DOM Primer Part 1

Contents

- 1. Introduction
  - 1.1 Motivation
  - 1.2 Basic ideas
  - 1.3 DOM Levels
- 2. Core and HTML
  - 2.1 Introduction
  - 2.2 Traversal methods
  - 2.3 Changing the structure of a document
- 3. HTML DOM, Attributes and Content
  - 3.1 Introduction
  - 3.2 Structure manipulation
  - 3.3 Methods for changing attributes
- 4. DOM Level 2
  - 4.1 Introduction
  - 4.2 Core
  - 4.3 Event model
- 5. Window Manipulation

Appendices

- A. The Core DOM
- B. The HTML DOM

1. Introduction

- 1.1 Motivation
- 1.2 Basic ideas
- 1.3 DOM levels

1.1 Motivation

After the Web had gained initial acceptance and popularity through the presentation of static pages of information, content creators and Web developers wanted more - namely dynamic content, content that would change over time and content that would change in response to user interaction with the Web page, for example moving the mouse over particular regions of the page. There are two obvious ways to achieve such enhancements:

- by adding extra tags to the markup language to enable dynamic effects to be described declaratively;
- by providing an interface and a scripting language to enable dynamic effects to be achieved by executing a program in the web client initiated when the Web page is loaded.

Early extensions to Netscape and Internet Explorer exemplified these approaches. Navigator 2 and the JavaScript language allowed Web pages to be modified on the client. Navigator 3 allowed the attributes of the img tag to be changed dynamically in response to mouse movements, for example:

```
<a href="dhtml1.htm"
  onMouseOver="document.images['logo'].src='up.gif'"
  onMouseOut="document.images['logo'].src='right.gif'">
<img name="logo" src="right.gif" height=32 width=32></a>
```

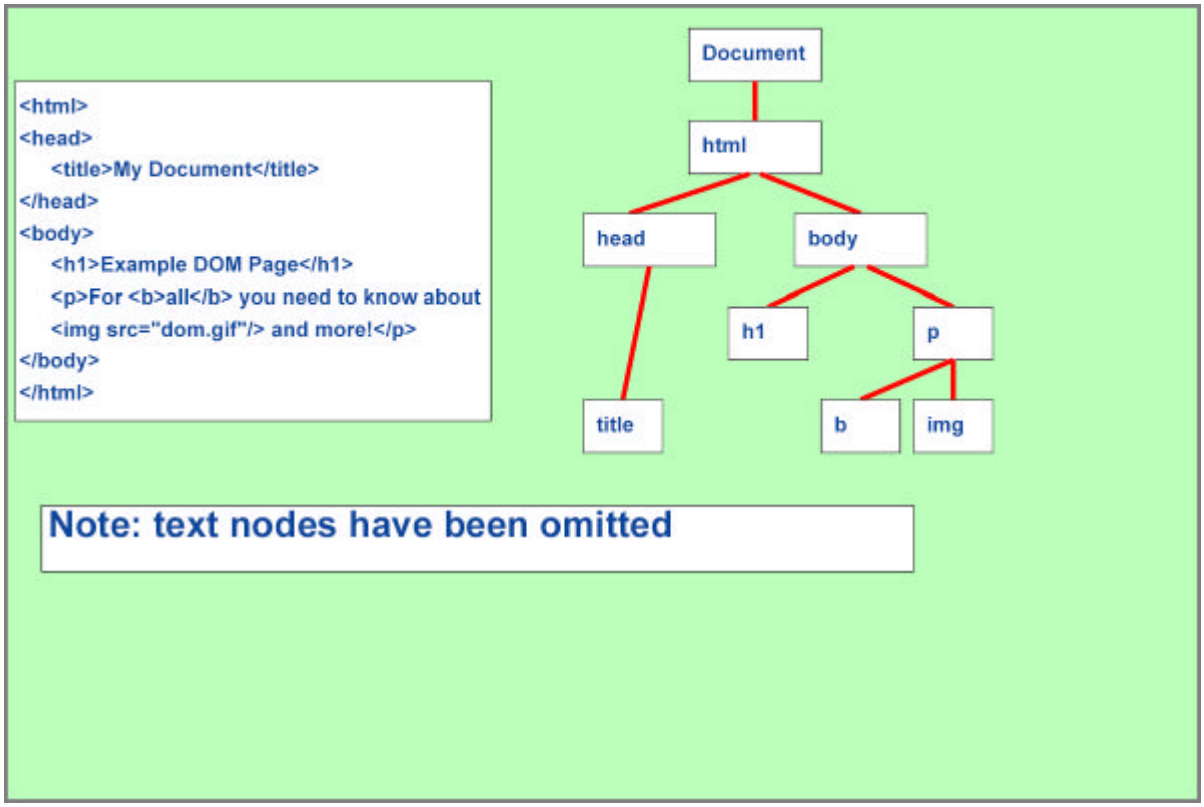
The effect is to change the image displayed to up.gif when the mouse enters the region bounded by the image right.gif and change it back to right.gif when the mouse moves out of the region.

