Chapter 13. Dynamic HTML: Object Model and Collections

The object model gives access to all elements of a Web page, whose properties and attributes can thus be retrieved or modified by scripting.

The value of the `id` attribute of an element becomes the name of the object representing that element.

The various HTML attributes of the element become properties of this object (which can be modified).

For example, the value of the `innerText` property of a `p` element is the text within that element.

So, if we have a `P` element with `id pText`, we can dynamically change the rendering of this element with, e.g.,

```
pText.innerText  =  “Good bye!”;
```

This is *dynamic content*.

In the following example, the function (not the `window` method) `alert` is used to pop up an alert box.

Example:

```
<html>

<head>
<title>object model</title>

<script type = "text/javascript">
   function start()
   {
      alert( pText.innerText );
      pText.innerText = "Good bye!";
   }
</script>

</head>

<body onload = "start()">

<p id = "pText">hello!</p>

</body>
</html>
```

When the page is loaded, the following appears both in the window and in an alert box:

Hello!

After the alert box is dismissed, the following appears in the window:

Good bye!

Collections

A *collection* is an array of related objects on a page.

The `all` collection of an element (syntactically a property) is a collection of all the elements in it in order of appearance.

This gives us reference even to elements that lack `ID` attributes.

Like all collections, it has a `length` property.

For example,

```
document.all[ i ]
```

references the i^{th} element in the document.

The following are needed for the next example.

The `innerHTML` property of a `p` element is like the `innerText` property but may contain HTML formatting.

The `tagName` property of an element is the name of the HTML element.

Example:

```
<html>

<!-- Using the all collection -->

<head>
<title>Object Model</title>

<script type = "text/javascript">
   var elements = "";

   function start()
   {
      for ( var loop = 0;
           loop < document.all.length; ++loop )
        elements += "<BR>" +
                    document.all[ loop ].tagName;

      pText.innerHTML += elements;
   }
</script>
</head>

<body onload = "start()">

<p id = "pText">Elements on this Web page:</p>

</body>
</html>
```

```
Elements on this Web page:
HTML
HEAD
TITLE
!
SCRIPT
BODY
P
```

Note that the `tagName` property of a comment is `!`.

The `children` collection for an object is like the `all` collection but contains only the next level down in the hierarchy.

For example, an `HTML` element has a `head` and a `body` child.

In the following example, function `child(object)` does a preorder traversal of the part of the object hierarchy rooted at `object`.

For every object with children, it appends on to global variable `elements`

- `<li>`,
- the name of the HTML element represented by the object,
- `<ul>`,
- similar information for the children (iteratively) and more distant descendants (recursively) of the object, and
- `</ul>`

The `body` tag is

```
<body  onload = "child( document.all[ 1 ] );
                 myDisplay.outerHTML += elements;">
```

When the page is loaded, this calls `child`, passing it the second object in the hierarchy.

(The first element is the comment at the top of the file.)

When control returns from the call, the string in global variable `elements` (containing the hierarchical description of the obects) is appended to the value of the `outerHTML` property of `P` element `myDisplay`.

Property `outerHTML` is like `innerHTML` but includes the enclosing tags.

Example:

```
<!-- The children collection -->

<head>
<title>Object Model</title>

<script type = "text/javascript">
   var elements = "<ul>";

   function child( object )
   {
      var loop = 0;

      elements += "<LI>" + object.tagName + "<UL>";

      for( loop = 0; loop < object.children.length;
           loop++ )

         if ( object.children[loop].children.length )
            child( object.children[ loop ] );
         else
            elements += "<LI>" +
                        object.children[ loop ].tagName
                        + "</LI>";

      elements += " </UL> ";
   }
</script>
</head>

<body onload = "child( document.all[ 1 ] );
                myDisplay.outerHTML += elements;">

<p>Welcome to our <strong>Web</strong> page!</p>

<p id = "myDisplay">
Elements on this Web page:
</p>

</body>
</html>
```

Welcome to our **Web** page!
Elements on this Web page:

- HTML
  - HEAD
    - TITLE
    - SCRIPT
  - BODY
    - P
      - STRONG
    - P

Dynamic Styles and Positioning

We can change an element's style dynamically.

