# 12

# Document Object Model (DOM): Objects and Collections

# 12.1 Introduction

- The Document Object Model gives you scripting access to *all* the elements on a web page. Using JavaScript, you can create, modify and remove elements in the page dynamically.

# 12.2 Modeling a Document: DOM Nodes and Trees

- `getElementById` method
  - Returns objects called DOM nodes
  - *Every* piece of an HTML5 page (elements, attributes, text, etc.) is modeled in the web browser by a DOM node
- The nodes in a document make up the page's DOM tree, which describes the relationships among elements
- Nodes are related to each other through child-parent relationships
- A node can have multiple children, but only one parent
- Nodes with the same parent node are referred to as siblings
- The `html` node in a DOM tree is called the root node, because it has no parent

| Browser | Command to display developer tools |
|---|---|
| Chrome | Windows/Linux: *Control* + *Shift* + *i*<br>Mac OS X: *Command* + *Option* + *i* |
| Firefox | Windows/Linux: *Control* + *Shift* + *i*<br>Mac OS X: *Command* + *Shift* + *i* |
| Internet Explorer | *F12* |
| Opera | Windows/Linux: *Control* + *Shift* + *i*<br>Mac OS X: *Command* + *Option* + *i* |
| Safari | Windows/Linux: *Control* + *Shift* + *i*<br>Mac OS X: *Command* + *Option* + *i* |

**Fig. 12.1** | Commands for displaying developer tools in desktop browsers.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 12.2: domtree.html -->
4  <!-- Demonstration of a document's DOM tree. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>DOM Tree Demonstration</title>
9     </head>
10    <body>
11       <h1>An HTML5 Page</h1>
12       <p>This page contains some basic HTML5 elements. The DOM tree
13          for the document contains a DOM node for every element</p>
14       <p>Here's an unordered list:</p>
15       <ul>
16          <li>One</li>
17          <li>Two</li>
18          <li>Three</li>
19       </ul>
20    </body>
21 </html>
```

**Fig. 12.2** | Demonstration of a document's DOM tree. (Part 1 of 3.)


**Fig. 12.2** | Demonstration of a document's DOM tree. (Part 2 of 3.)

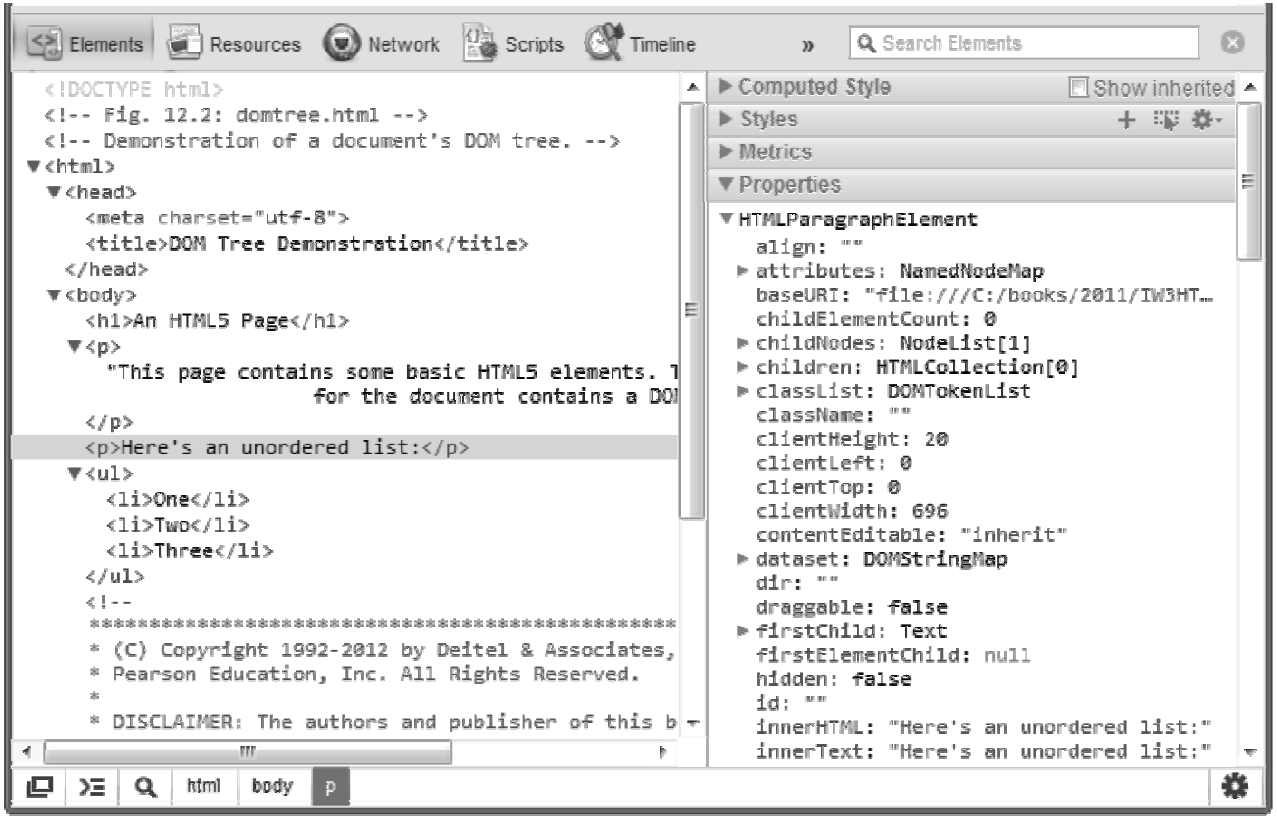


**Fig. 12.2** | Demonstration of a document's DOM tree. (Part 3 of 3.)

# 12.3 Traversing and Modifying a DOM Tree

- The next example demonstrates several DOM node features and two additional document-object methods.
  - It allows you to highlight, modify, insert and remove elements.
- CSS class `highlighted` is applied dynamically to elements in the document as we add, remove and select elements using the `form`.

```
1  /* Fig. 12.3: style.css */
2  /* CSS for dom.html. */
3  h1, h3       { text-align: center;
4                 font-family: tahoma, geneva, sans-serif; }
5  p            { margin-left: 5%;
6                 margin-right: 5%;
7                 font-family: arial, helvetica, sans-serif; }
8  ul           { margin-left: 10%; }
9  a            { text-decoration: none; }
10 a:hover      { text-decoration: underline; }
11 .nav         { width: 100%;
12                border-top: 3px dashed blue;
13                padding-top: 10px; }
14 .highlighted { background-color: yellow; }
15 input        { width: 150px; }
16 form > p     { margin: 0px; }
```

**Fig. 12.3** | CSS for basic DOM functionality example.

# 12.3 Traversing and Modifying a DOM Tree (Cont.)

- We'll manipulate the HTML5 document dynamically by modifying its DOM.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 12.4: dom.html -->
4  <!-- Basic DOM functionality. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Basic DOM Functionality</title>
9        <link rel = "stylesheet" type = "text/css" href = "style.css">
10       <script src = "dom.js"></script>
11    </head>
12    <body>
13       <h1 id = "bigheading" class = "highlighted">
14          [bigheading] DHTML Object Model</h1>
15       <h3 id = "smallheading">[smallheading] Element Functionality</h3>
16       <p id = "para1">[para1] The Document Object Model (DOM) allows for
17          quick, dynamic access to all elements in an HTML5 document for
18          manipulation with JavaScript.</p>
```

**Fig. 12.4** | HTML5 document that's used to demonstrate DOM functionality for dynamically adding, removing and selecting elements. (Part 1 of 4.)

```
19          <p id = "para2">[para2] For more information, check out the
20             "JavaScript and the DOM" section of Deitel's
21             <a id = "link" href = "http://www.deitel.com/javascript">
22                [link] JavaScript Resource Center.</a></p>
23          <p id = "para3">[para3] The buttons below demonstrate:(list)</p>
24          <ul id = "list">
25             <li id = "item1">[item1] getElementById and parentNode</li>
26             <li id = "item2">[item2] insertBefore and appendChild</li>
27             <li id = "item3">[item3] replaceChild and removeChild</li>
28          </ul>
29          <div id = "nav" class = "nav">
30             <form onsubmit = "return false" action = "#">
31                <p><input type = "text" id = "gbi" value = "bigheading">
32                   <input type = "button" value = "Get By id"
33                      id = "byIdButton"></p>
34                <p><input type = "text" id = "ins">
35                   <input type = "button" value = "Insert Before"
36                      id = "insertButton"></p>
37                <p><input type = "text" id = "append">
38                   <input type = "button" value = "Append Child"
39                      id = "appendButton"></p>
```

**Fig. 12.4** | HTML5 document that's used to demonstrate DOM functionality for dynamically adding, removing and selecting elements. (Part 2 of 4.)

```
40                <p><input type = "text" id = "replace">
41                   <input type = "button" value = "Replace Current"
42                      id = "replaceButton()"></p>
43                <p><input type = "button" value = "Remove Current"
44                      id = "removeButton"></p>
45                <p><input type = "button" value = "Get Parent"
46                      id = "parentButton"></p>
47             </form>
48          </div>
49       </body>
50    </html>
```

**Fig. 12.4** | HTML5 document that's used to demonstrate DOM functionality for dynamically adding, removing and selecting elements. (Part 3 of 4.)

The document when it first loads. It begins with the large heading highlighted.

