12. Document Object Model (DOM): Objects and Collections

Though leaves are many, the root is one.

—William Butler Yeats

Most of us become parents long before we have stopped being children.

—Mignon McLaughlin

Sibling rivalry is inevitable. The only sure way to avoid it is to have one child.

—Nancy Samalin

Objectives

In this chapter you will:

- Use JavaScript and the W3C Document Object Model to create dynamic web pages.
- Learn the concept of DOM nodes and DOM trees.
- Traverse, edit and modify elements in an HTML5 document.
- Change CSS styles dynamically.
- Create JavaScript animations.

Outline

12.1. Introduction

In this chapter we introduce the **Document Object Model (DOM)**. The DOM gives you scripting access to *all* the elements on a web page. Inside the browser, the whole web page—paragraphs, forms, tables, etc.—is represented in an **object hierarchy**. Using JavaScript, you can dynamically create, modify and remove elements in the page.

We introduce the concepts of DOM nodes and DOM trees. We discuss properties and methods of DOM nodes and cover additional methods of the `document` object. We show how to dynamically change style properties, which enables you to create effects, such as user-defined background colors and animations.

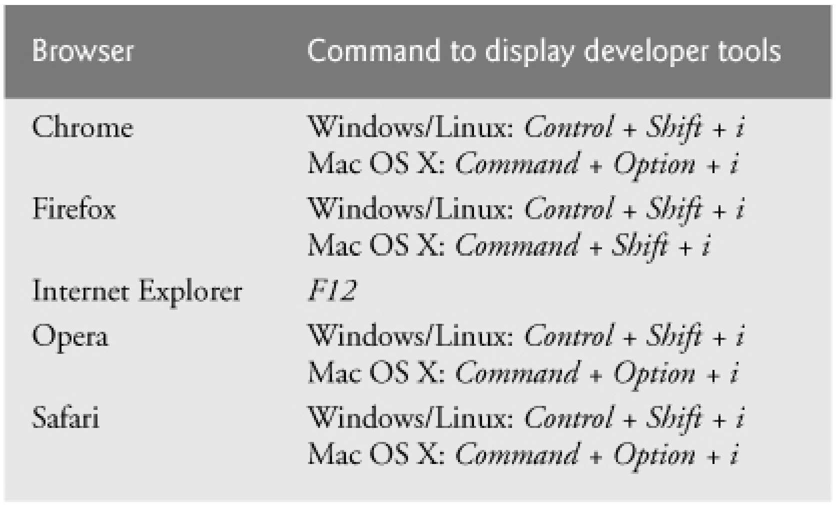
SOFTWARE ENGINEERING OBSERVATION 12.1

With the DOM, HTML5 elements can be treated as objects, and many attributes of HTML5 elements can be treated as properties of those objects. Then objects can be scripted with JavaScript to achieve dynamic effects.

12.2. Modeling a Document: DOM Nodes and Trees

As we saw in previous chapters, the `document`’s `getElementById` method is the simplest way to access a specific element in a page. The 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. All 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. An HTML5 element *inside* another element is said to be its **child**—the containing element is known as the **parent**. A node can have multiple children but only one parent. Nodes with the same parent node are referred to as **siblings**.

Today's desktop browsers provide developer tools that can display a visual representation of a document's DOM tree. Figure 12.1 shows how to access the developer tools for each of the desktop browsers we use for testing web apps in this book. For the most part, the developer tools are similar across the browsers. [*Note:* For Firefox, you must first install the DOM Inspector add-on from https://addons.mozilla.org/en-US/firefox/addon/dom-inspector-6622/. Other developer tools are available in the Firefox menu's **Web Developer** menu item, and more Firefox web-developer add-ons are available from https://addons.mozilla.org/en-US/firefox/collections/mozilla/webdeveloper/.]



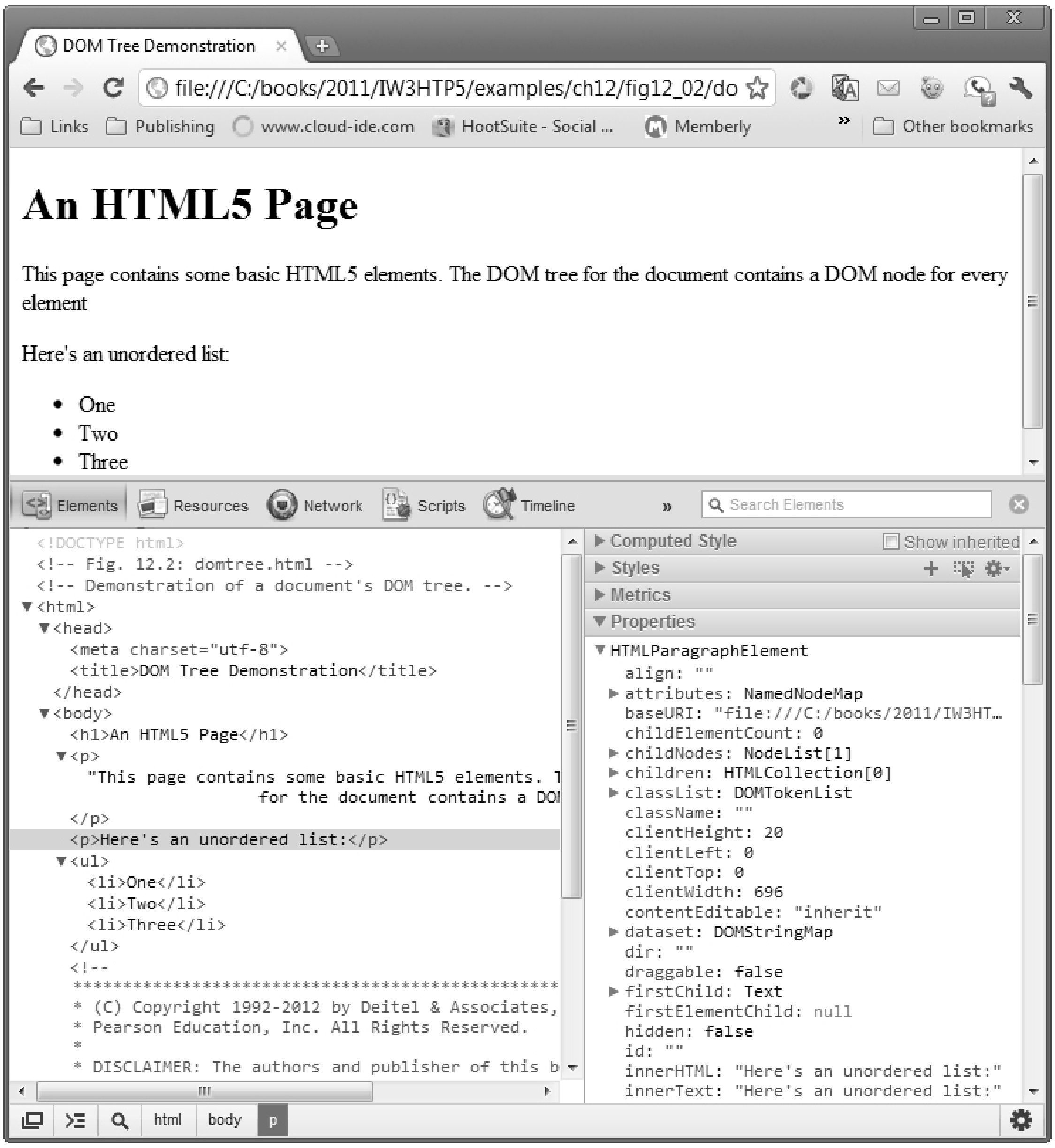
Browser	Command to display developer tools
Chrome	Windows/Linux: *Control + Shift + i* Mac OS X: *Command + Option + i*
Firefox	Windows/Linux: *Control + Shift + i* Mac OS X: *Command + Shift + i*
Internet Explorer	*F12*
Opera	Windows/Linux: *Control + Shift + i* Mac OS X: *Command + Option + i*
Safari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

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

Viewing a Document's DOM

Figure 12.2 shows an HTML5 document in the Chrome web browser. At the bottom of the window, the document's DOM tree is displayed in the **Elements** tab of the Chrome developer tools. The HTML5 document contains a few simple elements. A node can be expanded and collapsed using the ▶ and ▼ arrows next to a given node. Figure 12.2 shows all the nodes in the document fully expanded. The **html** node at the top of the tree is called the **root node**, because it has no parent. Below the **html** node, the **head** node is indented to signify that the **head** node is a child of the **html** node. The **html** node represents the `html` element (lines 5–21).

```
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>
```



An HTML5 Page

This page contains some basic HTML5 elements. The DOM tree for the document contains a DOM node for every element

Here's an unordered list:

- One
- Two
- Three

```
<!DOCTYPE html>
<!-- Fig. 12.2: domtree.html -->
<!-- Demonstration of a document's DOM tree. -->
▼<html>
  ▼<head>
      <meta charset="utf-8">
      <title>DOM Tree Demonstration</title>
    </head>
  ▼<body>
      <h1>An HTML5 Page</h1>
    ▼<p>
        "This page contains some basic HTML5 elements. T
                 for the document contains a DOI
      </p>
      <p>Here's an unordered list:</p>
    ▼<ul>
        <li>One</li>
        <li>Two</li>
        <li>Three</li>
      </ul>
      <!--
      ****************************************************
      * (C) Copyright 1992-2012 by Deitel & Associates,
      * Pearson Education, Inc. All Rights Reserved.
      *
      * DISCLAIMER: The authors and publisher of this b
```

```
▼HTMLParagraphElement
   align: ""
 ▶attributes: NamedNodeMap
   baseURI: "file:///C:/books/2011/IW3HT…
   childElementCount: 0
 ▶childNodes: NodeList[1]
 ▶children: HTMLCollection[0]
 ▶classList: DOMTokenList
   className: ""
   clientHeight: 20
   clientLeft: 0
   clientTop: 0
   clientWidth: 696
   contentEditable: "inherit"
 ▶dataset: DOMStringMap
   dir: ""
   draggable: false
 ▶firstChild: Text
   firstElementChild: null
   hidden: false
   id: ""
   innerHTML: "Here's an unordered list:"
   innerText: "Here's an unordered list:"
```

Fig. 12.2. Demonstration of a document's DOM tree.

The **head** and **body** nodes are siblings, since they're both children of the **html** node. The **head** contains the **meta** and **title** nodes. The **body** node contains nodes representing each of the elements in the document's `body` element. The **li** nodes are children of the **ul** node, since they're nested inside it.