The usual problem in computing soon emerged: different manufacturers produced different extensions and the job of authoring pages that would behave correctly across different browsers became exceedingly difficult, if not impossible. Furthermore, different scripting languages came along, including JavaScript, Python and Perl. XML appeared, and markup languages described within the XML framework such as Scalable Vector Graphics (SVG) and Mathematical Markup Language (MathML) started to be developed and these too required dynamic facilities. Thus there was a need to develop a standard language-neutral (i.e. not tied to any particular scripting language) Application Program Interface (API) through which a scripting language could access all the information in a Web document. The Document Object Model (DOM) is the result.

1.2 Basic Ideas

The key idea in the DOM is to represent a Web document as a **tree structured collection of objects**. If one thinks of the parse tree that would be generated by parsing, say, a HTML document, one can see where this idea stems from. It is important to understand that this representation of a Web document is a **logical** representation. The DOM does not prescribe that Web browser manufacturers **must** implement Web documents as tree structured collections of objects. This is just the way in which the interface is described. How browsers represent documents internally and provide the functionality of the DOM API is for individual manufacturers to decide. Methods are provided in the DOM to enquire about the object tree and to modify it. Such methods are typically invokved from a scripting language, such as JavaScript, in response to user input.

The document tree for a simple HTML document is shown below. The text content of the title, h1 and p nodes appears as text nodes in the tree, but these are not shown in this figure which is intended to capture the overall shape of the tree.



A Web document is therefore represented as a **document tree**. Each node in the tree is an **object** in the object-oriented programming sense. Data are hidden so that they cannot be manipulated directly and functions (methods) are associated with the object to manipulate the data it contains. The DOM thus identifies:

- the interfaces and objects used to represent and manipulate a document;
- the semantics of these interfaces and objects - including both behaviour and attributes;
- the relationships and collaborations among these interfaces and objects.

A DOM object is a collection of named pieces of data. Named values are called **properties**, for example the property nodeName holds the name of a node object. **Methods** are properties which are functions, for example appendChild is a method that can be applied to a node to add a new node to the sequence of child nodes of that node.

The DOM as noted earlier, is language independent. The DOM API can, in principle, be **bound** to any scripting language. In what language, therefore, should the DOM itself be defined? The approach taken is to define the DOM using a language called IDL - Interface Definition Language - which was developed by the Object Management Group (OMG) in connection with the distributed object system, CORBA. Essentially IDL provides a language for defining datatypes and objects in terms of their properties and the signatures (parameter types and result type) of their methods. An extract from the IDL definition of the Node interface (which includes the appendChild method) is given below. Objects in this class have a number of properties (readonly attributes) and methods including appendChild and hasChildNodes. The appendChild method takes a parameter of type Node and returns value of type Node. The method hasChildNodes takes no parameters and returns a value of type boolean.

```
interface Node {
  //NodeType
  const    unsigned short  ELEMENT_NODE = 1;
  ...
  readonly attribute DOMString nodeName;
  ...
  readonly attribute Node     parentNode;
  readonly attribute NodeList childNodes;
  ...
  Node     appendChild(in Node newChild)
               raises(DOMException);
  boolean  hasChildNodes();
};
```

Language bindings to ECMAScript (a standardised JavaScript) and Java are defined in the W3C DOM documents. The ECMAScript binding for part of the IDL interface given above follows.

```
Object Node
The Node object has the following properties:
  nodeName
    This property is of type String.
  parentNode
    This property is of type Node.
  childNodes
    This property is of type NodeList.
  ...
The Node object has the following methods:
    appendChild(newChild)
      This method returns a Node.
      The newChild parameter is of type Node.
    hasChildNodes()
      This method returns a boolean.
  ...
```

To simplify this Primer, the DOM is explained using the ECMAScript binding, rather than the IDL definitions.