Most HTML elements have a `style` object as a property.

The names of the properties of this object used in JavaScript are generally their HTML names modified to avoid the "-" (seen as subtraction in JavaScript) – e.g.,

HTML	JavaScript
`background-color`	`backgroundColor`
`border-width`	`borderWidth`
`font-family`	`fontFamily`

We can make assignments to these properties, dynamically changing the element's rendering – e.g.,

```
document.body.style.fontSize  =  16;
```

Suppose an element's CSS `position` property is declared to be either `absolute` or `relative`.

Then we can move it by manipulating any of the `top`, `left`, `right`, or `bottom` CSS properties of its style object.

– This is *dynamic positioning*.

Example

Suppose in the body we have

```
<p id = "pText1"
    style = "position: absolute; top: 35">
XXX</p>
```

and in the script we have

```
pText1.style.left = 100;
```

Then the rendering `XXX` of the `pText1` element will be shifted right 100 pixels.

We can also change the `class` attribute of an element by assigning the name of a class we have defined to the element's `className` property.

Example

Suppose in the body we have

```
<p id = "pText2">CCC</p>
```

in the style sheet we have defined

```
.bigText  { font-size: 2em }
```

and in the script we have

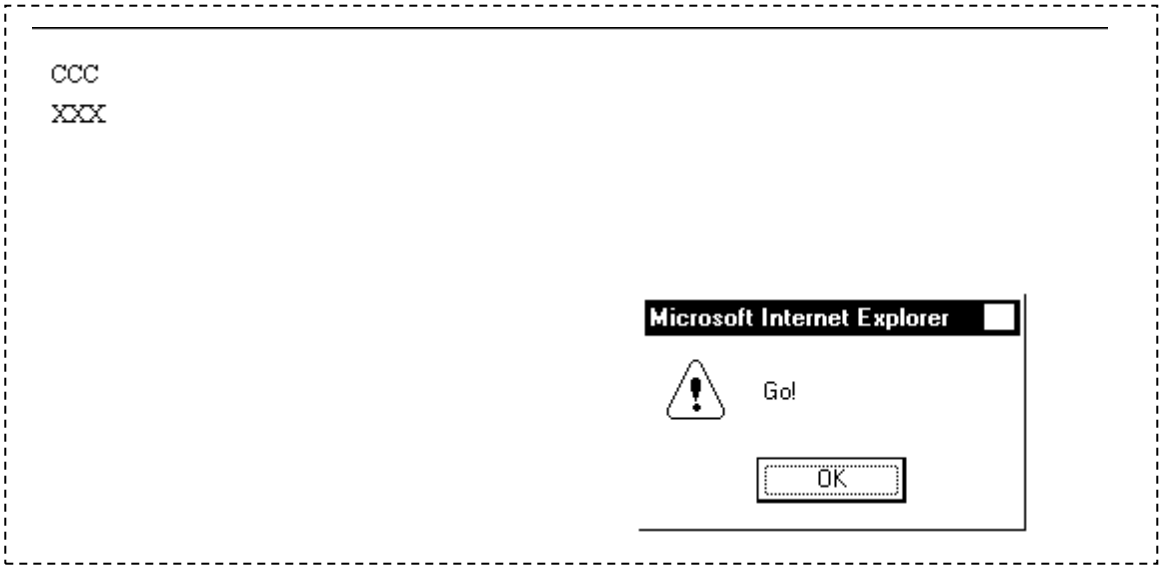
```
pText2.className = "bigText";
```

Then the rendering of `XXX` will be twice as large as the surrounding text.