When you select a node in the left side of the developer tools **Elements** tab, the node's details are displayed in the right side. In Fig. 12.2, we selected the **p** node just before the unordered list. In the **Properties** section, you can see values for that node's many properties, including the `innerHTML` property that we've used in many examples.

In addition to viewing a document's DOM structure, the developer tools in each browser typically enable you to view and modify styles, view and debug JavaScripts used in the document, view the resources (such as images) used by the document, and more. See each browser's developer tools documentation online for more detailed information.

12.3. Traversing and Modifying a DOM Tree

The DOM enables you to programmatically access a document's elements, allowing you to modify its contents dynamically using JavaScript. This section introduces some of the DOM-node properties and methods for traversing the DOM tree, modifying nodes and creating or deleting content dynamically.

The example in Figs. 12.3–12.5 demonstrates several DOM node features and two additional `document` -object methods. It allows you to highlight, modify, insert and remove elements.

CSS

Figure 12.3 contains the CSS for the example. The CSS class highlighted (line 14) is applied dynamically to elements in the document as we add, remove and select elements using the `form` in Fig. 12.4.

```
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; }
```

```
.highlighted { background-color: yellow; }
input        { width: 150px; }
form > p     { margin: 0px; }
```

Fig. 12.3. CSS for basic DOM functionality example.

HTML5 Document

Figure 12.4 contains the HTML5 document that we'll manipulate dynamically by modifying its DOM. Each element in this example has an `id` attribute, which we also display at the beginning of the element in square brackets. For example, the `id` of the `h1` element in lines 13–14 is set to `bigheading`, and the heading text begins with `[bigheading]`. This allows you to see the `id` of each element in the page. The `body` also contains an `h3` heading, several `p` elements, and an unordered list. A `div` element (lines 29–48) contains the remainder of the document. Line 30 begins a `form`. Lines 32–46 contain the controls for modifying and manipulating the elements on the page. The `click` event handlers (registered in Fig. 12.5) for the six `button`s call corresponding functions to perform the actions described by the `button`s' `value`s.

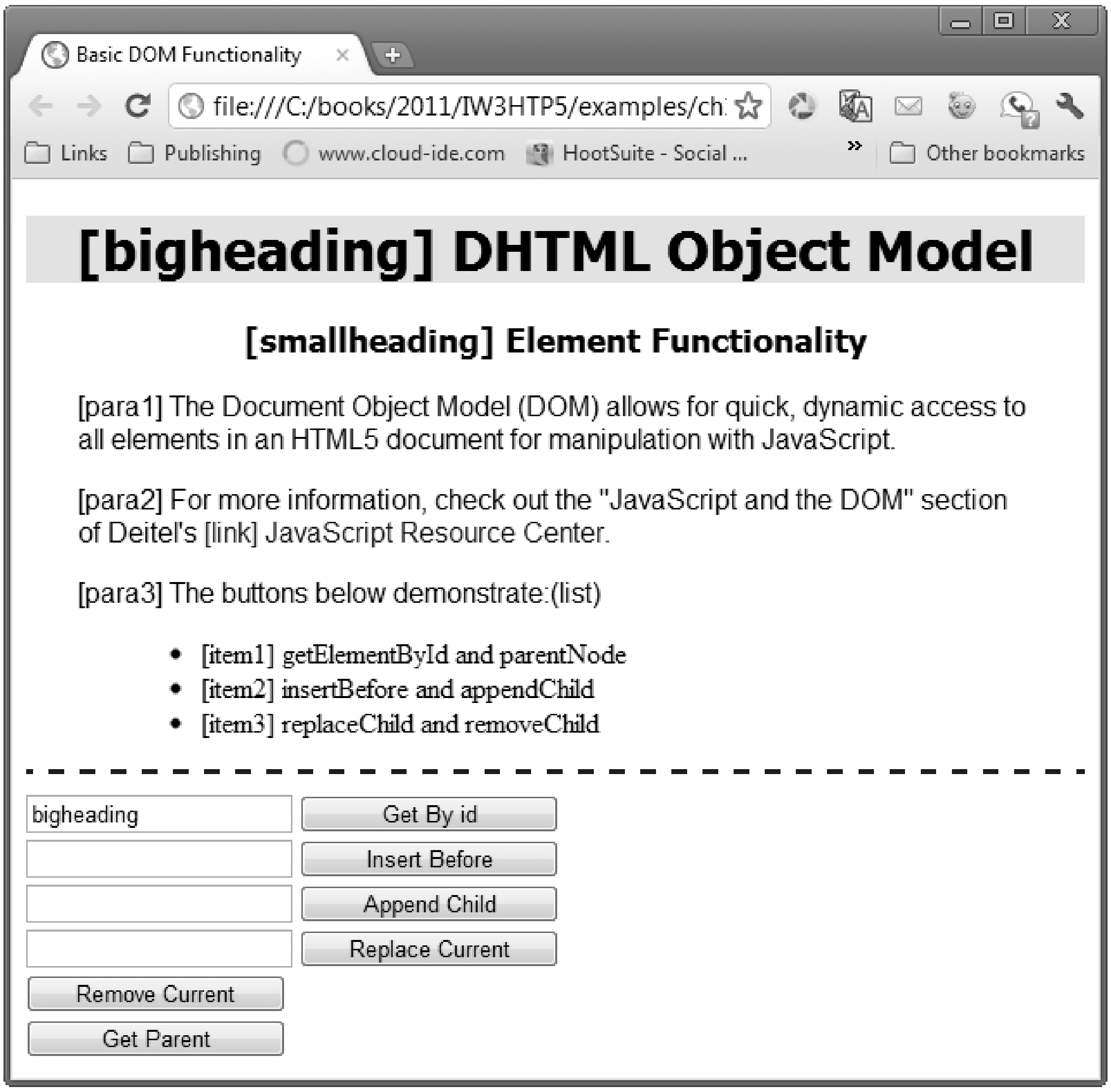
JavaScript

The JavaScript code (Fig. 12.5) begins by declaring two variables. Variable `currentNode` (line 3) keeps track of the currently highlighted node—the functionality of each `button` depends on which node is currently selected. Function `start` (lines 7–24) registers the evvent handlers for the document's `button`s, then initializes `currentNode` to the `h1` element with `id` `bigheading`. This function is set up to be called when the `window`'s `load` event (line 27) occurs. Variable `idcount` (line 4) 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. We now discuss each `button` and its corresponding event handler in detail.

```
<!DOCTYPE html>

<!-- Fig. 12.4: dom.html -->
<!-- Basic DOM functionality. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Basic DOM Functionality</title>
    <link rel = "stylesheet" type = "text/css" href = "style.css">
     <script src = "dom.js"></script>
   </head>
   <body>
     <h1 id = "bigheading" class = "highlighted">
       [bigheading] DHTML Object Model</h1>
     <h3 id = "smallheading">[smallheading] Element Functionality</h3>
     <p id = "para1">[para1] The Document Object Model (DOM) allows for
       quick, dynamic access to all elements in an HTML5 document for
       manipulation with JavaScript.</p>
     <p id = "para2">[para2] For more information, check out the
       "JavaScript and the DOM" section of Deitel's
       <a id = "link" href = "http://www.deitel.com/javascript">
         [link] JavaScript Resource Center.</a></p>
     <p id = "para3">[para3] The buttons below demonstrate:(list)</p>
     <ul id = "list">
       <li id = "item1">[item1] getElementById and parentNode</li>
       <li id = "item2">[item2] insertBefore and appendChild</li>
       <li id = "item3">[item3] replaceChild and removeChild</li>
     </ul>
     <div id = "nav" class = "nav">
       <form onsubmit = "return false" action = "#">
         <p><input type = "text" id = "gbi" value = "bigheading">
           <input type = "button" value = "Get By id"
             id = "byIdButton"></p>
         <p><input type = "text" id = "ins">
           <input type = "button" value = "Insert Before"
```

```
                id = "insertButton"></p>
            <p><input type = "text" id = "append">
              <input type = "button" value = "Append Child"
                id = "appendButton"></p>
            <p><input type = "text" id = "replace">
              <input type = "button" value = "Replace Current"
                id = "replaceButton()"></p>
            <p><input type = "button" value = "Remove Current"
                id = "removeButton"></p>
            <p><input type = "button" value = "Get Parent"
                id = "parentButton"></p>
          </form>
        </div>
      </body>
    </html>
```

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



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

Fig. 12.4. HTML5 document that's used to demonstrate DOM functionality for dynamically adding, removing and selecting elements.