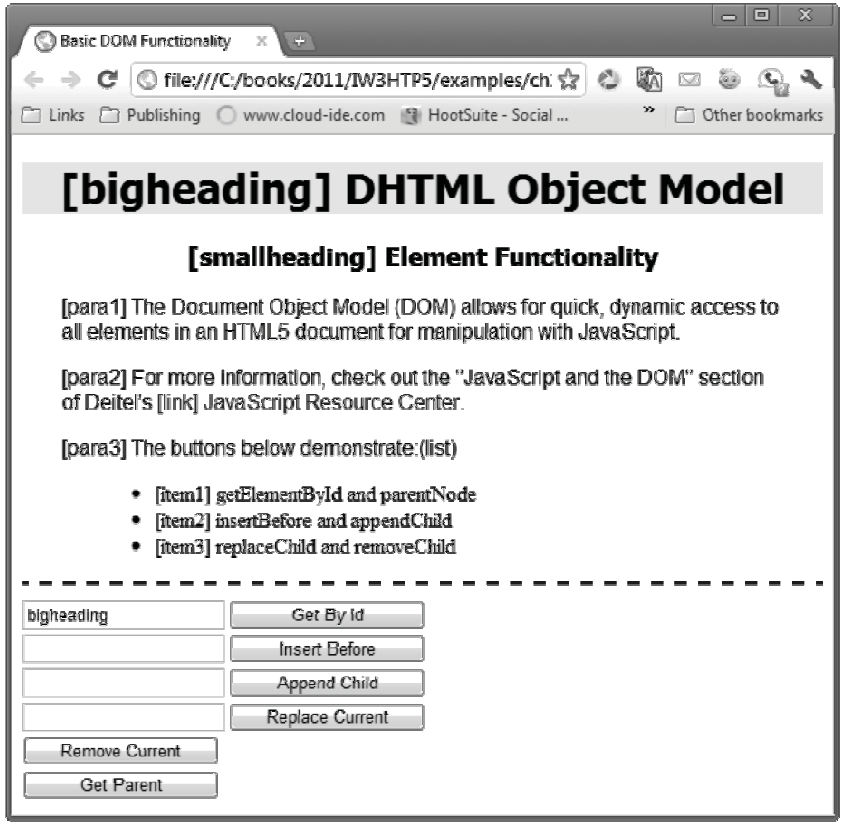


**Fig. 12.4** | HTML5 document that's used to demonstrate DOM functionality for dynamically adding, removing and selecting elements. (Part 4 of 4.)

# 12.3 Traversing and Modifying a DOM Tree (Cont.)

- The JavaScript code declares two variables
  - Variable `currentNode` keeps track of the currently highlighted node—the functionality of each button depends on which node is currently selected.
  - Variable `idcount` is used to assign a unique `id` to any new elements that are created.
- The remainder of the JavaScript code contains event-handling functions for the `button`s and two helper functions that are called by the event handlers.

```
1  // Fig. 12.5: dom.js
2  // Script to demonstrate basic DOM functionality.
3  var currentNode; // stores the currently highlighted node
4  var idcount = 0; // used to assign a unique id to new elements
5
6  // register event handlers and initialize currentNode
7  function start()
8  {
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 1 of 6.)

```
9      document.getElementById( "byIdButton" ).addEventListener(
10        "click", byId, false );
11     document.getElementById( "insertButton" ).addEventListener(
12        "click", insert, false );
13     document.getElementById( "appendButton" ).addEventListener(
14        "click", appendNode, false );
15     document.getElementById( "replaceButton" ).addEventListener(
16        "click", replaceCurrent, false );
17     document.getElementById( "removeButton" ).addEventListener(
18        "click", remove, false );
19     document.getElementById( "parentButton" ).addEventListener(
20        "click", parent, false );
21
22     // initialize currentNode
23     currentNode = document.getElementById( "bigheading" );
24  } // end function start
25
26  // call start after the window loads
27  window.addEventListener( "load", start, false );
28
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 2 of 6.)

```
29  // get and highlight an element by its id attribute
30  function byId()
31  {
32     var id = document.getElementById( "gbi" ).value;
33     var target = document.getElementById( id );
34
35     if ( target )
36        switchTo( target );
37  } // end function byId
38
39  // insert a paragraph element before the current element
40  // using the insertBefore method
41  function insert()
42  {
43     var newNode = createNewNode(
44        document.getElementById( "ins" ).value );
45     currentNode.parentNode.insertBefore( newNode, currentNode );
46     switchTo( newNode );
47  } // end function insert
48
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 3 of 6.)

```
49  // append a paragraph node as the child of the current node
50  function appendNode()
51  {
52     var newNode = createNewNode(
53        document.getElementById( "append" ).value );
54     currentNode.appendChild( newNode );
55     switchTo( newNode );
56  } // end function appendNode
57
58  // replace the currently selected node with a paragraph node
59  function replaceCurrent()
60  {
61     var newNode = createNewNode(
62        document.getElementById( "replace" ).value );
63     currentNode.parentNode.replaceChild( newNode, currentNode );
64     switchTo( newNode );
65  } // end function replaceCurrent
66
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 4 of 6.)

```
67  // remove the current node
68  function remove()
69  {
70     if ( currentNode.parentNode == document.body )
71        alert( "Can't remove a top-level element." );
72     else
73     {
74        var oldNode = currentNode;
75        switchTo( oldNode.parentNode );
76        currentNode.removeChild( oldNode );
77     }
78  } // end function remove
79
80  // get and highlight the parent of the current node
81  function parent()
82  {
83     var target = currentNode.parentNode;
84
85     if ( target != document.body )
86        switchTo( target );
87     else
88        alert( "No parent." );
89  } // end function parent
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 5 of 6.)