Example:

```
<html>

<head>
<title>Dynamic Styles</title>

<style type = "text/css">

  .bigText { font-size: 2em }

</style>

<script type = "text/javascript">
  function start()
  {
    alert( "Go!" );
    document.body.style.fontSize = 16;
    pText1.style.left = 100;
    pText2.className = "bigText";
  }
</script>

</head>

<body onload = "start()">

<p id = "pText1"
   STYLE = "position: absolute; top: 35">
XXX</p>

<p id = "pText2">CCC</p>

</body>
</html>
```

The initial rendering is:



After the alert dialog box is dismissed, the rendering is:

To get perceptible dynamic effects, we need some way to control when element attributes and properties are changed.

The `setInterval` method of the `window` object, used as

`window.setInterval( “`*function_name*`()”, `*msec*`)`,

invokes function *function_name* every *msecs* milliseconds.

Method `setTimeout` has the same parameters, but it waits *msecs* milliseconds before invoking *function_name* only once.

Both `setInterval` and `setTimeout` return values, which can be assigned to variables.

Method `clearInterval` takes the value returned by `setInterval` and terminates the timed function’s executions.

Method `clearTimeout` takes the value returned by `setTimeout` and stops the timer before it fires (if it hasn’t already).

Example

```
timer  =  window.setInterval( “f()”, 1000);
…
window.clearInterval( timer );
```

Example

```
<html>

<head>
<title>Dynamic Positioning</title>

<script language = "javascript">
   var speed = 5;
   var count = 10;
   var direction = 1;
   var firstLine = "Text growing";
   var fontStyle =
       [ "serif", "sans-serif", "monospace" ];
   var fontStylecount = 0;

   function start()
   {
      window.setInterval( "run()", 100 );
   }
```

Continued next page

Continued from previous page

```
   function run()
   {
      count += speed;

      if ( ( count % 200 ) == 0 ) {
         speed *= -1;
         direction = !direction;

         pText.style.color =
            ( speed < 0 ) ? "red" : "blue" ;
         firstLine =
            ( speed < 0 ) ? "Text shrinking" :
                            "Text growing";
         pText.style.fontFamily =
            fontStyle[ ++fontStylecount % 3 ];
      }

      pText.style.fontSize = count / 3;
      pText.style.left = count;
      pText.innerHTML = firstLine +
                        "<BR> Font size: " +
                        count + "px";
   }
</script>
</head>

<body onload = "start()">

<p id = "pText"
   STYLE = "position: absolute; left: 0;
            font-family: serif; color: blue">
Welcome!</p>

</body>
</html>
```

Cross-Frame Referencing

With frame sets, we have the problem of referencing elements that are on different pages.

In a page loaded into a frame, `parent` references the parent page (containing a `frame` element with the current page's URL as the value of its `src` attribute).

Then `parent.frames` is the collection of frames in the parent of the current page.

We can reference any page that is the source of some frame in the parent's frame set either

- by using the ordinal (0 to the number of frames minus one) of the frame within the frame set as an index or
- by using the value of the `name` attribute of the desired frame as an argument.

Example

```
parent.frames[ 1 ]
parent.frames( “lower” )
```

This gets us to the `document` level of the page.

If no page is loaded into the selected frame, the frame's `document` object is still defined – designating the space rendered for that frame in the rendering of the parent.

Example

```
parent.frames( “lower” ).document.writeln(
                                “<p>lower</p>” );
```