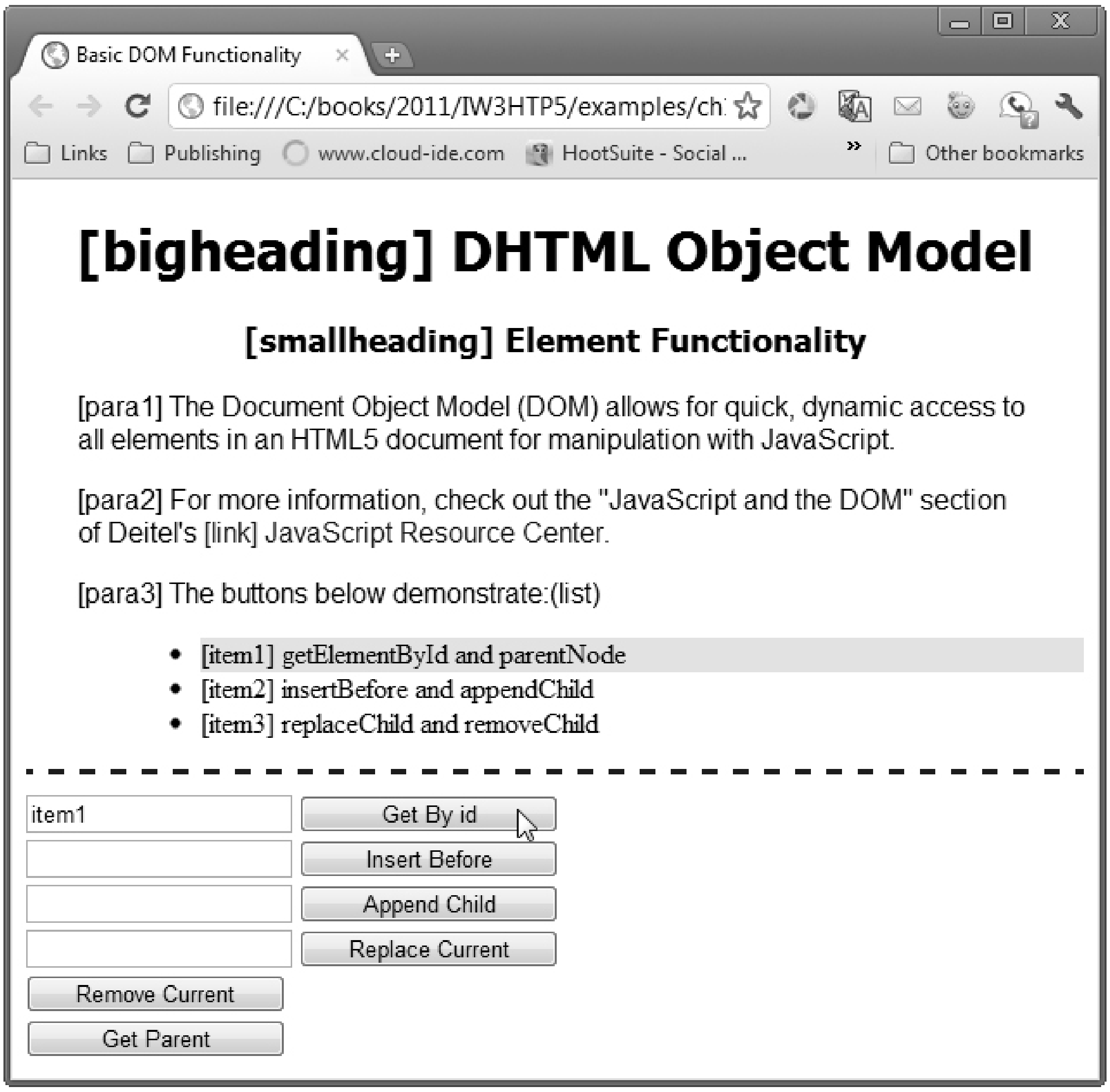
```
// Fig. 12.5: dom.js
// Script to demonstrate basic DOM functionality.
var currentNode; // stores the currently highlighted node
var idcount = 0; // used to assign a unique id to new elements

// register event handlers and initialize currentNode
function start()
{
  document.getElementById( "byIdButton" ).addEventListener(
     "click", byId, false );
  document.getElementById( "insertButton" ).addEventListener(
    "click", insert, false );
  document.getElementById( "appendButton" ).addEventListener(
    "click", appendNode, false );
  document.getElementById( "replaceButton" ).addEventListener(
    "click", replaceCurrent, false );
  document.getElementById( "removeButton" ).addEventListener(
    "click", remove, false );
  document.getElementById( "parentButton" ).addEventListener(
    "click", parent, false );

  // initialize currentNode
  currentNode = document.getElementById( "bigheading" );
} // end function start

// call start after the window loads
window.addEventListener( "load", start, false );

// get and highlight an element by its id attribute
function byId()
{
  var id = document.getElementById( "gbi" ).value;
  var target = document.getElementById( id );

  if ( target )
    switchTo( target );
} // end function byId

// insert a paragraph element before the current element
// using the insertBefore method
function insert()
{
  var newNode = createNewNode(
    document.getElementById( "ins" ).value );
  currentNode.parentNode.insertBefore( newNode, currentNode );
  switchTo( newNode );
```

```
} // end function insert

// append a paragraph node as the child of the current node
function appendNode()
{
  var newNode = createNewNode(
    document.getElementById( "append" ).value );
  currentNode.appendChild( newNode );
  switchTo( newNode );
} // end function appendNode

// replace the currently selected node with a paragraph node
function replaceCurrent()
{
  var newNode = createNewNode(
    document.getElementById( "replace" ).value );
  currentNode.parentNode.replaceChild( newNode, currentNode );
  switchTo( newNode );
} // end function replaceCurrent

// remove the current node
function remove()
{
  if ( currentNode.parentNode == document.body )
    alert( "Can't remove a top-level element." );
  else
  {
    var oldNode = currentNode;
    switchTo( oldNode.parentNode );
    currentNode.removeChild( oldNode );
  }
} // end function remove

// get and highlight the parent of the current node
function parent()
{
   var target = currentNode.parentNode;

   if ( target != document.body )
      switchTo( target );
   else
      alert( "No parent." );
} // end function parent

// helper function that returns a new paragraph node containing
// a unique id and the given text
function createNewNode( text )
{
   var newNode = document.createElement( "p" );
   nodeId = "new" + idcount;
   ++idcount;
   newNode.setAttribute( "id", nodeId ); // set newNode's id
   text = "[" + nodeId + "] " + text;
   newNode.appendChild( document.createTextNode( text ) );
   return newNode;
} // end function createNewNode

// helper function that switches to a new currentNode
function switchTo( newNode )
{
   currentNode.setAttribute( "class", "" ); // remove old highlighting
   currentNode = newNode;
   currentNode.setAttribute( "class", "highlighted" ); // highlight
   document.getElementById( "gbi" ).value =
      currentNode.getAttribute( "id" );
} // end function switchTo
```

Fig. 12.5. Script to demonstrate basic DOM functionality.

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

The first row of the `form` (Fig. 12.4, lines 31–33) 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, as shown in Fig. 12.6. The `button`'s `click` event calls function `byId`.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

item1 Get By id

Insert Before

Append Child

Replace Current

Remove Current

Get Parent

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

The `byId` function (Fig. 12.5, lines 30–37) uses `getElementById` to assign the contents of the text field to variable `id`. Line 33 uses `getElement-ById` to find the element whose `id` attribute matches variable `id` and assign it to variable `target`. If an element is found with the given `id`, an object is returned; otherwise, `null` is returned. Line 35 checks whether `target` is an object—any object used as a boolean expression is `true`, while `null` is `false`. If `target` evaluates to `true`, line 36 calls the `switchTo` function with `target` as its argument.

The `switchTo` function (lines 105–112) is used throughout the script to highlight an element in the page. The current element is given a yellow background using the style class `highlighted`, defined in the CSS styles. This function introduces the DOM element methods **`setAttribute`** and **`getAttribute`**, which allow you to modify an attribute value and get an attribute value, respectively. Line 107 uses `setAttribute` to set the current node's `class` attribute to the empty string. This clears the `class` attribute to remove the `highlighted` class from the `currentNode` before we highlight the new one.

Line 108 assigns the `newNode` object (passed into the function as a parameter) to variable `currentNode`. Line 109 uses `setAttribute` to set the new node's class attribute to the CSS class `highlighted`.

Finally, lines 110–111 use `getAttribute` to get the `currentNode`'s `id` and assign it to the input field's `value` property. While this isn't necessary when `switchTo` is called by `byId`, we'll see shortly that other functions call `switchTo`. This line ensures that the text field's `value` contains the currently selected node's `id`. Notice that we did not use `setAttribute` to change the `value` of the input field. Methods `setAttribute` and `getAttribute` do not work for user-modifiable content, such as the value displayed in an input field.

Creating and Inserting Elements Using `insertBefore` and `appendChild`

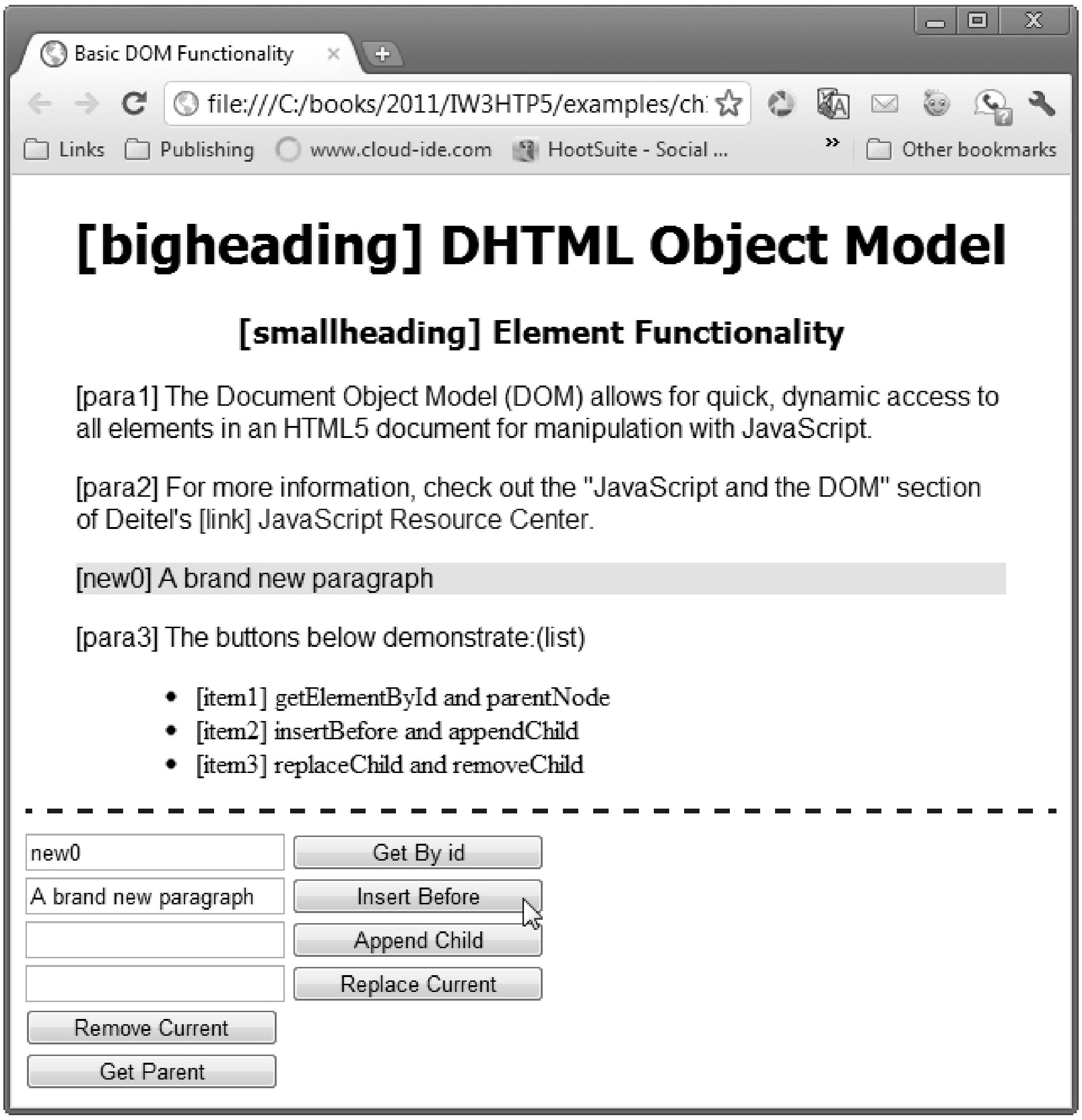
The second and third rows in the form (Fig. 12.4, lines 34–39) allow the user to create a new element and insert it before or as a child of the current node, respectively. If the user enters text in the second text field and clicks **Insert Before**, the text is placed in a new paragraph element, which is inserted into the document before the currently selected element, as in Fig. 12.7. The `button`'s `click` event calls function `insert` (Fig. 12.5, lines 41–47).

