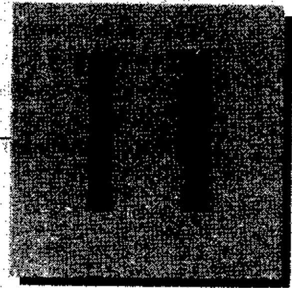
As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1s. When the 6 command (print) is given, display an asterisk or some other character of your choosing wherever there is a 1 in the array. Wherever there is a zero, display a blank. Write a script to implement the turtle-graphics capabilities discussed here. Write several turtle-graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle-graphics language.



JavaScript: Objects

My object all sublime
I shall achieve in time.
—W. S. Gilbert

Is it a world to hide virtues in?
—William Shakespeare

Good as it is to inherit a library, it is better to collect one.
—Augustine Birrell

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
—Alfred North Whitehead

OBJECTIVES

In this chapter you will learn:

- Object-based programming terminology and concepts.
- The concepts of encapsulation and data hiding.
- The value of object orientation.
- To use the JavaScript objects **`Math`**, **`String`**, **`Date`**, **`Boolean`** and **`Number`**.
- To use the browser's **`document`** and **`window`** objects.
- To use cookies.
- To represent objects simply using JSON.

Outline

11.1 Introduction
11.2 Introduction to Object Technology
11.3 Math Object
11.4 String Object
11.4.1 Fundamentals of Characters and Strings
11.4.2 Methods of the String Object
11.4.3 Character-Processing Methods
11.4.4 Searching Methods
11.4.5 Splitting Strings and Obtaining Substrings
11.4.6 XHTML Markup Methods
11.5 Date Object
11.6 Boolean and Number Objects
11.7 document Object
11.8 window Object
11.9 Using Cookies
11.10 Final JavaScript Example
11.11 Using JSON to Represent Objects
11.12 Web Resources
Summary | Terminology | Self-Review Exercise | Exercises
Special Section: Challenging String Manipulation Exercises

11.1 Introduction

Most of the JavaScript programs we've demonstrated illustrate basic programming concepts. These programs provide you with the foundation you need to build powerful and complex scripts as part of your web pages. As you proceed beyond this chapter, you will use JavaScript to manipulate every element of an XHTML document from a script.

This chapter presents a more formal treatment of **objects**. We begin by giving a brief introduction to the concepts behind object-orientation. The remainder of the chapter overviews—and serves as a reference for—several of JavaScript's built-in objects and demonstrates many of their capabilities. We also provide a brief introduction to JSON, a means for creating JavaScript objects. In the chapters on the Document Object Model and Events that follow this chapter, you will be introduced to many objects provided by the browser that enable scripts to interact with the elements of an XHTML document.

11.2 Introduction to Object Technology

This section provides a general introduction to object orientation. The terminology and technologies discussed here support various chapters that come later in the book. Here, you'll learn that objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents. In Chapters 6–10, we used built-in JavaScript objects—`Math` and `Array`—and objects provided by the web browser—`document` and

window—to perform tasks in our scripts. JavaScript uses objects to perform many tasks and therefore is referred to as an **object-based programming language**. As we have seen, JavaScript also uses constructs from the "conventional" structured programming methodology supported by many other programming languages. The first five JavaScript chapters concentrated on these conventional parts of JavaScript because they are important components of all JavaScript programs. Our goal here is to help you develop an object-oriented way of thinking. Many concepts in this book, including CSS, JavaScript, Ajax, Ruby on Rails, ASP.NET, and JavaServer Faces are based on at least some of the concepts introduced in this section.

Basic Object-Technology Concepts

We begin our introduction to object technology with some key terminology. Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers, monitors and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are "alive" in some sense—they move around and do things. Inanimate objects do not move on their own. Objects of both types, however, have some things in common. They all have **attributes** (e.g., size, shape, color and weight), and they all exhibit **behaviors** (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We'll study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults, and between humans and chimpanzees.

Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of **inheritance** relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOD **encapsulates** (i.e., wraps) attributes and **operations** (behaviors) into objects—an object's attributes and operations are intimately tied together. Objects have the property of **information hiding**. This means that objects may know how to communicate with one another across well-defined **interfaces**, but normally they are not allowed to know how

other objects are implemented—implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we'll see, is crucial to good software engineering.

Like the designers of an automobile, the designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects). You'll soon see that the browser's `document` object contains attributes and behaviors that provide access to every element of an XHTML document. Similarly, JavaScript provides objects that encapsulate various capabilities in a script. For example, the JavaScript `Array` object provides attributes and behaviors that enable a script to manipulate a collection of data. The `Array` object's `length` property (attribute) contains the number of elements in the `Array`. The `Array` object's `sort` method (behavior) orders the elements of the `Array`.

Some programming languages—like Java, Visual Basic, C# and C++—are **object oriented**. Programming in such a language is called **object-oriented programming (OOP)**, and it allows computer programmers to implement object-oriented designs as working software systems. Languages like C, on the other hand, are **procedural**, so programming tends to be **action oriented**. In procedural languages, the unit of programming is the **function**. In object-oriented languages, the unit of programming is the **class** from which objects are eventually **instantiated** (an OOP term for "created"). Classes contain functions that implement operations and data that comprises attributes.

Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in procedural languages, but the view is that data exists primarily in support of the actions that functions perform. The **verbs** in a system specification help a procedural programmer determine the set of functions that work together to implement the system.

Classes, Properties and Methods

Object-oriented programmers concentrate on creating their own **user-defined types** called **classes**. Each class contains data as well as the set of functions that manipulate that data and provide services to **clients** (i.e., other classes or functions that use the class). The data components of a class are called properties. For example, a bank account class might include an account number and a balance. The function components of a class are called methods. For example, a bank account class might include methods to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. You use built-in types (and other user-defined types) as the "building blocks" for constructing new user-defined types (classes). The **nouns** in a system specification help you determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses—a class is a "plan" for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the "bank teller" class needs to relate to other classes, such as the "customer" class, the "cash drawer" class, the "safe" class, and so on. These relationships are called **associations**.

Packaging software as classes makes it possible for future software systems to **reuse** the classes. Groups of related classes are often packaged as reusable **components**. Just as realtors often say that the three most important factors affecting the price of real estate are "location, location and location," some people in the software development community say that the three most important factors affecting the future of software development are "reuse, reuse and reuse.

Indeed, with object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts. Now that we've introduced the terminology associated with object-orientation, you'll see it used in the upcoming discussions of some of JavaScript's objects.

11.3 Math Object

The `Math` object's methods allow you to perform many common mathematical calculations. As shown previously, an object's methods are called by writing the name of the object followed by a dot (`.`) and the name of the method. In parentheses following the method name is the argument (or a comma-separated list of arguments) to the method. For example, to calculate and display the square root of `900.0` you might write

```
document.writeln( Math.sqrt( 900.0 ) );
```

which calls method `Math.sqrt` to calculate the square root of the number contained in the parentheses (`900.0`), then outputs the result. The number `900.0` is the argument of the `Math.sqrt` method. The preceding statement would display `30.0`. Some **`Math`** object methods are summarized in Fig. 11.1.

[illegible]	[illegible]	[illegible]

`abs( x )`	absolute value of x	`abs( 7.2 )` is `7.2`
		`abs( 0.0 )` is `0.0`
		`abs( -5.6 )` is `5.6`

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`

Fig. 11.1 | Math object methods. (Part 1 of 2.)

`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of x (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`max( x, y )`	larger value of x and y	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of x and y	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow( x, y )`	x raised to power y (x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, .5 )` is `3.0`
`round( x )`	rounds x to the closest integer	`round( 9.75 )` is `10` `round( 9.25 )` is `9`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0.0`

Fig. 11.1 | `Math` object methods. (Part 2 of 2.)

Common Programming Error 11.1

Forgetting to invoke a `Math` method by preceding the method name with the object name `Math` and a dot (.) is an error.

Software Engineering Observation 11.1

The primary difference between invoking a standalone function and invoking a method of an object is that an object name and a dot are not required to call a standalone function.

The `Math` object defines several commonly used mathematical constants, summarized in Fig. 11.2. [*Note:* By convention, the names of constants are written in all uppercase letters so they stand out in a program.]

`Math.E`	Base of a natural logarithm (*e*).	Approximately 2.718
`Math.LN2`	Natural logarithm of 2	Approximately 0.693
`Math.LN10`	Natural logarithm of 10	Approximately 2.302

Fig. 11.2 | Properties of the `Math` object. (Part 1 of 2.)

Math.LOG2E	Base 2 logarithm of *e*	Approximately 1.442
Math.LOG10E	Base 10 logarithm of *e*	Approximately 0.434
Math.PI	π—the ratio of a circle's circumference to its diameter	Approximately 3.141592653589793
Math.SQRT1_2	Square root of 0.5	Approximately 0.707
Math.SQRT2	Square root of 2.0	Approximately 1.414

Fig. 11.2 | Properties of the `Math` object. (Part 2 of 2.)

Good Programming Practice 11.1

Use the mathematical constants of the `Math` object rather than explicitly typing the numeric value of the constant.

11.4 `String` Object

In this section, we introduce JavaScript's string- and character-processing capabilities. The techniques discussed here are appropriate for processing names, addresses, telephone numbers, and similar items.

11.4.1 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of JavaScript programs. Every program is composed of a sequence of characters grouped together meaningfully that is interpreted by the computer as a series of instructions used to accomplish a task.

A string is a series of characters treated as a single unit. A string may include letters, digits and various **special characters**, such as +, -, *, /, and $. JavaScript supports the set of characters called **Unicode**, which represents a large portion of the world's languages. A string is an object of type **`String`**. **String literals** or **string constants** (often called **anonymous `String` objects**) are written as a sequence of characters in double quotation marks or single quotation marks, as follows:

```
"John Q. Doe"                  (a name)
'9999 Main Street'             (a street address)
"Waltham, Massachusetts"       (a city and state)
'(201) 555-1212'               (a telephone number)
```

A `String` may be assigned to a variable in a declaration. The declaration

```
var color = "blue";
```

initializes variable `color` with the `String` object containing the string `"blue"`. `String`s can be compared via the relational (<, <=, > and >=) and equality operators (== and !=). Strings are compared using the Unicode values of the corresponding characters. For example, the expression `"hello" < "Hello"` evaluates to false because lowercase letters have higher Unicode values.

11.4.2 Methods of the String Object

The `String` object encapsulates the attributes and behaviors of a string of characters. It provides many methods (behaviors) that accomplish useful tasks such as selecting characters from a string, combining strings (called **concatenation**), obtaining substrings of a string, searching for substrings within a string, tokenizing strings (i.e., splitting strings into individual words) and converting strings to all uppercase or lowercase letters. The `String` object also provides several methods that generate XHTML tags. Figure 11.3 summarizes many `String` methods. Figures 11.4–11.7 demonstrate some of these methods.

[illegible]	[illegible]
`charAt(` *index* `)`	Returns a string containing the character at the specified *index*. If there is no character at the *index*, `charAt` returns an empty string. The first character is located at *index* 0.
`charCodeAt(` *index* `)`	Returns the Unicode value of the character at the specified *index*, or `NaN` (not a number) if there is no character at that *index*.
`concat(` *string* `)`	Concatenates its argument to the end of the string that invokes the method. The string invoking this method is not modified; instead a new `String` is returned. This method is the same as adding two strings with the string-concatenation operator + (e.g., `s1.concat(s2)` is the same as `s1 + s2`).
`fromCharCode(` *value1*, *value2*, ... `)`	Converts a list of Unicode values into a string containing the corresponding characters.
`indexOf(` *substring*, *index* `)`	Searches for the first occurrence of *substring* starting from position *index* in the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from index 0 in the source string.
`lastIndexOf(` *substring*, *index* `)`	Searches for the last occurrence of *substring* starting from position *index* and searching toward the beginning of the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from the end of the source string.
`replace(` *searchString*, *replaceString* `)`	Searches for the substring *searchString*, and replaces the first occurrence with *replaceString* and returns the modified string, or the original string if no replacement was made.
`slice(` *start*, *end* `)`	Returns a string containing the portion of the string from index *start* through index *end*. If the *end* index is not specified, the method returns a string from the *start* index to the end of the source string. A negative *end* index specifies an offset from the end of the string, starting from a position one past the end of the last character (so –1 indicates the last character position in the string).

Fig. 11.3 | Some `String` object methods. (Part 1 of 2.)

[illegible]	[illegible]
`split( string )`	Splits the source string into an array of strings (tokens), where its *string* argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).
`substr( start, length )`	Returns a string containing *length* characters starting from index *start* in the source string. If *length* is not specified, a string containing characters from *start* to the end of the source string is returned.
`substring( start, end )`	Returns a string containing the characters from index *start* up to but not including index *end* in the source string.
`toLowerCase()`	Returns a string in which all uppercase letters are converted to lowercase letters. Nonletter characters are not changed.
`toUpperCase()`	Returns a string in which all lowercase letters are converted to uppercase letters. Nonletter characters are not changed.
Methods that generate XHTML tags	
`anchor( name )`	Wraps the source string in an anchor element (`<a></a>`) with *name* as the anchor name.
`fixed()`	Wraps the source string in a `<tt></tt>` element (same as `<pre></pre>`).
`link( url )`	Wraps the source string in an anchor element (`<a></a>`) with *url* as the hyperlink location.
`strike()`	Wraps the source string in a `<strike></strike>` element.
`sub()`	Wraps the source string in a `<sub></sub>` element.
`sup()`	Wraps the source string in a `<sup></sup>` element.

Fig. 11.3 | Some `String` object methods. (Part 2 of 2.)

11.4.3 Character-Processing Methods

The script in Fig. 11.4 demonstrates some of the `String` object's character-processing methods, including **`charAt`** (returns the character at a specific position), **`charCodeAt`** (returns the Unicode value of the character at a specific position), **`fromCharCode`** (returns a string created from a series of Unicode values), **`toLowerCase`** (returns the lowercase version of a string) and **`toUpperCase`** (returns the uppercase version of a string).

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC [illegible]
3     "http://www.w3.org/TR/xhtml1/[illegible]
4
5  <!-- Fig. 11.4: [illegible] -->
6  <!-- String methods charAt, [illegible]
7     toUpperCase. -->
```

Fig. 11.4 | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 1 of 2.)

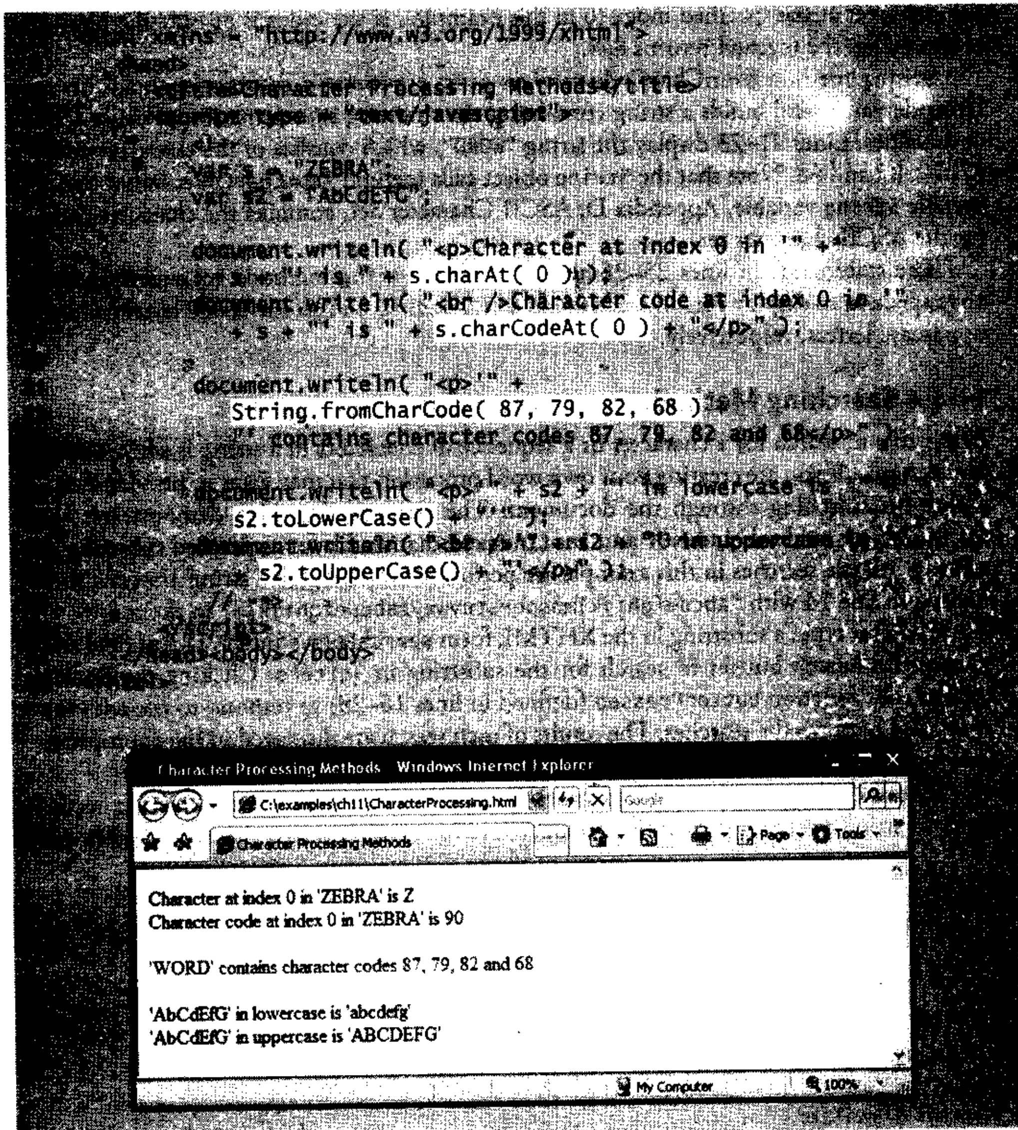


Fig. 11.4 | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 2 of 2.)

Lines 16–17 display the first character in `String s ("ZEBRA")` using `String` method `charAt`. Method **`charAt`** returns a string containing the character at the specified index (0 in this example). Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

Lines 18–19 display the character code for the first character in `String s ("ZEBRA")` by calling `String` method `charCodeAt`. Method `charCodeAt` returns the Unicode value of

the character at the specified index (0 in this example). If the index is outside the bounds of the string, the method returns NaN.

String method fromCharCode receives as its argument a comma-separated list of Unicode values and builds a string containing the character representation of those Unicode values. Lines 21–23 display the string "WORD", which consists of the character codes 87, 79, 82 and 68. Note that the String object calls method fromCharCode, rather than a specific String variable. Appendix D, ASCII Character Set, contains the character codes for the ASCII character set.

The statements in lines 25–26 and 27–28 use String methods **toLowerCase** and **toUpperCase** to display versions of String s2 ("AbCdEfG") in all lowercase letters and all uppercase letters, respectively.

11.4.4 Searching Methods

Being able to search for a character or a sequence of characters in a string is often useful. For example, if you are creating your own word processor, you may want to provide a capability for searching through the document. The script in Fig. 11.5 demonstrates the String object methods **indexOf** and **lastIndexOf** that search for a specified substring in a string. All the searches in this example are performed on the global string letters (initialized in line 14 with "abcdefghijklmnopqrstuvwxyzabcdefghijklm" in the script).

The user types a substring in the XHTML form searchForm's inputVal text field and presses the **Search** button to search for the substring in letters. Clicking the **Search** button calls function buttonPressed (defined in lines 16–29) to respond to the onclick event and perform the searches. The results of each search are displayed in the appropriate text field of searchForm.

Lines 21–22 use String method indexOf to determine the location of the first occurrence in string letters of the string inputVal.value (i.e., the string the user typed in the

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 11.5: SearchingStrings.html -->
6  <!-- String searching with indexOf and lastIndexOf. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>
10          Searching Strings with indexOf and lastIndexOf
11       </title>
12       <script type = "text/javascript">
13          <!--
14          var letters = "abcdefghijklmnopqrstuvwxyzabcdefghijklm";
15
16          function buttonPressed()
17          {
18             var searchForm = document.getElementById( "searchForm" );
19             var inputVal = document.getElementById( "inputVal" );
```

Fig. 11.5 | String searching with indexOf and lastIndexOf. (Part 1 of 3.)