1.3 DOM Levels

The World Wide Web Consortium (W3C) are developing the DOM specification. The first specification, known as DOM Level 1 was published on 1 October 1998, and a second edition (with editorial, but not technical changes) is (still) under development. The second level of the specification was published in November 2000 and work on a third level is in hand! The specifications are freely available as HTML, pdf and PostScript documents at the W3C Web site http://www.w3.org/DOM/. Robin Covers' XML Pages http://www.oasis-open.org/cover/dom.html contain good background and summary information on the DOM.

DOM Level 1 is in two parts: Core and HTML. The Core provides a set of low-level fundamental interfaces that can represent any structured document and extended interfaces for representing an XML document. The HTML section provides additional, higher level, interfaces that are used with the Core interfaces to provide a more convenient view of a HTML document.

DOM Level 2 is an extension of DOM Level 1. The Core interfaces of Level 1 have been extended to include support for XML namespaces. (Namespaces are a mechanism for identifying and distinguishing elements belonging to different types of markup in a document, for example SMIL and SVG.) Other parts of DOM Level 2 deal with Events, Style, Traversal and Range, and Views. A sixth part, HTML, is under development, but nearing completion.

DOM Level 3 is an extension of Level 2 and includes access to entities, DTDs and Schemas and keyboard events. Four parts: Core, Load and Save, Validation, Events, and XPath are at Working Draft stage. Events and XPath are the most advanced.

DOM specifications to provide specific functionality for Scalable Vector Graphics (SVG), Mathematical Markup Language (MathML) and Synchronized Multimedia Language (SMIL) are also available as W3C Recommendations and are published in these standards.

In this Primer we focus on the Core and HTML interfaces.

The Appendices to this document give a fairly complete description of the ECMAScript bindings of the main interfaces defined in the Core (levels 1 and 2) and HTML (level 1) DOM. It is important to realize that not all browsers yet implement all the interfaces as specified and some implement proprietary extensions/variations.

2. General Methods

- 2.1 Introduction
- 2.2 Traversal methods
- 2.3 Changing the structure of a document

2.1 Introduction

The functionality of the DOM falls into three major groups:

1. methods for traversing the document tree;
2. methods for changing the structure of the document tree;
3. methods for retrieving and changing the values associated with the nodes of the tree.

In addition, the HTML DOM provides methods specific to particular types of node, for example to get the attribute values of an HTML element node.

There are variations between browsers and versions of browsers. The examples in this Primer are written with Internet Explorer 6.0 in mind, though many also work with Mozilla. Opera currently has rather less DOM functionality than IE and Mozilla.

2.2 Traversal methods

Traversal methods enable a script to locate particular nodes within a document tree. The root of the document tree is accessed by the property:

```
document
```

The most important traversal methods are listed below. The notation name:Type is used to indicate that the parameter entity name has data type Type.

nodeName
Returns name of node
nodeType
Returns type of node. (Values of node types are listed in Appendix A)
nodeValue
Returns value of node
parentNode
Returns reference to parent node
firstChild
Returns reference to first child in list of child nodes
lastChild
Returns reference to last child in list of child nodes
previousSibling
Returns reference to node preceding this node. If none, returns null
nextSibling
Returns reference to node after this node. If none, returns null
hasChildNodes()
Returns true if node has child nodes
childNodes
Returns list of child nodes
getElementById(id:String)
Returns reference to object with id "id"
getElementsByTagName(tag:String)
Returns array of references to objects with tag name "tag"

The following methods are available for manipulating lists of object references.

item(num)
[num]
Return reference to item num in list of child nodes (starting from 0)
length
Returns length of a list

Consider the following Web document:

```
<html lang="en-uk">
<head>
<title>...</title>
<link ...>
</head>
<body>
<table class="slide" width="100%">
...
</table>

<div class="slidebody">

 <script language="JavaScript">
 ...
 </script>
 <h2 id="myh2">My test line</h2>
 ...
```

The corresponding DOM tree has the shape:

```
document
html [0]
 head [0]
  title [0]
  link [1]
 body [1]
  table [0]
        ...
  div [1]
   script [0]
   h2 [1]
```

The numbers in brackets after each element indicate the position of the element in its parent's list of children. The object corresponding to the h2 element can be located using:

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

Addressing the h2 tag works as follows:

- document identifies the root of the tree.
- childNodes[0] identifies the node corresponding to the html element.
- childNodes[1] identifies the node corresponding to the body element.
- childNodes[1] identifies the node corresponding to the div element.
- childNodes[1] identifies the node corresponding to the h2 element.