Lines 43–44 call the function `createNewNode`, passing it the value of the `"ins"` input field as an argument. Function `createNewNode`, defined in lines 93–102, creates a paragraph node containing the text passed to it. Line 95 creates a `p` element using the `document` object's **`createElement`**

method, which creates a new DOM node, taking the tag name as an argument. Though `createElement` *creates* an element, it does not *insert* the element on the page.

Line 96 creates a unique `id` for the new element by concatenating `"new"` and the value of `idcount` before incrementing `idcount`. Line 98 uses `setAttribute` to set the `id` of the new element. Line 99 concatenates the element's `id` in square brackets to the beginning of `text` (the parameter containing the paragraph's text).

Line 100 introduces two new methods. The `document`'s **`createTextNode` method** creates a node that contains only text. Given a string argument, `createTextNode` inserts the string into the text node. We create a new text node containing the contents of variable `text`. This new node is then used as the argument to the **`appendChild` method**, which is called on the new paragraph's node. Method `appendChild` inserts a child node (passed as an argument) after any existing children of the node on which it's called.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

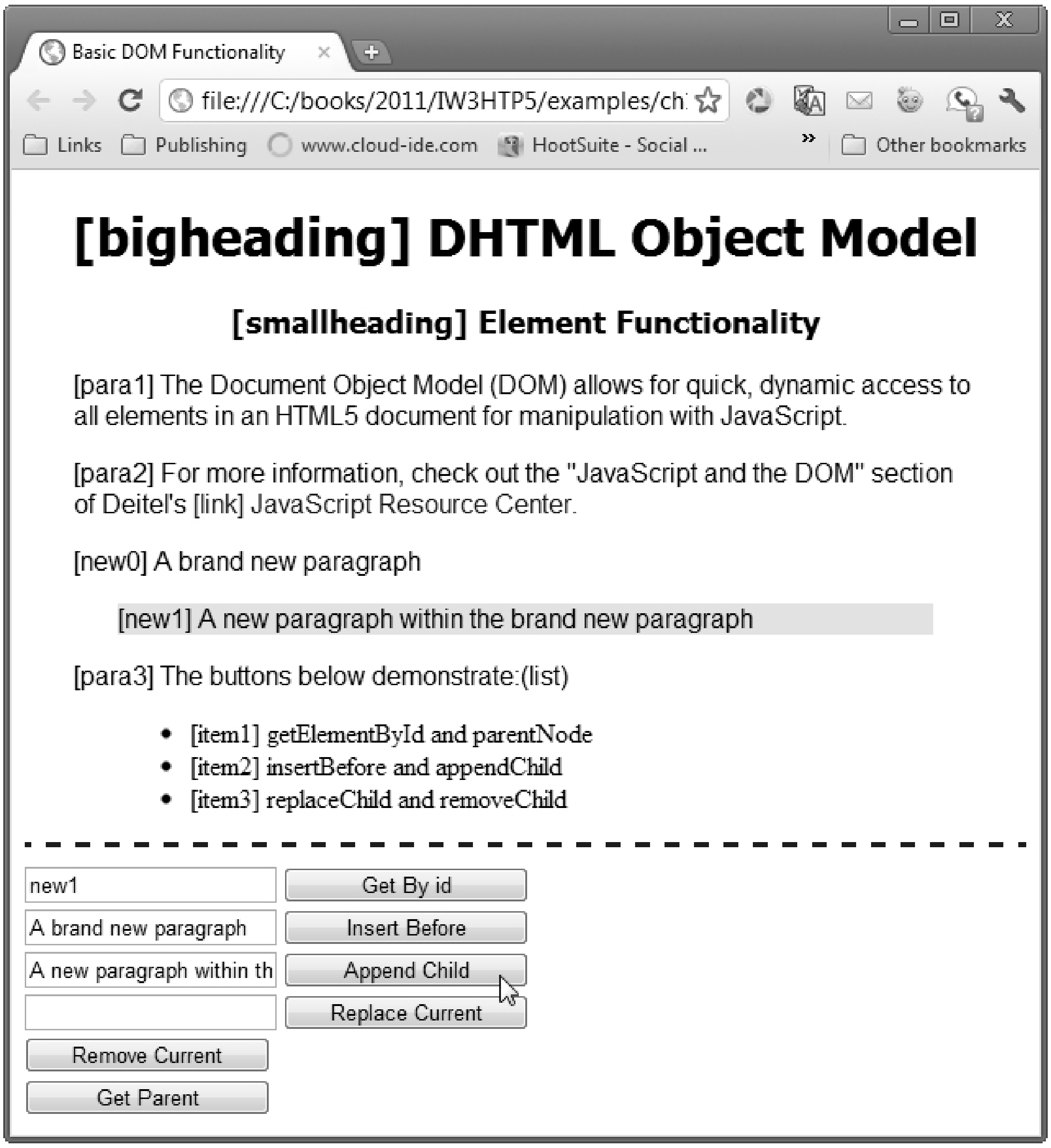
new0	Get By id
A brand new paragraph	Insert Before
	Append Child
	Replace Current
Remove Current	
Get Parent	

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`.

After the `p` element is created, line 101 returns the node to the calling function `insert`, where it's assigned to `newNode` (line 43). Line 45 inserts the new node before the currently selected one. Property **`parentNode`** contains the node's parent. In line 45, we use this property to get `currentNode`'s parent. Then we call the **`insertBefore` method** (line 45) on the parent with `newNode` and `currentNode` as its arguments. This inserts `newNode` as a child of the parent directly before `currentNode`. Line 46 uses our `switchTo` function to update the `currentNode` to the newly inserted node and highlight it in the document.

The input field and `button` in the third table row allow the user to append a new paragraph node as a child of the current element (Fig. 12.8).

This feature uses a procedure similar to the `insert` function. Lines 52–53 in function `appendNode` create a new node, line 54 inserts it as a child of the current node, and line 55 uses `switchTo` to update `currentNode` and highlight the new node.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph

[new1] A new paragraph within the brand new paragraph

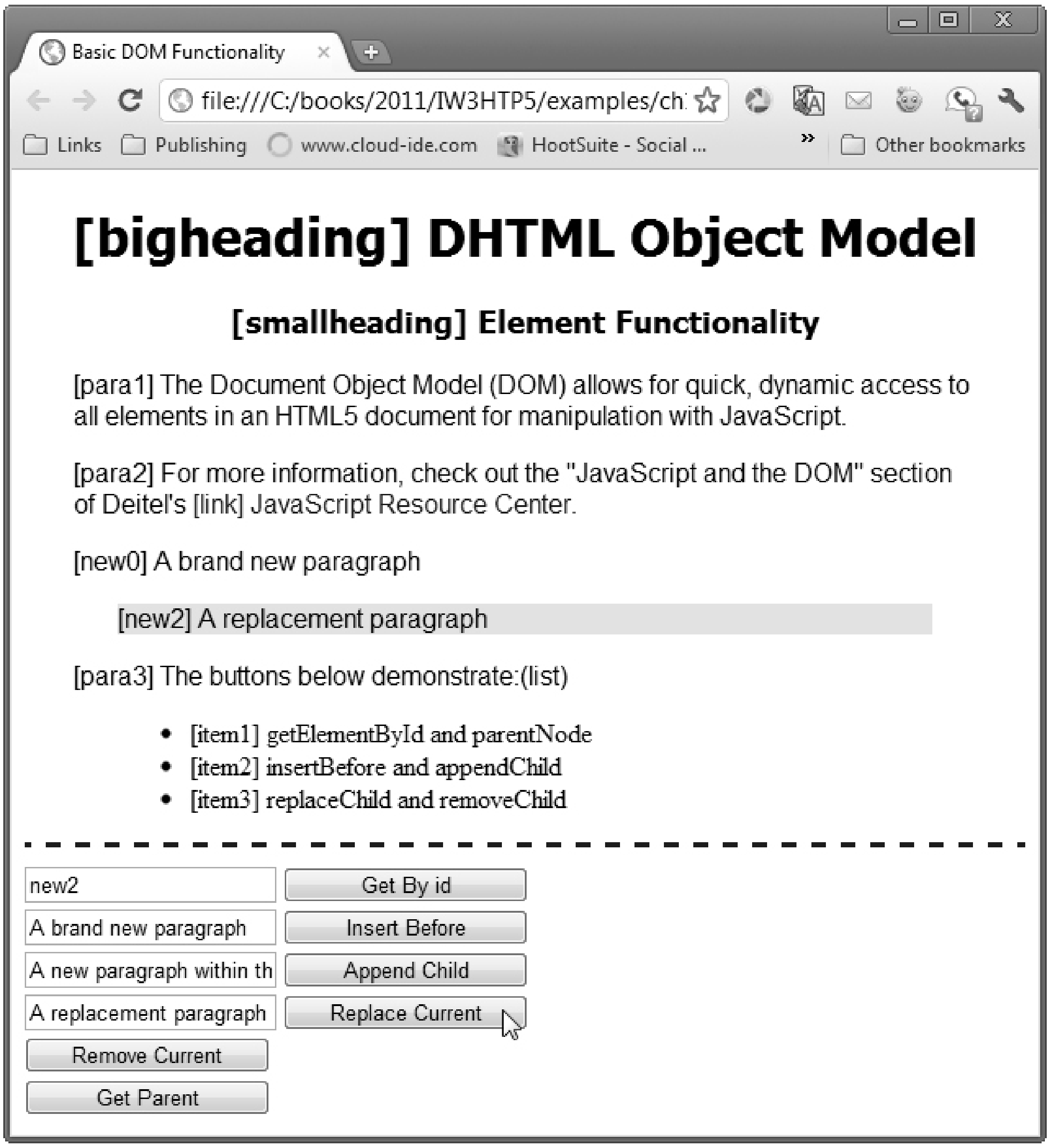
[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

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.