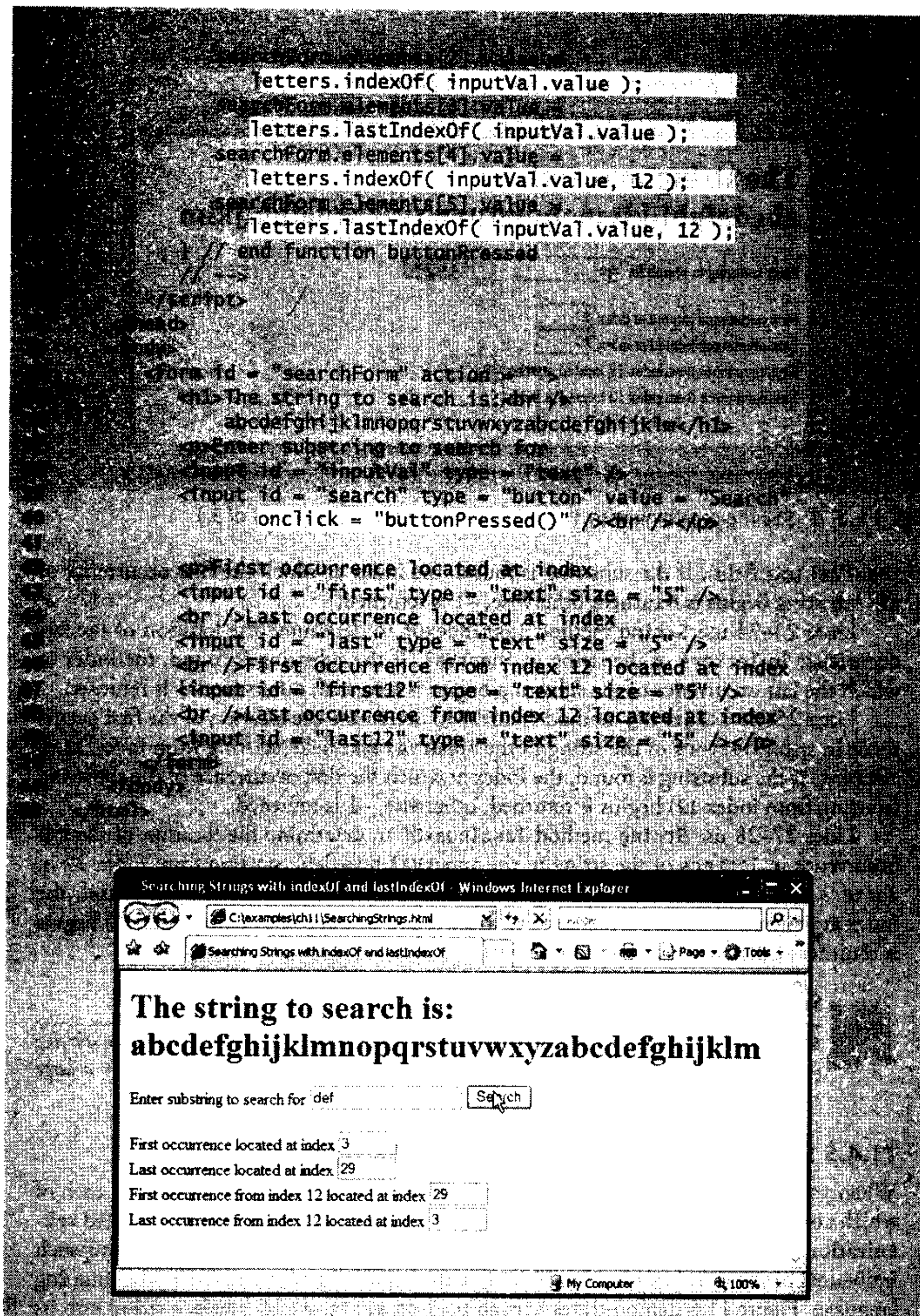


Fig. 11.5 | `String` searching with `indexOf` and `lastIndexOf`. (Part 2 of 3.)

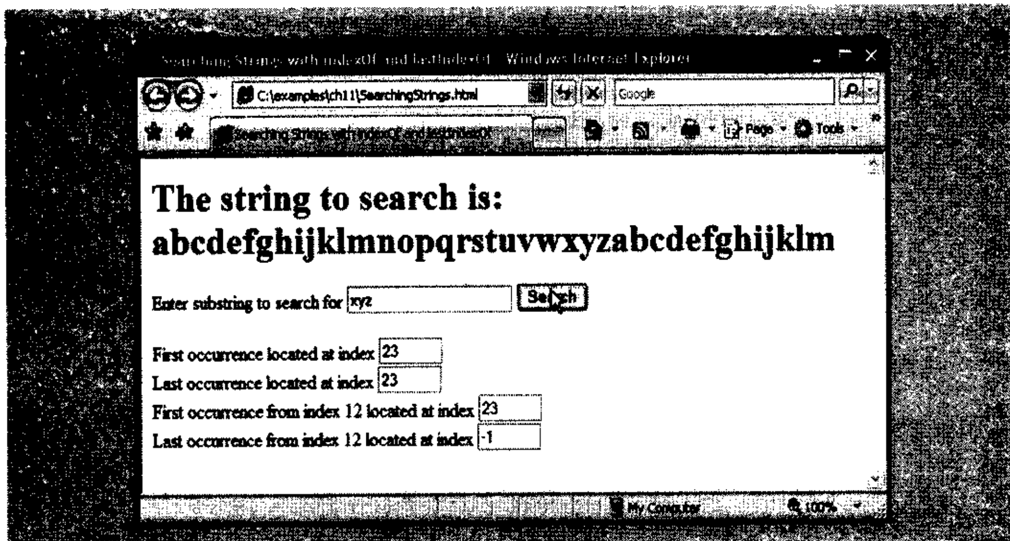


Fig. 11.5 | `String` searching with `indexOf` and `lastIndexOf`. (Part 3 of 3.)

`inputVal` text field). If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, –1 is returned.

Lines 23–24 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in `inputVal`. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, –1 is returned.

Lines 25–26 use `String` method `indexOf` to determine the location of the first occurrence in string `letters` of the string in the `inputVal` text field, starting from index `12` in `letters`. If the substring is found, the index at which the first occurrence of the substring (starting from index `12`) begins is returned; otherwise, –1 is returned.

Lines 27–28 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in the `inputVal` text field, starting from index `12` in `letters` and moving toward the beginning of the input. If the substring is found, the index at which the first occurrence of the substring (if one appears before index `12`) begins is returned; otherwise, –1 is returned.

Software Engineering Observation 11.2

`String` methods `indexOf` and `lastIndexOf`, with their optional second argument (the starting index from which to search), are particularly useful for continuing a search through a large amount of text.

11.4.5 Splitting Strings and Obtaining Substrings

When you read a sentence, your mind breaks it into individual words, or **tokens**, each of which conveys meaning to you. The process of breaking a string into tokens is called **tokenization.** Interpreters also perform tokenization. They break up statements into such individual pieces as keywords, identifiers, operators and other elements of a programming language. Figure 11.6 demonstrates `String` method **`split`**, which breaks a string into its component tokens. Tokens are separated from one another by **delimiters**, typically

white-space characters such as blanks, tabs, newlines and carriage returns. Other characters may also be used as delimiters to separate tokens. The XHTML document displays a form containing a text field where the user types a sentence to tokenize. The results of the tokenization process are displayed in an XHTML `textarea` GUI component. The script also demonstrates `String` method **`substring`**, which returns a portion of a string.

The user types a sentence into the text field with `id inputVal` text field and presses the **Split** button to tokenize the string. Function `splitButtonPressed` (lines 12–21) handles the button's `onclick` event.

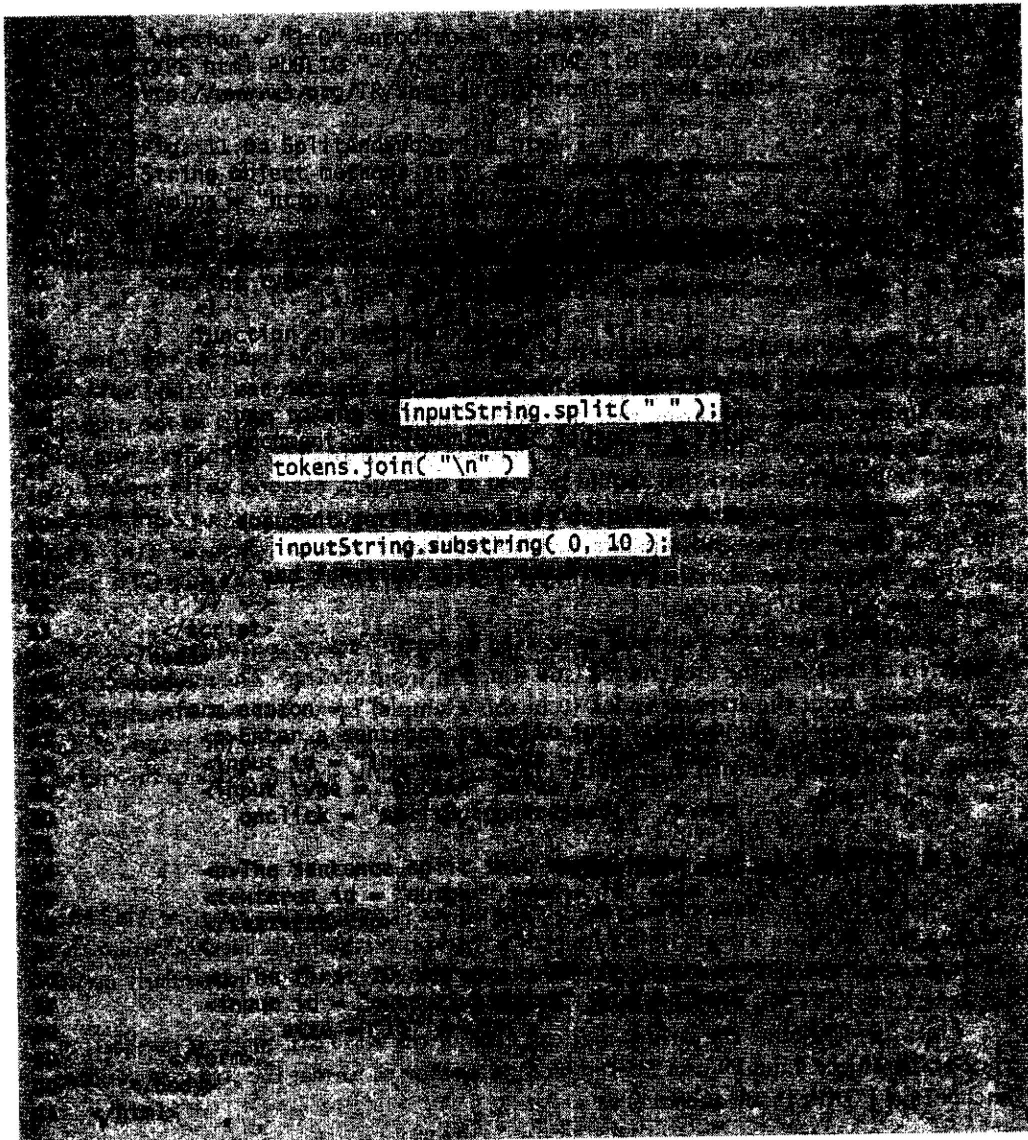


Fig. 11.6 | `String` object methods `split` and `substring`. (Part 1 of 2.)

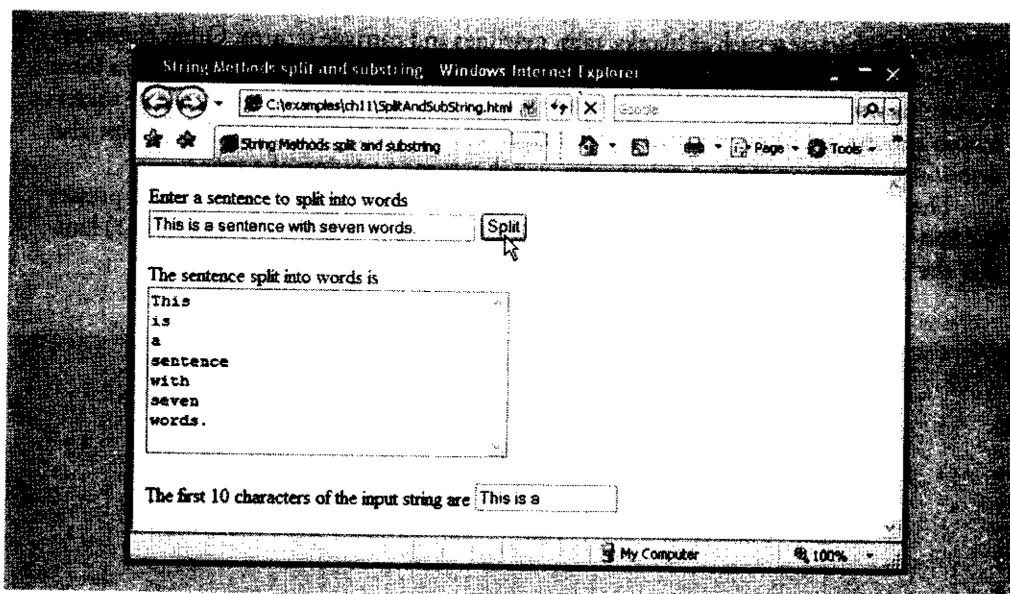


Fig. 11.6 | String object methods split and substring. (Part 2 of 2.)

Line 14 gets the value of the input field and stores it in variable inputString. Line 15 calls String method split to tokenize inputString. The argument to method split is the **delimiter string**—the string that determines the end of each token in the original string. In this example, the space character delimits the tokens. The delimiter string can contain multiple characters that should be used as delimiters. Method split returns an array of strings containing the tokens. Line 17 uses Array method join to combine the tokens in array tokens and separate each token with a newline character (\n). The resulting string is assigned to the value property of the XHTML form's output GUI component (an XHTML textarea).

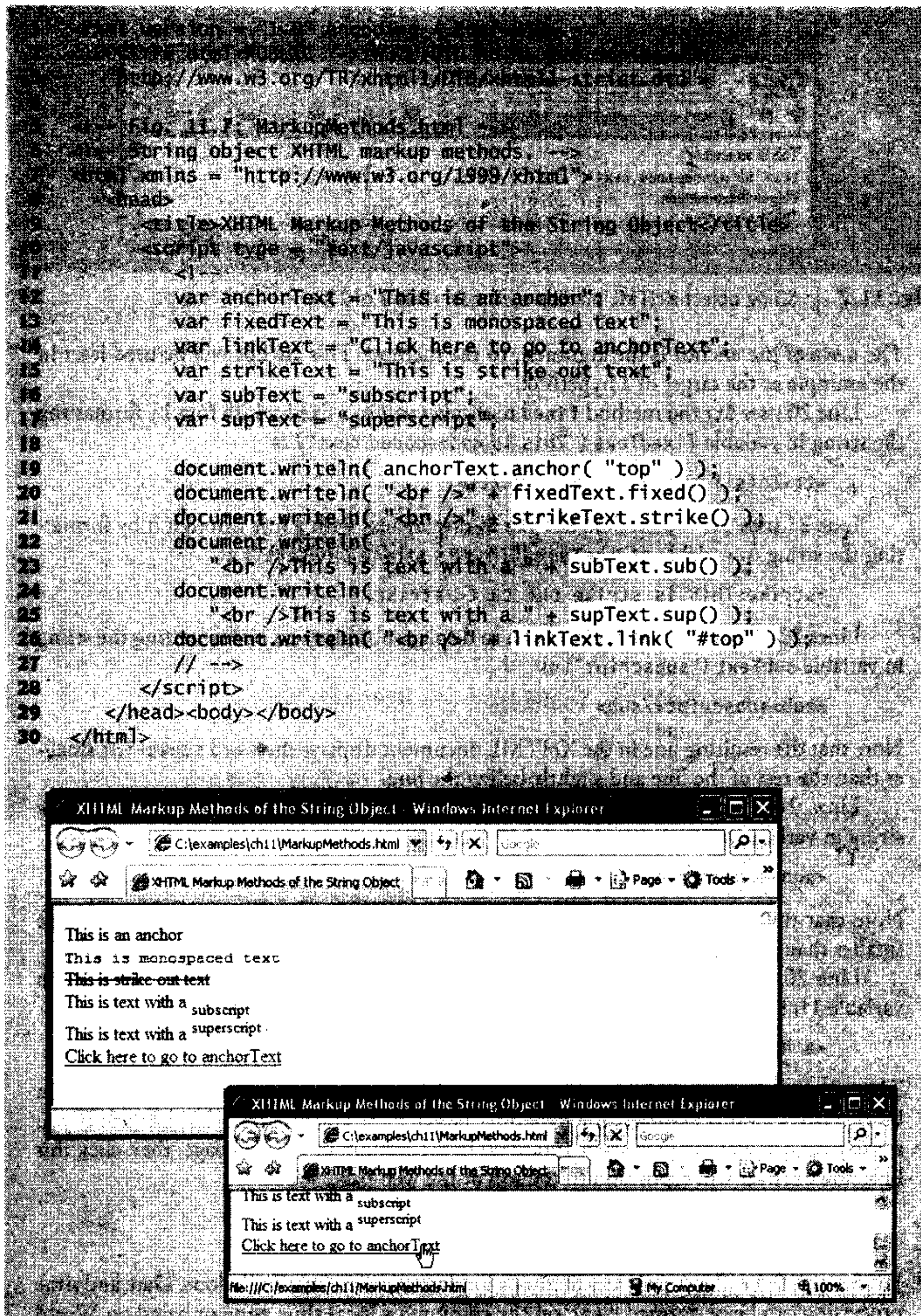
Lines 19–20 use String method substring to obtain a string containing the first 10 characters of the string the user entered (still stored in inputString). The method returns the substring from the **starting index** (0 in this example) up to but not including the **ending index** (10 in this example). If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string.

11.4.6 XHTML Markup Methods

The script in Fig. 11.7 demonstrates the String object's methods that generate XHTML markup tags.. When a String object invokes a markup method, the method wraps the String's contents in the appropriate XHTML tag. These methods are particularly useful for generating XHTML dynamically during script processing.

Lines 12–17 define the strings that call each of the XHTML markup methods of the String object. Line 19 uses String method **anchor** to format the string in variable anchorText ("This is an anchor") as

```
<a name = "top">This is an anchor</a>
```



```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 11.7: MarkupMethods.html -->
 6  <!-- String object XHTML markup methods. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>XHTML Markup Methods of the String Object</title>
10        <script type = "text/javascript">
11           <!--
12           var anchorText = "This is an anchor";
13           var fixedText = "This is monospaced text";
14           var linkText = "Click here to go to anchorText";
15           var strikeText = "This is strike out text";
16           var subText = "subscript";
17           var supText = "superscript";
18
19           document.writeln( anchorText.anchor( "top" ) );
20           document.writeln( "<br />" + fixedText.fixed() );
21           document.writeln( "<br />" + strikeText.strike() );
22           document.writeln(
23              "<br />This is text with a " + subText.sub() );
24           document.writeln(
25              "<br />This is text with a " + supText.sup() );
26           document.writeln( "<br />" + linkText.link( "#top" ) );
27           // -->
28        </script>
29     </head><body></body>
30  </html>
```


Fig. 11.7 | String object XHTML markup methods. (Part 1 of 2.)

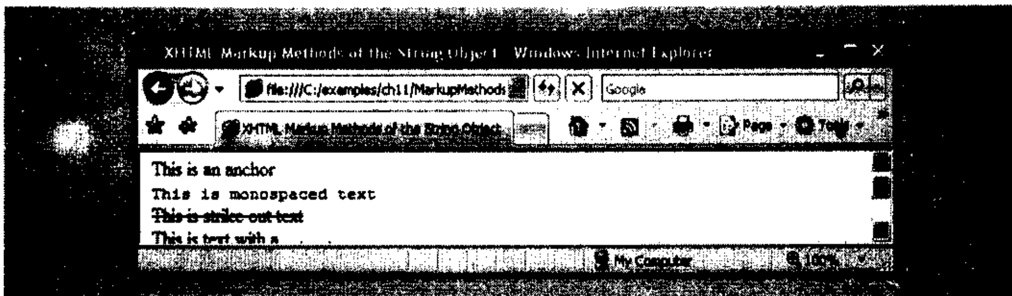


Fig. 11.7 | String object XHTML markup methods. (Part 2 of 2.)

The name of the anchor is the argument to the method. This anchor will be used later in the example as the target of a hyperlink.

Line 20 uses String method **fixed** to display text in a fixed-width font by formatting the string in variable fixedText ("This is monospaced text") as

```
<tt>This is monospaced text</tt>
```

Line 21 uses String method **strike** to display text with a line through it by formatting the string in variable strikeText ("This is strike out text") as

```
<strike>This is strike out text</strike>
```

Lines 22–23 use String method **sub** to display subscript text by formatting the string in variable subText ("subscript") as

```
<sub>subscript</sub>
```

Note that the resulting line in the XHTML document displays the word subscript smaller than the rest of the line and slightly below the line.