Alternatively the node corresponding to the h2 element can be located using:

```
document.getElementsByTagName("h2")[0]
```

or

```
document.getElementById("myh2")
```

Some points to note about these different approaches:

1. The first approach requires the precise structure of the DOM tree to be known. It is not unknown in the current state of DOM implementations for different browsers to produce different DOM representations of the same Web document. Any change to the document is likely to require changes to the script.
2. In the second approach, if the document contains more than one node of a particular type, the script writer needs to know which one is required. Changing the document dynamically may also change which is required.
3. In the third approach, element identifiers are unique within a document and hence this approach gives an easy way to locate a particular element, but clearly requires collaboration between script writer and document author to assign identifiers to appropriate elements.

2.3 Changing the structure of a document

The structure of a document can be changed by inserting new elements into the document, and replacing existing elements. The following Core DOM methods are available for this purpose.

createElement
Creates an element of the defined type
createTextNode
Creates a text node (the content of, say, an h2 element)
appendChild
Adds a new child at the end of the list of children
insertBefore
Inserts child node before the one specified
replaceChild
Replaces a child node by the one specified

Consider a HTML document containing:

```
<ul id="ulorig">
   <li>First item</li>
   <li id="listrepl">The one before the one to be replaced</li>
   <li>Third item</li>
   <li id="listorig">The one after the insertion</li>
   <li>Fourth item</li>
   <li>Fifth element</li>
 </ul>
```

This corresponds to:

- First item
- The one before the one to be replaced
- Third item
- The one after the insertion
- Fourth item
- Fifth element

A new item can be inserted before the list element with identifier 'listorig' ("The one after the insertion") using the JavaScript code:

```
var newitem=document.createElement("li");
var newtext=document.createTextNode("This is a new item");
newitem.appendChild(newtext);
document.getElementById("ulorig").insertBefore(
  newitem,document.getElementById("listorig"));
```

The result is:

- First item
- The one before the one to be replaced
- Third item
- This is a new item
- The one after the insertion
- Fourth item
- Fifth element

Some points to note:

1. The first two lines of code create new object nodes in the document tree, representing a list item node and a text node respectively. The createElement and createTextNode methods do not link the nodes created into the tree. They create in effect two separate one node trees.
2. The variables newitem and newtext contain object references to the nodes created.
3. The third line of code links the "li" node and the text node, so that the text node becomes a child of the "li" node.
4. The fourth line of code links the "li" node into the document tree, as a child of the ul element with identifier "ulorig". The node is inserted before the specified child (the child with identifier "listorig").
5. Note that the getElementById method takes an argument which is a string, whereas the two arguments of insertBefore are references to nodes.

A new child can be appended at the end of the list using the code:

```
var newitem1=document.createElement("li");
var newtext1=document.createTextNode("This is a new item");
newitem1.appendChild(newtext1);
document.getElementById("ulorig").appendChild(newitem1);
```

A new subtree (li node and text content node) is created as in the previous example, and appended to the list of children of the "ul" tag by applying the appendChild method to the node representing the "ul" element. When this code is invoked after the previous code, the result is:

- First item
- The one before the one to be replaced
- Third item
- This is a new list item
- The one after the insertion
- Fourth item
- Fifth element
- This is a new item

The third element in the list ("The one to be replaced") can be replaced using the code:

```
var newitem2=document.createElement("li");
var newtext2=document.createTextNode("This is a replacement item");
newitem2.appendChild(newtext2);
document.getElementById("ulorig").replaceChild(newitem2,
        document.getElementById("repl"));
```

When this code is invoked after the previous code the result is:

- First item
- The one before the one to be replaced
- This is a replacement item
- This is a new item
- The one after the insertion
- Fourth item
- Fifth element
- This is the last item

Just to illustrate the use of other traversal methods, the element to be replaced could also have been specified as:

```
document.getElementById("listrepl").nextSibling
```

The method splitText can be applied to a textNode to split the node into two. The node is split after a specified character offset (which starts from zero). The example below splits the string "Add emphasis here" at the start of the word "here", then inserts an em node with the word "here" as its text content.