```
90
91  // helper function that returns a new paragraph node containing
92  // a unique id and the given text
93  function createNewNode( text )
94  {
95     var newNode = document.createElement( "p" );
96     nodeId = "new" + idcount;
97     ++idcount;
98     newNode.setAttribute( "id", nodeId ); // set newNode's id
99     text = "[" + nodeId + "] " + text;
100    newNode.appendChild( document.createTextNode( text ) );
101    return newNode;
102 } // end function createNewNode
103
104 // helper function that switches to a new currentNode
105 function switchTo( newNode )
106 {
107    currentNode.setAttribute( "class", "" ); // remove old highlighting
108    currentNode = newNode;
109    currentNode.setAttribute( "class", "highlighted" ); // highlight
110    document.getElementById( "gbi" ).value =
111       currentNode.getAttribute( "id" );
112 } // end function switchTo
```

**Fig. 12.5** | Script to demonstrate basic DOM functionality. (Part 6 of 6.)

# 12.3 Traversing and Modifying a DOM Tree (Cont.)

*Finding and Highlighting an Element Using `getElementById`, `setAttribute` and `getAttribute`*

- The first row of the `form` allows the user to enter the `id` of an element into the text field and click the Get By Id button to find and highlight the element.

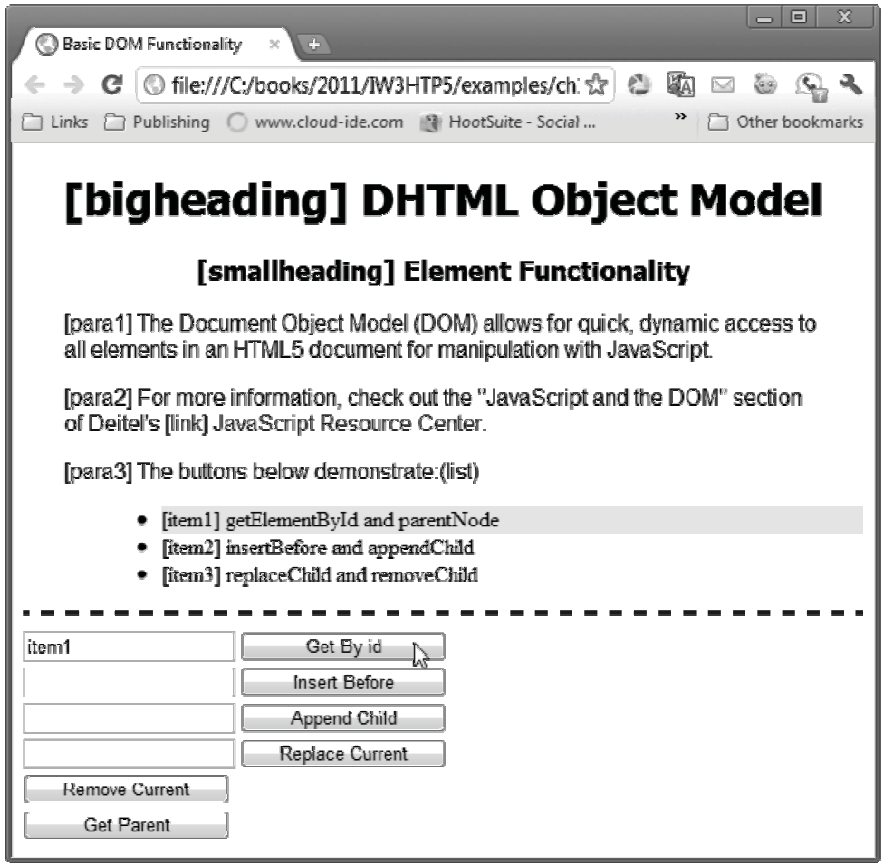


**Fig. 12.6** | The document of Figure 12.4 after using the **Get By id** button to select `item1`.

# 12.3 Traversing and Modifying a DOM Tree (Cont.)

- The DOM element methods `setAttribute` and `getAttribute` allow you to modify an attribute value and get an attribute value, respectively.

# 12.3 Traversing and Modifying a DOM Tree (Cont.)

- document object `createElement` method
  - Creates a new DOM node, taking the tag name as an argument. It does not *insert* the element on the page.
- document object `createTextNode` method
  - Creates a DOM node that contains only text. Given a string argument, `createTextNode` inserts the string into the text node.
- Method `appendChild`
  - Inserts a child node (passed as an argument) after any existing children of the node on which it's called
- Property `parentNode` contains the node's parent
- `insertBefore` method
  - Inserts `newNode` as a child of the parent directly before `currentNode`.
- `replaceChild` method
  - Receives as its first argument the new node to insert and as its second argument the node to replace.
- `removeChild` method
  - Remove the `oldNode` (a child of the new `currentNode`) from its place in the HTML5 document.

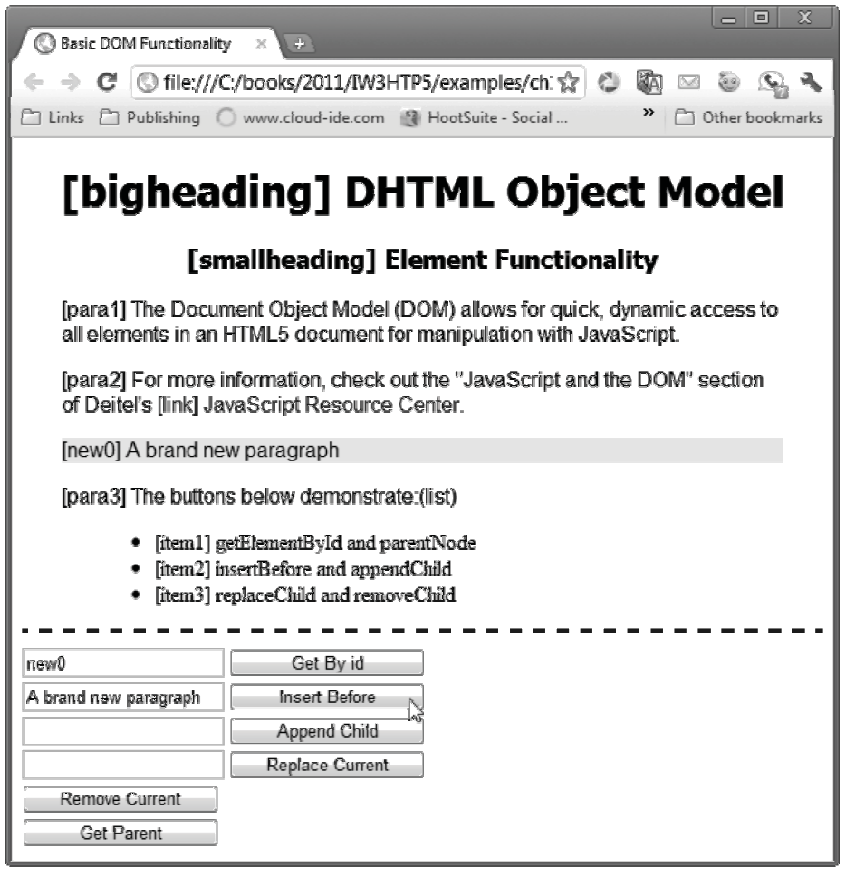


**Fig. 12.7** | The document of Figure 12.4 after selecting `para3` with the **Get By id** `button`, then using the **Insert Before** `button` to insert a new paragraph before `para3`.

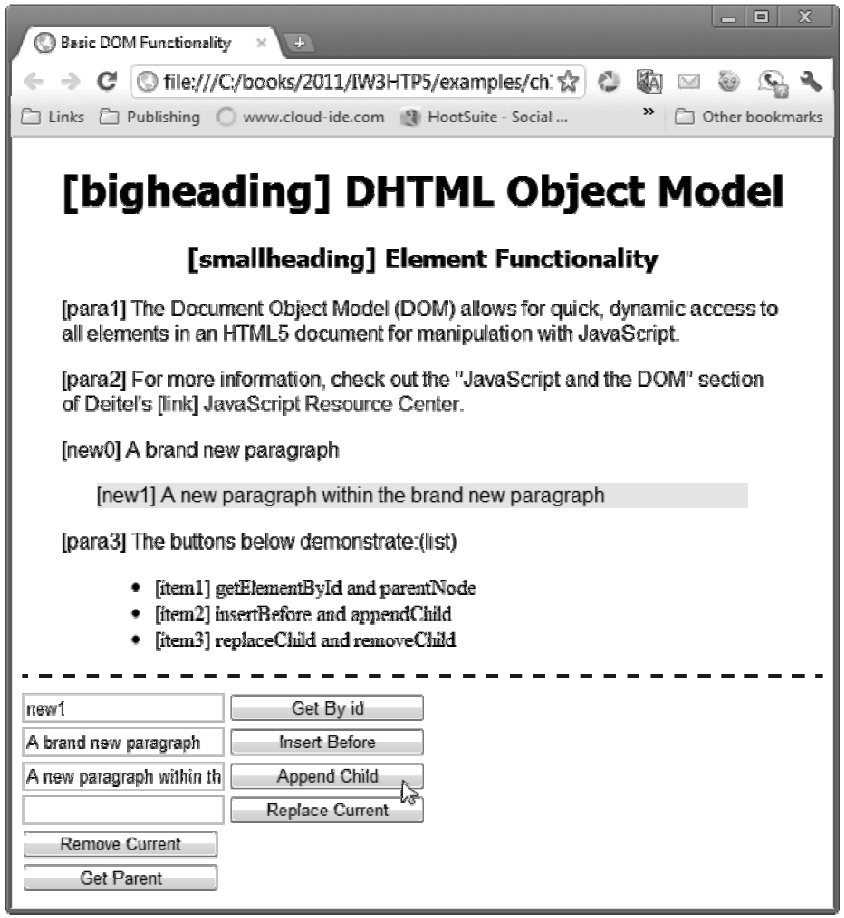


**Fig. 12.8** | The document of Figure 12.4 after using the **Append Child** button to append a child to the new paragraph in Figure 12.7.

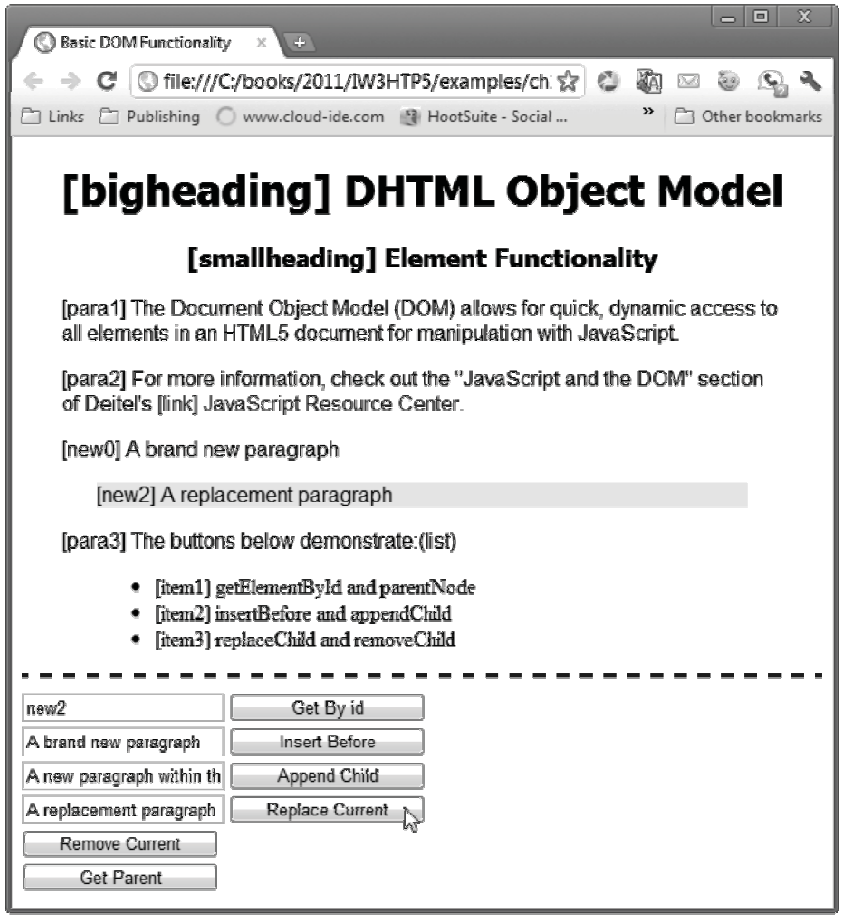


**Fig. 12.9** | The document of Figure 12.4 after using the **Replace Current** `button` to replace the paragraph created in Figure 12.8.

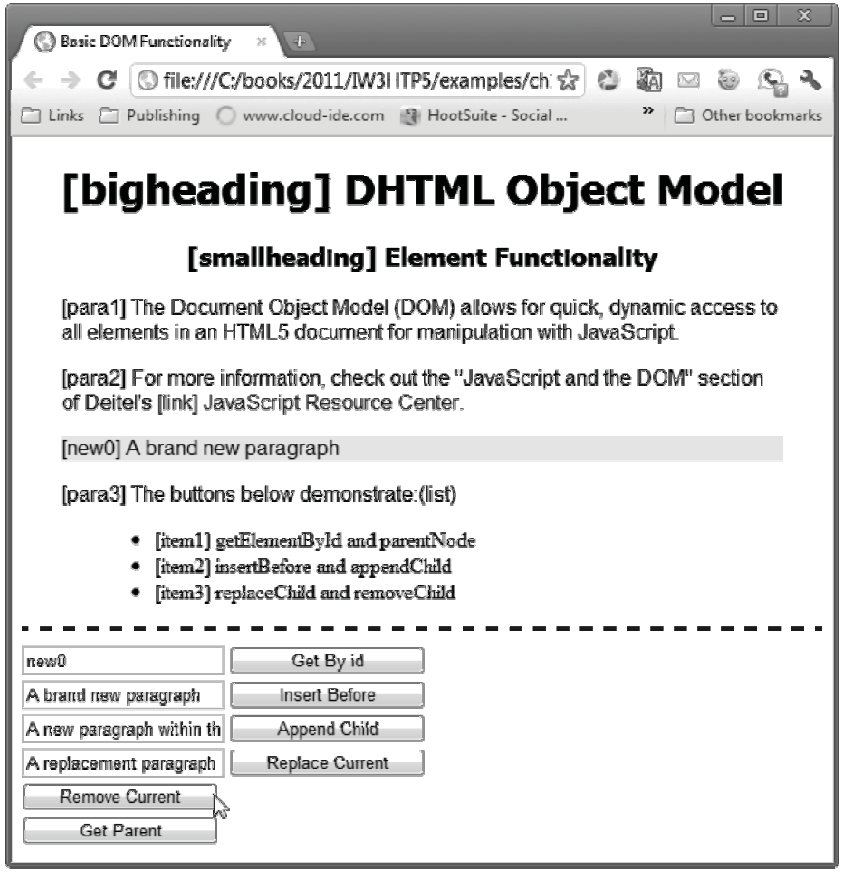


**Fig. 12.10** | The document of Figure 12.4 after using the **Remove Current** `button` to remove the paragraph highlighted in Figure 12.9.

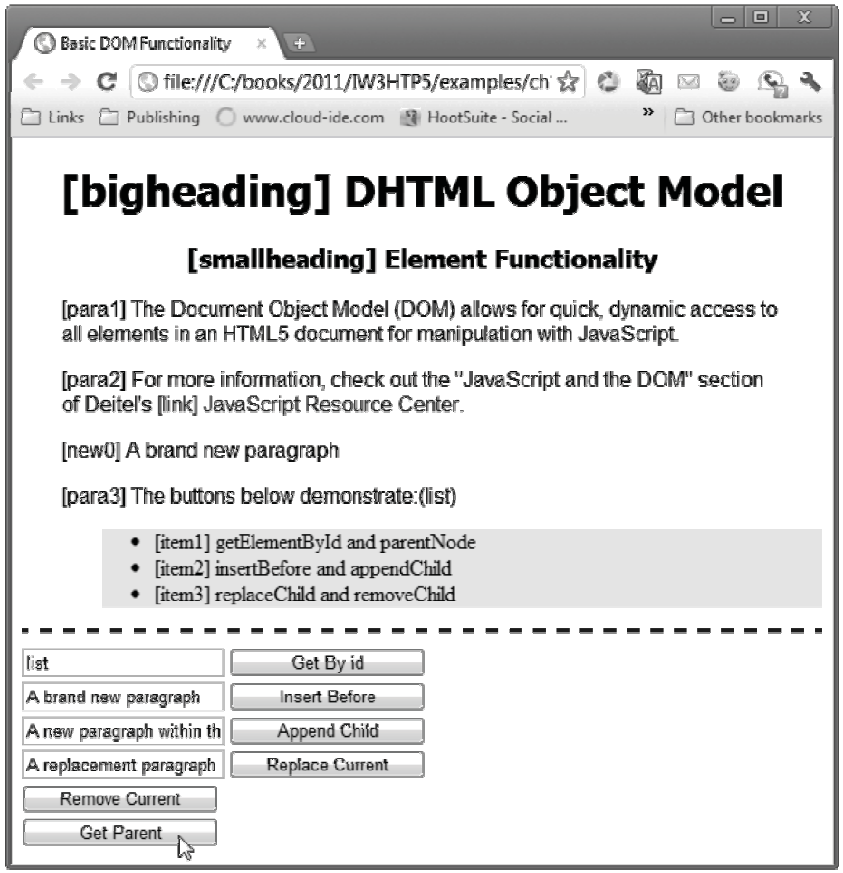


**Fig. 12.11** | The document of Figure 12.4 after using the **Get By id** `button` to `item2`, then using the **Get Parent** `button` to select `item2`'s parent—the unordered list.

# 12.4 DOM Collections

- DOM has collections—groups of related objects on a page
- DOM collections are accessed as properties of DOM objects such as the `document` object or a DOM node
- The `document` object has properties containing the `images` collection, `links` collection, `forms` collection and `anchors` collection
  - Contain all the elements of the corresponding type on the page
- The collection's `length` property specifies the number of items in the collection

# 12.4 DOM Collections (Cont.)

- You access the elements of the collection using indices in square brackets
- `item` method of a DOM collection
  - An alternative to the square bracketed indices
  - Receives an integer argument and returns the corresponding item in the collection.
- `namedItem` method
  - receives an element `id` as an argument and finds the element with that `id` in the collection.
- `href` property of a DOM link node
  - Refers to the link's `href` attribute
- Collections allow easy access to all elements of a single type in a page
  - Useful for gathering elements into one place and for applying changes across an entire page