Replacing and Removing Elements Using `replaceChild` and `removeChild`

The next two table rows (Fig. 12.4, lines 40–44) allow the user to replace the current element with a new `p` element or simply remove the current element. When the user clicks **Replace Current** (Fig. 12.9), function `replaceCurrent` (Fig. 12.5, lines 59–65) is called.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph

[new2] A replacement paragraph

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

new2	Get By id
A brand new paragraph	Insert Before
A new paragraph within th	Append Child
A replacement paragraph	Replace Current
Remove Current	
Get Parent	

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

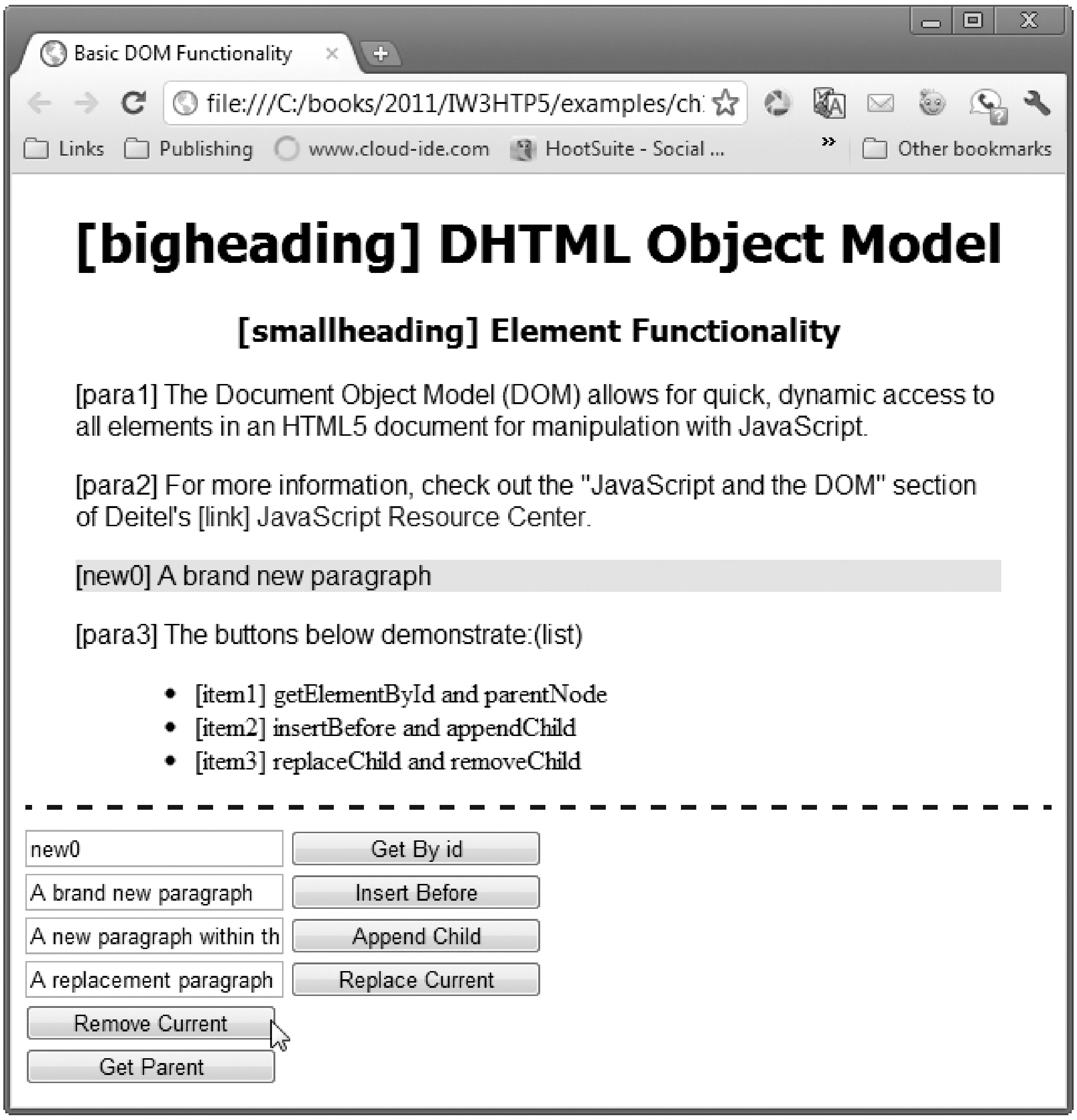
In function `replaceCurrent`, lines 61–62 call `createNewNode`, in the same way as in `insert` and `appendNode`, getting the text from the correct input field. Line 63 gets the parent of `currentNode`, then calls the `replaceChild` method on the parent. The **`replaceChild` method** receives as its first argument the new node to insert and as its second argument the node to replace.

Clicking the **Remove Current** `button` (Fig. 12.10) calls the `remove` function (Fig. 12.5, lines 68–77) to remove the current element entirely and highlights the parent. If the node's parent is the `body` element, line 71 displays an error message—the program does not allow the entire `body` element to be selected. Otherwise, lines 74–76 remove the current element. Line 74 stores the old `currentNode` in variable `oldNode`. We do this to maintain a reference to the node to be removed after we've changed the value of `currentNode`. Line 75 calls `switchTo` to highlight the parent node. Line 76 uses the **`removeChild` method** to remove the `oldNode` (a child of the new `currentNode`) from its place in the HTML5 document. In general,

parent.removeChild(*child*);

looks in *parent*'s list of children for *child* and removes it.

The `form`'s **Get Parent** `button` selects and highlights the parent element of the currently highlighted element (Fig. 12.11) by calling the `parent` function (Fig. 12.5, lines 81–89). The function simply gets the parent node (line 83), makes sure it's not the `body` element and calls `switchTo` to highlight the parent; otherwise, we display an error if the parent node is the `body` element.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

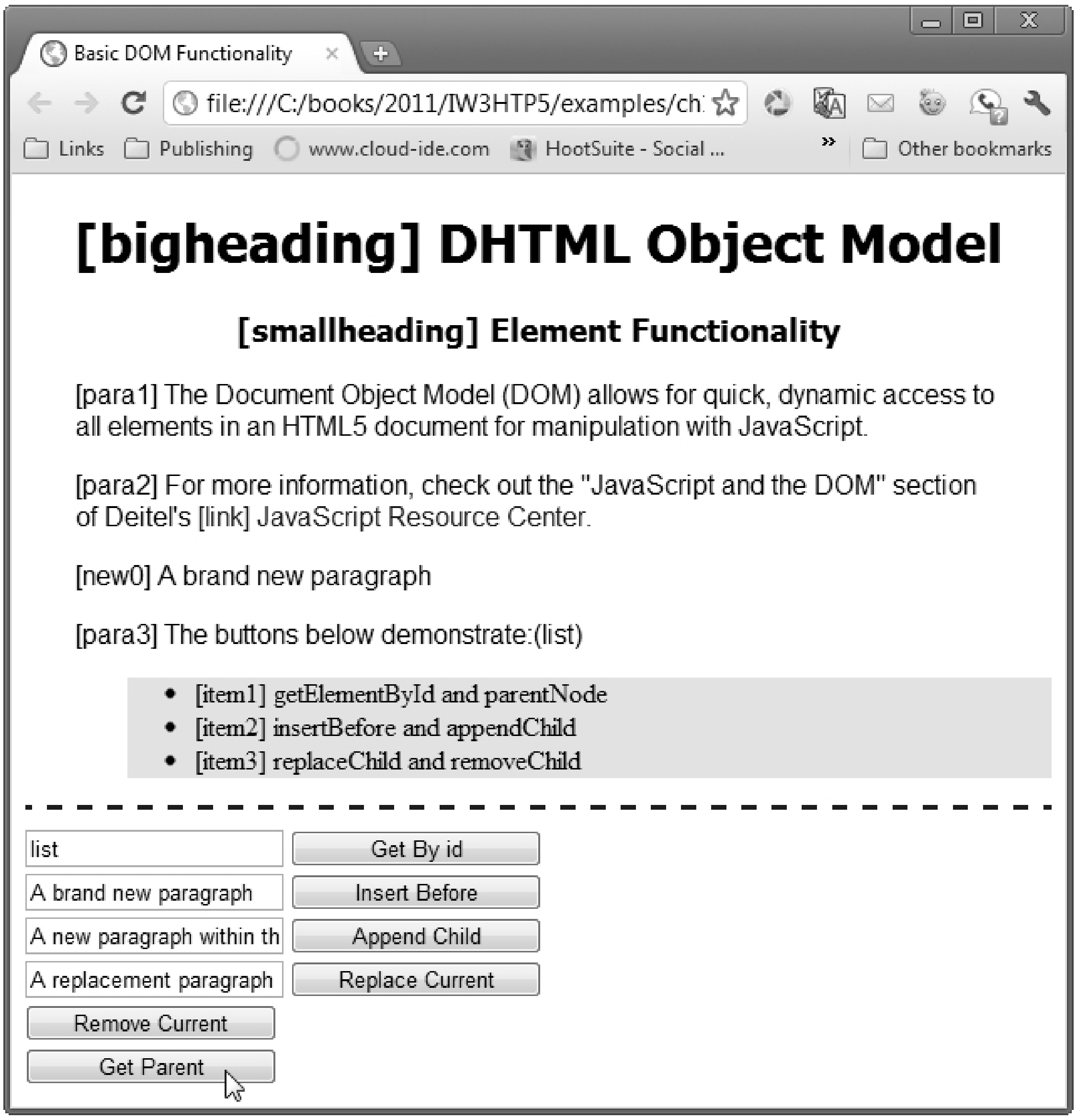
[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

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



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an HTML5 document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

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

The Document Object Model contains several **collections**, which are 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**

- **`anchors` collection**

These collections contain all the elements of the corresponding type on the page. The example of Figs. 12.12–12.14 uses the `links` collection to extract all the links on a page and display them at the bottom of the page.

CSS

Figure 12.12 contains the CSS for the example.

