  cols="65"  rows="16">
Put your HTML code fragment here.                          (1)
Anything that the HTML body
element may contain is fine.                               (2)
</textarea>
<p><input name="result" type="button"
          value="Show result" onclick="show()" />
   <input type="button" value=" Cancel "
          onclick="goBack()" /></p>
<p>You can enter and edit code in this window,
and click on "Show result" to see the result in
the window on the right.  Click "Cancel" to go
back to the hands-on page.</p>
</body></html>
```

The HTML source contained in `textarea` (lines 1-2) can be anything here. The user will enter the code interactively for experimentation. The Javascript code in `code.html` supports the desired effects. The `Show result` button calls `show()`. It obtains the `document` object in the window for the frameset (line `A`). Note `window` is the window for the `codeframe` and from this frame `window.top` (or `window.parent` is the window for the frameset. From

the `resultframe` element (line `B`), we obtain its `document` object (line `C`) and the `<body>` element (line `D`) in that document to show the code. We applied the `HTMLDocument` method `getElementsByTagName` (Figure 10.14) to get the `body` element.

The HTML source in the `textarea` is assigned as the content `innerHTML` of the `<body>` element (lines `E-F`).

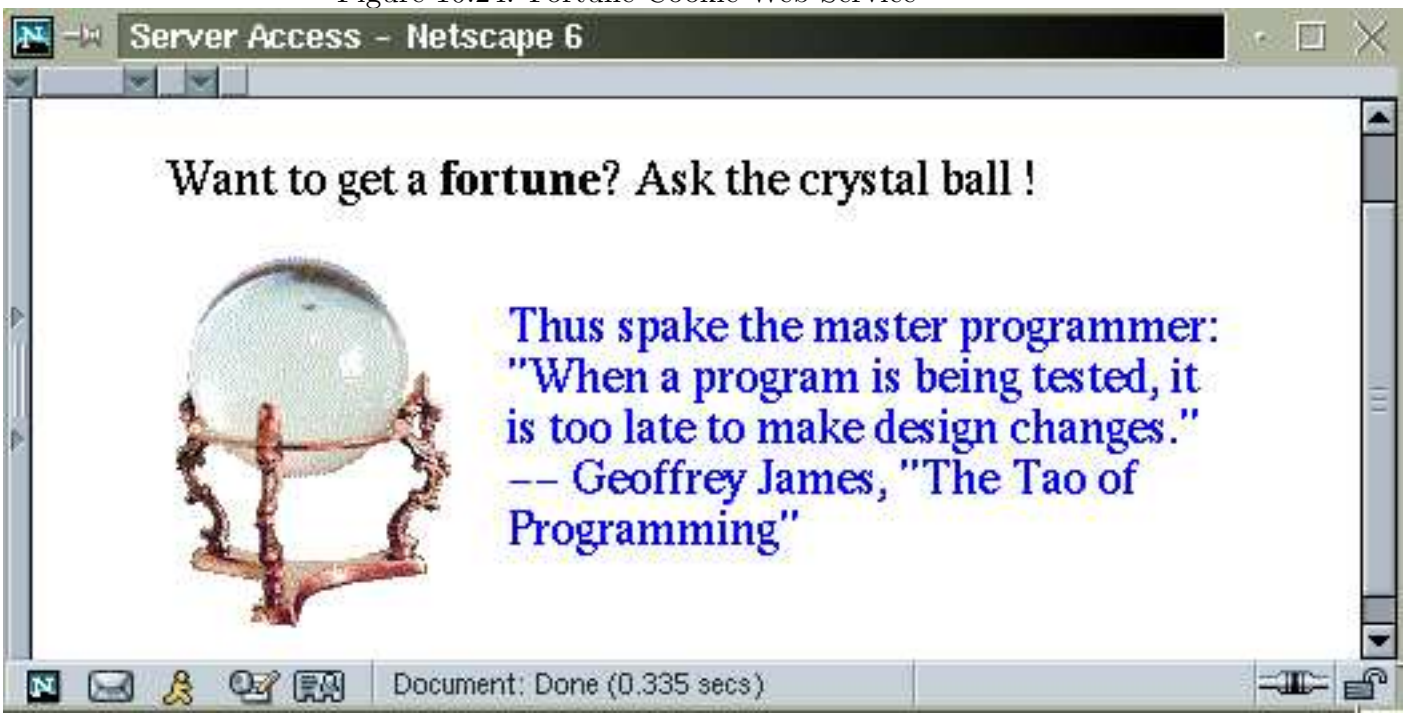
```
<head>
<script type="text/javascript">
function show()
{   var b = window.top.document;                        // (A)
    var f = b.getElementById("resultframe");            // (B)
    var d = f.contentDocument;                          // (C)
    var bb = d.getElementsByTagName("body").item(0); // (D)
    var c= document.getElementById("code");             // (E)
    bb.innerHTML=c.value;                               // (F)
}