```
1  /* Fig. 12.12: style.css */
2  /* CSS for collections.html. */
3  body           { font-family: arial, helvetica, sans-serif }
4  h1             { font-family: tahoma, geneva, sans-serif;
5                   text-align: center }
6  p a            { color: DarkRed }
7  ul             { font-size: .9em; }
8  li             { display: inline;
9                   list-style-type: none;
10                  border-right: 1px solid gray;
11                  padding-left: 5px; padding-right: 5px; }
12 li:first-child { padding-left: 0px; }
13 li:last-child  { border-right: none; }
14 a              { text-decoration: none; }
15 a:hover        { text-decoration: underline; }
```

**Fig. 12.12** | CSS for `collections.html`.

```
<!DOCTYPE html>

<!-- Fig. 12.13: collections.html -->
<!-- Using the links collection. -->
<html>
   <head>
      <meta charset="utf-8">
      <title>Using Links Collection</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "collections.js"></script>
   </head>
   <body>
      <h1>Deitel Resource Centers</h1>
      <p><a href = "http://www.deitel.com/">Deitel's website</a>
         contains a growing
         <a href = "http://www.deitel.com/ResourceCenters.html">list
         of Resource Centers</a> on a wide range of topics. Many
         Resource centers related to topics covered in this book,
         <a href = "http://www.deitel.com/books/iw3htp5">Internet &
         World Wide Web How to Program, 5th Edition</a>. We have
         Resource Centers on
         <a href = "http://www.deitel.com/Web2.0">Web 2.0</a>,
         <a href = "http://www.deitel.com/Firefox">Firefox</a> and
         <a href = "http://www.deitel.com/IE9">Internet Explorer 9</a>,
```

**Fig. 12.13** | Using the `links` collection. (Part 1 of 2.)

```
25            <a href = "http://www.deitel.com/HTML5">HTML5</a>, and
26            <a href = "http://www.deitel.com/JavaScript">JavaScript</a>.
27            Watch for related new Resource Centers.</p>
28         <p>Links in this page:</p>
29         <div id = "links"></div>
30      </body>
31   </html>
```

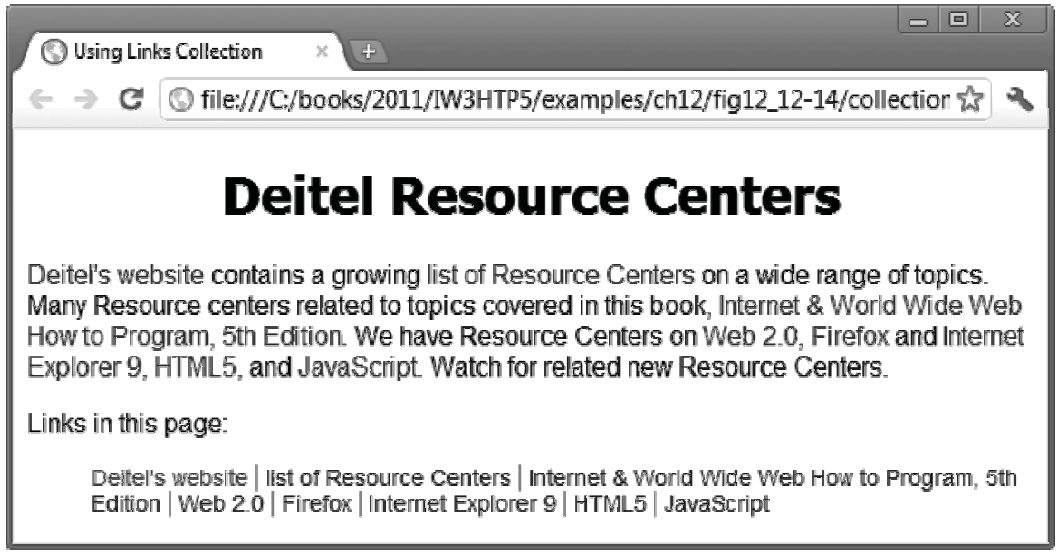


**Fig. 12.13** | Using the `links` collection. (Part 2 of 2.)

```
1  // Fig. 12.14: collections.js
2  // Script to demonstrate using the links collection.
3  function processLinks()
4  {
5     var linksList = document.links; // get the document's links
6     var contents = "<ul>";
7
8     // concatenate each link to contents
9     for ( var i = 0; i < linksList.length; ++i )
10    {
11       var currentLink = linksList[ i ];
12       contents += "<li><a href='" + currentLink.href + "'>" +
13          currentLink.innerHTML + "</li>";
14    } // end for
15
16    contents += "</ul>";
17    document.getElementById( "links" ).innerHTML = contents;
18 } // end function processLinks
19
20 window.addEventListener( "load", processLinks, false );
```

**Fig. 12.14** | Script to demonstrate using the links collection.

# 12.5 Dynamic Styles

- An element's style can be changed dynamically
  - E.g., in response to user events
  - Can create mouse-hover effects, interactive menus and animations
- The document object's body property
  - Refers to the body element
- The `setAttribute` method is used to set the `style` attribute with the user-specified color for the `background-color` CSS property.
- If you have predefined CSS style classes defined for your document, you can also use the `setAttribute` method to set the `class` attribute.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 12.15: dynamicstyle.html -->
4  <!-- Dynamic styles. -->
5  <html>
6     <head>
7        <meta charset="utf-8">
8        <title>Dynamic Styles</title>
9        <script src = "dynamicstyle.js"></script>
10    </head>
11    <body>
12       <p>Welcome to our website!</p>
13    </body>
14 </html>
```