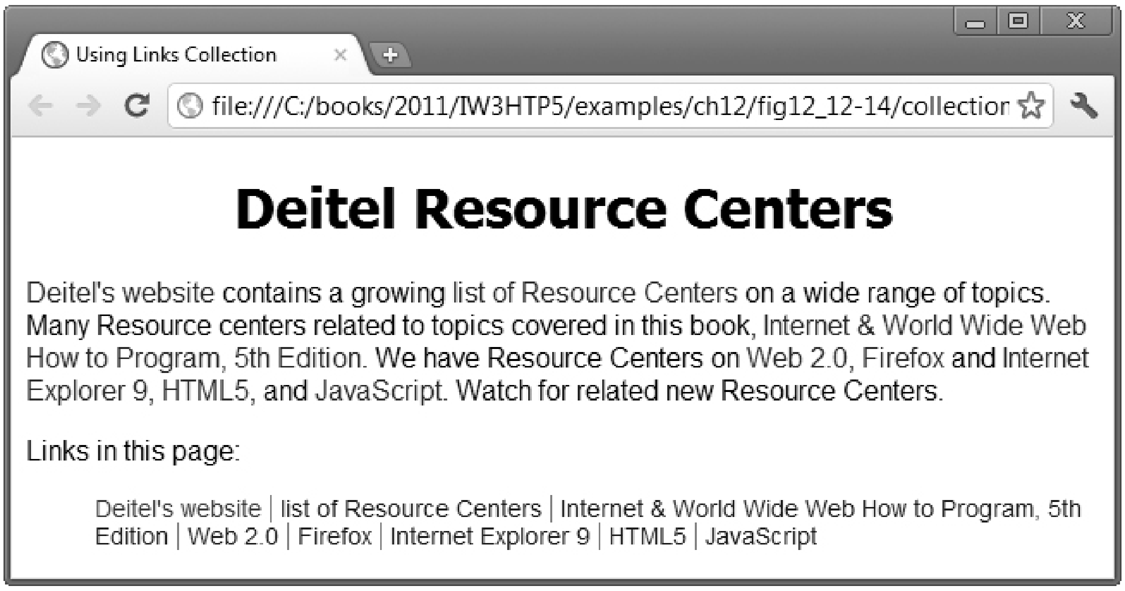
```
/* Fig. 12.12: style.css */
/* CSS for collections.html. */
body           { font-family: arial, helvetica, sans-serif }
h1             { font-family: tahoma, geneva, sans-serif;
                 text-align: center }
p a            { color: DarkRed }
ul             { font-size: .9em; }
li       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

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

HTML5 Document

Figure 12.13 presents the example's HTML5 document. The `body` contains two paragraphs (lines 14–28) with links at various places in the text and an empty `div` (line 29) with the `id` `"links"`.

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


Using Links Collection
file:///C:/books/2011/IW3HTP5/examples/ch12/fig12_12-14/collection

Deitel Resource Centers

Deitel's website contains a growing list of Resource Centers on a wide range of topics. Many Resource centers related to topics covered in this book, Internet & World Wide Web How to Program, 5th Edition. We have Resource Centers on Web 2.0, Firefox and Internet Explorer 9, HTML5, and JavaScript. Watch for related new Resource Centers.

Links in this page:

Deitel's website | list of Resource Centers | Internet & World Wide Web How to Program, 5th Edition | Web 2.0 | Firefox | Internet Explorer 9 | HTML5 | JavaScript

Fig. 12.13. Using the `links` collection.

JavaScript

Function `processlinks` (Fig. 12.14) is called when the `window`'s `load` event occurs (as specified in line 20). The function declares variable `linksList` (line 5) to store the `document`'s `links` collection, which is accessed with the `links` property of the `document` object. Line 6 creates the string (`contents`) that will contain all the document's links as an unordered list, to be inserted into the `"links" div` later. Lines 9–14 iterate through the `links` collection. The collection's **`length` property** specifies the number of items in the collection.

Line 11 stores the current link. You access the elements of the collection using indices in square brackets, just as we did with arrays. DOM collection objects have one property and two methods—the `length` property, the **`item` method** and the **`namedItem` method**. The `item` method—an alternative to the square bracketed indices—receives an an integer argument and returns the corresponding item in the collection. The `namedItem` method receives an element `id` as an argument and finds the element with that `id` in the collection.

```
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.

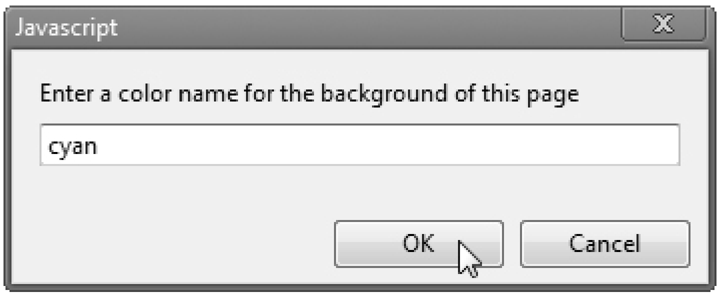
Lines 12–13 add to the `contents` string an `li` element containing the current link. Variable `currentLink` (a DOM node representing an `a` element) has an **`href` property** representing the link's `href` attribute. Line 17 inserts the contents into the empty `div` with `id "links"` to show all the links on the page in one location.

Collections allow easy access to all elements of a single type in a page. This is useful for gathering elements into one place and for applying changes to those elements across an entire page. For example, the `forms` collection could be used to disable all form inputs after a `submit` button has been pressed to avoid multiple submissions while the next page loads.

12.5. Dynamic Styles

An element's style can be changed dynamically. Often such a change is made in response to user events, which we discuss in Chapter 13. Style changes can create mouse-hover effects, interactive menus and animations. The example in Figs. 12.15–12.16 changes the document `body`'s `background-color` style property in response to user input. The document (Fig. 12.15) contains just a paragraph of text.

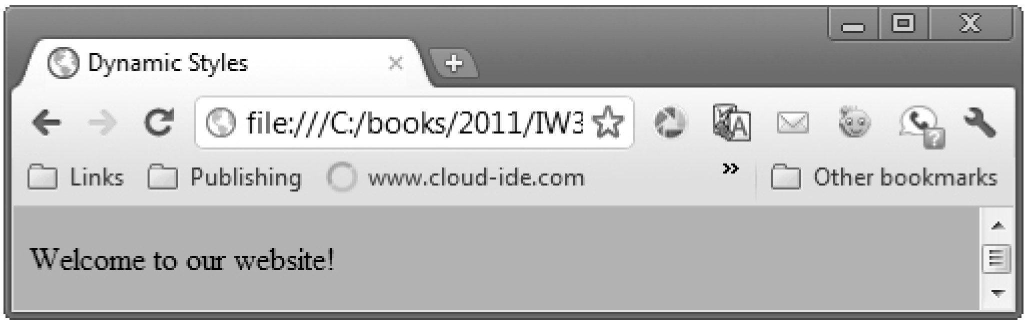
```
<!DOCTYPE html>

<!-- Fig. 12.15: dynamicstyle.html -->
<!-- Dynamic styles. -->
<html>
  <head>
    <meta charset="utf-8">
    <title>Dynamic Styles</title>
    <script src = "dynamicstyle.js"></script>
   </head>
   <body>
     <p>Welcome to our website!</p>
   </body>
 </html>
```



Javascript

Enter a color name for the background of this page

cyan



Welcome to our website!

Fig. 12.15. Dynamic styles.

Function `start` (Fig. 12.16) is called when the window's `load` event occurs (as specified in line 11). The function `prompt` s the user to enter a color name, then sets the `body` element's background color to that value. [*Note:* An error occurs if the value entered is not a valid color. See Appendix B, HTML Colors, for a list of color names.] The `document` object's **`body` property** refers to the `body` element. We then use the `setAttribute` method 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. So, if you had a class named `.red` you could set the `class` attribute's value to `"red"` to apply the style class.

```
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 example of Figs. 12.17–12.19 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.

CSS

Figure 12.17 contains the CSS styles used in the example.

```
/* 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`.

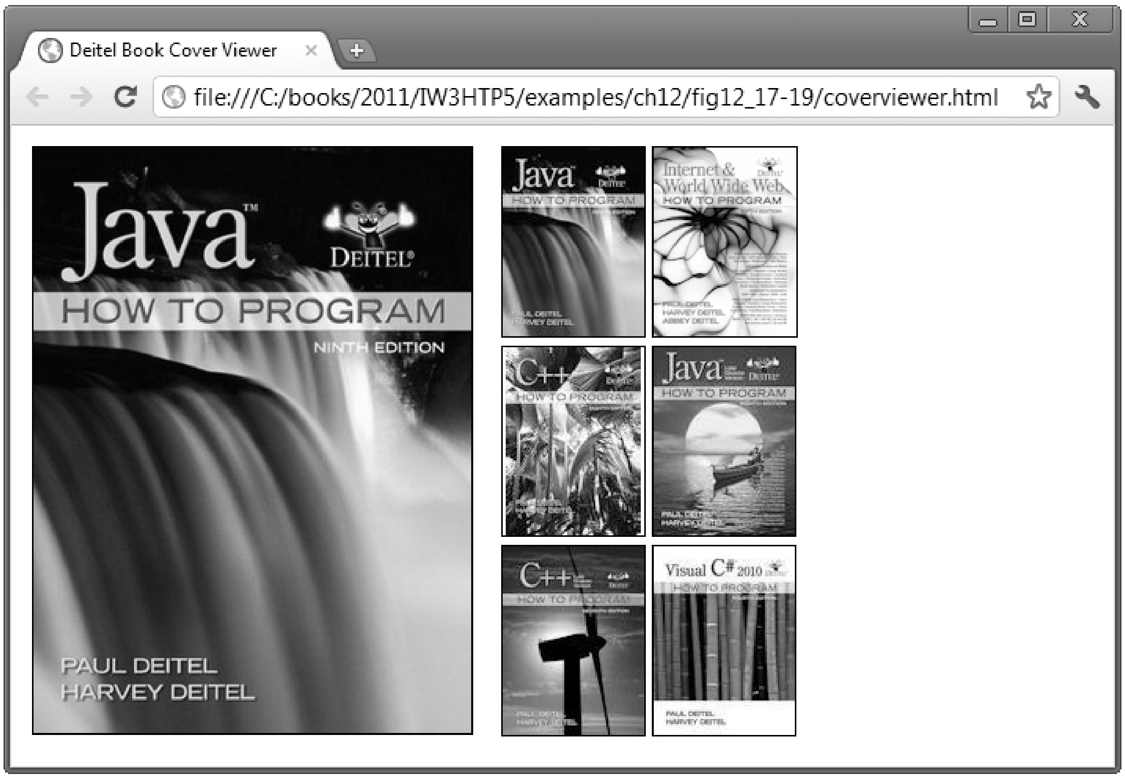
HTML5 Document

The HTML5 document (Fig. 12.18) contains two `div` elements, both floated `left` using styles defined in Fig. 12.17 to present them side by side. The left `div` contains the full-size image `jhtp.jpg`, which appears when the page loads. The right `div` contains six thumbnail images. Each responds to its click event by calling the `display` function (as registered in Fig. 12.19) and passing it the filename of the corresponding full-size image.