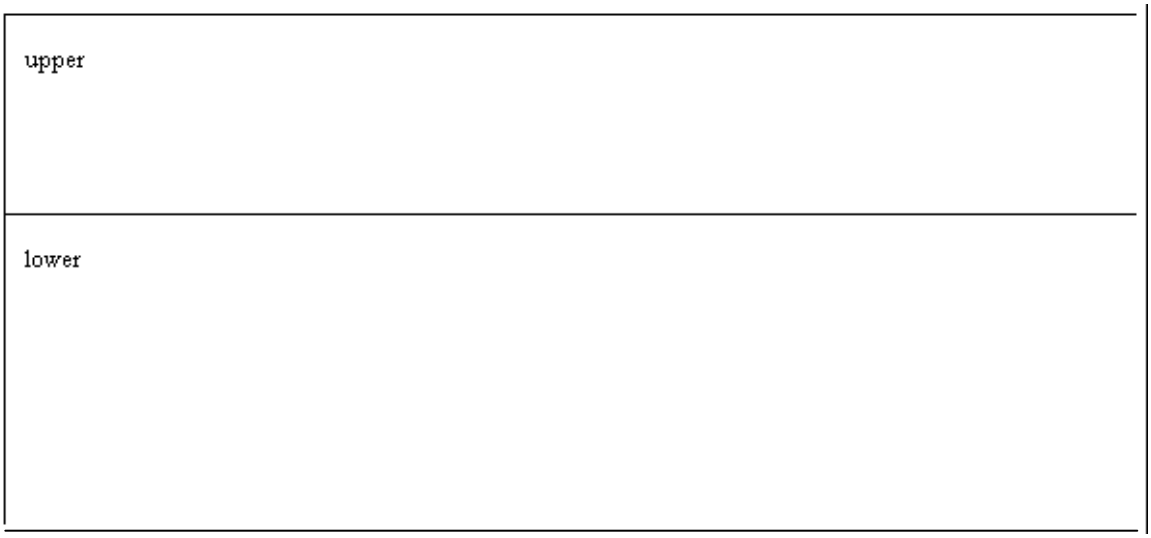
Example:

The following is a page that defines a frame set. One of its frames has “frameset2.html” as the value of its SRC attribute.

```
<html>

<head>
   <title>Frames collection</title>
</HEAD>

<frameset rows = "100, *">
   <frame src = "frameset2.html" name = "upper">
   <frame src = "" name = "lower">
</frameset>

</html>
```

The following is file `frameset2.html`:

```
<html>

<head>
<title>The frames collection</title>

<script type = "text/javascript">
   function start()
   {
      parent.frames( "lower" ).document.write(
                    "<p>lower</p>" );
   }
</script>
</head>

<body onload = "start()">

<p>upper</p>

</body>
</html>
```


upper
lower

The navigator Object

Both Netscape's Navigator and Microsoft's Internet Explorer support the `navigator` object.

It contains information about the browser that's viewing the page.

Property `navigator.appName` is

`"Microsoft Internet Explorer"` if the application is Internet Explorer and

`"Netscape"` if the application is Netscape's Navigator.

Property `navigator.appVersion` is a string of various information, starting with the version number.

For the following example, note that

`document.location`

is the URL of the document being viewed.

Example:

```
function start()
   {
      if ( navigator.appName ==
           "Microsoft Internet Explorer" ) {

        if ( navigator.appVersion.substring(1,0)
             >= "4" )
           document.location =
                    "newIEversion.html";
        else
           document.location =
                    "oldIEversion.html";
     }
     else
        document.location = "NSversion.html";
   }
```