**Fig. 12.15** | Dynamic styles. (Part 1 of 2.)

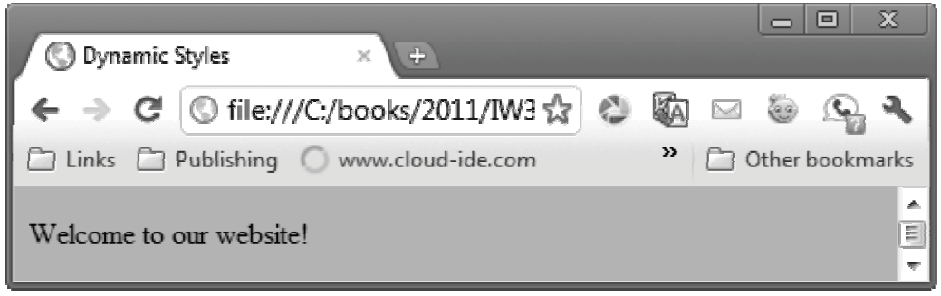


**Fig. 12.15** | Dynamic styles. (Part 2 of 2.)

```
1  // Fig. 12.16: dynamicstyle.js
2  // Script to demonstrate dynamic styles.
3  function start()
4  {
5     var inputColor = prompt( "Enter a color name for the " +
6        "background of this page", "" );
7     document.body.setAttribute( "style",
8        "background-color: " + inputColor );
9  } // end function start
10
11 window.addEventListener( "load", start, false );
```

**Fig. 12.16** | Script to demonstrate dynamic styles.

# 12.6 Using a Timer and Dynamic Styles to Create Animated Effects

- The next example introduces the `window` object's `setInterval` and `clearInterval` methods, combining them with dynamic styles to create animated effects.
- This example is a basic image viewer that allows you to select a book cover and view it in a larger size. When the user clicks a thumbnail image, the larger version grows from the top-left corner of the main image area.

```
/* Fig. 12.17: style.css */
/* CSS for coverviewer.html. */
#thumbs   { width: 192px;
            height: 370px;
            padding: 5px;
            float: left }
#mainimg  { width: 289px;
            padding: 5px;
            float: left }
#imgCover { height: 373px }
img       { border: 1px solid black }
```

**Fig. 12.17** | CSS for coverviewer.html.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 12.18: coverviewer.html -->
4  <!-- Dynamic styles used for animation. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Deitel Book Cover Viewer</title>
9        <link rel = "stylesheet" type = "text/css" href = "style.css">
10       <script src = "coverviewer.js"></script>
11    </head>
```

**Fig. 12.18** | Dynamic styles used for animation. (Part 1 of 6.)

```
12      <body>
13         <div id = "mainimg">
14            <img id = "imgCover" src = "fullsize/jhtp.jpg"
15               alt = "Full cover image">
16         </div>
17         <div id = "thumbs" >
18            <img src = "thumbs/jhtp.jpg" id = "jhtp"
19               alt = "Java How to Program cover">
20            <img src = "thumbs/iw3htp.jpg" id = "iw3htp"
21               alt = "Internet & World Wide Web How to Program cover">
22            <img src = "thumbs/cpphtp.jpg" id = "cpphtp"
23               alt = "C++ How to Program cover">
24            <img src = "thumbs/jhtplov.jpg" id = "jhtplov"
25               alt = "Java How to Program LOV cover">
26            <img src = "thumbs/cpphtplov.jpg" id = "cpphtplov"
27               alt = "C++ How to Program LOV cover">
28            <img src = "thumbs/vcsharphtp.jpg" id = "vcsharphtp"
29               alt = "Visual C# How to Program cover">
30         </div>
31      </body>
32   </html>
```

**Fig. 12.18** | Dynamic styles used for animation. (Part 2 of 6.)

a) The cover viewer page loads with the cover of *Java How to Program, 9/e*


**Fig. 12.18** | Dynamic styles used for animation. (Part 3 of 6.)

b) When the user clicks the thumbnail of *Internet & World Wide Web How to Program, 5/e*, the full-size image begins growing from the top-left corner of the window


**Fig. 12.18** | Dynamic styles used for animation. (Part 4 of 6.)

c) The cover continues to grow


**Fig. 12.18** | Dynamic styles used for animation. (Part 5 of 6.)

d) The animation finishes when the cover reaches its full size


**Fig. 12.18** | Dynamic styles used for animation. (Part 6 of 6.)

```
1  // Fig. 12.19: coverviewer.js
2  // Script to demonstrate dynamic styles used for animation.
3  var interval = null; // keeps track of the interval
4  var speed = 6; // determines the speed of the animation
5  var count = 0; // size of the image during the animation
6
7  // called repeatedly to animate the book cover
8  function run()
9  {
10    count += speed;
11
12     // stop the animation when the image is large enough
13    if ( count >= 375 )
14    {
15       window.clearInterval( interval );
16       interval = null;
17    } // end if
18
19    var bigImage = document.getElementById( "imgCover" );
20    bigImage.setAttribute( "style", "width: " + (0.7656 * count + "px;") +
21       "height: " + (count + "px;") );
22 } // end function run
23
```

**Fig. 12.19** | Script to demonstrate dynamic styles used for animation. (Part 1 of 3.)

```
24  // inserts the proper image into the main image area and
25  // begins the animation
26  function display( imgfile )
27  {
28     if ( interval )
29        return;
30
31     var bigImage = document.getElementById( "imgCover" );
32     bigImage.setAttribute( "style", "width: 0px; height: 0px;" );
33     bigImage.setAttribute( "src", "fullsize/" + imgfile );
34     bigImage.setAttribute( "alt", "Large version of " + imgfile );
35     count = 0; // start the image at size 0
36     interval = window.setInterval( "run()", 10 ); // animate
37  } // end function display
```

**Fig. 12.19** | Script to demonstrate dynamic styles used for animation. (Part 2 of 3.)

```
38
39  // register event handlers
40  function start()
41  {
42     document.getElementById( "jhtp" ).addEventListener(
43        "click", function() { display( "jhtp.jpg" ); }, false );
44     document.getElementById( "iw3htp" ).addEventListener(
45        "click", function() { display( "iw3htp.jpg" ); }, false );
46     document.getElementById( "cpphtp" ).addEventListener(
47        "click", function() { display( "cpphtp.jpg" ); }, false );
48     document.getElementById( "jhtplov" ).addEventListener(
49        "click", function() { display( "jhtplov.jpg" ); }, false );
50     document.getElementById( "cpphtplov" ).addEventListener(
51        "click", function() { display( "cpphtplov.jpg" ); }, false );
52     document.getElementById( "vcsharphtp" ).addEventListener(
53        "click", function() { display( "vcsharphtp.jpg" ); }, false );
54  } // end function start
55
56  window.addEventListener( "load", start, false );
```

**Fig. 12.19** | Script to demonstrate dynamic styles used for animation. (Part 3 of 3.)

# 12.6 Using a Timer and Dynamic Styles to Create Animated Effects (Cont.)

- `setInterval` method of the `window` object
  - Repeatedly executes a statement on a certain interval
  - Takes two parameters
    - A statement to execute repeatedly
    - An integer specifying how often to execute it, in milliseconds
  - Returns a unique identifier to keep track of that particular interval.
- `window` object's `clearInterval` method
  - Stops the repetitive calls of object's `setInterval` method
  - Pass to `clearInterval` the interval identifier that `setInterval` returned
- Anonymous function
  - Defined with no name—it's created in nearly the same way as any other function, but with no identifier after the keyword function.

# Chapter 13
# JavaScript Events

Internet & World Wide Web
How to Program, 5/e

# 13.1 Introduction

- JavaScript events
  - allow scripts to respond to user interactions and modify the page accordingly
- Events and event handling
  - help make web applications more dynamic and interactive

# 13.2 Reviewing the load Event

- The window object's load event fires when the window finishes loading successfully (i.e., all its children are loaded and all external files referenced by the page are loaded)
- *Every* DOM element has a load event, but it's most commonly used on the window object.
- The next example reviews the load event.
- The load event's handler creates an interval timer that updates a span with the number of seconds that have elapsed since the document was loaded. The document's paragraph contains the span.

```
<!DOCTYPE html>

<!-- Fig. 13.1: onload.html -->
<!-- Demonstrating the load event. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>load Event</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "load.js"></script>
   </head>
   <body>
      <p>Seconds you have spent viewing this page so far:
      <span id = "soFar">0</span></p>
   </body>
</html>
```

**Fig. 13.1** | Demonstrating the `window`'s `load` event. (Part 1 of 2.)


**Fig. 13.1** | Demonstrating the window's load event. (Part 2 of 2.)

```
1  // Fig. 13.2: load.js
2  // Script to demonstrate the load event.
3  var seconds = 0;
4
5  // called when the page loads to begin the timer
6  function startTimer()
7  {
8     window.setInterval( "updateTime()", 1000 );
9  } // end function startTimer
10
11 // called every 1000 ms to update the timer
12 function updateTime()
13 {
14    ++seconds;
15    document.getElementById( "soFar" ).innerHTML = seconds;
16 } // end function updateTime
17
18 window.addEventListener( "load", startTimer, false );
```

**Fig. 13.2** | Script that registers window's `load` event handler and handles the event.

# 13.2 Reviewing the load Event (Cont.)

- An **event handler** is a function that responds to an event.
- Assigning an event handler to an event on a DOM node is called **registering an event handler**
- Method `addEventListener` can be called multiple times on a DOM node to register more than one event-handling method for an event.
- It's also possible to remove an event listener by calling **`removeEventListener`** with the same arguments that you passed to `addEventListener` to register the event handler.
- If a script in the head attempts to get a DOM node for an HTML element in the body, `getElementById` returns `null` because the body has not yet loaded

# 13.2 Reviewing the load Event (Cont.)

- Two models for registering event handlers
  - Inline model treats events as attributes of HTML elements
  - Traditional model assigns the name of the function to the event property of a DOM node
- The inline model places calls to JavaScript functions directly in HTML code.
- The following code indicates that JavaScript function `start` should be called when the body element loads:

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

- The traditional model uses a property of an object to specify an event handler.
- The following JavaScript code indicates that function start should be called when document loads:

```
document.onload = "start()";
```

# 13.3 Event mouseMove and the event Object

- mousemove event occurs whenever the user moves the mouse over the web page
- The next example creates a simple drawing program that allows the user to draw inside a table element in red or blue by holding down the *Shift* key or *Ctrl* key and moving the mouse over the box.
  - ctrlKey property contains a boolean which reflects whether the *Ctrl* key was pressed during the event
  - shiftKey property reflects whether the *Shift* key was pressed during the event