function goBack() { window.top.back(); }
</script>
```

This example also shows the usage of frame objects and how they can interact under DOM.

## 10.20 DHTML Access to Web Services

DHTML can do many things to make Web pages more responsive and more functional. But it is, up to this point, restricted to operating on data already in the page and input by the user from the keyboard. DHTML can be much more powerful and useful if it can manipulate data from other Web pages and perhaps even obtain data from server-side programs. The data thus obtained can be incorporated at appropriate places in a Web page. For example, a stock price can be obtained and inserted in an article on stock trading; a travel page can display weather forecast information of the departure and arrival cities for the travel dates; maps and driving directions can be included for stores and businesses, etc.

Figure 10.24: Fortune Cookie Web Service




Thus, *Web services* provide well-defined data and DHTML can fetch such data dynamically and insert them at designated places on the DOM tree. A Web service does not have to return a complete HTML page. It may return just an HTML fragment suitable for inclusion in another Web page. In time such usage will be common-place and standardized.

To show how this can work with current DHTML techniques, let's look at a *fortune cookie* program (Figure 10.24). This program Ex: **Fortune** works with two frames as follows.

- When the end user clicks on the picture of a crystal ball, a request is made to a Web service that returns a `fortune cookie` message.
- The fortune cookie page is loaded in a hidden frame and contains Javascript code which will deposit the fortune cookie HTML fragment into the viewing frame.

The frameset code hides the first frame (line 1) and shows only the second frame (line 2).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
```

```
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head><title>Web Service Access</title></head>
<frameset rows="0,*">
   <frame id="hide" name="hide" src="empty.html" />        (1)
   <frame id="show" name="show" src="fortune.html" />      (2)
</frameset>
</html>
```

The `empty.html` file is very simple and can be anything similar to

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head><title>hidden access</title></head>
<body></body></html>
```

The `fortune.html` page displays the crystal ball and ties it to the access of the fortune cookie service (line 3).

```
<head>
<title>Web Service Demo: Fortune Cookie</title>
<script type="text/javascript" src="service.js"></script>
</head>
<body style="font-size: Large; margin: 50px 10%">
<p >Want to get a <b>fortune</b>?
    Ask the crystal ball !</p>
<p>
<img onclick="comp('/cgi-bin/fortune.pl')"          // (3)
     style="float: left; margin-right: 1em;
            margin-bottom: 1em;"
     src="img/crystalball_stand.gif"
     width="110" height="142"
     alt="Crystal ball on a stand" /></p>
<p id="ans" class="ans">
000&lt;*@!+_($&amp;0000000&lt;*@!+_($    // random letters
&amp;0000000000&lt;*@!+_($&amp;</p>
</body>
```

The `comp` function is in `service.js`. It obtains the frameset window (line 4), then the `window` object for the hidden frame (line 5) and sets its location to the target URL `t`.

```
function comp(t)
{   tw = window.top;                    // (4)
    afw = tw.frames["hide"];            // (5)
    afw.location=t;
}
```

The loading of the result page from the `fortune.pl` service is completely hidden. And the result page contains Javascript code to modify the DOM tree of the `show` frame.

The Perl code for `fortune.pl` is as follows.

```
#!/usr/bin/perl

my $xhtml_front =
'<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">';

$ft = `/usr/games/fortune`;                   ## (6)
print <<END;
Content-type: text/html

$xhtml_front
<head><title>Fortune</title>
<script type="text/javascript">
function act()
{   nd = document.getElementById("fortune");  // (7)
    code = nd.innerHTML;                      // (8)
    td = window.top.document;                 // (9)
    f = td.getElementById("show");
    afd = f.contentDocument;
    bb = afd.getElementById("ans");
    bb.innerHTML=code;                        // (10)
}
</script></head>
<body onload="act()" id="fortune">
<p style="color: blue"> $ft </p>
```

```
</body></html>
END
exit;
```

The Perl program calls the UNIX program **fortune** (line 6) which generates a randomly selected "fortune cookie" message each time it is called. The "fortune" is sent in the `<body>` of the returned page. The Javascript function `act` is called when the returned page is loaded to obtain the HTML code in the `<body>` (lines 7-8) and place the HTML code in the `ans` element of the `show` frame (lines 9-10).

Due to security concerns, the Web service must be provided by the same Web server that supplied the `show` frame page. Only the Javascript code in such pages are allowed to access and modify the `show` frame page.

The list of files involved in this example are:

- `webservice.html`—the frameset
- `fortune.html`—the `show` frame page
- `empty.html`—the `hide` frame page
- `service.js`—Javascript included in `fortune.html`
- `fortune.pl`—Perl CGI script delivering the fortune cookie and Javascript code

The technique considered here can be applied together with the code experimenter (Section 10.19) to load different pages for experimentation.

## 10.21 For More Information

This chapter gets you started in standard-based DHTML. As DOM, Javascript, and CSS standards grow and evolve, and as browser compliance becomes more complete and wide spread, DHTML will be an increasingly more powerful and effective tool for delivering dynamic Web pages.

For more information on the W3C standards see `www.w3c.org`. For DOM bindings to the ECMA Script see