The document contains initially:

```
<p id="p1">Add emphasis here</p>
```

The code to modify the document tree is:

```
var p1node = document.getElementById("p1");
var textnode = p1node.childNodes.item(0);
textnode.splitText(13);
var textnode1 = p1node.childNodes.item(1);
var bnode = document.createElement("em");
bnode.appendChild(textnode1.cloneNode(true));
p1node.replaceChild(bnode, textnode1);
```

The method cloneNode(deep:boolean) returns a duplicate of the specified node. The method serves as a generic copy constructor for nodes of any type. The node returned has no parent, i.e. it is not linked into the DOM tree. When a node is cloned, all the attributes and values of the node are copied. The parameter deep determines whether a deep (true) or shallow (false) copy is made. A deep copy recursively clones the subtree under the specified node. A shallow copy just clones the node itself. Note that according to the DOM Recommendation, if the node is a text node, the text will only be copied if a deep clone is made, since the text is contained in a child text node.

The difference between deep and shallow copy can be seen in the next example. Consider the function clone defined as follows.

```
function clone(elemid, deep) {
  var para1ref = document.getElementById("p1");
  var divref = document.getElementById(elemid);
  divref.appendChild(para1ref.cloneNode(deep));
}
```

If this function is applied to the following HTML fragment:

```
<div id="d1">
  <p id="p1">Some text emphasised <em>here</em></p>
</div>
<div id="d2"></div>
<div id="d3"></div>
```

invoked as clone('d2', true) the result is to clone a copy of the p element and its subtree and append the copy as a child of the div element with id attribte d2. The presentation effect is:

Some text emphasised **here**

Some text emphasised **here**

If the function is then invoked as clone('d3', false), only the p node is copied, there is no content and hence the visual appearance of the document does not change.

The DOM trees corresponding to these cases are shown in the table below.

Initial tree	Deep clone clone('d2', true)	Deep followed by Shallow clone clone('d2', true); clone('d3', false)
DIV P #text EM #text DIV DIV	DIV P #text EM #text DIV P #text EM #text DIV	DIV P #text EM #text DIV P #text EM #text DIV P

3. HTML DOM, Attributes and Content

- 3.1 Introduction
- 3.2 Structure manipulation
- 3.3 Methods for changing attributes

3.1 Introduction

Most of the objects, properties and methods described so far are in the Core DOM. The HTML DOM extends the Core DOM with properties and methods for manipulating HTML content. Some of these manipulations can be achieved using Core DOM methods, but the HTML DOM methods may be more convenient.

3.2 Structure manipulation

The root of a HTML document is a **HTMLDocument** object. Properties of this object give links into elements inthe document and the object provides methods which allow the structure of the document to be extended or replaced using strings of unparsed HTML. The HTML DOM defines objects corresponding to the HTML element types. HTML attributes are then exposed as properties on the element objects.

The main properties of **HTMLDocument** object are shown below.

title
Document title as specified in TITLE element
URL
Complete URI of the document
body
Element that contains the content (BODY or outermost FRAMESET element)
images
List of all IMG elements in document
links
List of all AREA and anchor (A) elements with a value for **href** attribute
forms
List of all the forms of a document
anchors
List of all anchor (A) elements with a value for the **name** attribute
cookies
Cookies associated with a document

Thus the title of a document can be retrieved using:

```
document.title
```

The width and height of the first image in a document can be retrieved using:

```
document.images.item(0).width
document.images.item(0).height
```

The content of a document can be completely replaced using methods on the HTMLDocument object. The text parameter is an unparsed string of HTML text. The following methods to support this are available on HTMLDocument.

open
Opens a document stream for writing
write(text:String)
Add a text string (which is parsed according to the document's structure model) to the document stream
writeln(text:String)
As above, but adds newline character to end of string
close
Close document stream

The example below illustrates their usage.

```
document.write("<html><head><title>New Page</title> </head><body><h1>New
Heading</h1></body></html>");
```

will replace the current document with the new document specified in the parameter to the write method. The writing stream is closed using the close method.

```
document.close
```

3.3 Methods for changing attributes

The HTML DOM provides methods for changing the attributes of an element and the content of an element. As noted earlier, each HTML element has a corresponding object type. HTML attributes are exposed as properties on the element object. Property names are independent of the case of the attribute in the source document and follow some naming rules.