Lines 24–25 call String method **sup** to display superscript text by formatting the string in variable supText ("superscript") as

```
<sup>superscript</sup>
```

Note that the resulting line in the XHTML document displays the word superscript smaller than the rest of the line and slightly above the line.

Line 26 uses String method **link** to create a hyperlink by formatting the string in variable linkText ("Click here to go to anchorText") as

```
<a href = "#top">Click here to go to anchorText</a>
```

The target of the hyperlink (#top in this example) is the argument to the method and can be any URL. In this example, the hyperlink target is the anchor created in line 19. If you make your browser window short and scroll to the bottom of the web page, then click this link, the browser will reposition to the top of the web page.

11.5 Date Object

JavaScript's **Date** object provides methods for date and time manipulations. Date and time processing can be performed based on the computer's **local time zone** or based on World Time Standard's **Coordinated Universal Time** (abbreviated UTC)—formerly called

Greenwich Mean Time (GMT). Most methods of the Date object have a local time zone and a UTC version. The methods of the Date object are summarized in Fig. 11.8.

getDate() getUTCDate()	Returns a number from 1 to 31 representing the day of the month in local time or UTC.
getDay() getUTCDay()	Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC.
getFullYear() getUTCFullYear()	Returns the year as a four-digit number in local time or UTC.
getHours() getUTCHours()	Returns a number from 0 to 23 representing hours since midnight in local time or UTC.
getMilliseconds() getUTCMilliSeconds()	Returns a number from 0 to 999 representing the number of milliseconds in local time or UTC, respectively. The time is stored in hours, minutes, seconds and milliseconds.
getMinutes() getUTCMinutes()	Returns a number from 0 to 59 representing the minutes for the time in local time or UTC.
getMonth() getUTCMonth()	Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC.
getSeconds() getUTCSeconds()	Returns a number from 0 to 59 representing the seconds for the time in local time or UTC.
getTime()	Returns the number of milliseconds between January 1, 1970, and the time in the Date object.
getTimezoneOffset()	Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).
setDate(*val*) setUTCDate(*val*)	Sets the day of the month (1 to 31) in local time or UTC.
setFullYear(*y*, *m*, *d*) setUTCFullYear(*y*, *m*, *d*)	Sets the year in local time or UTC. The second and third arguments representing the month and the date are optional. If an optional argument is not specified, the current value in the Date object is used.
setHours(*h*, *m*, *s*, *ms*) setUTCHours(*h*, *m*, *s*, *ms*)	Sets the hour in local time or UTC. The second, third and fourth arguments, representing the minutes, seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the Date object is used.
setMilliSeconds(*ms*) setUTCMilliseconds(*ms*)	Sets the number of milliseconds in local time or UTC.

Fig. 11.8 | Date object methods. (Part 1 of 2.)

`setMinutes( m, s, ms )` `setUTCMinutes( m, s, ms )`	Sets the minute in local time or UTC. The second and third arguments, representing the seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the `Date` object is used.
`setMonth( m, d )` `setUTCMonth( m, d )`	Sets the month in local time or UTC. The second argument, representing the date, is optional. If the optional argument is not specified, the current date value in the `Date` object is used.
`setSeconds( s, ms )` `setUTCSeconds( s, ms )`	Sets the second in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current millisecond value in the `Date` object is used.
`setTime( ms )`	Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.
`toLocaleString()`	Returns a string representation of the date and time in a form specific to the computer's locale. For example, September 13, 2007, at 3:42:22 PM is represented as *09/13/07 15:47:22* in the United States and *13/09/07 15:47:22* in Europe.
`toUTCString()`	Returns a string representation of the date and time in the form: *15 Sep 2007 15:47:22 UTC*
`toString()`	Returns a string representation of the date and time in a form specific to the locale of the computer (*Mon Sep 17 15:47:22 EDT 2007* in the United States).
`valueOf()`	The time in number of milliseconds since midnight, January 1, 1970. (Same as `getTime`.)

Fig. 11.8 | `Date` object methods. (Part 2 of 2.)

The script of Fig. 11.9 demonstrates many of the local time zone methods in Fig. 11.8. Line 12 creates a new `Date` object. The new operator allocates the memory for the `Date` object. The empty parentheses indicate a call to the `Date` object's **constructor** with no arguments. A constructor is an initializer method for an object. Constructors are called automatically when an object is allocated with new. The `Date` constructor with no arguments initializes the `Date` object with the local computer's current date and time.

Software Engineering Observation 11.3

When an object is allocated with new, the object's constructor is called automatically to initialize the object before it is used in the program.

Lines 16–19 demonstrate the methods `toString`, `toLocaleString`, `toUTCString` and `valueOf`. Note that method `valueOf` returns a large integer value representing the total number of milliseconds between midnight, January 1, 1970, and the date and time stored in `Date` object `current`.

Lines 23–32 demonstrate the `Date` object's *get* methods for the local time zone. Note that method `getFullYear` returns the year as a four-digit number. Note as well that method `getTimeZoneOffset` returns the difference in minutes between the local time zone and UTC time (i.e., a difference of four hours in our time zone when this example was executed).

```
var current = new Date();

document.writeln(
"<h1>String representations and valueOf</h1>" );
document.writeln( "toString: " + current.toString() +
"<br />toLocaleString: " + current.toLocaleString() +
"<br />toUTCString: " + current.toUTCString() +
"<br />valueOf: " + current.valueOf() );

document.writeln(
"<h1>Get methods for local time zone</h1>" );
document.writeln( "getDate: " + current.getDate() +
... current.getDay() +
... current.getMonth() +
... current.getFullYear() +
... current.getTime() +
"<br />getHours: " + current.getHours() +
... current.getMinutes() +
"<br />getSeconds: " + current.getSeconds() +
"<br />getMilliseconds: " + current.getMilliseconds() +
"<br />getTimezoneOffset: " + current.getTimezoneOffset() );

...

anotherDate.setDate( 31 );
anotherDate.setMonth( 11 );
anotherDate.setFullYear( 2007 );
anotherDate.setHours( 23 );
anotherDate.setMinutes( 59 );
anotherDate.setSeconds( 59 );
```

Fig. 11.9 | Date and time methods of the `Date` object. (Part 1 of 2.)

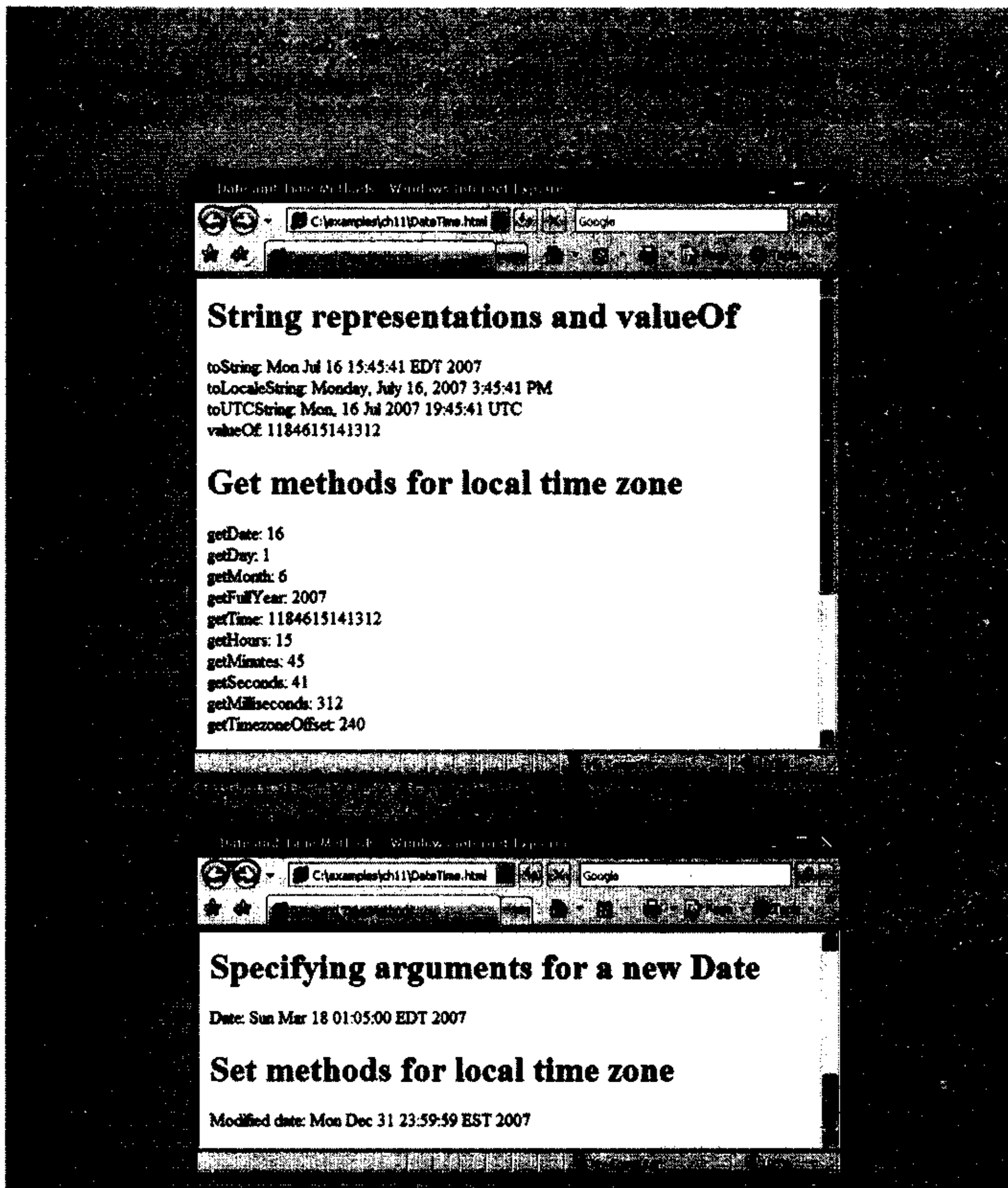


Fig. 11.9 | Date and time methods of the `Date` object. (Part 2 of 2.)

Line 36 demonstrates creating a new `Date` object and supplying arguments to the `Date` constructor for *year*, *month*, *date*, *hours*, *minutes*, *seconds* and *milliseconds*. Note that the *hours*, *minutes*, *seconds* and *milliseconds* arguments are all optional. If any one of these arguments is not specified, a zero is supplied in its place. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments is specified, it too must be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified).

Lines 40–45 demonstrate the `Date` object *set* methods for the local time zone. `Date` objects represent the month internally as an integer from 0 to 11. These values are off by one from what you might expect (i.e., 1 for January, 2 for February, ..., and 12 for December). When creating a `Date` object, you must specify 0 to indicate January, 1 to indicate February, ..., and 11 to indicate December.

Common Programming Error 11.2

Assuming that months are represented as numbers from 1 to 12 leads to off-by-one errors when you are processing `Dates`*.*

The `Date` object provides two other methods that can be called without creating a new `Date` object—**`Date.parse`** and **`Date.UTC`**. Method `Date.parse` receives as its argument a string representing a date and time, and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time. This value can be converted to a `Date` object with the statement

```
var theDate = new Date( numberOfMilliseconds );
```

which passes to the `Date` constructor the number of milliseconds since midnight, January 1, 1970, for the `Date` object.

Method `parse` converts the string using the following rules:

- Short dates can be specified in the form MM-DD-YY, MM-DD-YYYY, MM/DD/YY or MM/DD/YYYY. The month and day are not required to be two digits.
- Long dates that specify the complete month name (e.g., "January"), date and year can specify the month, date and year in any order.
- Text in parentheses within the string is treated as a comment and ignored. Commas and white-space characters are treated as delimiters.
- All month and day names must have at least two characters. The names are not required to be unique. If the names are identical, the name is resolved as the last match (e.g., "Ju" represents "July" rather than "June").
- If the name of the day of the week is supplied, it is ignored.
- All standard time zones (e.g., EST for Eastern Standard Time), Coordinated Universal Time (UTC) and Greenwich Mean Time (GMT) are recognized.
- When specifying hours, minutes and seconds, separate each by colons.
- When using a 24-hour-clock format, "PM" should not be used for times after 12 noon.

`Date` method `UTC` returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required *year*, *month* and *date*, and the optional *hours*, *minutes*, *seconds* and *milliseconds*. If any of the *hours*, *minutes*, *seconds* or *milliseconds* arguments is not specified, a zero is supplied in its place. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments in the argument list is specified, that argument must also be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified). As

with the result of `Date.parse`, the result of `Date.UTC` can be converted to a `Date` object by creating a new `Date` object with the result of `Date.UTC` as its argument.

11.6 Boolean and Number Objects

JavaScript provides the **Boolean** and **Number** objects as object **wrappers** for boolean `true`/`false` values and numbers, respectively. These wrappers define methods and properties useful in manipulating boolean values and numbers. Wrappers provide added functionality for working with simple data types.

When a JavaScript program requires a boolean value, JavaScript automatically creates a `Boolean` object to store the value. JavaScript programmers can create `Boolean` objects explicitly with the statement

```
var b = new Boolean( booleanValue );
```

The constructor argument *booleanValue* specifies whether the value of the `Boolean` object should be `true` or `false`. If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or an empty string (""), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`. Figure 11.10 summarizes the methods of the `Boolean` object.

JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program. JavaScript programmers can create a `Number` object with the statement

```
var n = new Number( numericValue );
```

The constructor argument *numericValue* is the number to store in the object. Although you can explicitly create `Number` objects, normally the JavaScript interpreter creates them as needed. Figure 11.11 summarizes the methods and properties of the `Number` object.

[illegible]	[illegible]
`toString()`	Returns the string `"true"` if the value of the `Boolean` object is true; otherwise, returns the string `"false"`.
`valueOf()`	Returns the value `true` if the `Boolean` object is true; otherwise, returns `false`.

Fig. 11.10 | `Boolean` object methods.

[illegible]	Description
`toString(` *radix* `)`	Returns the string representation of the number. The optional *radix* argument (a number from 2 to 36) specifies the number's base. For example, radix 2 results in the binary representation of the number, 8 results in the octal representation, 10 results in the decimal representation and 16 results in the hexadecimal representation.

Fig. 11.11 | `Number` object methods and properties. (Part 1 of 2.)

valueOf()	Returns the numeric value.
Number.MAX_VALUE	This property represents the largest value that can be stored in a JavaScript program—approximately 1.79E+308.
Number.MIN_VALUE	This property represents the smallest value that can be stored in a JavaScript program—approximately 5.00E–324.
Number.NaN	This property represents *not a number*—a value returned from an arithmetic expression that does not result in a number (e.g., the expression parseInt("hello") cannot convert the string "hello" into a number, so parseInt would return Number.NaN. To determine whether a value is NaN, test the result with function isNaN, which returns true if the value is NaN; otherwise, it returns false.
Number.NEGATIVE_INFINITY	This property represents a value less than -Number.MAX_VALUE.
Number.POSITIVE_INFINITY	This property represents a value greater than Number.MAX_VALUE.

Fig. 11.11 | Number object methods and properties. (Part 2 of 2.)

11.7 document Object

The **document** object is used to manipulate the document that is currently visible in the browser window. The document object has many properties and methods, such as methods document.write and document.writeln, which have both been used in prior JavaScript examples. Figure 11.12 shows the methods and properties of the document objects that are used in this chapter. You can learn more about the properties and methods of the document object in our JavaScript Resource Center (www.deitel.com/javascript).

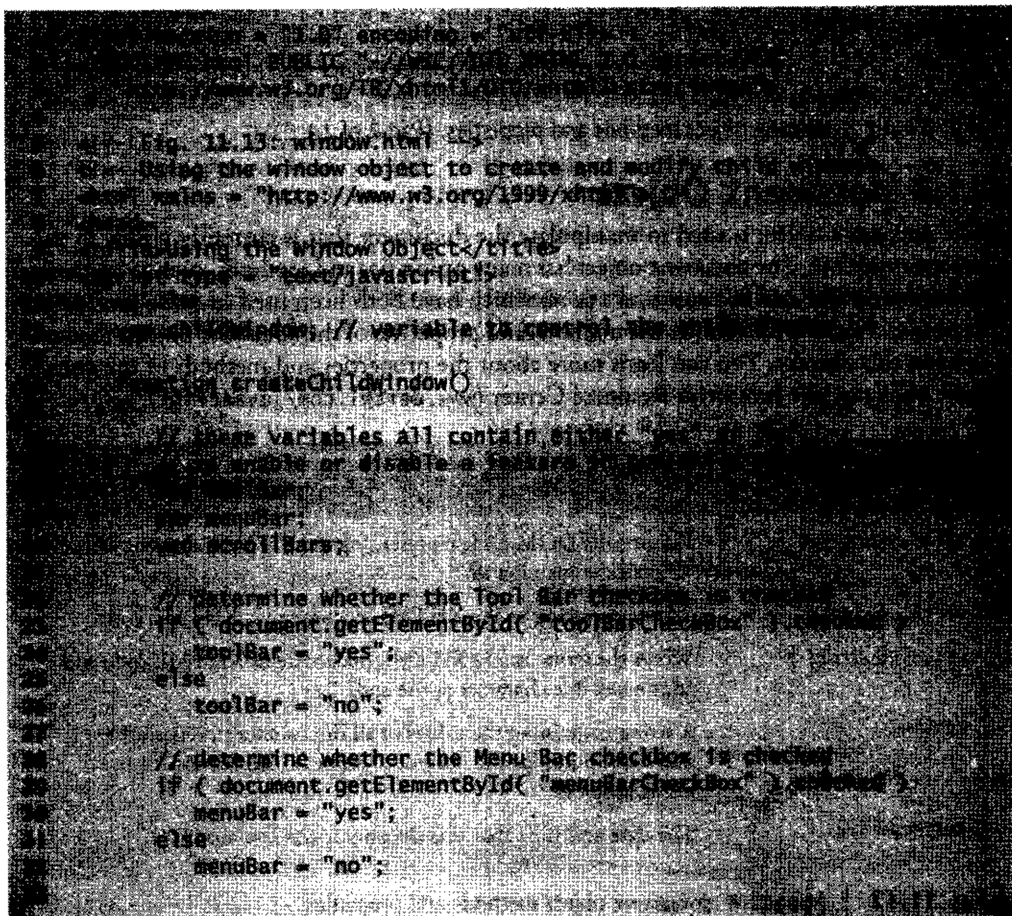
getElementById(*id*)	Returns the DOM node representing the XHTML element whose id attribute matches *id*.
write(*string*)	Writes the string to the XHTML document as XHTML code.
writeln(*string*)	Writes the string to the XHTML document as XHTML code and adds a newline character at the end.
cookie	A string containing the values of all the cookies stored on the user's computer for the current document. See Section 11.9, Using Cookies.
lastModified	The date and time that this document was last modified.

Fig. 11.12 | Important document object methods and properties.

11.8 window Object

The **window** object provides methods for manipulating browser windows. The following script shows many of the commonly used properties and methods of the `window` object and uses them to create a website that spans multiple browser windows. Figure 11.13 allows the user to create a new, fully customized browser window by completing an XHTML form and clicking the **Submit** button. The script also allows the user to add text to the new window and navigate the window to a different URL.

The script starts in line 10. Line 12 declares a variable to refer to the new window. We refer to the new window as the **child window** because it is created and controlled by the main, or **parent**, window in this script. Lines 14–50 define the `createChildWindow` function, which determines the features that have been selected by the user and creates a child window with those features (but does not add any content to the window). Lines 18–20 declare several variables to store the status of the checkboxes on the page. Lines 23–38 set each variable to "yes" or "no" based on whether the corresponding checkbox is checked or unchecked.



```
[illegible]
<!-- Fig. 11.13: window.html -->
<!-- Using the window object to create and modify child windows. -->
[illegible] xmlns = "http://www.w3.org/1999/[illegible]
[illegible]
[illegible]Using the Window Object</title>
[illegible] type = "text/javascript">
[illegible]
[illegible]; // variable to control the [illegible]

[illegible] createChildWindow()
[illegible]
// [illegible] variables all contain [illegible]
// [illegible] enable or disable a feature [illegible]
[illegible]
[illegible] menuBar;
[illegible] scrollBars;

// determine whether the Tool Bar checkbox [illegible]
if ( document.getElementById( "toolBarCheckBox" [illegible] )
   toolBar = "yes";
else
   toolBar = "no";

// determine whether the Menu Bar checkbox is checked
if ( document.getElementById( "menuBarCheckBox" [illegible] )
   menuBar = "yes";
else
   menuBar = "no";
```

Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 1 of 4.)

```
//display window with selected features
childWindow = window.open( "", "",
   ",toolbar = " + toolBar +
   ",menubar = " + menuBar +
   ",scrollbars = " + scrollBars );

childWindow.closed

childWindow.document.write(
   document.getElementById( "textForChild" ).value );

childWindow.closed

childWindow.close();

childWindow.closed

childWindow.location =
   document.getElementById( "myChildURL" ).value;
```

Fig. 11.13 | Using the window object to create and modify child windows. (Part 2 of 4.)