```
<!DOCTYPE html>

<!-- Fig. 13.3: draw.html -->
<!-- A simple drawing program. -->
<html>
   <head>
      <meta charset="utf-8">
      <title>Simple Drawing Program</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "draw.js"></script>
   </head>
   <body>
      <table id = "canvas">
         <caption>Hold <em>Ctrl</em> (or <em>Control</em>) to draw blue.
            Hold <em>Shift</em> to draw red.</caption>
         <tbody id = "tablebody"></tbody>
      </table>
   </body>
</html>
```

**Fig. 13.3** | Simple drawing program. (Part 1 of 3.)

a) User holds the *Shift* key and moves the mouse to draw in red.


**Fig. 13.3** | Simple drawing program. (Part 2 of 3.)

b) User holds the *Ctrl* key and moves the mouse to draw in blue.

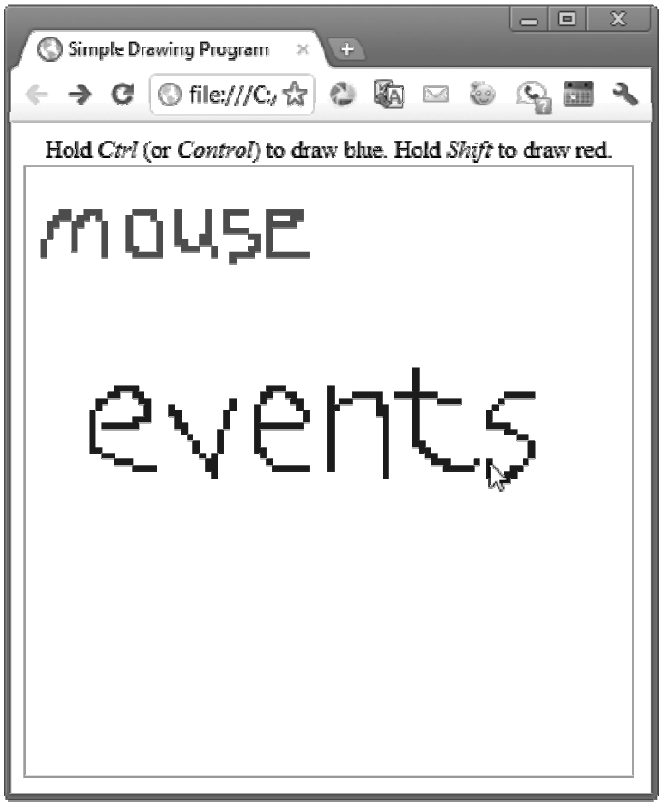


**Fig. 13.3** | Simple drawing program. (Part 3 of 3.)

```
1  // Fig. 13.4: draw.js
2  // A simple drawing program.
3  // initialization function to insert cells into the table
4  function createCanvas()
5  {
6     var side = 100;
7     var tbody = document.getElementById( "tablebody" );
8
9     for ( var i = 0; i < side; ++i )
10    {
11       var row = document.createElement( "tr" );
12
13       for ( var j = 0; j < side; ++j )
14       {
15          var cell = document.createElement( "td" );
16          row.appendChild( cell );
17       } // end for
18
19       tbody.appendChild( row );
20    } // end for
21
```

**Fig. 13.4** | JavaScript code for the simple drawing program. (Part 1 of 2.)

```
22      // register mousemove listener for the table
23      document.getElementById( "canvas" ).addEventListener(
24         "mousemove", processMouseMove, false );
25   } // end function createCanvas
26
27   // processes the onmousemove event
28   function processMouseMove( e )
29   {
30      if ( e.target.tagName.toLowerCase() == "td" )
31      {
32         // turn the cell blue if the Ctrl key is pressed
33         if ( e.ctrlKey )
34         {
35            e.target.setAttribute( "class", "blue" );
36         } // end if
37
38         // turn the cell red if the Shift key is pressed
39         if ( e.shiftKey )
40         {
41            e.target.setAttribute( "class", "red" );
42         } // end if
43      } // end if
44   } // end function processMouseMove
45
46   window.addEventListener( "load", createCanvas, false );
```

**Fig. 13.4** | JavaScript code for the simple drawing program. (Part 2 of 2.)

| Property | Description |
| --- | --- |
| `altKey` | This value is `true` if the *Alt* key was pressed when the event fired. |
| `cancelBubble` | Set to `true` to prevent the event from bubbling. Defaults to `false`. (See Section 13.7, Event Bubbling.) |
| `clientX` and `clientY` | The coordinates of the mouse cursor inside the client area (i.e., the active area where the web page is displayed, excluding scrollbars, navigation buttons, etc.). |
| `ctrlKey` | This value is `true` if the *Ctrl* key was pressed when the event fired. |
| `keyCode` | The ASCII code of the key pressed in a keyboard event. See Appendix D for more information on the ASCII character set. |
| `screenX` and `screenY` | The coordinates of the mouse cursor on the screen coordinate system. |
| `shiftKey` | This value is `true` if the *Shift* key was pressed when the event fired. |
| `target` | The DOM object that received the event. |
| `type` | The name of the event that fired. |

**Fig. 13.5** | Some event-object properties.

# 13.4 Rollovers with `mouseover` and `mouseout`

- When the mouse cursor enters an element, an `mouseover` event occurs for that element
- When the mouse cursor leaves the element, a `mouseout` event occurs for that element
- Creating an `Image` object and setting its `src` property preloads the image

```
<!DOCTYPE html>

<!-- Fig 13.6: mouseoverout.html -->
<!-- Events mouseover and mouseout. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Events mouseover and mouseout</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "mouseoverout.js"></script>
   </head>
   <body>
      <h1><img src = "heading1.png" id = "heading"
         alt = "Heading Image"></h1>
      <p>Can you tell a color from its hexadecimal RGB code
      value? Look at the hex code, guess its color. To see
      what color it corresponds to, move the mouse over the
      hex code. Moving the mouse out of the hex code's table
      cell will display the color name.</p>
```

**Fig. 13.6** | HTML5 document to demonstrate mouseover and mouseout. (Part 1 of 6.)

```
20          <div>
21             <ul>
22                <li id = "Black">#000000</li>
23                <li id = "Blue">#0000FF</li>
24                <li id = "Magenta">#FF00FF</li>
25                <li id = "Gray">#808080</li>
26                <li id = "Green">#008000</li>
27                <li id = "Lime">#00FF00</li>
28                <li id = "Maroon">#800000</li>
29                <li id = "Navy">#000080</li>
30                <li id = "Olive">#808000</li>
31                <li id = "Purple">#800080</li>
32                <li id = "Red">#FF0000</li>
33                <li id = "Silver">#C0C0C0</li>
34                <li id = "Cyan">#00FFFF</li>
35                <li id = "Teal">#008080</li>
36                <li id = "Yellow">#FFFF00</li>
37                <li id = "White">#FFFFFF</li>
38             </ul>
39          </div>
40       </body>
41    </html>
```

**Fig. 13.6** | HTML5 document to demonstrate `mouseover` and `mouseout`. (Part 2 of 6.)

a) The page loads with the blue heading image and all the hex codes in black.

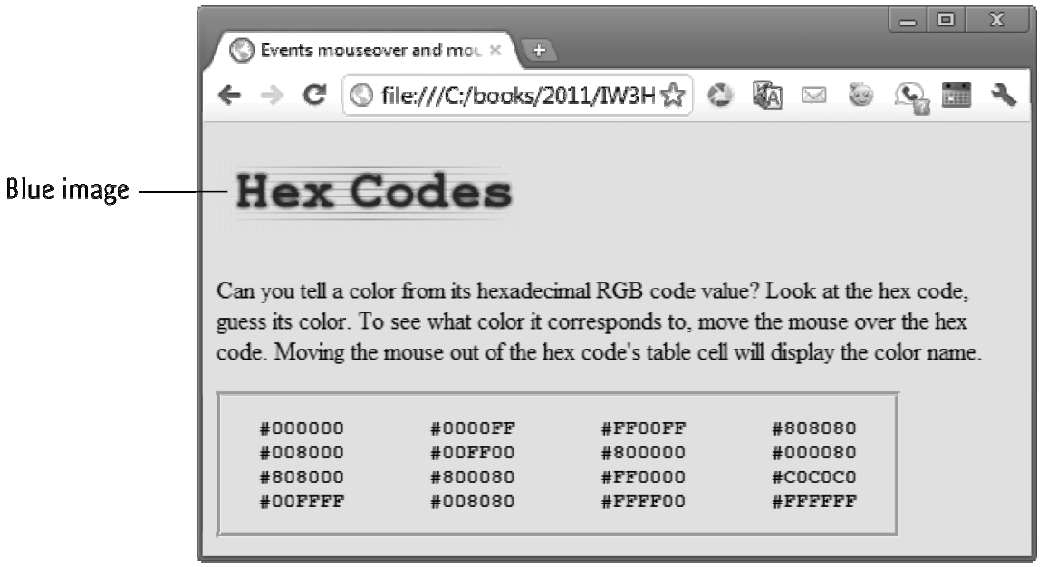


**Fig. 13.6** | HTML5 document to demonstrate `mouseover` and `mouseout`. (Part 3 of 6.)

b) The heading image switches to an image with green text when the mouse rolls over it.


**Fig. 13.6** | HTML5 document to demonstrate `mouseover` and `mouseout`. (Part 4 of 6.)

c) When mouse rolls over a hex code, the text color changes to the color represented by the hex code. Notice that the heading image has become blue again because the mouse is no longer over it.


**Fig. 13.6** | HTML5 document to demonstrate `mouseover` and `mouseout`. (Part 5 of 6.)

d) When the mouse leaves the hex code's table cell, the text changes to the name of the color.

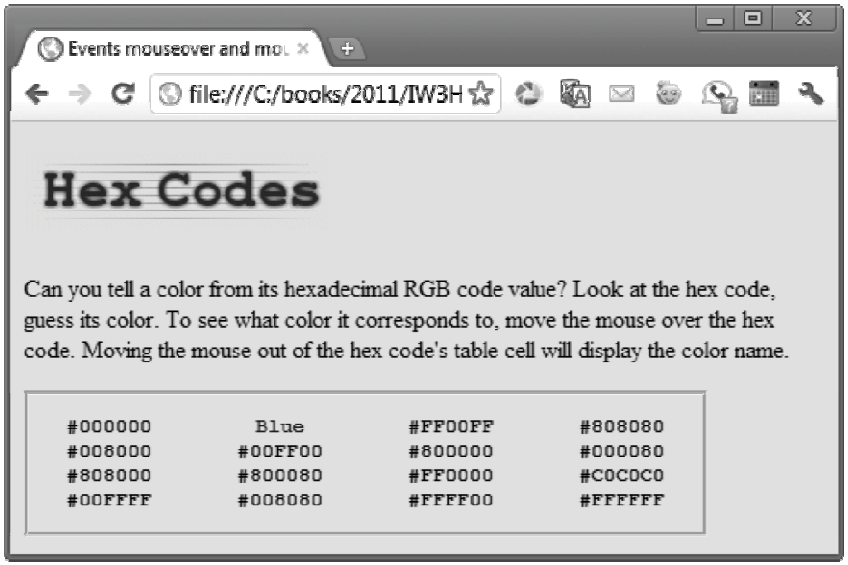


**Fig. 13.6** | HTML5 document to demonstrate `mouseover` and `mouseout`. (Part 6 of 6.)