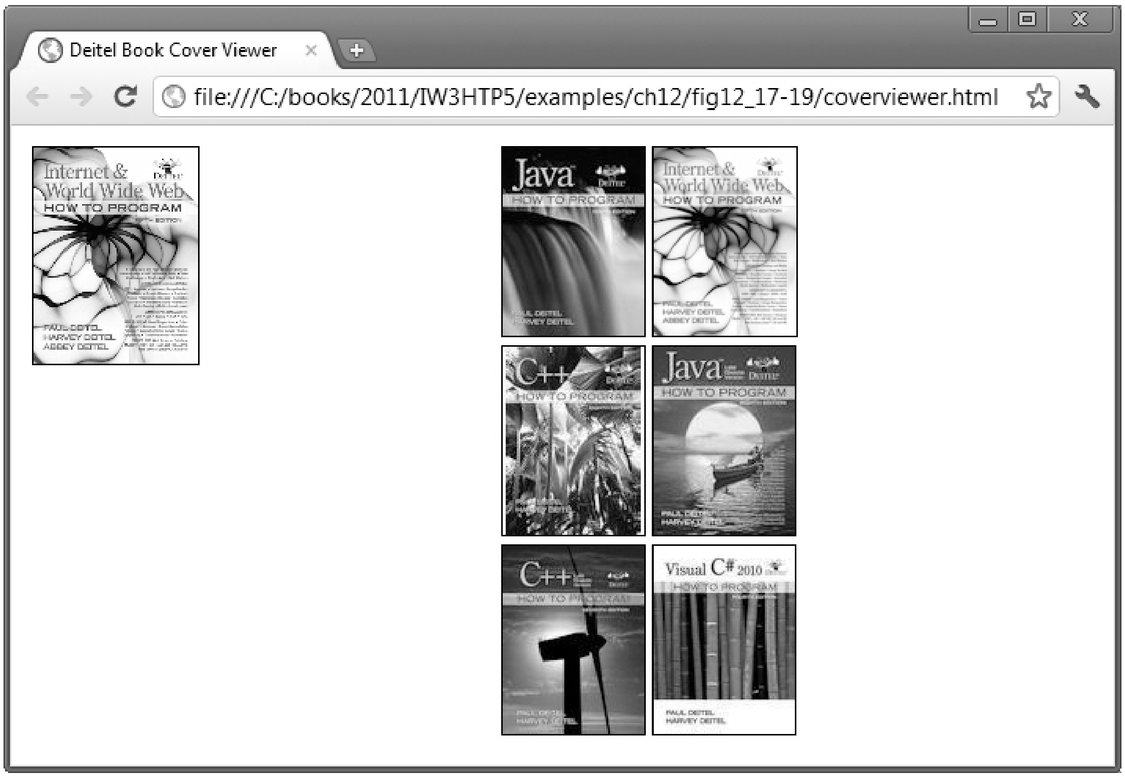
```
<!DOCTYPE html>

<!-- Fig. 12.18: coverviewer.html -->
<!-- Dynamic styles used for animation. -->
```

```
<html>
  <head>
    <meta charset = "utf-8">
    <title>Deitel Book Cover Viewer</title>
    <link rel = "stylesheet" type = "text/css" href = "style.css">
     <script src = "coverviewer.js"></script>
   </head>
   <body>
     <div id = "mainimg">
       <img id = "imgCover" src = "fullsize/jhtp.jpg"
         alt = "Full cover image">
     </div>
     <div id = "thumbs" >
       <img src = "thumbs/jhtp.jpg" id = "jhtp"
         alt = "Java How to Program cover">
       <img src = "thumbs/iw3htp.jpg" id = "iw3htp"
         alt = "Internet & World Wide Web How to Program cover">
       <img src = "thumbs/cpphtp.jpg" id = "cpphtp"
         alt = "C++ How to Program cover">
       <img src = "thumbs/jhtplov.jpg" id = "jhtplov"
         alt = "Java How to Program LOV cover">
       <img src = "thumbs/cpphtplov.jpg" id = "cpphtplov"
         alt = "C++ How to Program LOV cover">
       <img src = "thumbs/vcsharphtp.jpg" id = "vcsharphtp"
         alt = "Visual C# How to Program cover">
     </div>
   </body>
</html>
```

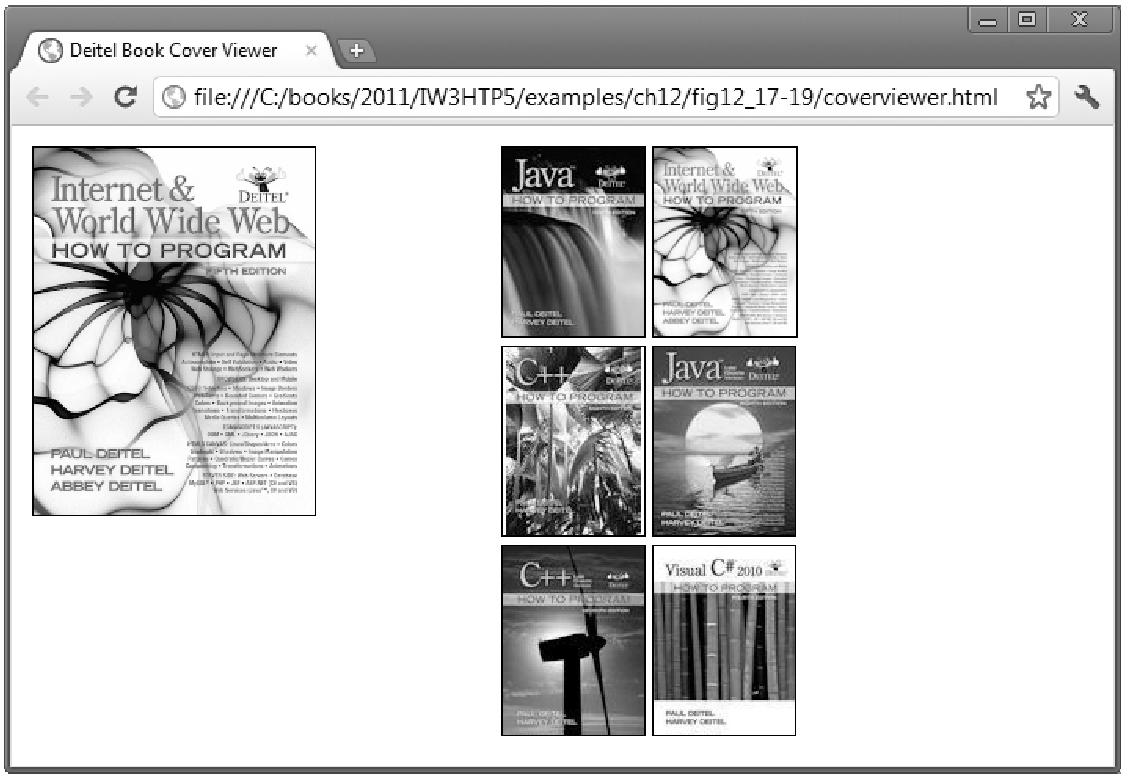
a) The cover viewer page loads with the cover of *Java How to Program, 9/e*


Deitel Book Cover Viewer
file:///C:/books/2011/IW3HTP5/examples/ch12/fig12_17-19/coverviewer.html

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


Deitel Book Cover Viewer
file:///C:/books/2011/IW3HTP5/examples/ch12/fig12_17-19/coverviewer.html

c) The cover continues to grow


Deitel Book Cover Viewer
file:///C:/books/2011/IW3HTP5/examples/ch12/fig12_17-19/coverviewer.html

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

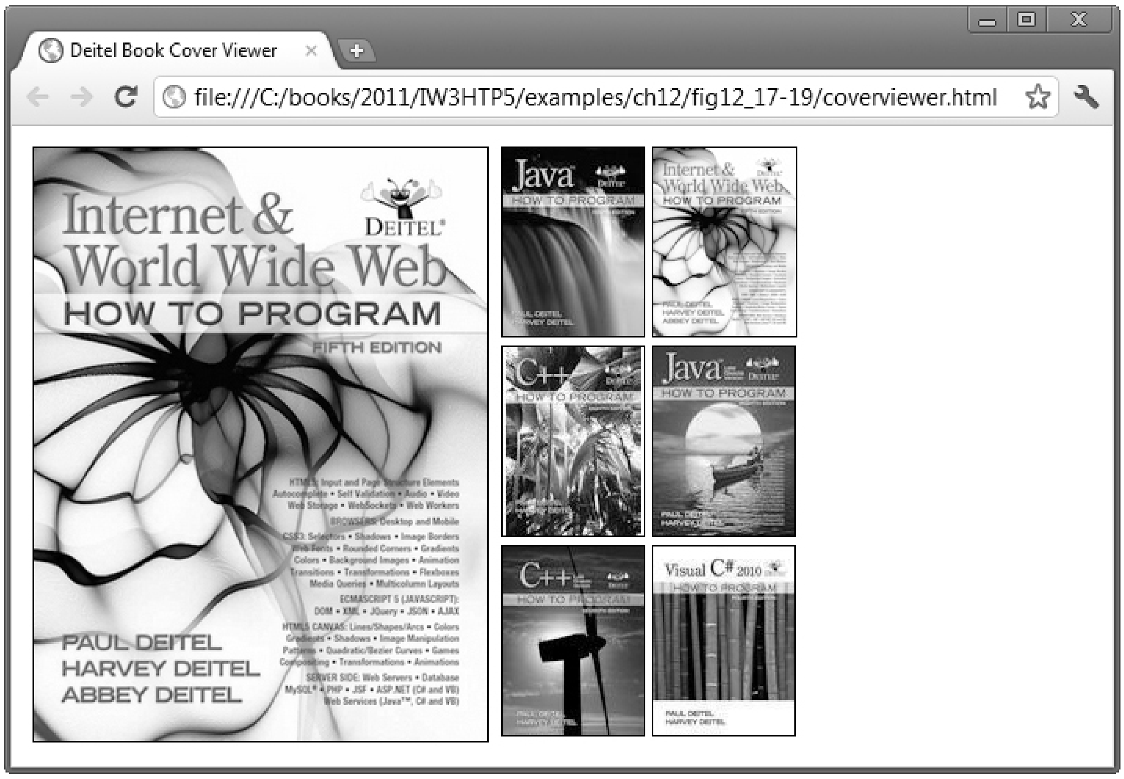

Deitel Book Cover Viewer
file:///C:/books/2011/IW3HTP5/examples/ch12/fig12_17-19/coverviewer.html

Fig. 12.18. Dynamic styles used for animation.