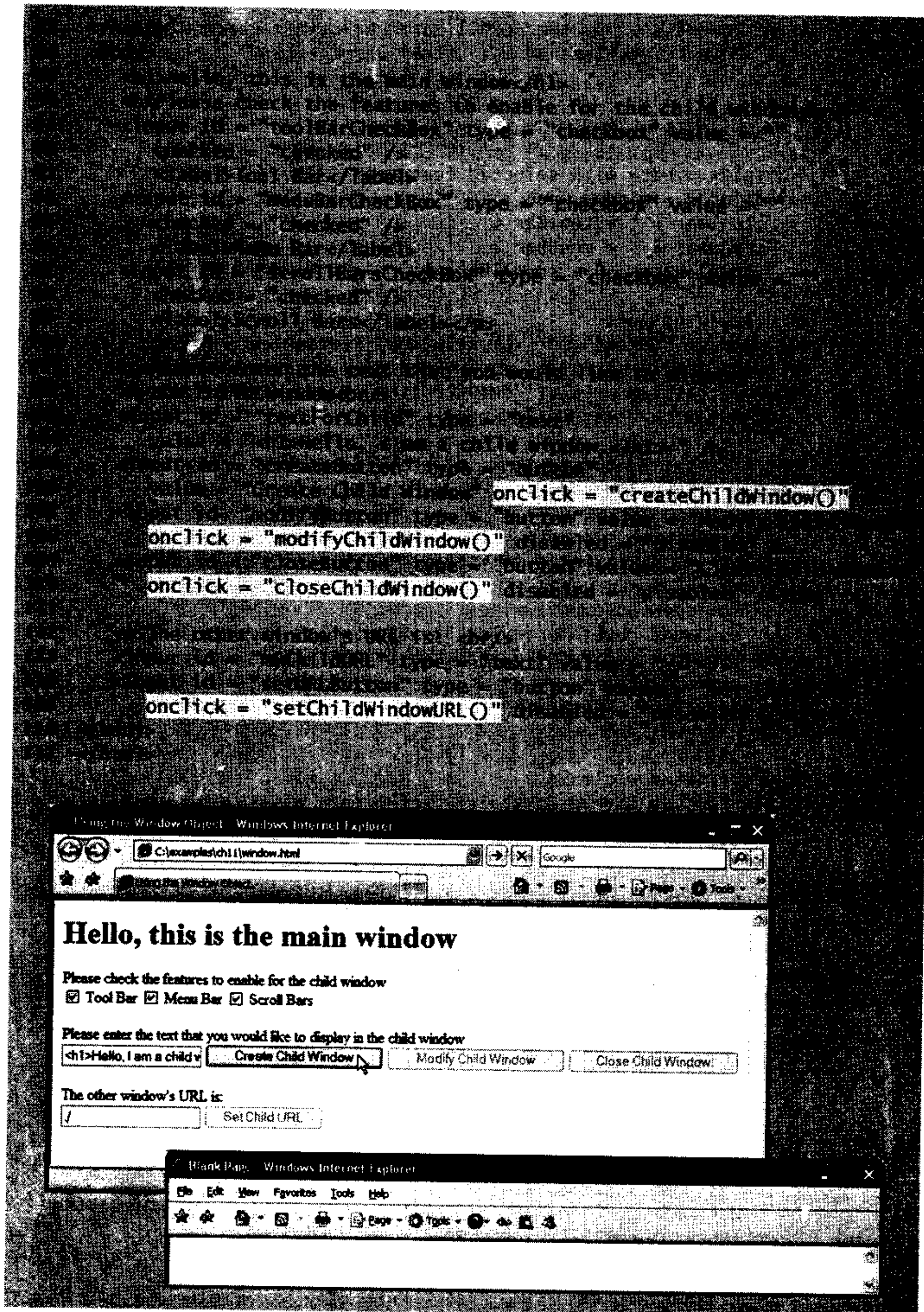


Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 3 of 4.)

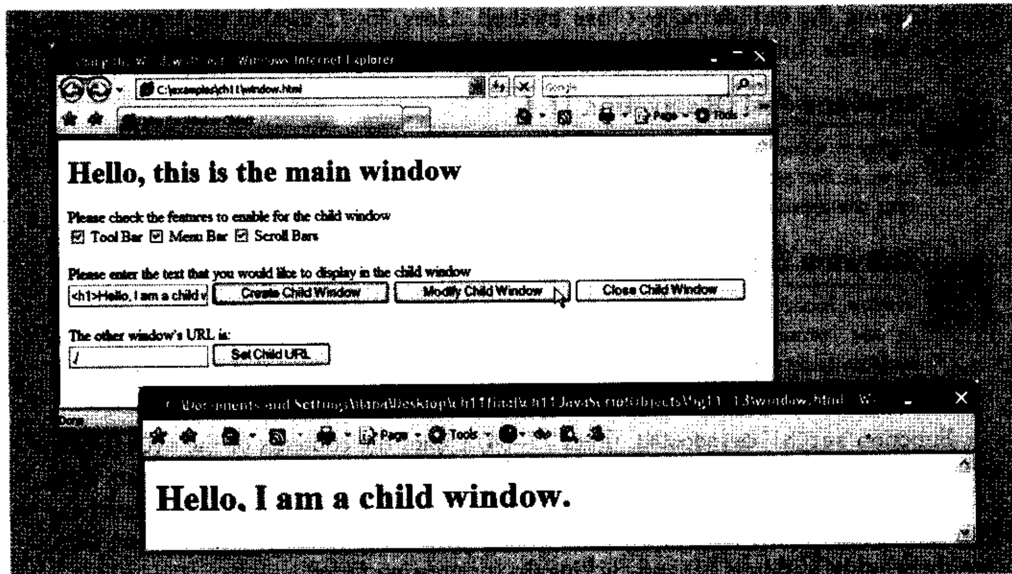


Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 4 of 4.)

The statement in lines 41–44 uses the `window` object's `open` method to create the requested child window. Method `open` has three parameters. The first parameter is the URL of the page to open in the new window, and the second parameter is the name of the window. If you specify the `target` attribute of an `a` (anchor) element to correspond to the name of a window, the `href` of the link will be opened in the window. In our example, we pass `window.open` empty strings as the first two parameter values because we want the new window to open a blank page, and we use a different method to manipulate the child window's URL.

The third parameter of the `open` method is a string of comma-separated, all-lowercase feature names, each followed by an = sign and either `"yes"` or `"no"` to determine whether that feature should be displayed in the new window. If these parameters are omitted, the browser defaults to a new window containing an empty page, no title and all features visible. [*Note:* If your menu bar is normally hidden in IE7, it will not appear in the child window. Press the *Alt* key to display it.] Lines 47–49 enable the buttons for manipulating the child window—these are initially disabled when the page loads.

Lines 53–60 define the function `modifyChildWindow`, which adds a line of text to the content of the child window. In line 55, the script determines whether the child window is closed. Function `modifyChildWindow` uses property `childWindow.closed` to obtain a boolean value that is `true` if `childWindow` is closed and `false` if the window is still open. If the window is closed, an alert box is displayed notifying the user that the window is currently closed and cannot be modified. If the child window is open, lines 58–59 obtain text from the `textForChild` input (lines 103–104) in the XHTML form in the parent window and uses the child's `document.write` method to write this text to the child window.

Function `closeChildWindow` (lines 63–73) also determines whether the child window is closed before proceeding. If the child window is closed, the script displays an alert box telling the user that the window is already closed. If the child window is open, line 68

closes it using the `childWindow.close` method. Lines 70–72 disable the buttons that interact with the child window.

Look-and-Feel Observation 11.1

Popup windows should be used sparingly. Many users dislike websites that open additional windows, or that resize or reposition the browser. Some some users have popup blockers that will prevent new windows from opening.

Software Engineering Observation 11.4

`window.location` is a property that always contains a string representation of the URL displayed in the current window. Typically, web browsers will allow a script to retrieve the `window.location` property of another window only if the script belongs to the same website as the page in the other window.

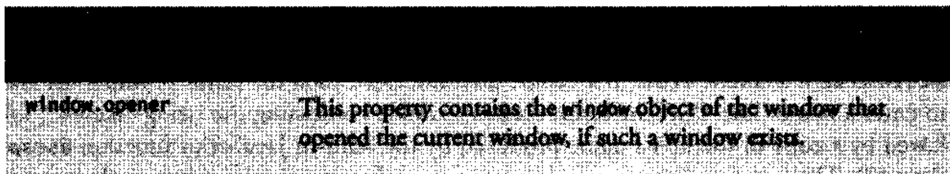
Function `setChildWindowURL` (lines 77–84) copies the contents of the `myChildURL` text field to the `location` property of the child window. If the child window is open, lines 81–82 set property `location` of the child window to the string in the `myChildURL` textbox. This action changes the URL of the child window and is equivalent to typing a new URL into the window's address bar and clicking **Go** (or pressing *Enter*).

The script ends in line 86. Lines 88–116 contain the body of the XHTML document, comprising a form that contains checkboxes, buttons, textboxes and form field labels. The script uses the form elements defined in the body to obtain input from the user. Lines 106, 108, 110, and 115 specify the `onclick` attributes of XHTML buttons. Each button is set to call a corresponding JavaScript function when clicked.

Figure 11.14 contains a list of some commonly used methods and properties of the `window` object.

[illegible]	[illegible]
`open( url, name, options )`	Creates a new window with the URL of the window set to *url*, the name set to *name* to refer to it in the script, and the visible features set by the string passed in as *option*.
`prompt( prompt, default )`	Displays a dialog box asking the user for input. The text of the dialog is *prompt*, and the default value is set to *default*.
`close()`	Closes the current window and deletes its object from memory.
`focus()`	This method gives focus to the window (i.e., puts the window in the foreground, on top of any other open browser windows).
`blur()`	This method takes focus away from the window (i.e., puts the window in the background).
`window.document`	This property contains the document object representing the document currently inside the window.
`window.closed`	This property contains a boolean value that is set to true if the window is closed, and false if it is not.

Fig. 11.14 | Important `window` object methods and properties. (Part 1 of 2.)



window.opener	This property contains the window object of the window that opened the current window, if such a window exists.

Fig. 11.14 | Important window object methods and properties. (Part 2 of 2.)

11.9 Using Cookies

Cookies provide web developers with a tool for personalizing web pages. A **cookie** is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions. A website may store a cookie on the client's computer to record user preferences or other information that the website can retrieve during the client's subsequent visits. For example, a website can retrieve the user's name from a cookie and use it to display a personalized greeting.

Microsoft Internet Explorer and Mozilla Firefox store cookies as small text files on the client's hard drive. When a user visits a website, the browser locates any cookies written by scripts on that site and makes them available to any scripts located on the site. Note that cookies may be accessed only by scripts belonging to the same website from which they originated (i.e., a cookie set by a script on `amazon.com` can be read only by other scripts on `amazon.com`).

Cookies are accessible in JavaScript through the `document` object's **`cookie`** property. JavaScript treats a cookie as a string of text. Any standard string function or method can manipulate a cookie. A cookie has the syntax "*identifier=value*," where *identifier* is any valid JavaScript variable identifier, and *value* is the value of the cookie variable. When multiple cookies exist for one website, *identifier-value* pairs are separated by semicolons in the `document.cookie` string.

Cookies differ from ordinary strings in that each cookie has an expiration date, after which the web browser deletes it. This date can be defined by setting the **`expires`** property in the cookie string. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.

The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it. Thus, if we set two cookies

```
document.cookie = "name1=value1;";
document.cookie = "name2=value2;";
```

`document.cookie` will contain `"name1=value1; name2=value2"`.

Figure 11.15 uses a cookie to store the user's name and displays a personalized greeting. This example improves upon the functionality in the dynamic welcome page example of Fig. 6.17 by requiring the user to enter a name only during the first visit to the web page. On each subsequent visit, the script can display the user name that is stored in the cookie.

Line 10 begins the script. Lines 12–13 declare the variables needed to obtain the time, and line 14 declares the variable that stores the name of the user. Lines 16–27 contain the same `if...else` statement used in Fig. 6.17 to display a time-sensitive greeting.

Lines 30–66 contain the code used to manipulate the cookie. Line 30 determines whether a cookie exists on the client computer. The expression `document.cookie` evaluates to `true` if a cookie exists. If a cookie does not exist, then the script prompts the user to enter a name (line 45). The script creates a cookie containing the string `"name="`, followed by a copy of the user's name produced by the built-in JavaScript function **escape** (line 49). The function `escape` converts any non-alphanumeric characters, such as spaces

```
[illegible]
         document.cookie
[illegible]
         var myCookie = unescape( document.cookie );
[illegible]
         var cookieTokens = myCookie.split( "=" );
[illegible]
         name = cookieTokens[ 1 ];
[illegible]
```

Fig. 11.15 | Using cookies to store user identification data. (Part 1 of 3.)

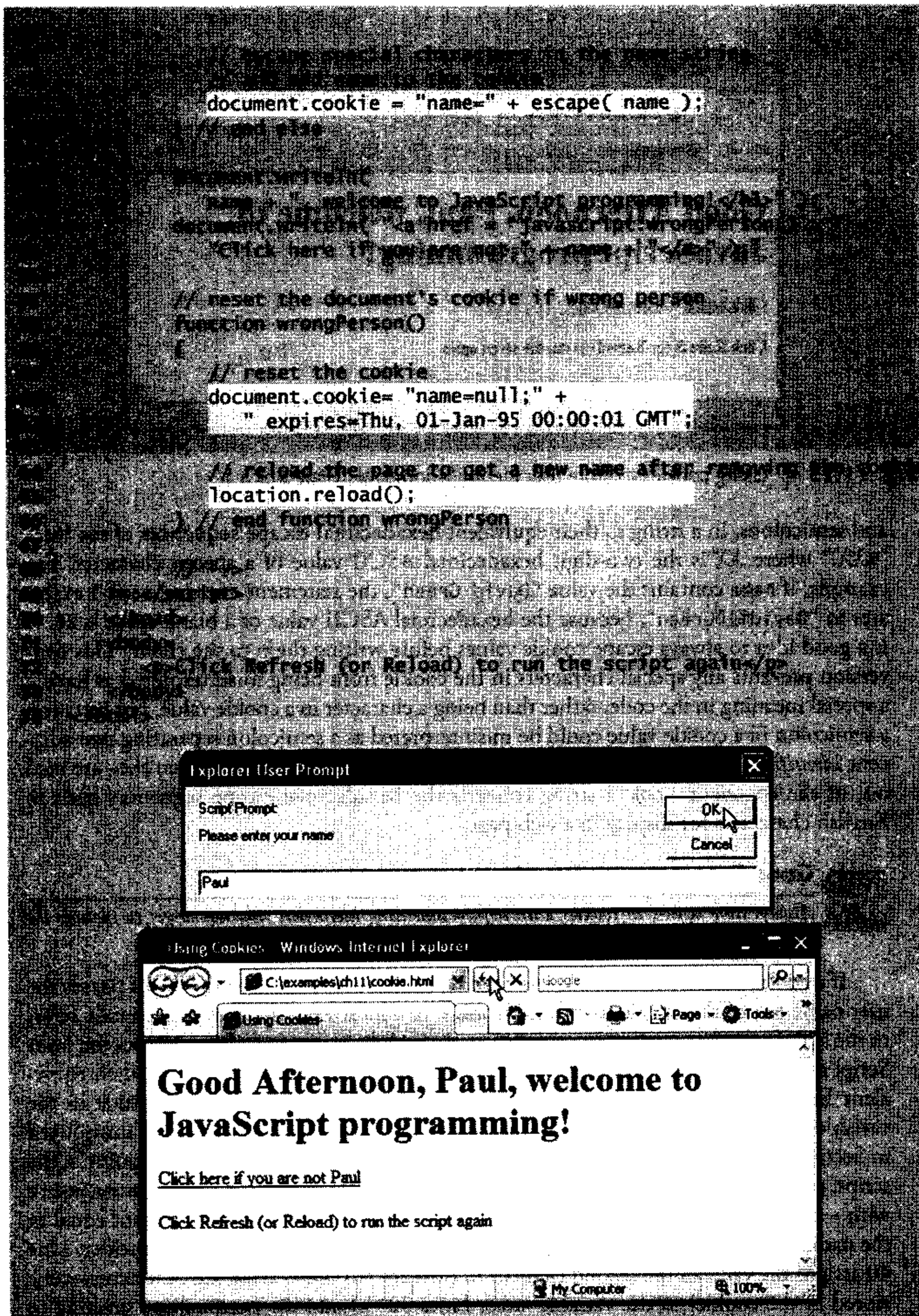


Fig. 11.15 | Using cookies to store user identification data. (Part 2 of 3.)

Fig. 11.15 | Using cookies to store user identification data. (Part 3 of 3.)

and semicolons, in a string to their equivalent **hexadecimal escape sequences** of the form "%*XX*," where *XX* is the two-digit hexadecimal ASCII value of a special character. For example, if `name` contains the value `"David Green"`, the statement `escape( name )` evaluates to `"David%20Green"`, because the hexadecimal ASCII value of a blank space is `20`. It is a good idea to always escape cookie values before writing them to the client. This conversion prevents any special characters in the cookie from being misinterpreted as having a special meaning in the code, rather than being a character in a cookie value. For instance, a semicolon in a cookie value could be misinterpreted as a semicolon separating two adjacent *identifier-value* pairs. Applying the function **`unescape`** to cookies when they are read out of the `document.cookie` string converts the hexadecimal escape sequences back to English characters for display in a web page.

Good Programming Practice 11.2

Always store values in cookies with self-documenting identifiers. Do not forget to include the identifier followed by an `=` *sign before the value being stored.*

If a cookie exists (i.e., the user has been to the page before), then the script **parses** the user name out of the cookie string and stores it in a local variable. Parsing generally refers to the act of splitting a string into smaller, more useful components. Line 34 uses the JavaScript function `unescape` to replace all the escape sequences in the cookie with their equivalent English-language characters. The script stores the unescaped cookie value in the variable `myCookie` (line 34) and uses the JavaScript function `split` (line 37), introduced in Section 11.4.5, to break the cookie into identifier and value tokens. At this point in the script, `myCookie` contains a string of the form `"name=`*`value`*`"`. We call `split` on `myCookie` with `=` as the delimiter to obtain the `cookieTokens` array, with the first element equal to the name of the identifier and the second element equal to the value of the identifier. Line 40 assigns the value of the second element in the `cookieTokens` array (i.e., the actual value stored in the cookie) to the variable `name`. Lines 52–53 add the personalized greeting to the web page, using the user's name stored in the cookie.

The script allows the user to reset the cookie, which is useful in case someone new is using the computer. Lines 54–55 create a hyperlink that, when clicked, calls the JavaScript function `wrongPerson` (lines 58–66). Lines 61–62 set the cookie `name` to `null` and the `expires` property to January 1, 1995 (though any date in the past will suffice). Internet Explorer detects that the `expires` property is set to a date in the past and deletes the cookie from the user's computer. The next time this page loads, no cookie will be found. The **`reload`** method of the `location` object forces the page to refresh (line 65), and, unable to find an existing cookie, the script prompts the user to enter a new name.

11.10 Final JavaScript Example

The past few chapters have explored many JavaScript concepts and how they can be applied on the web. The next JavaScript example combines many of these concepts into a single web page. Figure 11.16 uses functions, cookies, arrays, loops, the `Date` object, the `window` object and the `document` object to create a sample welcome screen containing a personalized greeting, a short quiz, a random image and a random quotation. We have seen all of these concepts before, but this example illustrates how they work together on one web page.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 11.16: final.html -->
6  <!-- Rich welcome page using several JavaScript concepts. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Putting It All Together</title>
10       <script type = "text/javascript">
11          <!--
12          var now = new Date(); // current date and time
13          var hour = now.getHours(); // current hour
14
15          // array with names of the images that will be randomly selected
16          var pictures =
17             [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];
18
19          // array with the quotes that will be randomly selected
20          var quotes = [ "Form ever follows function.<br/>" +
21             " Louis Henri Sullivan", "E pluribus unum." +
22             " (One composed of many.) <br/> Virgil", "Is it a" +
23             " world to hide virtues in?<br/> William Shakespeare" ];
24
25          // write the current date and time to the web page
26          document.write( "<p>" + now.toLocaleString() + "<br/></p>" );
27
28          // determine whether it is morning
29          if ( hour < 12 )
30             document.write( "<h2>Good Morning, " );
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 1 of 5.)

```
31          else
32          {
33             hour = hour - 12; // convert from 24-hour clock to PM time
34
35             // determine whether it is afternoon or evening
36             if ( hour < 6 )
37                document.write( "<h2>Good Afternoon, " );
38             else
39                document.write( "<h2>Good Evening, " );
40          } // end else
41
42          // determine whether there is a cookie
43          if ( document.cookie )
44          {
45             // convert escape characters in the cookie string to their
46             // English notation
47             var myCookie = unescape( document.cookie );
48
49             // split the cookie into tokens using = as delimiter
50             var cookieTokens = myCookie.split( "=" );
51
52             // set name to the part of the cookie that follows the = sign
53             name = cookieTokens[ 1 ];
54          } // end if
55          else
56          {
57             // if there was no cookie, ask the user to input a name
58             name = window.prompt( "Please enter your name", "Paul" );
59
60             // escape special characters in the name string
61             // and add name to the cookie
62             document.cookie = "name =" + escape( name );
63          } // end else
64
65          // write the greeting to the page
66          document.writeln(
67             name + ", welcome to JavaScript programming!</h2>" );
68
69          // write the link for deleting the cookie to the page
70          document.writeln( "<a href = \"javascript:wrongPerson()\" > " +
71             "Click here if you are not " + name + "</a><br/>" );
72
73          // write the random image to the page
74          document.write ( "<img src = \"" +
75             pictures[ Math.floor( Math.random() * 7 ) ] +
76             ".gif\" /> <br/>" );
77
78          // write the random quote to the page
79          document.write ( quotes[ Math.floor( Math.random() * 3 ) ] );
80
81          // create a window with all the quotes in it
82          function allQuotes()
83          {
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 2 of 5.)