```
www.w3.org/TR/REC-DOM-Level-1/ecma-script-language-binding.html
```

For the Netscape implementation of DOM see the Gecko DOM reference

```
www.mozilla.org/docs/dom/domref/dom_shortTOC.html
```

For standard-based DHTML sample codes see, for example,

```
www.mozilla.org/docs/dom/samples/

dmoz.org/Computers/Programming/Languages/
         JavaScript/W3C_DOM/Sample_Code/

webfx.eae.net
```

## 10.22 Summary

DHTML is a technique that combines Javascript, CSS, and HTML. DOM is a W3C recommended standard API for accessing and modifying HTML and XML documents. DHTML with DOM results in powerful and cross-platform code.

Browsers support the DOM specified API and provides the required objects and methods for Javascript programming. The `window.document` object implements the HTMLDocument interface and gives you the root node of a Web page. Each element on this DOM tree corresponds to an HTML element in the source document and implements the `HTMLElement` interface, derived from `Element` which extends `Node`. The `document.getElementById` method is handy for obtaining the DOM object representing any HTML element with an assigned `id`. Starting from any node *el* on the DOM tree you can follow the child (*el*.`childNodes`, *el*.`firstChild`), parent (*el*.`parentNode`) , and sibling (*el*.`nextSibling`, *el*.`previousSibling`)

relations to traverse the entire tree or any parts of it. You can also access and modify element attributes (*el*.`getAttribute(`*attr*`)`, *el*.`setAttribute(`*attr*`,` *value*`)`) and styles (*el*.`style.`*property*).

The DOM API allows you to systematically access, modify, delete and augment the DOM tree resulting in altered page display: *e*.`removeChild(`*node*`)`, *e*.`appendChild(`*node*`)`, *e*.`replaceChild(`*new*`,` *old*`)`, *e*.`insertBefore(`*new*`,` *node*`)`. New tree nodes can be created in Javascript with `document.createElement(`*tagName*`)`, `document.createTextNode(`*string*`)`.

When you combine event handling, including those generated by `window.setTimeout()`, and style manipulations, many interesting and effective dynamic effects can be achieved for your Web pages.

Accessing Web services with DHTML promises to allow developers to add value to Web pages by including information dynamically produced by various services on the Web.

# Exercises

## Review Questions

1. What is DHTML? Explain in your own words.

2. What are three important enabling technologies for standard-based DHTML?

3. What is DOM? the DOM tree? the most basic `Node` object?

4. Name important types of nodes on the DOM tree and describe the DOM tree for a Web page in detail.

5. Write down the Javascript code for obtaining the DOM node for an HTML element with a given *id* and for determining its node type.

6. What is the `nodeName` and `nodeValue` for an `HTMLElement`?

7. Describe the `HTMLElement` interface.

8. Explain the fields and methods in the `HTMLDocument` interface. Which object available by Javascript supports this interface?

9. How does Javascript modify the presentation style of an element on the DOM tree?

10. Compare the `window` object described in Section 10.18 and Section 9.12.

11. Explain the concept of Web service and the access of Web services with DHTML.

## Assignments

1. Improve the Ex: **DomHello** in Section 10.2 and make the mouseover action also change the text "`over the phrase`" to "`out of the phrase`". Test and see how well it works.

2. Modify the Ex: **DomCalc** in Section 10.6 and present a display in the form *string* `=` *result*.

3. Test the `normalize` method of `Node` (Section 10.7) to see how well it is supported by your browser.

4. Consider Ex: **DomDft** in Section 10.8). The traversal does not take comments into account. Modify the `dft` function to remedy this and test the traversal on pages that contain comments. (Hint: `Node.COMMENT_NODE` is `8`.)

5. Consider the visual navigation of DOM tree (Section 10.9). Take (Ex: **DomNav**) and add a *tree display* of the table. As the user navigates the table, show the current position also on the DOM tree display.

6. Consider the guided form example in Section 10.11. Add the correctness check of email from Section 9.15) to it.

7. Follow the in-place fade-in example in Section 10.12 and write the code to achieve fade-out.

8. Take the "disappearing country code" problem described at the end of Section 10.16 and fashion a solution.

9. Add a *unmove* button to the TicTacToe program in Section 10.18 to take away the last move made.

10. Improve the TicTacToe program in Section 10.18 by adding the ability to play with the computer. (Hint: add move generation.)

11. Construct a pocket calculator using DHTML. Layout the LCD window and calculator buttons with a `table` and simulate the functions of the common calculator with `onclick` events on the buttons.