Methods on element objects include:

getAttribute(name:String)
Returns attribute value as String
setAttribute(name:String, value:String)
Sets specified attribute to specified value
removeAttribute(name:String)
Removes specified attribute

Attribute values can also be retrieved and set using the property directly

```
elem.propertyName
elem.propertyName = value
```

Consider:

```
<script>
 function RunScript1(){
 document.getElementById("mylink").href="ref2.htm";
 }
 function RunScript2(){
 document.getElementById("mylink").setAttribute("href", "ref2.htm");
 }
</script>

<h2 id="myh2"><a id="mylink" href="ref1.htm">My test line</a></h2>
```

The effect of both functions is to change the value of the href attribute to ref2.htm

Consider the following:

```
<style type="text/css">
#pid1 {color:red}
#pid2 {color: yellow}
</style>
<script language="JavaScript" type="text/javascript">
function change1() {
  document.getElementById("pid2").id = "pid1";
}

function change2() {
  document.getElementById("pid1").id = "pid2";
}
</script>

<p id="pid1">Text string</p>
```

The effect is to change the value of the id attribute of the p element, in the first case from pid2 to pid1 and in the second the reverse. Note how changing the attribute changes the appearance of the primitive. The style sheet declares that an element with an id attribute with value pid1 has colour red, whereas an element with an id attribute with value pid2 has colour yellow. Changing the attribute changes the colour.

Attributes can also be removed from elements. For example:

```
<script language="JavaScript">
function RemAttr(){
  document.getElementById("auth1").removeAttribute("style");
}
</script>

<p id="auth1" style="color:green;">Author: David Duce</p>
```

The effect is to remove the style from the p element and which causes its appearance to revert to the normal style for that element.

The content of an element can be hidden in the presentation by setting the visibility property of the element to hidden. For example applying the function:

```
function hide(){
  document.getElementById("auth1").style.visibility="hidden";
}
```

to:

```
<p>The DOM Made Complicated</p>
<p id="auth1" style="visibility:visible;">Author: David Duce</p>
<p>Date: October 2001</p>
<p>
```

will produce the presentation:

The DOM Made Complicated

Date: October 2001

Note that although the text "Author: David Duce" is now hidden, space has been allocated where the text would have appeared. In order to remove space allocated to the text, i.e. to reflow the document, the display property has to be set to the value none.

```
function reflow() {
  document.getElementById("auth1").style.display="none";
}
```

This will produce the presentation:

The DOM Made Complicated

Date: October 2001

The appearance of the document could be restored using:

```
function reset() {
  document.getElementById("auth1").style.visibility="visible";
  document.getElementById("auth1").style.display="";
}
```

The innerText method is used to change the content text of a HTML element, although this method does not appear to be defined in the W3C DOM Recommendation. For example the content "My Test Title" of the h2 element:

```
<h2 id="myh2">My Test Title</h2>
```

can be changed to "My New Test Title" using:

```
<document.getElementById("myh2").innerText="My New Test Title";
```

Using DOM methods, the same effect can be achieved with:

```
var h1ref = document.getElementById("myh2");
var oldtextref = h1ref.firstChild;
var newtextref = document.createTextNode("My New Test Title");
h1ref.replaceChild(newtextref, oldtextref);
```

This code first creates a new text node element using createTextNode, then replaces the link to the old text content with this new text content element.