```
 84          // create the child window for the quotes
 85          var quoteWindow = window.open( "", "", "resizable=yes, " +
 86             "toolbar=no, menubar=no, status=no, location=no, " +
 87             " scrollBars=yes" );
 88          quoteWindow.document.write( "<p>" )
 89
 90          // loop through all quotes and write them in the new window
 91          for ( var i = 0; i < quotes.length; i++ )
 92             quoteWindow.document.write( ( i + 1 ) + ".) " +
 93                quotes[ i ] + "<br/><br/>");
 94
 95          // write a close link to the new window
 96          quoteWindow.document.write( "</p><br/><a href = " +
 97             "\"javascript:window.close()\">Close this window</a>" );
 98       } // end function allQuotes
 99
100       // reset the document's cookie if wrong person
101       function wrongPerson()
102       {
103          // reset the cookie
104          document.cookie= "name=null;" +
105             " expires=Thu, 01-Jan-95 00:00:01 GMT";
106
107          // reload the page to get a new name after removing the cookie
108          location.reload();
109       } // end function wrongPerson
110
111       // open a new window with the quiz2.html file in it
112       function openQuiz()
113       {
114          window.open( "quiz2.html", "", "toolbar = no, " +
115             "menubar = no, scrollBars = no" );
116       } // end function openQuiz
117       // -->
118       </script>
119    </head>
120    <body>
121       <p><a href = "javascript:allQuotes()">View all quotes</a></p>
122
123       <p id = "quizSpot">
124          <a href = "javascript:openQuiz()">Please take our quiz</a></p>
125
126       <script type = "text/javascript">
127          // variable that gets the last modification date and time
128          var modDate = new Date( document.lastModified );
129
130          // write the last modified date and time to the page
131          document.write ( "This page was last modified " +
132             modDate.toLocaleString() );
133       </script>
134    </body>
135 </html>
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 3 of 5.)

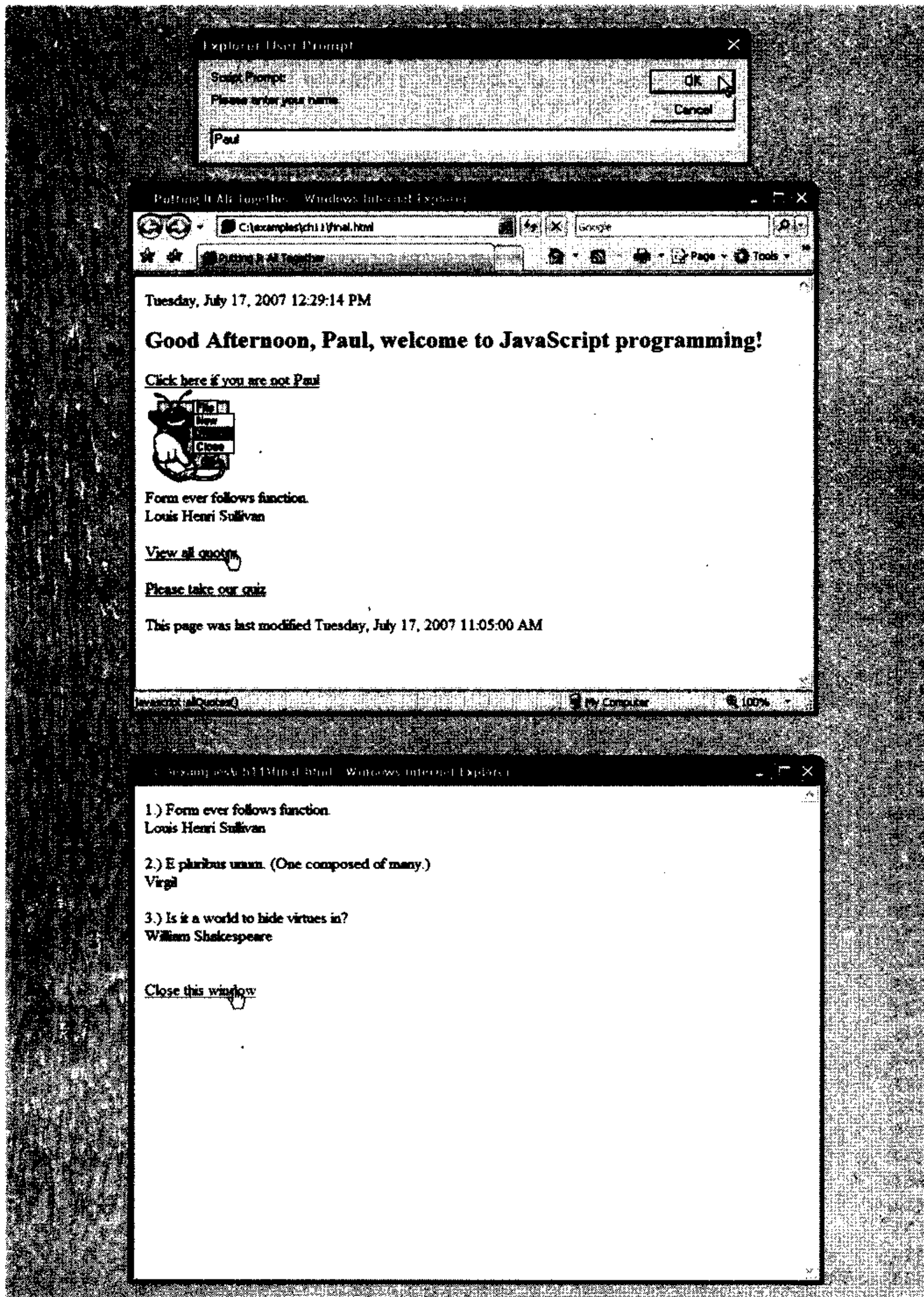


Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 4 of 5.)

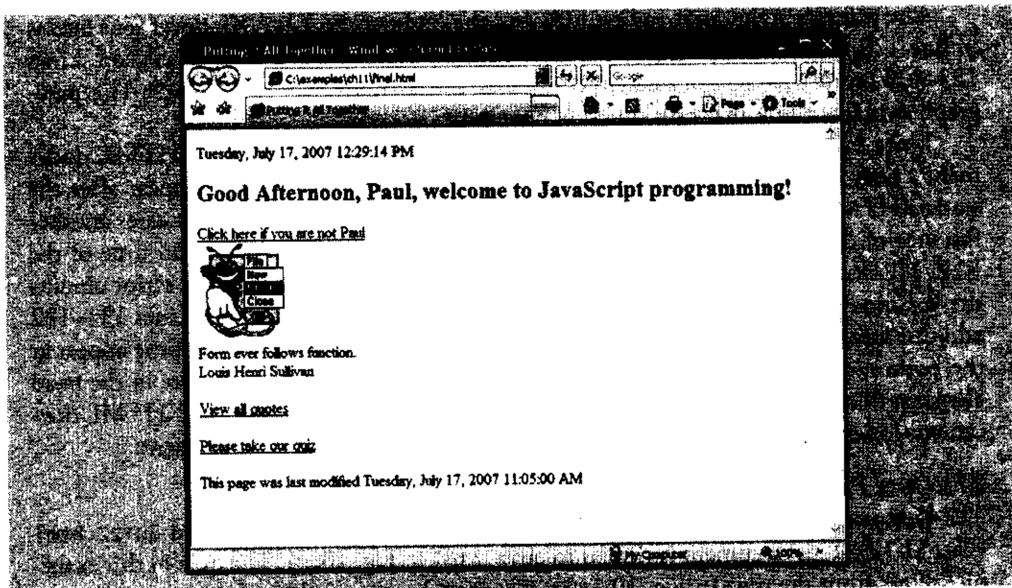


Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 5 of 5.)

The script that builds most of this page starts in line 10. Lines 12–13 declare variables needed for determining the time of day. Lines 16–23 create two arrays from which content is randomly selected. This web page contains both an image (whose filename is randomly selected from the `pictures` array) and a quote (whose text is randomly selected from the `quotes` array). Line 26 writes the user's local date and time to the web page using the `Date` object's `toLocaleString` method. Lines 29–40 display a time-sensitive greeting using the same code as Fig. 6.17. The script either uses an existing cookie to obtain the user's name (lines 43–54) or prompts the user for a name, which the script then stores in a new cookie (lines 55–63). Lines 66–67 write the greeting to the web page, and lines 70–71 produce the link for resetting the cookie. This is the same code used in Fig. 11.15 to manipulate cookies. Lines 74–79 write the random image and random quote to the web page. The script chooses each by randomly selecting an index into each array. This code is similar to the code used in Fig. 10.7 to display a random image using an array.

Function `allQuotes` (lines 82–98) uses the `window` object and a `for` loop to open a new window containing all the quotes in the `quotes` array. Lines 85–87 create a new window called `quoteWindow`. The script does not assign a URL or a name to this window, but it does specify the window features to display. Line 88 opens a new paragraph in `quoteWindow`. A `for` loop (lines 91–93) traverses the quotes array and writes each quote to `quoteWindow`. Lines 96–97 close the paragraph in `quoteWindow`, insert a new line and add a link at the bottom of the page that allows the user to close the window. Note that `allQuotes` generates a web page and opens it in an entirely new window with JavaScript.

Function `wrongPerson` (lines 101–109) resets the cookie storing the user's name. This function is identical to function `wrongPerson` in Fig. 11.15.

Function `openQuiz` (lines 112–116) opens a new window to display a sample quiz. Using the `window.open` method, the script creates a new window containing `quiz2.html` (lines 114–115). We discuss `quiz2.html` later in this section.

The primary script ends in line 118, and the body of the XHTML document begins in line 120. Line 121 creates the link that calls function `allQuotes` when clicked. Lines 123–124 create a paragraph element containing the attribute `id = "quizSpot"`. This paragraph contains a link that calls function `openQuiz`.

Lines 126–133 contain a second script. This script appears in the XHTML document's `body` because it adds a dynamic footer to the page, which must appear after the static XHTML content contained in the first part of the `body`. This script creates another instance of the `Date` object, but the date is set to the last modified date and time of the XHTML document, rather than the current date and time (line 128). The script obtains the last modified date and time using property `document.lastModified`. Lines 131–132 add this information to the web page. Note that the last modified date and time appear at the bottom of the page, after the rest of the `body` content. If this script were in the `head` element, this information would be displayed before the entire `body` of the XHTML document. Lines 133–135 close the `script`, the `body` and the XHTML document.

The Quiz Page

The quiz used in this example is in a separate XHTML document named `quiz2.html` (Fig. 11.17). This document is similar to `quiz.html` in Fig. 10.14. The quiz in this example differs from the quiz in Fig. 10.14 in that it shows the result in the main window in the example, whereas the earlier quiz example alerts the result. After the **Submit** button in the quiz window is clicked, the main window changes to reflect that the quiz was taken, and the quiz window closes.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 11.17: quiz2.html -->
<!-- Online quiz in a child window. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Online Quiz</title>
      <script type = "text/JavaScript">
         <!--
         function checkAnswers()
         {
            // determine whether the answer is correct
            if ( document.getElementById( "myQuiz" ).elements[1].checked )
               window.opener.document.getElementById( "quizSpot" ).
                  innerHTML = "Congratulations, your answer is correct";
            else // if the answer is incorrect
               window.opener.document.getElementById( "quizSpot" ).
                  innerHTML = "Your answer is incorrect. " +
                  "Please try again <br /> <a href = " +
                  \"javascript:openQuiz()\">Please take our quiz</a>";

            window.opener.focus();
            window.close();
         } // end function checkAnswers
```

Fig. 11.17 | Online quiz in a child window. (Part 1 of 3.)

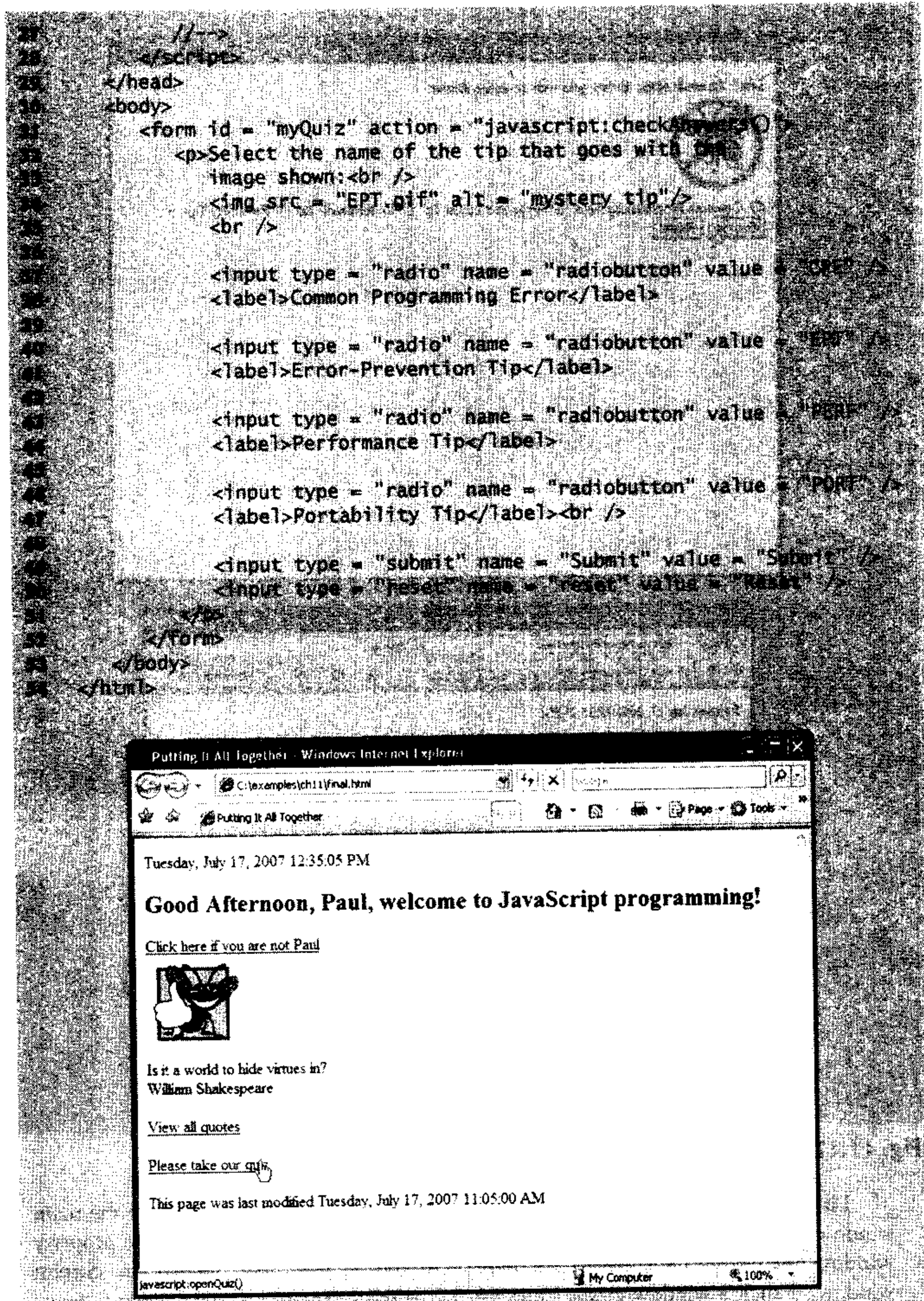


Fig. 11.17 | Online quiz in a child window. (Part 2 of 3.)

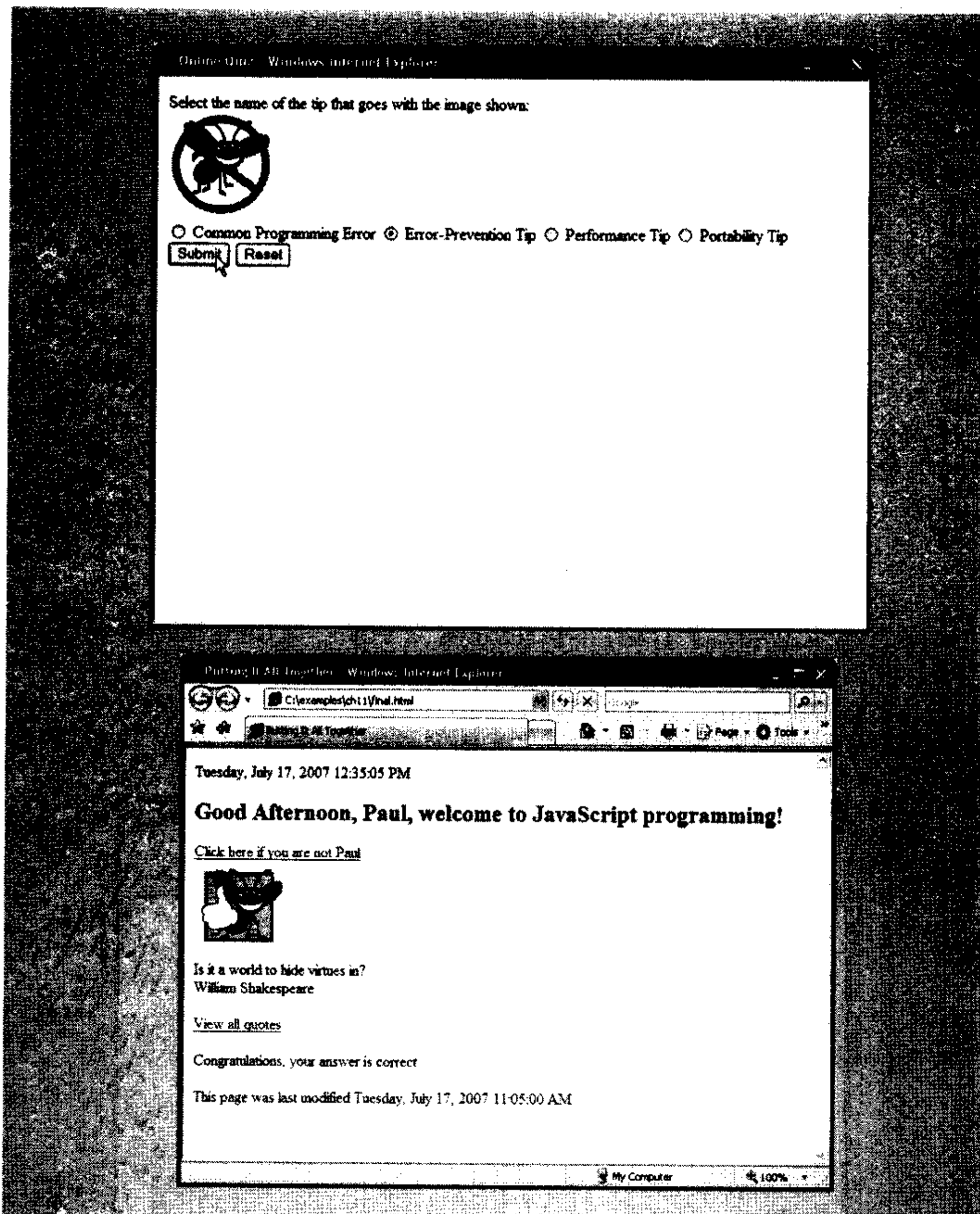


Fig. 11.17 | Online quiz in a child window. (Part 3 of 3.)

Lines 15–22 of this script check the user's answer and output the result to the main window. Lines 16–17 use `window.opener` to write to the main window. The property **window.opener** always contains a reference to the window that opened the current window, if such a window exists. Lines 16–17 write to property `window.opener.document.getElementById("quizSpot").innerHTML`. Recall that `quizSpot` is the `id` of the

paragraph in the main window that contains the link to open the quiz. Property **innerHTML** refers to the HTML code inside the quizSpot paragraph (i.e., the code between <p> and </p>). Modifying the innerHTML property dynamically changes the XHTML code in the paragraph. Thus, when lines 16–17 execute, the link in the main window disappears, and the string "Congratulations, your answer is correct." appears. Lines 19–22 modify window.opener.document.getElementById("quizSpot").innerHTML. Lines 19–22 use the same technique to display "Your answer is incorrect. Please try again", followed by a link to try the quiz again.

After checking the quiz answer, the script gives focus to the main window (i.e., puts the main window in the foreground, on top of any other open browser windows), using the method **focus** of the main window's window object. The property window.opener references the main window, so window.opener.focus() (line 24) gives the main window focus, allowing the user to see the changes made to the text of the main window's quizSpot paragraph. Finally, the script closes the quiz window, using method window.close (line 25).

Lines 28–29 close the script and head elements of the XHTML document. Line 30 opens the body of the XHTML document. The body contains the form, image, text labels and radio buttons that comprise the quiz. Lines 52–54 close the form, the body and the XHTML document.

11.11 Using JSON to Represent Objects

In 1999, **JSON (JavaScript Object Notation)**—a simple way to represent JavaScript objects as strings—was introduced as an alternative to XML as a data-exchange technique. JSON has gained acclaim due to its simple format, making objects easy to read, create and parse. Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

{ *propertyName1* : *value1*, *propertyName2* : *value2* }

Arrays are represented in JSON with square brackets in the following format:

[*value1*, *value2*, *value3*]

Each value can be a string, a number, a JSON object, true, false or null. To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries from Chapter 15:

```
[ { first: 'Cheryl', last: 'Black' },
  { first: 'James', last: 'Blue' },
  { first: 'Mike', last: 'Brown' },
  { first: 'Meg', last: 'Gold' } ]
```

JSON provides a straightforward way to manipulate objects in JavaScript, and many other programming languages now support this format. In addition to simplifying object creation, JSON allows programs to extract data easily and to efficiently transmit data across the Internet. JSON integrates especially well with Ajax applications, discussed in Chapter 15. See Section 15.7 for a more detailed discussion of JSON, as well as an Ajax-specific example. For more information on JSON, visit our JSON Resource Center at www.deitel.com/json.

11.12 Web Resources

www.deitel.com/javascript/
The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (www.deitel.com/xhtml/) and CSS 2.1 (www.deitel.com/css21/).

Summary

Section 11.1 Introduction

- The chapter describes several of JavaScript's built-in objects, which will serve as a basis for understanding browser objects in the chapters on Dynamic HTML.

Section 11.2 Introduction to Object Technology

- Objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents.
- JavaScript uses objects to perform many tasks and therefore is referred to as an object-based programming language.
- Objects have attributes and exhibit behaviors.
- Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics.
- OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects.
- OOD also models communication between objects.
- OOD encapsulates attributes and operations (behaviors) into objects.
- Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- The designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects).
- Some programming languages—like Java, Visual Basic, C# and C++—are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement object-oriented designs as working software systems.
- Languages like C are procedural, so programming tends to be action oriented.
- In procedural languages, the unit of programming is the function.
- In object-oriented languages, the unit of programming is the class from which objects are eventually instantiated. Classes contain functions that implement operations and data that implements attributes.

- Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs.
- Object-oriented programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients.
- The data components of a class are called properties.
- The function components of a class are called methods.
- The nouns in a system specification help you determine the set of classes from which objects are created that work together to implement the system.
- Classes are to objects as blueprints are to houses.
- Classes can have relationships with other classes. These relationships are called associations.
- Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components.
- With object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts.

Section 11.3 `Math` Object

- `Math` object methods allow you to perform many common mathematical calculations.
- An object's methods are called by writing the name of the object followed by a dot operator (.) and the name of the method. In parentheses following the method name is the argument (or a comma-separated list of arguments) to the method.

Section 11.4 `String` Object

- Characters are the fundamental building blocks of JavaScript programs. Every program is composed of a sequence of characters grouped together meaningfully that is interpreted by the computer as a series of instructions used to accomplish a task.
- A string is a series of characters treated as a single unit.
- A string may include letters, digits and various special characters, such as +, -, *, /, and $.
- JavaScript supports Unicode, which represents a large portion of the world's languages.
- String literals or string constants (often called anonymous `String` objects) are written as a sequence of characters in double quotation marks or single quotation marks.
- Combining strings is called concatenation.
- String method `charAt` returns the character at a specific index in a string. Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.
- `String` method `charCodeAt` returns the Unicode value of the character at a specific index in a string. If the index is outside the bounds of the string, the method returns `NaN`. `String` method `fromCharCode` creates a string from a list of Unicode values.
- `String` method `toLowerCase` returns the lowercase version of a string. `String` method `toUpperCase` returns the uppercase version of a string.
- `String` method `indexOf` determines the location of the first occurrence of its argument in the string used to call the method. If the substring is found, the index at which the first occurrence

of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.

- `String` method `lastIndexOf` determines the location of the last occurrence of its argument in the string used to call the method. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.
- The process of breaking a string into tokens is called tokenization. Tokens are separated from one another by delimiters, typically white-space characters such as blank, tab, newline and carriage return. Other characters may also be used as delimiters to separate tokens.
- `String` method `split` breaks a string into its component tokens. The argument to method `split` is the delimiter string—the string that determines the end of each token in the original string. Method `split` returns an array of strings containing the tokens.
- `String` method `substring` returns the substring from the starting index (its first argument) up to but not including the ending index (its second argument). If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string.
- `String` method `anchor` wraps the string that calls the method in XHTML element `<a></a>` with the name of the anchor supplied as the argument to the method.
- `String` method `fixed` displays text in a fixed-width font by wrapping the string that calls the method in a `<tt></tt>` XHTML element.
- `String` method `strike` displays struck-out text (i.e., text with a line through it) by wrapping the string that calls the method in a `<strike></strike>` XHTML element.
- `String` method `sub` displays subscript text by wrapping the string that calls the method in a `<sub></sub>` XHTML element.
- `String` method `sup` displays superscript text by wrapping the string that calls the method in a `<sup></sup>` XHTML element.
- `String` method `link` creates a hyperlink by wrapping the string that calls the method in XHTML element `<a></a>`. The target of the hyperlink (i.e, value of the `href` property) is the argument to the method and can be any URL.

Section 11.5 `Date` Object

- JavaScript's `Date` object provides methods for date and time manipulations.
- Date and time processing can be performed based either on the computer's local time zone or on World Time Standard's Coordinated Universal Time (abbreviated UTC)—formerly called Greenwich Mean Time (GMT).
- Most methods of the `Date` object have a local time zone and a UTC version.
- `Date` method `parse` receives as its argument a string representing a date and time and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.
- `Date` method `UTC` returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required year, month and date, and the optional hours, minutes, seconds and milliseconds. If any of the hours, minutes, seconds or milliseconds arguments is not specified, a zero is supplied in its place. For the hours, minutes and seconds arguments, if the argument to the right of any of these arguments is specified, that argument must also be specified (e.g., if the minutes argument is specified, the hours argument must be specified; if the milliseconds argument is specified, all the arguments must be specified).

Section 11.6 `Boolean` *and* `Number` *Objects*

- JavaScript provides the `Boolean` and `Number` objects as object wrappers for boolean `true`/`false` values and numbers, respectively.
- When a boolean value is required in a JavaScript program, JavaScript automatically creates a `Boolean` object to store the value.
- JavaScript programmers can create `Boolean` objects explicitly with the statement

  `var b = new Boolean( booleanValue );`

  The argument *booleanValue* specifies the value of the `Boolean` object (`true` or `false`). If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or the empty string (""), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`.
- JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program.
- JavaScript programmers can create a `Number` object with the statement

  `var n = new Number( numericValue );`

  The argument *numericValue* is the number to store in the object. Although you can explicitly create `Number` objects, normally they are created when needed by the JavaScript interpreter.

Section 11.7 `document` *Object*

- JavaScript provides the `document` object for manipulating the document that is currently visible in the browser window.

Section 11.8 `window` *Object*

- JavaScript's `window` object provides methods for manipulating browser windows.

Section 11.9 Using Cookies

- A cookie is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions.
- Cookies are accessible in JavaScript through the `document` object's `cookie` property.
- A cookie has the syntax "*identifier=value*," where *identifier* is any valid JavaScript variable identifier, and *value* is the value of the cookie variable. When multiple cookies exist for one website, *identifier-value* pairs are separated by semicolons in the `document.cookie` string.
- The `expires` property in a cookie string sets an expiration date, after which the web browser deletes the cookie. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.
- The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it.

Section 11.10 Final JavaScript Example

- `window.opener` always contains a reference to the window that opened the current window.
- The property `innerHTML` refers to the HTML code inside the current paragraph element.
- Method `focus` puts the window it references on top of all the others.
- The `window` object's `close` method closes the browser window represented by the `window` object.

Section 11.11 Using JSON to Represent Objects

- JSON (JavaScript Object Notation) is a simple way to represent JavaScript objects as strings.
- JSON was introduced in 1999 as an alternative to XML for data exchange.

- Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

  { *propertyName1* : *value1*, *propertyName2* : *value2* }

- Arrays are represented in JSON with square brackets in the following format:

  [*value1*, *value2*, *value3*]

- Values in JSON can be strings, numbers, JSON objects, `true`, `false` or `null`.

Terminology

`abs` method of `Math`
abstraction
action-oriented programming language
`anchor` method of `String`
anonymous `String` object
association
attribute (property)
behavior (method)
`blink` method of `String`
`Boolean` object
`ceil` method of `Math`
character
`charAt` method of `String`
`charCodeAt` method of `String`
class
`close` method of `window`
[illegible]
components
`concat` method of `String`
cookie
Coordinated Universal Time (UTC)
`cos` method of `Math`
date
`Date` object
delimiter
`document` object
`E` property of `Math`
empty string
encapsulation
`escape` function
`exp` method of `Math`
`fixed` method of `String`
`floor` method of `Math`
`focus` method of `window`
`fromCharCode` method of `String`
`getDate` method of `Date`
`getDay` method of `Date`
`getFullYear` method of `Date`
`getHours` method of `Date`
`getMilliseconds` method of `Date`
`getMinutes` method of `Date`
`getMonth` method of `Date`
`getSeconds` method of `Date`
`getTime` method of `Date`
`getTimezoneOffset` method of `Date`
`getUTCDate` method of `Date`
`getUTCDay` method of `Date`
`getUTCFullYear` method of `Date`
`getUTCHours` method of `Date`
`getUTCMilliseconds` method of `Date`
`getUTCMinutes` method of `Date`
`getUTCMonth` method of `Date`
`getUTCSeconds` method of `Date`
Greenwich Mean Time (GMT)
hexadecimal escape sequences
hiding
index in a string
`indexOf` method of `String`
information hiding
inheritance
`innerHTML` property
instantiation
interface
`lastIndexOf` method of `String`
`link` method of `String`
`LN10` property of `Math`
`LN2` property of `Math`
local time zone
`log` method of `Math`
`LOG10E` property of `Math`
`LOG2E` property of `Math`
`Math` object
`max` method of `Math`
`MAX_SIZE` property of `Number`
method
`min` method of `Math`
`MIN_SIZE` property of `Number`
`NaN` property of `Number`
`NEGATIVE_INFINITY` property of `Number`
`Number` object

object
object wrapper
object-based programming language
object-oriented design (OOD)
object-oriented programming (OOP)
open method of window
opener property of window
operation
parse method of Date
PI property of Math
POSITIVE_INFINITY property of Number
pow method of Math
property
round method of Math
[illegible]
sqrt method of Math
SQRT1_2 property of Math
SQRT2 property of Math
[illegible] method of String
string
string constant
string literal
String object
sub method of String
substr method of String
substring
substring method of String
sup method of String
tan method of Math
[illegible]
[illegible]
tokenization
toLocaleString method of Date
toLowerCase method of String
toString method of Date
toString method of String
toUpperCase method of String
toUTCString method of Date
[illegible] function
Unicode
user-defined type
UTC (Coordinated Universal Time)
UTC method of Date
valueOf method of Boolean
valueOf method of Date
valueOf method of Number
valueOf method of String
well-defined interface
window object
write to XHTML page

[illegible]

[illegible] JavaScript is commonly referred [illegible]

[illegible] allow you to perform many common mathematical [illegible]

[illegible] is referred to as ______.

[illegible] a sequence of characters in ______ of [illegible]

f) Indices for the characters in a string start at ________.
g) `String` methods ________ and ________ search for the first and last occurrences of a substring in a `String`, respectively.
h) The process of breaking a string into tokens is called ________.
i) `String` method ________ formats a `String` as a hyperlink.
j) Date and time processing can be performed based on the ________ or on World Time Standard's ________.
k) `Date` method ________ receives as its argument a string representing a date and time, and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.

Exercises

11.2 Create a web page that contains four XHTML buttons. Each button, when clicked, should cause an alert dialog to display a different time or date in relation to the current time. Create a `Now` button that alerts the current time and date and a `Yesterday` button that alerts the time and date 24 hours ago. The other two buttons should alert the time and date ten years ago and one week from today.

11.3 `Math` method `floor` may be used to round a number to a specific decimal place. For example, the statement

```
y = Math.floor( x * 10 + .5 ) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = Math.floor( x * 100 + .5 ) / 100;
```

rounds `x` to the hundredths position (i.e., the second position to the right of the decimal point). Write a script that defines four functions to round a number `x` in various ways:

a) `roundToInteger( number )`
b) `roundToTenths( number )`
c) `roundToHundredths( number )`
d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

11.4 Write a script that uses relational and equality operators to compare two `String`s input by the user through an XHTML form. Output in an XHTML `textarea` whether the first string is less than, equal to or greater than the second.

11.5 Write a script that uses random number generation to create sentences. Use four arrays of strings called `article`, `noun`, `verb` and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period.

The arrays should be filled as follows: the `article` array should contain the articles `"the"`, `"a"`, `"one"`, `"some"` and `"any"`; the `noun` array should contain the nouns `"boy"`, `"girl"`, `"dog"`, `"town"` and `"car"`; the `verb` array should contain the verbs `"drove"`, `"jumped"`, `"ran"`, `"walked"` and `"skipped"`; the `preposition` array should contain the prepositions `"to"`, `"from"`, `"over"`, `"under"` and `"on"`.

The program should generate 20 sentences to form a short story and output the result to an XHTML `textarea`. The story should begin with a line reading `"Once upon a time..."` and end with a line reading `"THE END"`.

11.6 *(Pig Latin)* Write a script that encodes English-language phrases in pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:

To form a pig Latin phrase from an English-language phrase, tokenize the phrase into an array of words using `String` method `split`. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus the word "`jump`" becomes "`umpjay`," the word "the" becomes "`hetay`" and the word "`computer`" becomes "`omputercay`." Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. Each token (i.e., word in the sentence) is passed to method `printLatinWord` to print the pig Latin word. Enable the user to input the sentence through an XHTML form. Keep a running display of all the converted sentences in an XHTML `textarea`.

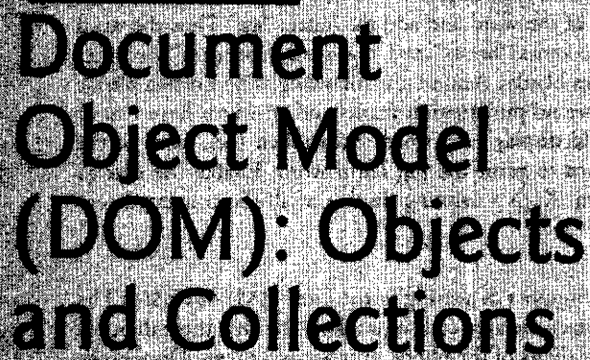
11.7 Write a script that inputs a telephone number as a string in the form `(555) 555-5555`. The script should use `String` method `split` to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. Display the area code in one text field and the seven-digit phone number in another text field.

11.8 Write a script that inputs a line of text, tokenizes it with `String` method `split` and outputs the tokens in reverse order.

11.9 Write a script that inputs text from an XHTML form and outputs it in uppercase and lowercase letters.

11.10 Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word "bathe" include the commonly used words "ate," "bat," "bet," "tab," "hat," "the" and "tea." Output the results in an XHTML `textarea`.

11.11 *(Printing Dates in Various Formats)* Dates are printed in several common formats. Write a script that reads a date from an XHTML form and creates a `Date` object in which to store it. Then use the various methods of the `Date` object that convert `Dates` into strings to display the date in several formats.

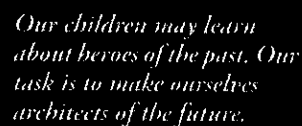


Document Object Model (DOM): Objects and Collections

OBJECTIVES

In this chapter you will learn:

- How to use JavaScript and the W3C Document Object Model to create dynamic web pages.
- The concept of DOM nodes and DOM trees.
- How to traverse, edit and modify elements in an XHTML document.
- How to change CSS styles dynamically.
- To create JavaScript animations.



Our children may learn about heroes of the past. Our task is to make ourselves architects of the future.
—Jomo Mzee Kenyatta

Though leaves are many, the root is one.
—William Butler Yeats

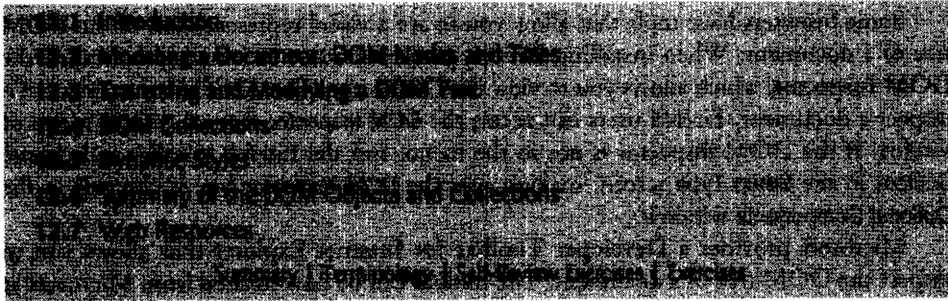
The thing that impresses me most about America is the way parents obey their children.
—Duke of Windsor

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

To write it, it took three months; to conceive it three minutes; to collect the data in it—all my life.
—F. Scott Fitzgerald

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

Outline



12.1 Introduction

In this chapter we introduce the **Document Object Model (DOM)**. The DOM gives you 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 create, modify and remove elements in the page dynamically.

Previously, both Internet Explorer and Netscape had different versions of Dynamic HTML, which provided similar functionality to the DOM. However, while they provided many of the same capabilities, these two models were incompatible with each other. In an effort to encourage cross-browser websites, the W3C created the standardized Document Object Model. Firefox 2, Internet Explorer 7, and most other major browsers implement *most* of the features of the W3C DOM.

This chapter begins by formally introducing the concept of DOM nodes and DOM trees. We then discuss properties and methods of DOM nodes and cover additional methods of the `document` object. We also discuss how to dynamically change style properties, which enables you to create many types of effects, such as user-defined background colors and animations. Then, we present a diagram of the extensive object hierarchy, with explanations of the various objects and properties, and we provide links to websites with further information on the topic.

Software Engineering Observation 12.1

With the DOM, XHTML elements can be treated as objects, and many attributes of XHTML elements can be treated as properties of those objects. Then, objects can be scripted (through their `id` *attributes) 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. In this section and the next, we discuss more thoroughly the objects returned by this method.

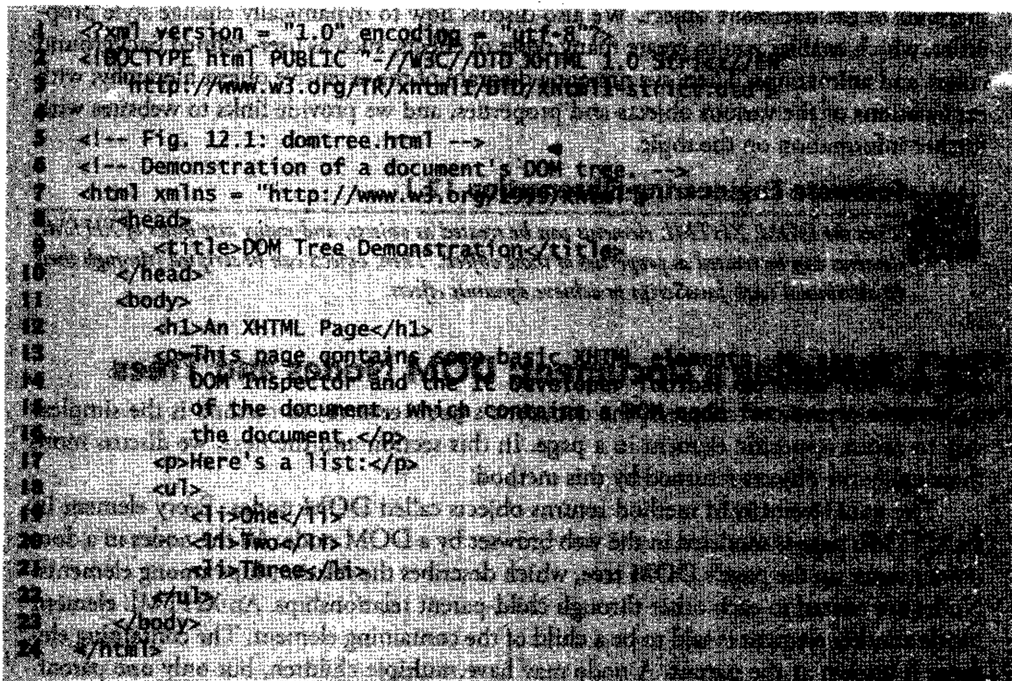
The `getElementById` method returns objects called **DOM nodes**. Every element in an XHTML page is modeled in the web browser by a DOM node. All the nodes in a document make up the page's **DOM tree**, which describes the relationships among elements. Nodes are related to each other through child-parent relationships. An XHTML element inside another element is said to be a **child** of the containing element. The containing element is known as the **parent**. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as **siblings**.

Some browsers have tools that allow you to see a visual representation of the DOM tree of a document. When installing Firefox, you can choose to install a tool called the **DOM Inspector**, which allows you to view the DOM tree of an XHTML document. To inspect a document, Firefox users can access the **DOM Inspector** from the **Tools** menu of Firefox. If the DOM inspector is not in the menu, run the Firefox installer and choose **Custom** in the **Setup Type** screen, making sure the **DOM Inspector** box is checked in the **Optional Components** window.

Microsoft provides a **Developer Toolbar** for Internet Explorer that allows you to inspect the DOM tree of a document. The toolbar can be downloaded from Microsoft at `go.microsoft.com/fwlink/?LinkId=92716`. Once the toolbar is installed, restart the browser, then click the » icon at the right of the toolbar and choose **IE Developer Toolbar** from the menu. Figure 12.1 shows an XHTML document and its DOM tree displayed in Firefox's DOM Inspector and in IE's Web Developer Toolbar.

The XHTML document contains a few simple elements. We explain the example based on the Firefox DOM Inspector—the IE Toolbar displays the document with only minor differences. A node can be expanded and collapsed using the + and - buttons next to the node's name. Figure 12.1(b) shows all the nodes in the document fully expanded. The document node (shown as **#document**) at the top of the tree is called the **root node**, because it has no parent. Below the document node, the **HTML** node is indented from the document node to signify that the **HTML** node is a child of the **#document** node. The **HTML** node represents the `html` element (lines 7–24).

The **HEAD** and **BODY** nodes are siblings, since they are both children of the **HTML** node. The **HEAD** contains two **#comment** nodes, representing lines 5–6. The **TITLE** node