JavaScript

Figure 12.19 contains the JavaScript code that creates the animation effect. The same effects can be achieved by declaring animations and transitions in CSS3, as we demonstrated in Sections 5.12–5.13.

```
// Fig. 12.19: coverviewer.js
// Script to demonstrate dynamic styles used for animation.
var interval = null; // keeps track of the interval
var speed = 6; // determines the speed of the animation
var count = 0; // size of the image during the animation

// called repeatedly to animate the book cover
function run()
{
   count += speed;

   // stop the animation when the image is large enough
   if ( count >= 375 )
   {
      window.clearInterval( interval );
      interval = null;
   } // end if

   var bigImage = document.getElementById( "imgCover" );
   bigImage.setAttribute( "style", "width: " + (0.7656 * count + "px;") +
      "height: " + (count + "px;") );
} // end function run

// inserts the proper image into the main image area and
// begins the animation
function display( imgfile )
{
   if ( interval )
      return;

   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
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.

The `display` function (lines 26–36) dynamically updates the image in the left `div` to the one the user clicked. Lines 28–29 prevent the rest of the function from executing if `interval` is defined (i.e., an animation is in progress.) Line 31 gets the left `div` by its `id`, `imgCover`. Line 32 sets the image's `style` attribute, using `0px` for the `width` and `height`—the initial size of the image before the animation begins. Next, line 33 sets the image's `src` attribute to the specified image file in the `fullsize` directory, and line 34 sets its required `alt` attribute. Line 35 sets `count`, the variable that controls the animation, to `0`.

Line 36 introduces the `window` object's **`setInterval` method**, which creates a timer that controls our animation. This method takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. We use `setInterval` to call function `run` (lines 8–22) every `10` milliseconds. The `setInterval` method returns a unique identifier to keep track of that particular interval timer—we assign this identifier to the variable `interval`. This identifier can be used later to stop the timer (and thus, the animation) when the image has finished growing.

The `run` function increases the height of the image by the value of `speed` and updates its width accordingly to keep the aspect ratio consistent. The `run` function is called every 10 milliseconds, so the image grows dynamically. Line 10 adds the value of `speed` (declared and initialized to `6` in line 4) to `count`, which keeps track of the animation's progress and determines the current size of the image. If the image has grown to its full `height` (`375`), line 15 uses the `window`'s **`clearInterval` method** to terminate the timer, which prevents function `run` from being called again until the user clicks another thumbnail image. We pass to `clearInterval` the interval-timer identifier (stored in `interval`) that `setInterval` created in line 36. Since each interval timer has its own unique identifier, scripts can keep track of multiple interval timers and choose which one to stop when calling `clearInterval`.

Line 19 gets the `imgCover` element, and lines 20–21 set its `width` and `height` CSS properties. Note that line 20 multiplies `count` by a scaling factor of `0.7656`—this is the aspect ratio of the width to the height for the images used in this example. Run the code example and click on a thumbnail image to see the full animation effect.

Function `start`—Using Anonymous `function`s

Function `start` (lines 40–54) registers the `click` event handlers for the `img` elements in the HTML5 document. In each case, we define an **anonymous function** to handle the event. An anonymous function is defined with no name—it's created in nearly the same way as any other function, but with no identifier after the keyword `function`. This notation is useful when creating a function for the sole purpose of assigning it to an event handler. It's also useful when you must provide arguments to the function, since you cannot provide a function call as the second argument to `addEventListener`—if you did, the JavaScript interpreter would call the function, then pass the result of the function call to `addEventListener`. In line 43, the code

```
function() { display( "jhtp.jpg" ); }
```

defines an anonymous function that calls function `display` with the name of the image file to display.

Summary

Section 12.1 Introduction

- The Document Object Model (p. 396) gives you access to all the elements on a web page. Using JavaScript, you can dynamically create, modify and remove elements in the page.

Section 12.2 Modeling a Document: DOM Nodes and Trees

- The `getElementById` method returns objects called DOM nodes (p. 396). Every element in an HTML5 page is modeled in the web browser by a DOM node.

- All the nodes in a document make up the page's DOM tree (p. 396), which describes the relationships among elements.

- Nodes are related to each other through child-parent relationships. An HTML5 element inside another element is said to be its child (p. 396)—the containing element is known as the parent (p. 396). A node can have multiple children but only one parent. Nodes with the same parent node are referred to as siblings (p. 396).

- The document node in a DOM tree is called the root node (p. 397), because it has no parent.

Section 12.3 Traversing and Modifying a DOM Tree

- DOM element methods `setAttribute` and `getAttribute` (p. 404) allow you to modify an attribute value and get an attribute value of an element, respectively.

- The `document` object's `createElement` method (p. 405) creates a new DOM node, taking the tag name as an argument. Note that while `createElement` *creates* an element, it does not *insert* the element on the page.

- The `document` 's `createTextNode` method (p. 405) creates a DOM node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node.

- Method `appendChild` (p. 406) is called on a parent node to insert a child node (passed as an argument) after any existing children.

- The `parentNode` property (p. 406) of any DOM node contains the node's parent.

- The `insertBefore` method (p. 406) is called on a parent having a new child and an existing child as arguments. The new child is inserted as a child of the parent directly before the existing child.

- The `replaceChild` method (p. 407) is called on a parent, taking a new child and an existing child as arguments. The method inserts the new child into its list of children in place of the existing child.

- The `removeChild` method (p. 407) is called on a parent with a child to be removed as an argument.

Section 12.4 DOM Collections

- The DOM contains several collections (p. 409), which are groups of related objects on a page. DOM collections are accessed as properties of DOM objects such as the `document` object (p. 409) or a DOM node.

- The `document` object has properties containing the `images` collection (p. 409), `links` collection (p. 409), `forms` collection and `anchors` collection (p. 409). These collections contain all the elements of the corresponding type on the page.

- To find the number of elements in the collection, use the collection's `length` property (p. 410).

- To access items in a collection, use square brackets just as you would with an array, or use the `item` method. The `item` method (p. 411) of a DOM collection is used to access specific elements in a collection, taking an index as an argument. The `namedItem` method (p. 411) takes a name as a parameter and finds the element in the collection, if any, whose `id` attribute or `name` attribute matches it.

- The `href` property of a DOM link node refers to the link's `href` attribute (p. 411).

Section 12.5 Dynamic Styles

- An element's style can be changed dynamically. Often such a change is made in response to user events. Such style changes can create many effects, including mouse-hover effects, interactive menus, and animations.

- A `document` object's `body` property refers to the `body` element (p. 412) in the HTML5 page.

- The `setInterval` method (p. 417) of the `window` object repeatedly executes a statement on a certain interval. It takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. The `setInterval` method returns a unique identifier to keep track of that particular interval.

- The `window` object's `clearInterval` method (p. 418) stops the repetitive calls of object's `setInterval` method. We pass to `clearInterval` the interval identifier that `setInterval` returned.

Self-Review Exercises

12.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a. Every HTML5 element in a page is represented by a DOM tree.

b. A text node cannot have child nodes.

c. The `document` node in a DOM tree cannot have child nodes.

d. You can change an element's style class dynamically by setting the `style` attribute.

e. The `createElement` method creates a new node and inserts it into the document.

f. The `setInterval` method calls a function repeatedly at a set time interval.

g. The `insertBefore` method is called on the document object, taking a new node and an existing one to insert the new one before.

h. The most recently started interval is stopped when the `clearInterval` method is called.

i. The collection `links` contains all the links in a document with specified `id` attribute.

12.2 Fill in the blanks for each of the following statements.

a. The ________ property refers to the text inside an element, including HTML5 tags.

b. A document's DOM ________ represents all the nodes in a document, as well as their relationships to each other.

c. The ________ property contains the number of elements in a collection.

d. The ________ method allows access to an individual element in a collection.

e. The ________ collection contains all the `img` elements on a page.

Answers to Self-Review Exercises

12.1

a. False. Every element is represented by a DOM *node*. Each node is a member of the document's DOM tree.

b. True.

c. False. The `document` is the root node, therefore has no parent node.

d. False. The style class is changed by setting the `class` attribute.

e. False. The `createElement` method creates a node but does not insert it into the DOM tree.

f. True.

g. False. `insertBefore` is called on the parent.

h. False. `clearInterval` takes an interval identifier as an argument to determine which interval to end.

i. False. The `links` collection contains all links in a document.

12.2

a. `innerHTML`.

b. tree.

c. `length`.

d. `item`.

e. `images`.

Exercises

12.3 Modify Fig. 12.13 to use a background color to highlight all the links in the page instead of displaying them in a box at the bottom.

12.4 Use a browser's developer tools to view the DOM tree of the document in Fig. 12.4. Look at the document tree of your favorite website. Explore the information these tools give you in the right panel(s) about an element when you click it.

12.5 Write a script that contains a button and a counter in a `div`. The button's event handler should increment the counter each time it's clicked.

12.6 Create a web page in which the user is allowed to select the page's background color and whether the page uses serif or sans serif fonts. Then change the `body` element's `style` attribute accordingly.

12.7 *(15 Puzzle)* Write a web page that enables the user to play the game of 15. There's a 4-by-4 board (implemented as an HTML5 table) for a total of 16 slots. One of the slots is empty. The other slots are occupied by 15 tiles, randomly numbered from 1 through 15. Any tile next to the currently empty slot can be moved into the currently empty slot by clicking on the tile. Your program should create the board with the tiles out of order. The user's goal is to arrange the tiles in sequential order row by row. Using the DOM and the `click` event, write a script that allows the user to swap the positions of the open position and an adjacent tile. [*Hint:* The `click` event should be specified for each table cell.]

12.8 Modify your solution to Exercise 12.7 to determine when the game is over, then prompt the user to determine whether to play again. If so, scramble the numbers using the `Math.random` method.

12.9 Modify your solution to Exercise 12.8 to use an image that's split into 16 pieces of equal size. Discard one of the pieces and randomly place the other 15 pieces in the HTML5 table.

Support Sign Out