Using these methods, we could write a JavaScript function to sort the rows of a table. For example the table:

The Voice of the Atlas	Aatabou, Najat	CDORBD 069
French Blues	Abshire, Nathan	CD 373
The Essential Roy Acuff	Acuff, Roy	48956
Qareeb	Akhtar, Najma	TERRACO 103
Aldina	Aldina	Fado C08
Segments	Allen, Geri	DIW-833
Azure-te	Allyson, Karrin	CCD-4641

could be sorted by the title column (first column) using the JavaScript function:

```
function insertionSort(t, iRowStart, iRowEnd, fReverse)
{
    var iRowInsertRow, iRowWalkRow;
    for ( iRowInsert = iRowStart + 1 ; iRowInsert <= iRowEnd ; iRowInsert++ )
    {
        textRowInsert = t.children[iRowInsert].innerText;

        for ( iRowWalk = iRowStart ; iRowWalk <= iRowInsert ; iRowWalk++ )
        {
            textRowCurrent = t.children[iRowWalk].innerText;

            if ( (   (!fReverse && textRowInsert <= textRowCurrent)
                 || ( fReverse && textRowInsert >= textRowCurrent) )
                 && (iRowInsert != iRowWalk) )
            {
                eRowInsert = t.children[iRowInsert];
                eRowWalk = t.children[iRowWalk];
                t.insertBefore(eRowInsert, eRowWalk);
                iRowWalk = iRowInsert; // done
            }
        }
    }
}
```

The sort function could be invoked using:

```
<input type=button value="Sort By Title"
  onclick="insertionSort(Table1.children[0], 0,
  Table1.rows.length - 1, false)">
```

4. DOM Level 2

- 4.1 Introduction
- 4.2 Core
- 4.3 Event model

4.1 Introduction

DOM Level 2 extends the API defined in DOM Level 1. DOM Level 1 is contained in DOM Level 2. We will not describe all the functionality of DOM Level 2 in this Primer, but aim instead to give a flavour of the key developments.

4.2 Core

DOM Level 2 caters fully for the core requirements of XML, as well as HTML documents. The main extension is to provide a way to handle documents containing elements from different XML namespaces. The significance of namespaces will become clearer later in the course (in the term 2 XML modules), for now it is enough to note that a document might contain, say, text, graphics and mathematics, and it is necessary to be able to distinguish between the nodes for each, given that there is no requirement in XML for element names to be unique across different languages.

The elements:

```
<a:elementName xmlns:a="SomeURI"/>
<b:elementName xmlns:b="AnotherURI"/>
```

each have the same name, "elementName", but they belong to different name spaces, and hence should yield distinguishable nodes in the DOM tree. This is achieved in DOM Level 2 Core through the addition of new properties and attributes to manage namespaces. The properties include: namespaceURI and prefix. New methods include getElementByTagNameNS and createElementNS.

4.3 Event model

The DOM Level 2 provides a rich generalization of the basic idea that the document tree can be modified in response to events such as mouse events. There are two components to this, the **event model** which defines the basic mechanisms for event handling independently from the particular type of event being handled and a set of **event modules** which populate the event model with particular types of event. These aspects of the DOM are described in Part 2 of this Primer.

5. Window Manipulation

Although not a part of the DOM, it is useful to note that browsers provide methods for manipulating windows. These notes refer particularly to IE 6.0.

The Window object is a global object and the properties of this object are global variables in JavaScript. The global variables window and self refer to the window object. Other properties include:

status
Text in status line
document
Reference to document object in window
history
Reference to user's browsing history for window
location
Reference to URL displayed; setting loads new document
name
Name of window

The methods include:

alert()
confirm()
prompt()
Display dialogue boxes
open(...)
Open window (see below for parameters)
close()
Close window
moveBy()
moveTo()
Move window
resizeBy()
resizeTo()
Resize window

An example using some of these methods follows.

```
var w = window.open("overview.htm", "newwin",
  "width=400, height=350, status=yes, resizeable=yes");
w.name = "My New Window";
w.status = "Demonstration in progress";
```

This will open a new resizeable window with width 400 pixels, height 350 pixels, in which the document overview.htm will be displayed. The title of the window is set to "My New Window" and the status line to "Demonstration in progress".

A. The Core DOM

This section is a quick reference to the Core DOM ECMAScript binding. The properties and methods that are most likely to be useful are listed. Note that this is NOT a complete listing of the Core DOM at either Level 1 or Level 2, nor does inclusion of a property or method in this section indicate that it is available on all browsers.

The notation name:Type is used to indicate that the parameter entity name has data type Type.