```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 12.1: domtree.html -->
6  <!-- Demonstration of a document's DOM tree. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>DOM Tree Demonstration</title>
10    </head>
11    <body>
12       <h1>An XHTML Page</h1>
13       <p>This page contains some basic XHTML [illegible]
14          DOM Inspector and the IE Developer [illegible]
15          of the document, which contains a [illegible]
16          the document.</p>
17       <p>Here's a list:</p>
18       <ul>
19          <li>One</li>
20          <li>Two</li>
21          <li>Three</li>
22       </ul>
23    </body>
24 </html>
```

Fig. 12.1 | Demonstration of a document's DOM tree. (Part 1 of 3.)

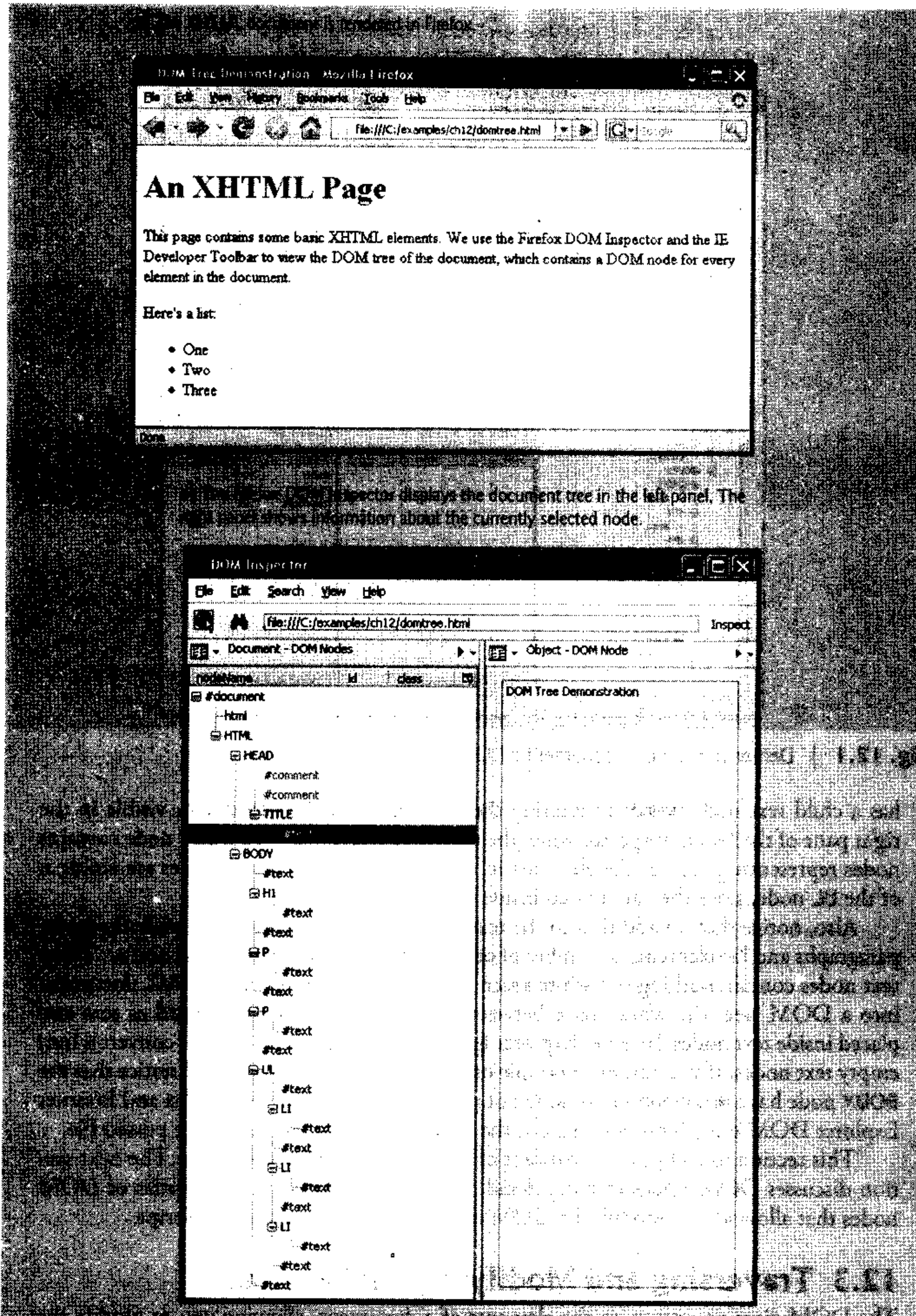


Fig. 12.1 | Demonstration of a document's DOM tree. (Part 2 of 3.)

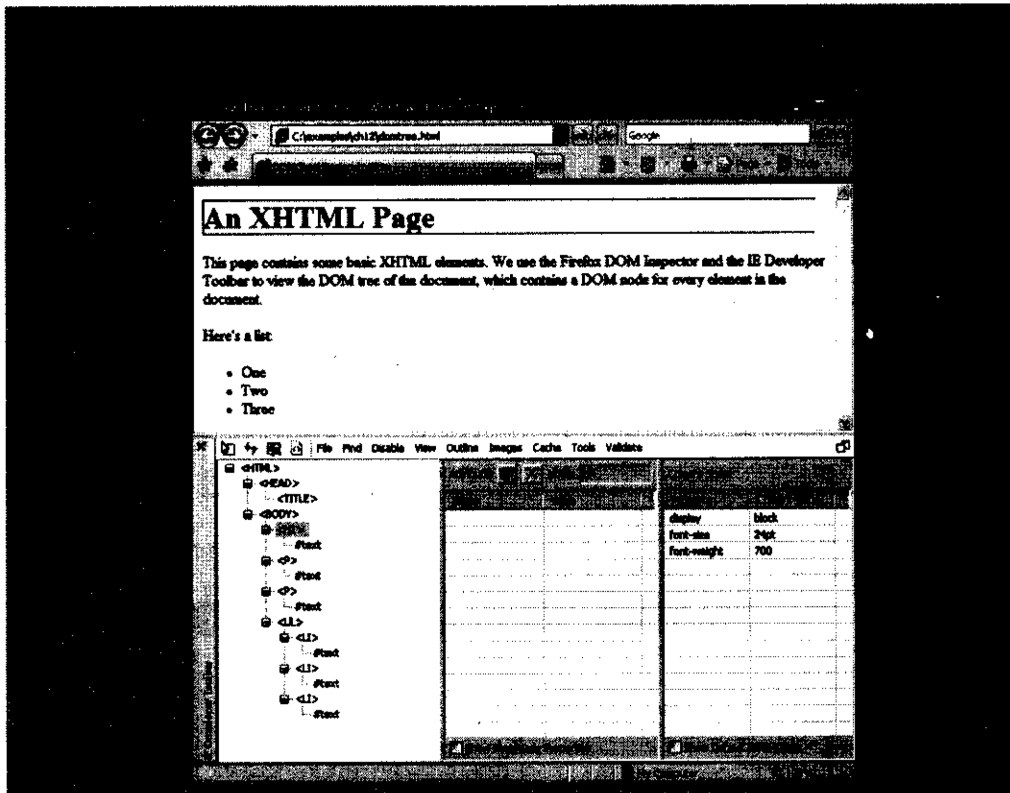


Fig. 12.1 | Demonstration of a document's DOM tree. (Part 3 of 3.)

has a child text node (**#text**) containing the text **DOM Tree Demonstration**, visible in the right pane of the DOM inspector when the text node is selected. The **BODY** node contains nodes representing each of the elements in the page. Note that the **LI** nodes are children of the **UL** node, since they are nested inside it.

Also, notice that, in addition to the text nodes representing the text inside the body, paragraphs and list elements, a number of other text nodes appear in the document. These text nodes contain nothing but white space. When Firefox parses an XHTML document into a DOM tree, the white space between sibling elements is interpreted as text and placed inside text nodes. Internet Explorer ignores white space and does not convert it into empty text nodes. If you run this example on your own computer, you will notice that the **BODY** node has a **#comment** child node not present above in both the Firefox and Internet Explorer DOM trees. This is a result of the copyright line at the end of the posted file.

This section introduced the concept of DOM nodes and DOM trees. The next section discusses DOM nodes in more detail, discussing methods and properties of DOM nodes that allow you to modify the DOM tree of a document using JavaScript.

12.3 Traversing and Modifying a DOM Tree

The DOM gives you access to the elements of a document, allowing you to modify the contents of a page dynamically using event-driven JavaScript. This section introduces

properties and methods of all DOM nodes that enable you to traverse the DOM tree, modify nodes and create or delete content dynamically.

Figure 12.2 showcases some of the functionality of DOM nodes, as well as two additional methods of the `document` object. The program allows you to highlight, modify, insert and remove elements.

Lines 117–132 contain basic XHTML elements and content. Each element has an `id` attribute, which is also displayed at the beginning of the element in square brackets. For example, the `id` of the `h1` element in lines 117–118 is set to `bigheading`, and the heading text begins with `[bigheading]`. This allows the user 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 133–162) contains the remainder of the XHTML `body`. Line 134 begins a `form` element, assigning the empty string to the required `action` attribute (because we're not submitting to a server) and returning `false` to the `onsubmit` attribute. When a `form`'s `onsubmit` handler returns `false`, the navigation to the address specified in the `action` attribute is aborted. This allows us to modify the page using JavaScript event handlers without reloading the original, unmodified XHTML.

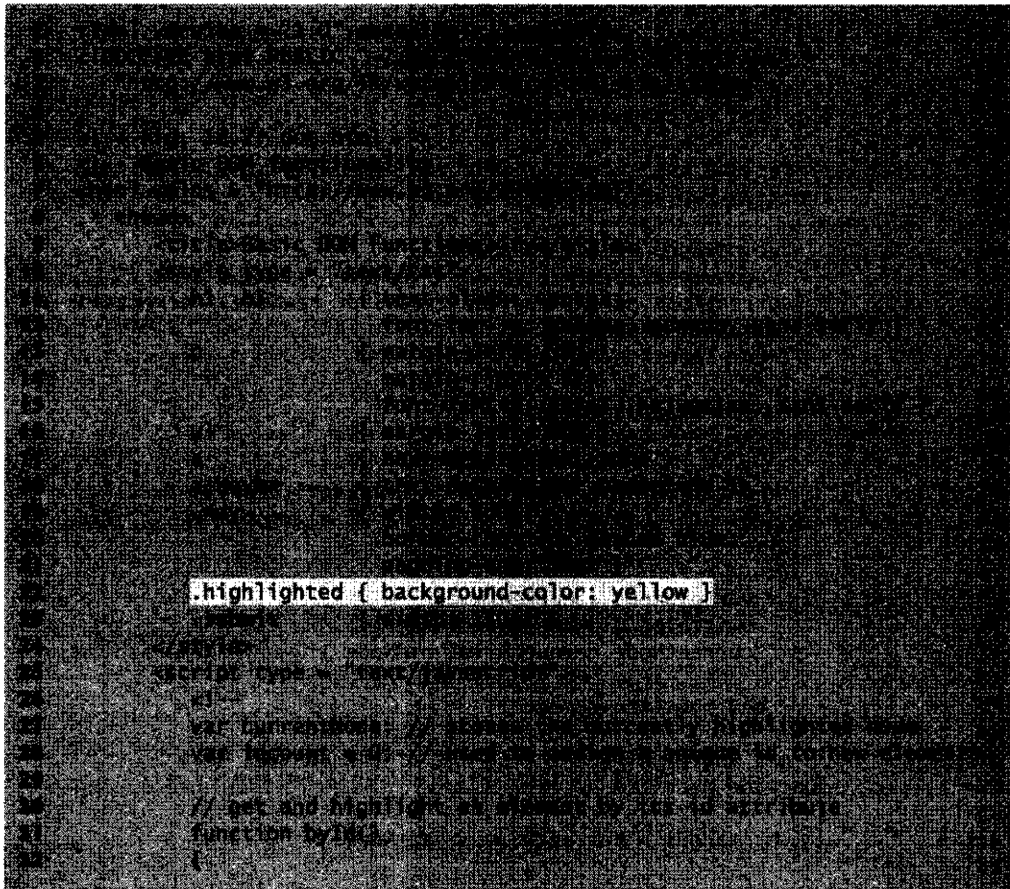


Fig. 12.2 | Basic DOM functionality. (Part 1 of 8.)

```
33          var id = document.getElementById( "gbi" ).value;
34          var target = document.getElementById( id );
35
36          if ( target )
37             switchTo( target );
38       } // end function byId
39
40       // insert a paragraph element before the current element
41       // using the insertBefore method
42       function insert()
43       {
44          var newNode = createNewNode(
45             document.getElementById( "ins" ).value );
46          currentNode.parentNode.insertBefore( newNode, currentNode );
47          switchTo( newNode );
48       } // end function insert
49
50       // append a paragraph node as the child of the current node
51       function appendNode()
52       {
53          var newNode = createNewNode(
54             document.getElementById( "append" ).value );
55          currentNode.appendChild( newNode );
56          switchTo( newNode );
57       } // end function appendNode
58
59       // replace the currently selected node with a paragraph node
60       function replaceCurrent()
61       {
62          var newNode = createNewNode(
63             document.getElementById( "replace" ).value );
64          currentNode.parentNode.replaceChild( newNode, currentNode );
65          switchTo( newNode );
66       } // end function replaceCurrent
67
68       // remove the current node
69       function remove()
70       {
71          if ( currentNode.parentNode == document.body )
72             alert( "Can't remove a top-level element." );
73          else
74          {
75             var oldNode = currentNode;
76             switchTo( oldNode.parentNode );
77             currentNode.removeChild( oldNode );
78          }
79       } // end function remove
80
81       // get and highlight the parent of the current node
82       function parent()
83       {
84          var target = currentNode.parentNode;
85
```

Fig. 12.2 | Basic DOM functionality. (Part 2 of 8.)

```
var newNode = document.createElement( "p" );

newNode.appendChild(document.createTextNode( text ) );

currentNode.className = "";