```
1  // Fig 13.7: mouseoverout.js
2  // Events mouseover and mouseout.
3  image1 = new Image();
4  image1.src = "heading1.png";
5  image2 = new Image();
6  image2.src = "heading2.png";
7
8  function mouseOver( e )
9  {
10    // swap the image when the mouse moves over it
11    if ( e.target.getAttribute( "id" ) == "heading" )
12    {
13       e.target.setAttribute( "src", image2.getAttribute( "src" ) );
14    } // end if
15
16    // if the element is an li, assign its id to its color
17    // to change the hex code's text to the corresponding color
18    if ( e.target.tagName.toLowerCase() == "li"  )
19    {
20       e.target.setAttribute( "style",
21          "color: " + e.target.getAttribute( "id" ) );
22    } // end if
23 } // end function mouseOver
```

**Fig. 13.7** | Processing the mouseover and mouseout events. (Part 1 of 2.)

```
24
25  function mouseOut( e )
26  {
27     // put the original image back when the mouse moves away
28     if ( e.target.getAttribute( "id" ) == "heading" )
29     {
30        e.target.setAttribute( "src", image1.getAttribute( "src" ) );
31     } // end if
32
33     // if the element is an li, assign its id to innerHTML
34     // to display the color name
35     if ( e.target.tagName.toLowerCase() == "li"  )
36     {
37        e.target.innerHTML = e.target.getAttribute( "id" );
38     } // end if
39  } // end function mouseOut
40
41  document.addEventListener( "mouseover", mouseOver, false );
42  document.addEventListener( "mouseout", mouseOut, false );
```

**Fig. 13.7** | Processing the mouseover and mouseout events. (Part 2 of 2.)

# 13.5 Form Processing with `focus` and `blur`

- `focus` event fires when an element gains focus
  - i.e., when the user clicks a form field or uses the *Tab* key to move between form elements
- `blur` fires when an element loses focus
  - i.e., when another control gains the focus

```
<!DOCTYPE html>

<!-- Fig. 13.8: focusblur.html -->
<!-- Demonstrating the focus and blur events. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>A Form Using focus and blur</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "focusblur.js"></script>
   </head>
   <body>
      <form id = "myForm" action = "">
         <p><label class = "fixed" for = "name">Name:</label>
            <input type = "text" id =  "name"
               placeholder = "Enter name"></p>
         <p><label class = "fixed" for = "email">E-mail:</label>
            <input type = "email" id = "email"
               placeholder = "Enter e-mail address"></p>
         <p><label>Click here if you like this site
            <input type = "checkbox" id =  "like"></label></p>
         <p><label for = "comments">Any comments?</p>
         <textarea id =  "comments"
            placeholder = "Enter comments here"></textarea>
```

**Fig. 13.8** | Demonstrating the focus and blur events. (Part 1 of 3.)

```
25            <p><input id = "submit" type = "submit">
26               <input id = "reset" type = "reset"></p>
27         </form>
28         <p id = "helpText"></p>
29      </body>
30   </html>
```

a) The blue message at the bottom of the page instructs the user to enter a name when the **Name:** field has the focus.

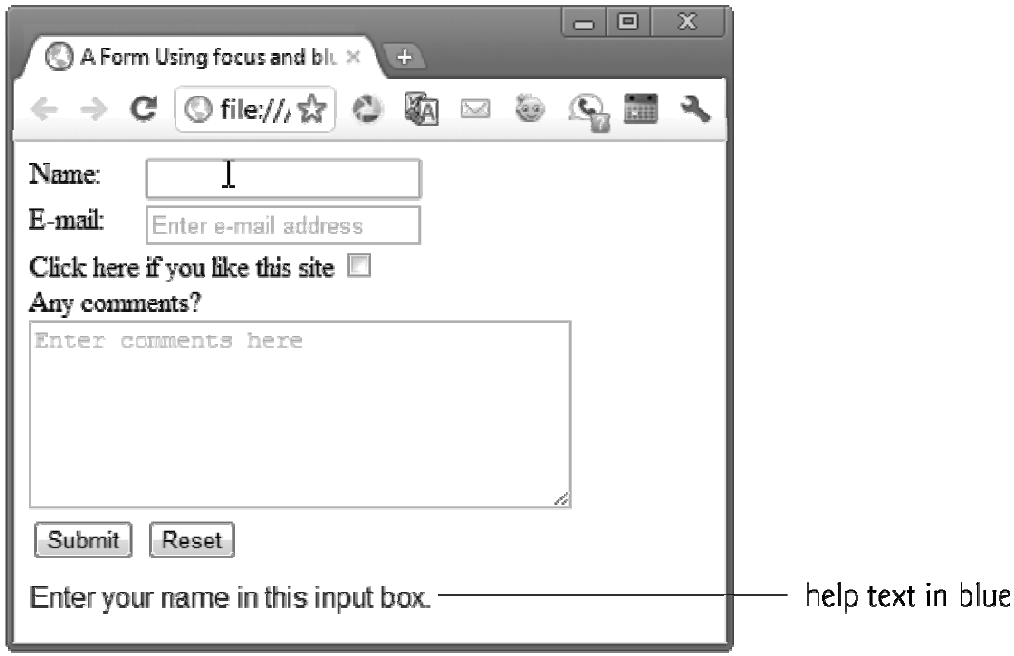


**Fig. 13.8** | Demonstrating the `focus` and `blur` events. (Part 2 of 3.)

b) The message changes depending on which field has focus—this window shows the help text for the `comments` `textarea`.

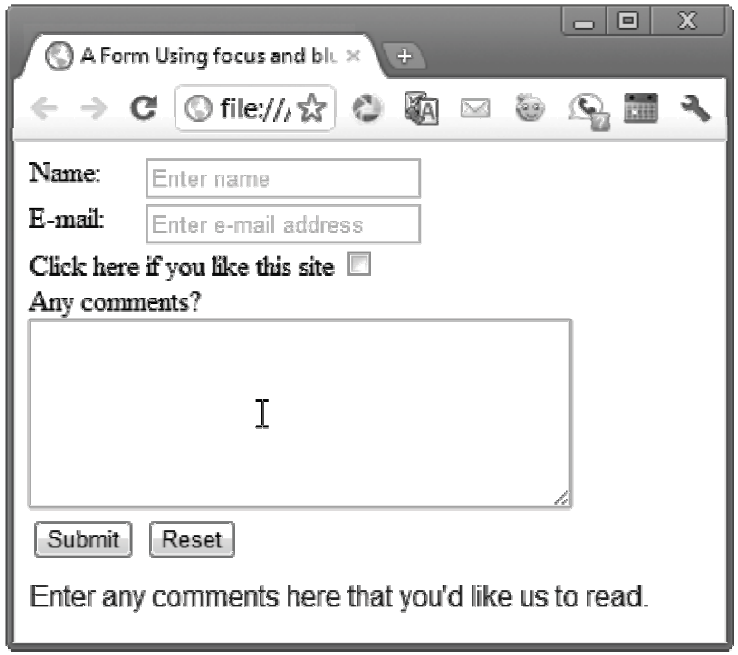


**Fig. 13.8** | Demonstrating the `focus` and `blur` events. (Part 3 of 3.)

```
1  // Fig. 13.9: focusblur.js
2  // Demonstrating the focus and blur events.
3  var helpArray = [ "Enter your name in this input box.",
4    "Enter your e-mail address in the format user@domain.",
5    "Check this box if you liked our site.",
6    "Enter any comments here that you'd like us to read.",
7    "This button submits the form to the server-side script.",
8    "This button clears the form.", "" ];
9  var helpText;
10
11 // initialize helpTextDiv and register event handlers
12 function init()
13 {
14    helpText = document.getElementById( "helpText" );
15
16    // register listeners
17    registerListeners( document.getElementById( "name" ), 0 );
18    registerListeners( document.getElementById( "email" ), 1 );
19    registerListeners( document.getElementById( "like" ), 2 );
20    registerListeners( document.getElementById( "comments" ), 3 );
21    registerListeners( document.getElementById( "submit" ), 4 );
22    registerListeners( document.getElementById( "reset" ), 5 );
23 } // end function init
24
```

**Fig. 13.9** | Demonstrating the `focus` and `blur` events. (Part 1 of 2.)