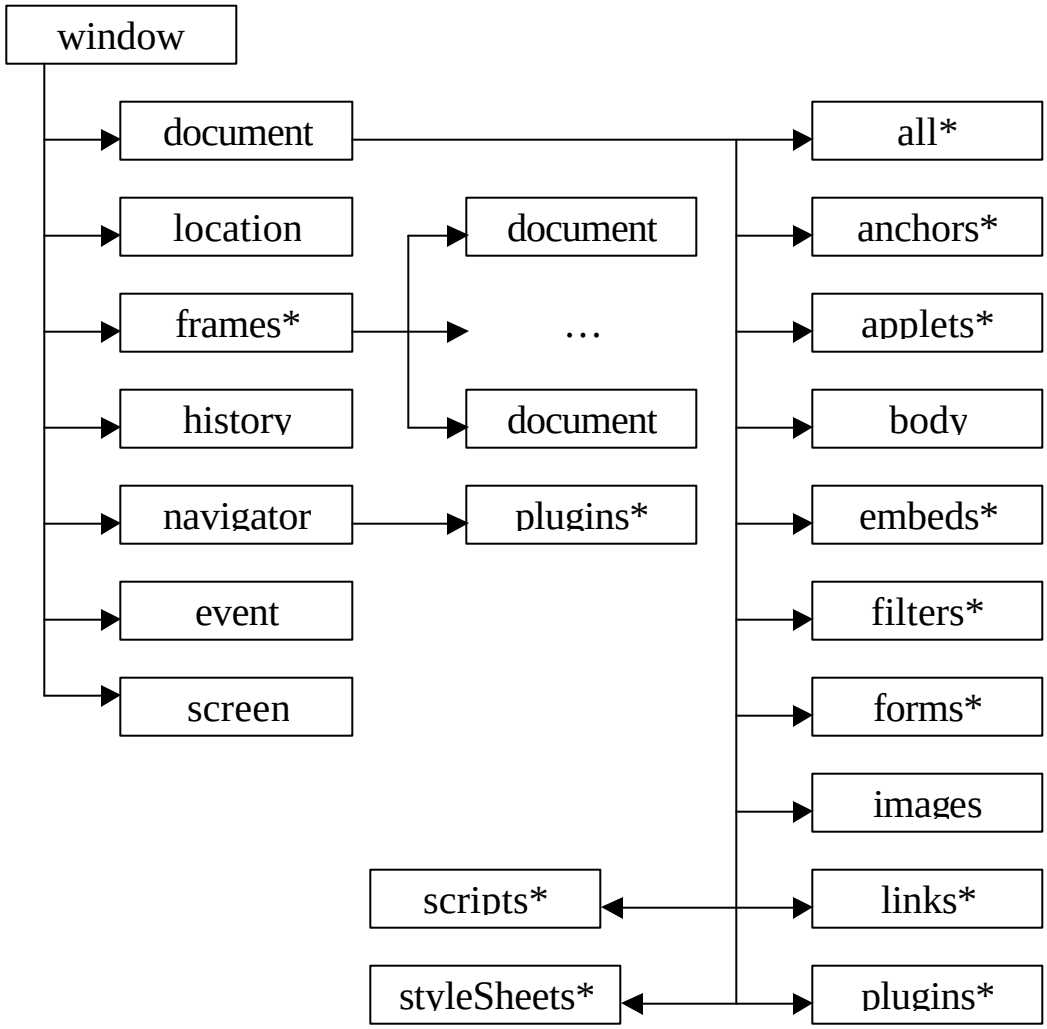
Summary of the DHTML Object Model

For comprehensive documentation on HTML, DHTML, and CSS, see
`http://msdn.microsoft.com/workshop/c-frame.htm#`
`/workshop/author/default.asp`

See the DHTML References section for detailed descriptions of every object, collection, and event used in DHTML

The following, for many of the important objects and properies, shows the what objects or collections are properties of others.

Collections are marked with a '*'.



The following are descriptions of some of the common objects and collections in the object model – see the text, Fig. 13.11, pp. 452-453 of a larger list.

`window` (object) represents the browser window and provides access to the `document` object contained in the window.

If the window contains frames, a separate `window` object is created for each frame, providing access to the `document` rendered in that frame.

`document` (object) provides access to every element in the HTML document.

`frames` (collection) contains `window` objects for each frame in the browser window.

`body` (object) provides access to the `body` element of the HTML document.

`location` (object) contains the URL of the rendered document.

`screen` (object) contains information on the screen of the computer on which the browser is running.

`forms` (collection) contains all the `form` elements of the document.

`images` (collection) contains all the `img` elements of the document.

`scripts` (collection) contains all the `script` elements of the document.

`styleSheets` (collection) contains all `styleSheet` objects, representing each `style` element in the document and each style sheet included via a link.

`links` (collection) contains all the anchor (`A`) with an `href` property and all `area` elements representing links in an image map.

`anchors` (collection) contains all anchor (`A`) elements with a `NAME` or `ID` property.

`event` (object) can be used in an event handler to get information about the event that just occurred.