Object Document		
Methods	**Return type**	**Explanation**
createElement(tagName:String)	Element	Creates new element
createDocumentFragment()	DocumentFragment	Creates a document fragment
createTextNode(data:String)	Text	Create text node
createAttribute(name:String)	Attr	Create attribute node
getElementsByTagName(tagName:String)	NodeList	Returns list of nodes with specified tag
getElementsById(elementId:String)	Element	Returns element with specified identifier

Object Node		
Properties	**Property type**	**Explanation**
nodeName	String	Name of node
nodeValue	String	Value of node
nodeType	short	Type of underlying object
parentNode	Node	Parent of node. Document, document fragment and Attr do not have parent node
childNodes	NodeList	References to all children of node
firstChild	Node	First child of node. null if none
lastChild	Node	Last child of node. null if none
previousSibling	Node	Node immediately preceeding this node. null if none
nextSibling	Node	Node immediately following this node. null if none
attributes	NamedNodeMap	Attributes of this node
ownerDocument	Document	document object associated with node
Methods	**Return type**	**Explanation**
insertBefore (newChild:Node, refChild:Node)	Node	Inserts newChild before existing node refChild
replaceChild (newChild:Node, oldChild:Node)	Node	Replaces child node oldChild with node newChild
removeChild (oldChild:Node)	Node	Removes the child indicated by oldChild
appendChild (newChild:Node)	Node	Add newChild to end of list of children of this node
hasChildNodes()	boolean	true if node has any children, false otherwise
cloneNode(deep:boolean)	Node	Returns a duplicate of the node. If deep is true, recursively clones the subtree under the node, if false clones only nodes (and attributes if an element)

Object NodeList		
Properties	**Property type**	**Explanation**
length	int	Number of nodes in list
Methods	**Return type**	**Explanation**
item(index: unsigned long)	Node	Returns specified item in list, or null if not a valid index

Object Attr (all the properties and methods of Node plus the following)		
Properties	**Property type**	**Explanation**
name	String	Name of attribute
specified	boolean	true if assigned value in document, false otherwise. If assigned default value in DTD then that value is held in value property
value	String	true if assigned value in document, false otherwise. If assigned default value in DTD then that value is held in value property

Object Element (properties and methods of Node plus those below (incomplete list))		
Properties	**Property type**	**Explanation**
tagName	String	Element tag name
Methods	**Return type**	**Explanation**
getAttribute(name:String)	String	Retrieves attribute value by name
setAttribute (name:String,value:String)	void	Adds new attribute, if already present, replaces existing value
removeAttribute(name:String)	void	Removes the specified attribute. If attribute has a default value, it is immediately replaced with that value

Object Text (incomplete list)		
Methods	**Return type**	**Explanation**
splitText(offset:int)	Text	Breaks text node to which applied into two. Afterwards this node contains text up to offset, next sibling inserted into tree by method contains remainder
setAttribute (name:String,value:String)	void	Adds new attribute, if already present, replaces existing value
removeAttribute(name:String)	void	Removes the specified attribute. If attribute has a default value, it is immediately replaced with that value

Node types (incomplete list)	
Node	**nodeType**
Element	1
Text	3
Document	9
Comment	8
Attr	2

B. The HTML DOM

The HTML DOM builds on the Core DOM. The main extension is that attributes of HTML elements are made available as properties. The names of the properties correspond to the attribute names. The lists of properties so created are extensive and will not be reproduced here.

Object HTMLDocument **(incomplete list))**		
Properties	**Property type**	**Explanation**
title	String	document title as specified in TITLE element
URL	String	complete URI of the document
body	HTMLElement	element that contains the content (BODY or outermost FRAMESET element)
images	HTMLCollection	list of all IMG elements in document
links	HTMLCollection	list of all AREA and anchor (A) elements with a value for **href** attribute
forms	HTMLCollection	list of all the forms of a document
anchors	HTMLCollection	list of all anchor (A) elements with a value for the **name** attribute
cookies	HTMLCollection	Cookies associated with a document
Methods	**Return type**	**Explanation**
open()	void	opens document stream for writing
close()	void	closes document stream and forces rendering
write(text: String)	void	write string of text to document stream. Text is parsed into document's structure model
writeln(text: String)	void	As **write** except string of text is followed by newline character

The object type HTMLCollection has the following properties and methods.

Object HTMLCollection		
Properties	**Property type**	**Explanation**
length	int	Number of items in collection
Methods	**Return type**	**Explanation**
item(index: int)	Node	Returns index'th item of collection (starting from 0)
namedItem(name)	void	Returns named item in collection