```
25  // utility function to help register events
26  function registerListeners( object, messageNumber )
27  {
28     object.addEventListener( "focus",
29        function() { helpText.innerHTML = helpArray[ messageNumber ]; },
30        false );
31     object.addEventListener( "blur",
32        function() { helpText.innerHTML = helpArray[ 6 ]; }, false );
33  } // end function registerListener
34
35  window.addEventListener( "load", init, false );
```

**Fig. 13.9** | Demonstrating the `focus` and `blur` events. (Part 2 of 2.)

# 13.6 More Form Processing with `submit` and `reset`

- `submit` and `reset` events fire when a `form` is submitted or reset, respectively
- The anonymous function executes in response to the user's submitting the form by clicking the **Submit** button or pressing the *Enter* key.
- `confirm` method asks the users a question, presenting them with an OK button and a Cancel button
  - If the user clicks OK, `confirm` returns `true`; otherwise, `confirm` returns `false`
- By returning either `true` or `false`, event handlers dictate whether the default action for the event is taken
- If an event handler returns `true` or does not return a value, the default action is taken once the event handler finishes executing

```
1  // Fig. 13.10: focusblur.js
2  // Demonstrating the focus and blur events.
3  var helpArray = [ "Enter your name in this input box.",
4    "Enter your e-mail address in the format user@domain.",
5    "Check this box if you liked our site.",
6    "Enter any comments here that you'd like us to read.",
7    "This button submits the form to the server-side script.",
8    "This button clears the form.", "" ];
9  var helpText;
10
11 // initialize helpTextDiv and register event handlers
12 function init()
13 {
14    helpText = document.getElementById( "helpText" );
15
16    // register listeners
17    registerListeners( document.getElementById( "name" ), 0 );
18    registerListeners( document.getElementById( "email" ), 1 );
19    registerListeners( document.getElementById( "like" ), 2 );
20    registerListeners( document.getElementById( "comments" ), 3 );
21    registerListeners( document.getElementById( "submit" ), 4 );
22    registerListeners( document.getElementById( "reset" ), 5 );
23
24    var myForm = document.getElementById( "myForm" );
```

**Fig. 13.10** | Demonstrating the focus and blur events. (Part 1 of 3.)

```
25      myForm.addEventListener( "submit",
26         function()
27         {
28            return confirm( "Are you sure you want to submit?" );
29         }, // end anonymous function
30         false );
31      myForm.addEventListener( "reset",
32         function()
33         {
34            return confirm( "Are you sure you want to reset?" );
35         }, // end anonymous function
36         false );
37   } // end function init
38
39   // utility function to help register events
40   function registerListeners( object, messageNumber )
41   {
42      object.addEventListener( "focus",
43         function() { helpText.innerHTML = helpArray[ messageNumber ]; },
44         false );
45      object.addEventListener( "blur",
46         function() { helpText.innerHTML = helpArray[ 6 ]; }, false );
47   } // end function registerListener
48
49   window.addEventListener( "load", init, false );
```

**Fig. 13.10** | Demonstrating the `focus` and `blur` events. (Part 2 of 3.)

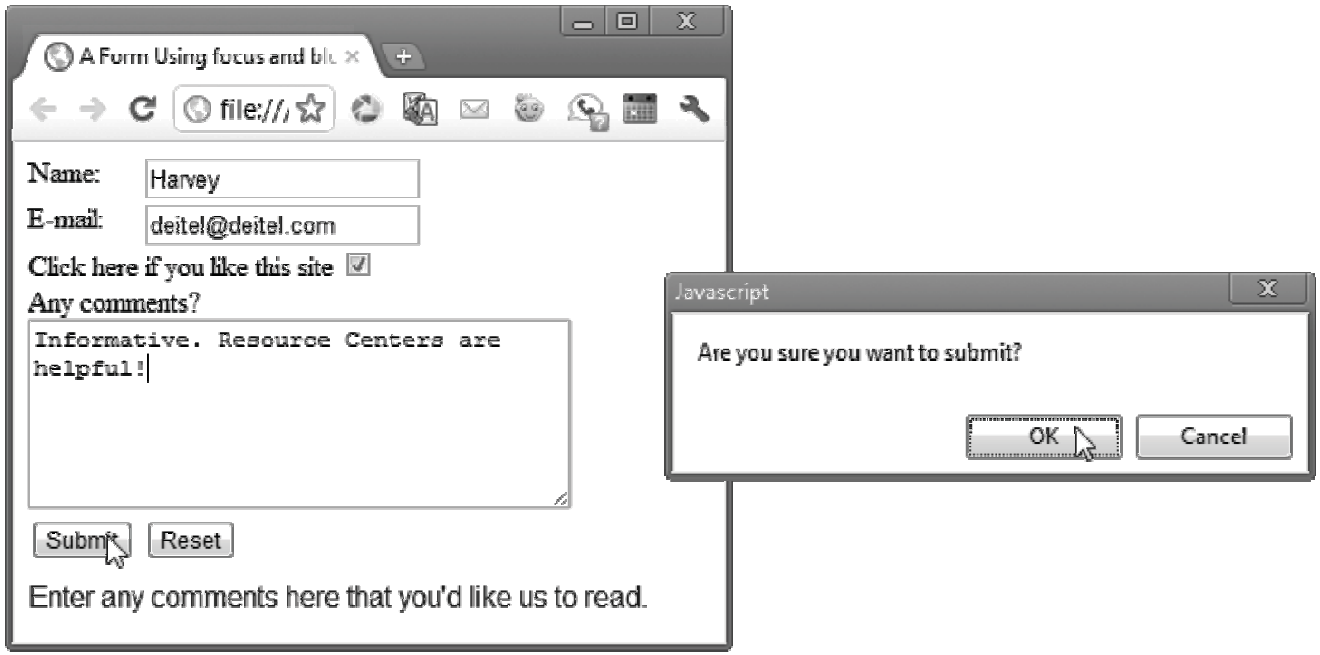


**Fig. 13.10** | Demonstrating the `focus` and `blur` events. (Part 3 of 3.)

# 13.7 Event Bubbling

- Event bubbling
  - The process whereby events fired on *child* elements "bubble" up to their *parent* elements
  - When an event is fired on an element, it is first delivered to the element's event handler (if any), then to the parent element's event handler (if any)
- *If you intend to handle an event in a child element alone, you should cancel the bubbling of the event in the child element's event-handling code by using the `cancelBubble` property of the event object*

```
<!DOCTYPE html>

<!-- Fig. 13.11: bubbling.html -->
<!-- Canceling event bubbling. -->
<html>
   <head>
      <meta charset="utf-8">
      <title>Event Bubbling</title>
      <script src = "bubbling.js">
   </head>
   <body>
      <p id = "bubble">Bubbling enabled.</p>
      <p id = "noBubble">Bubbling disabled.</p>
   </body>
</html>
```

**Fig. 13.11** | Canceling event bubbling. (Part 1 of 3.)

a) User clicks the first paragraph, for which bubbling is enabled.


b) Paragraph's event handler causes an alert.


c) Document's event handler causes another alert, because the event bubbles up to the document.

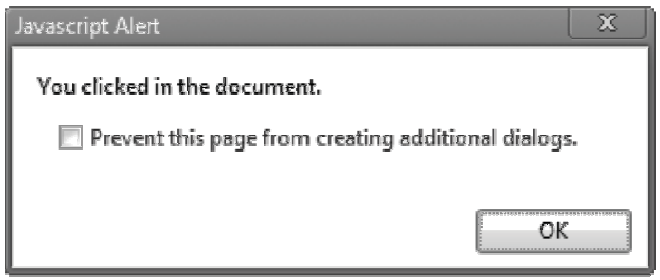


**Fig. 13.11** | Canceling event bubbling. (Part 2 of 3.)

d) User clicks the second paragraph, for which bubbling is disabled.


e) Paragraph's event handler causes an alert. The document's event handler is not called.


**Fig. 13.11** | Canceling event bubbling. (Part 3 of 3.)

```
1  // Fig. 13.12: bubbling.js
2  // Canceling event bubbling.
3  function documentClick()
4  {
5     alert( "You clicked in the document." );
6  } // end function documentClick
7
8  function bubble( e )
9  {
10    alert( "This will bubble." );
11    e.cancelBubble = false;
12 } // end function bubble
13
14 function noBubble( e )
15 {
16    alert( "This will not bubble." );
17    e.cancelBubble = true;
18 } // end function noBubble
19
```

**Fig. 13.12** | Canceling event bubbling. (Part 1 of 2.)

```
20  function registerEvents()
21  {
22     document.addEventListener( "click", documentClick, false );
23     document.getElementById( "bubble" ).addEventListener(
24        "click", bubble, false );
25     document.getElementById( "noBubble" ).addEventListener(
26        "click", noBubble, false );
27  } // end function registerEvents
28
29  window.addEventListener( "load", registerEvents, false );
```

**Fig. 13.12** | Canceling event bubbling. (Part 2 of 2.)

# 13.8 More Events

- The following slide lists some common events and their descriptions. The actual DOM event names begin with "on", but we show the names you use with `addEventListener` here.

| Event | Description |
| --- | --- |
| `abort` | Fires when image transfer has been interrupted by user. |
| `change` | Fires when a new choice is made in a `select` element, or when a text input is changed and the element loses focus. |
| `click` | Fires when the user clicks the mouse. |
| `dblclick` | Fires when the user double clicks the mouse. |
| `focus` | Fires when a form element gets the focus. |
| `keydown` | Fires when the user pushes down a key. |
| `keypress` | Fires when the user presses then releases a key. |
| `keyup` | Fires when the user releases a key. |
| `load` | Fires when an element and all its children have loaded. |
| `mousedown` | Fires when a mouse button is pressed. |
| `mousemove` | Fires when the mouse moves. |
| `mouseout` | Fires when the mouse leaves an element. |
| `mouseover` | Fires when the mouse enters an element. |

**Fig. 13.13** | Common events. (Part 1 of 2.)

| Event | Description |
| --- | --- |
| `mouseup` | Fires when a mouse button is released. |
| `reset` | Fires when a form resets (i.e., the user clicks a reset button). |
| `resize` | Fires when the size of an object changes (i.e., the user resizes a window or frame). |
| `select` | Fires when a text selection begins (applies to `input` or `textarea`). |
| `submit` | Fires when a form is submitted. |
| `unload` | Fires when a page is about to unload. |

**Fig. 13.13** | Common events. (Part 2 of 2.)