currentNode.id
```

Fig. 12.2 | Basic DOM functionality. (Part 3 of 8.)

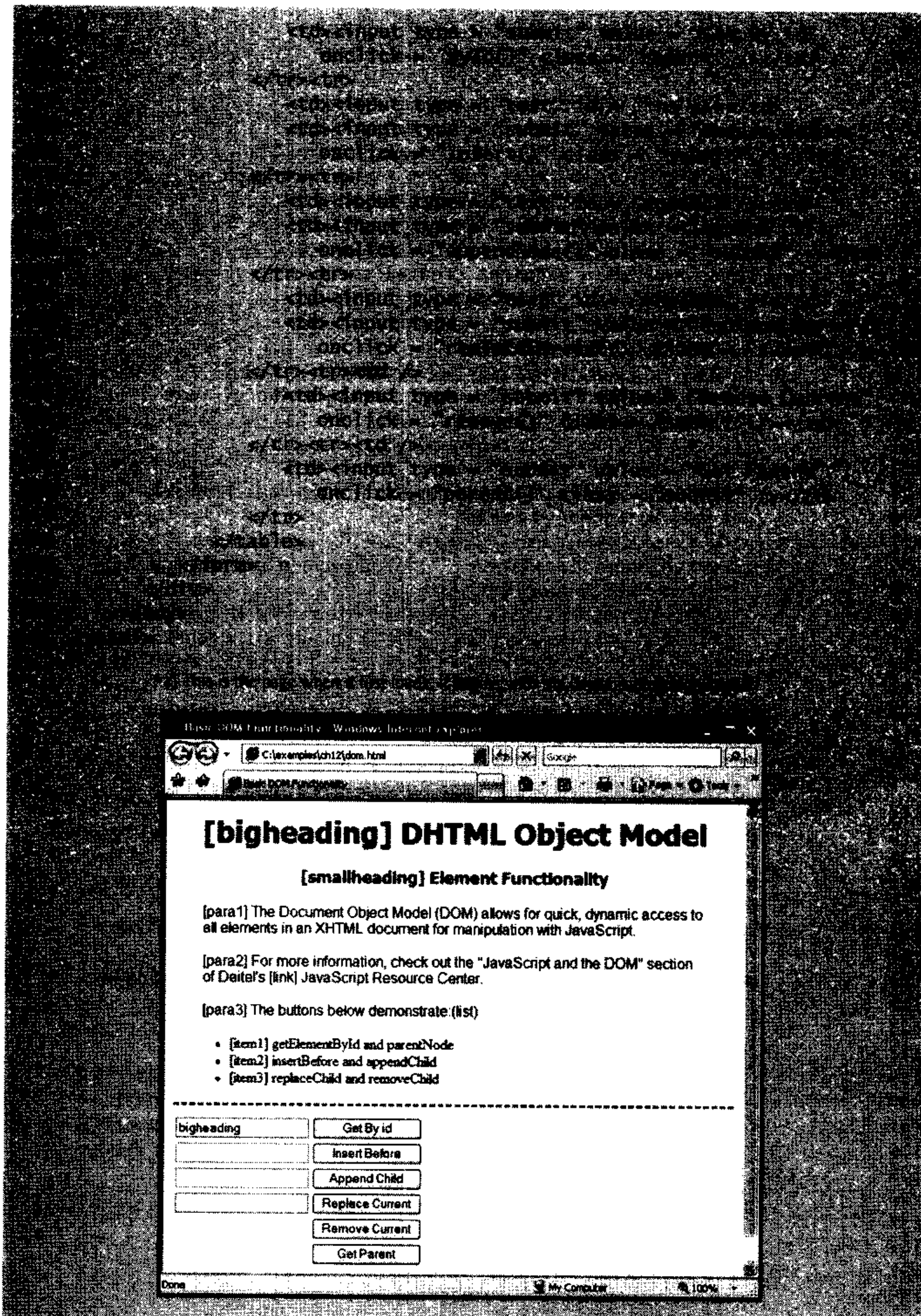


Fig. 12.2 | Basic DOM functionality. (Part 4 of 8.)

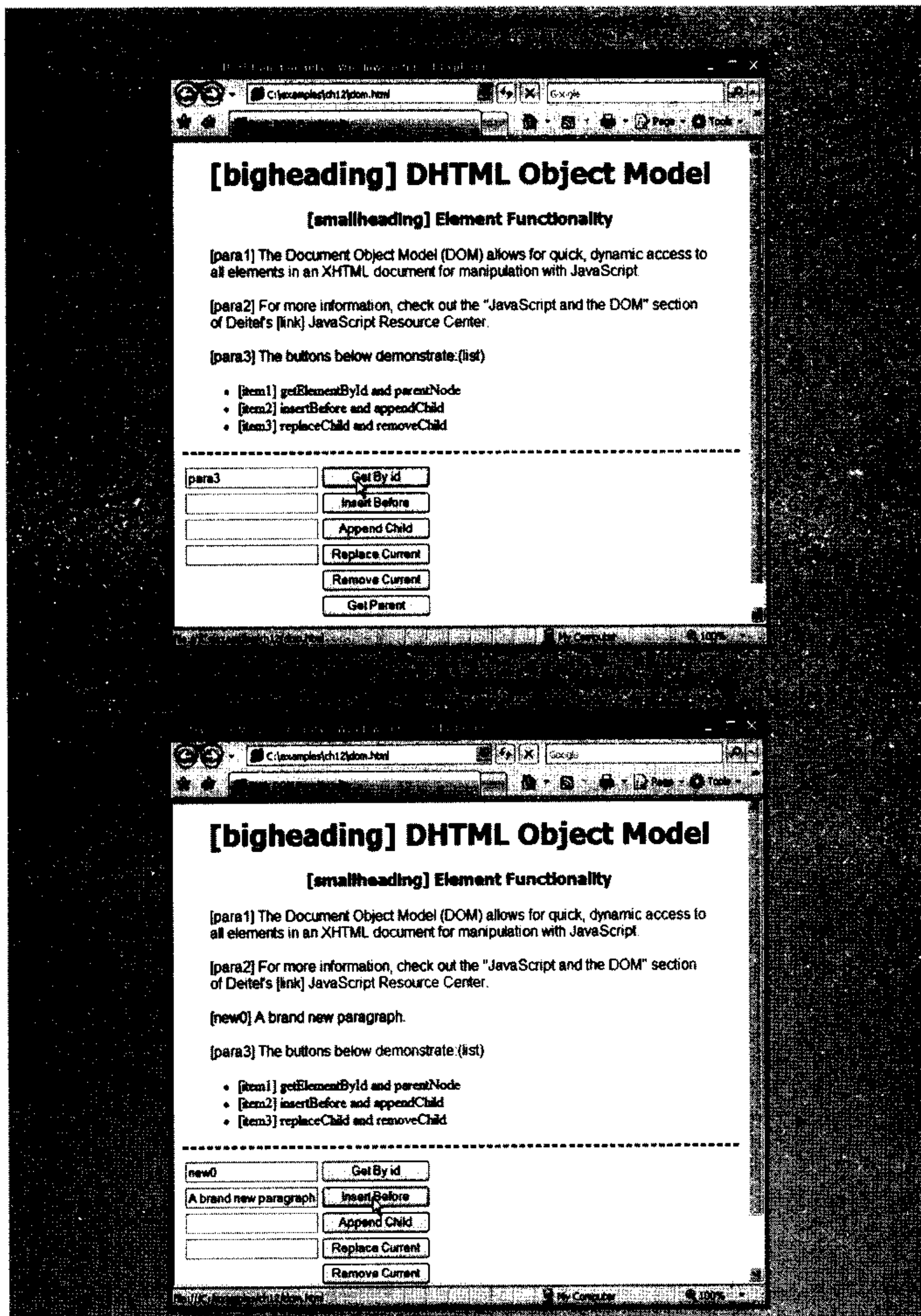


Fig. 12.2 | Basic DOM functionality. (Part 5 of 8.)

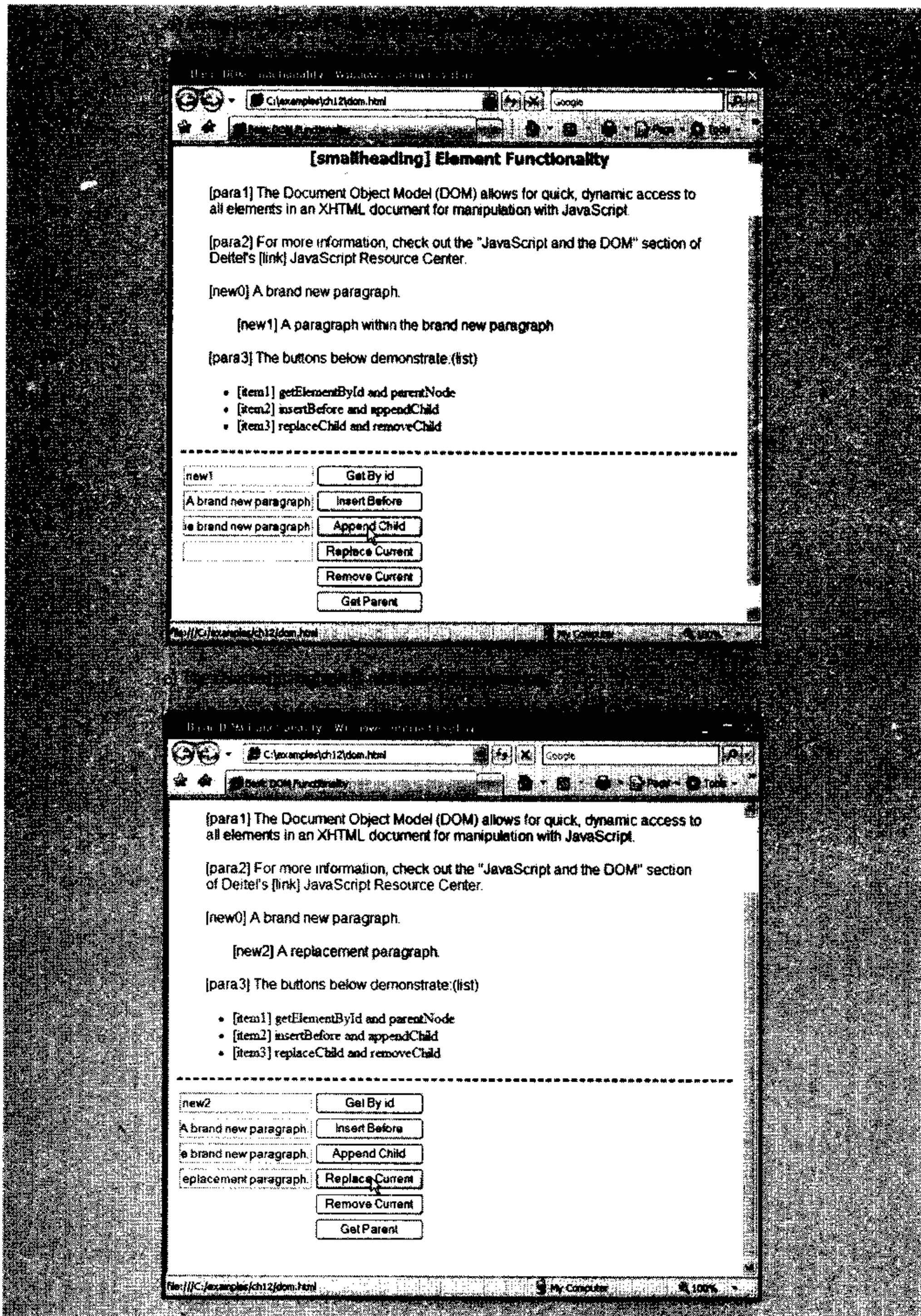


Fig. 12.2 | Basic DOM functionality. (Part 6 of 8.)

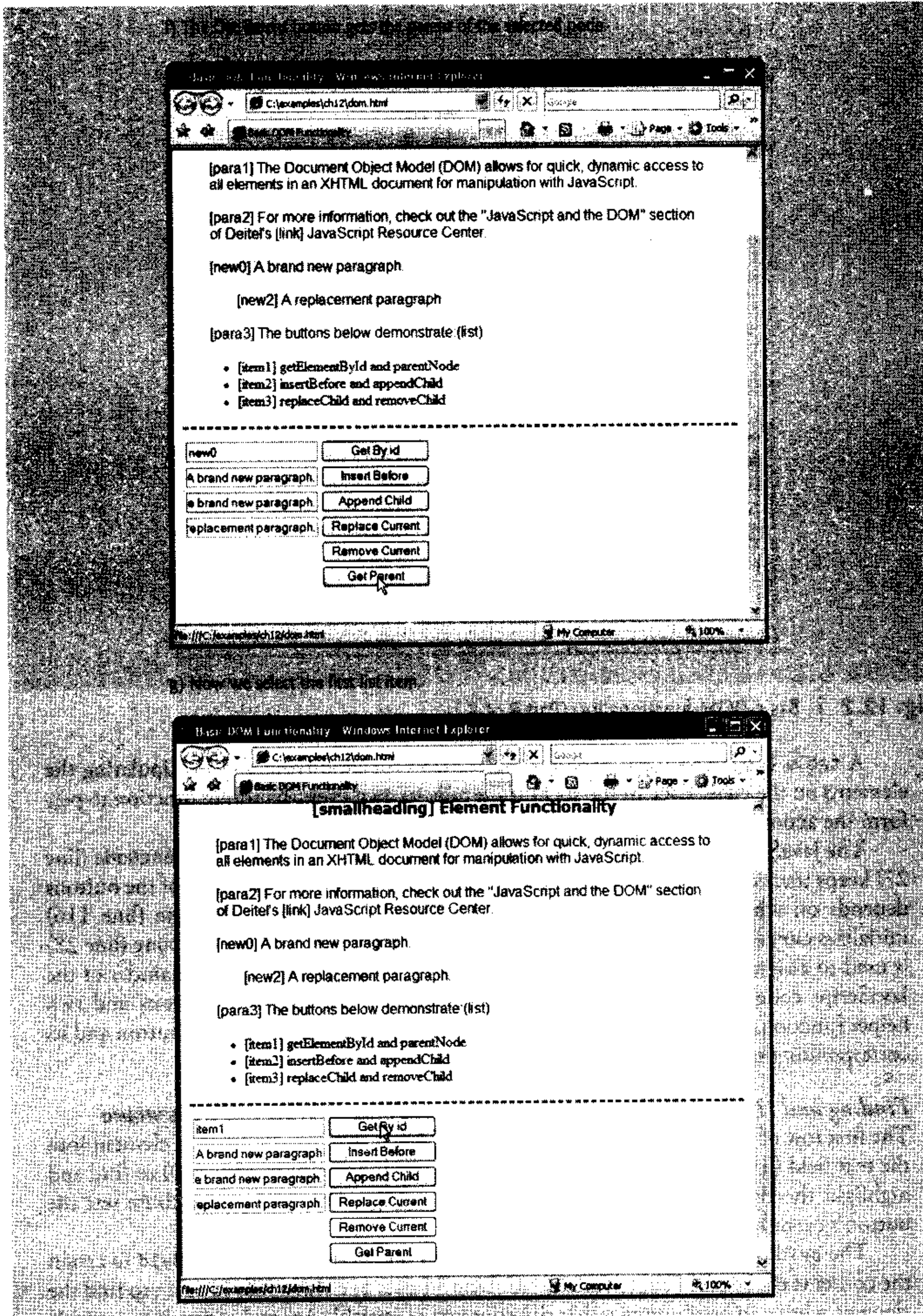


Fig. 12.2 | Basic DOM functionality. (Part 7 of 8.)

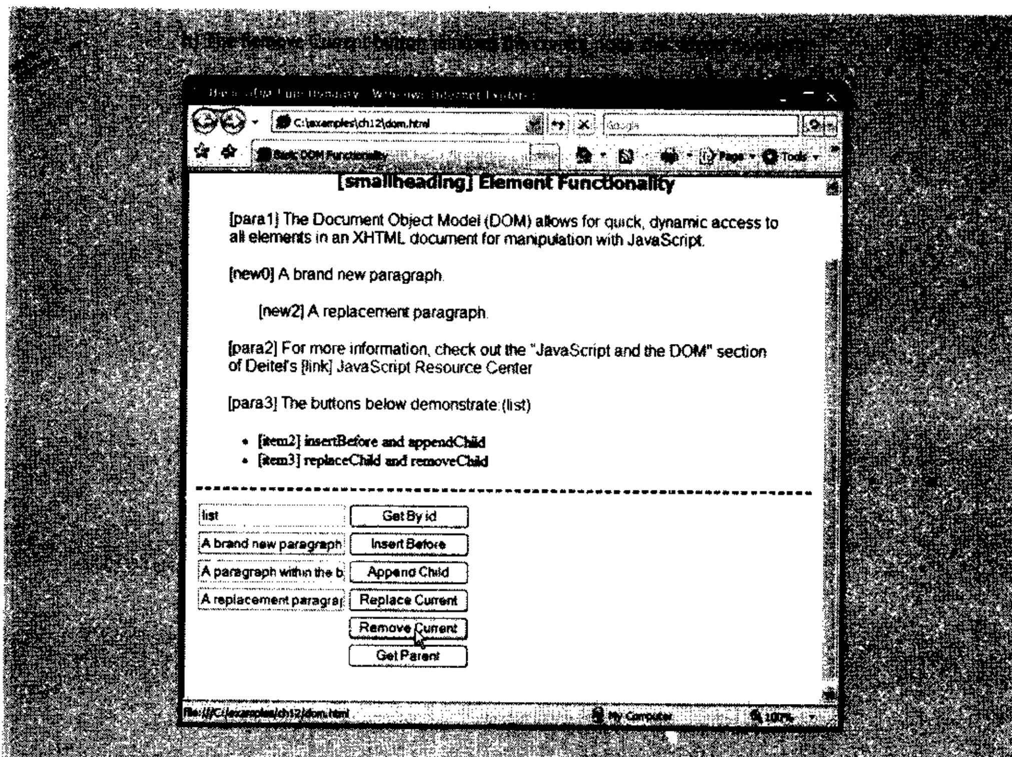


Fig. 12.2 | Basic DOM functionality. (Part 8 of 8.)

A `table` (lines 135–160) contains the controls for modifying and manipulating the elements on the page. Each of the six buttons calls its own event-handling function to perform the action described by its `value`.

The JavaScript code begins by declaring two variables. The variable `currentNode` (line 27) keeps track of the currently highlighted node, because the functionality of the buttons depends on which node is currently selected. The `body`'s `onload` attribute (line 116) initializes `currentNode` to the `h1` element with `id` `bigheading`. Variable `idcount` (line 28) 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 XHTML buttons and two helper functions that are called by the event handlers. We now discuss each button and its corresponding event handler in detail.

Finding and Highlighting an Element Using `getElementById` *and* `className`

The first row of the table (lines 136-141) allows the user to enter the `id` of an element into the text field (lines 137–138) and click the `Get By Id` button (lines 139–140) to find and highlight the element, as shown in Fig. 12.2(b) and (g). The `onclick` attribute sets the button's event handler to function `byId`.

The `byId` function is defined in lines 31–38. Line 33 uses `getElementById` to assign the contents of the text field to variable `id`. Line 34 uses `getElementById` again to find the element whose `id` attribute matches the contents of variable `id`, and assign it to variable `target`. If an element is found with the given `id`, `getElementById` returns an object

representing that element. If no element is found, `getElementById` returns `null`. Line 36 checks whether `target` is an object—recall that any object used as a boolean expression is `true`, while `null` is `false`. If `target` evaluates to `true`, line 37 calls the `switchTo` function with `target` as its argument.

The `switchTo` function, defined in lines 106–112, is used throughout the program to highlight a new element in the page. The current element is given a yellow background using the style class `highlighted`, defined in line 22. Line 108 sets the current node's `className` property to the empty string. The **`className` property** allows you to change an XHTML element's `class` attribute. In this case, we clear the `class` attribute in order to remove the `highlighted` class from the `currentNode` before we highlight the new one.

Line 109 assigns the `newNode` object (passed into the function as a parameter) to variable `currentNode`. Line 110 adds the `highlighted` style class to the new `currentNode` using the `className` property.

Finally, line 111 uses the **`id` property** to assign the current node's `id` to the input field's value property. Just as `className` allows access to an element's `class` attribute, the `id` property controls an element's `id` attribute. While this isn't necessary when `switchTo` is called by `byId`, we will see shortly that other functions call `switchTo`. This line makes sure that the text field's value is consistent with the currently selected node's `id`. Having found the new element, removed the highlighting from the old element, updated the `currentNode` variable and highlighted the new element, the program has finished selecting a new node by a user-entered `id`.

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

The next two table rows allow the user to create a new element and insert it before the current node or as a child of the current node. The second row (lines 141–145) allows the user to enter text into the text field and click the `Insert Before` button. The text is placed in a new paragraph element, which is then inserted into the document before the currently selected element, as in Fig. 12.2(c). The button in lines 143–144 calls the `insert` function, defined in lines 42–48.

Lines 44–45 call the function `createNewNode`, passing it the value of the input field (whose `id` is `ins`) as an argument. Function `createNewNode`, defined in lines 94–103, creates a paragraph node containing the text passed to it. Line 96 creates a `p` element using the `document` object's **`createElement` method**. The `createElement` method 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.

Line 97 creates a unique `id` for the new element by concatenating `"new"` and the value of `idcount` before incrementing `idcount` in line 98. Line 99 assigns the `id` to the new element. Line 100 concatenates the element's `id` in square brackets to the beginning of `text` (the parameter containing the paragraph's text).

Line 101 introduces two new methods. The `document`'s **`createTextNode` method** creates a node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node. In line 101, we create a new text node containing the contents of variable `text`. This new node is then used (still in line 101) as the argument to the **`appendChild` method**, which is called on the paragraph node. Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.

After the p element is created, line 102 returns the node to the calling function `insert`, where it is assigned to variable `newNode` in lines 44–45. Line 46 inserts the newly created node before the currently selected node. The **`parentNode` property** of any DOM node contains the node's parent. In line 46, we use the `parentNode` property of `currentNode` to get its parent.

We call the `insertBefore` method (line 46) on the parent with `newNode` and `currentNode` as its arguments to insert `newNode` as a child of the parent directly before `currentNode`. The general syntax of the **`insertBefore` method** is

```
parent.insertBefore( newChild, existingChild );
```

The method is called on a parent with the new child and an existing child as arguments. The node *newChild* is inserted as a child of *parent* directly before *existingChild*. Line 47 uses the `switchTo` function (discussed earlier in this section) to update the `currentNode` to the newly inserted node and highlight it in the XHTML page.

The third table row (lines 145–149) allows the user to append a new paragraph node as a child of the current element, demonstrated in Fig. 12.2(d). This features uses a similar procedure to the `insertBefore` functionality. Lines 53–54 in function `appendNode` create a new node, line 55 inserts it as a child of the current node, and line 56 uses `switchTo` to update `currentNode` and highlight the new node.

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

The next two table rows (lines 149–156) allow the user to replace the current element with a new p element or simply remove the current element. Lines 150–152 contain a text field and a button that replaces the currently highlighted element with a new paragraph node containing the text in the text field. This feature is demonstrated in Fig. 12.2(e).

The button in lines 151–152 calls function `replaceCurrent`, defined in lines 60–66. Lines 62–63 call `createNewNode`, in the same way as in `insert` and `appendNode`, getting the text from the correct input field. Line 64 gets the parent of `currentNode`, then calls the `replaceChild` method on the parent. The **`replaceChild` method** works as follows:

```
parent.replaceChild( newChild, oldChild );
```

The *parent*'s `replaceChild` method inserts *newChild* into its list of children in place of *oldChild*.

The `Remove Current` feature, shown in Fig. 12.2(h), removes the current element entirely and highlights the parent. No text field is required because a new element is not being created. The button in lines 154-155 calls the `remove` function, defined in lines 69–79. If the node's parent is the body element, line 72 alerts an error—the program does not allow the entire body element to be selected. Otherwise, lines 75–77 remove the current element. Line 75 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 76 calls `switchTo` to highlight the parent node.

Line 77 uses the **`removeChild` method** to remove the `oldNode` (a child of the new `currentNode`) from its place in the XHTML document. In general,

```
parent.removeChild( child );
```

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

The final button (lines 157–158) selects and highlights the parent element of the currently highlighted element by calling the parent function, defined in lines 82–90. Function parent simply gets the parent node (line 84), makes sure it is not the body element, (line 86) and calls switchTo to highlight it (line 87). Line 89 alerts an error if the parent node is the body element. This feature is shown in Fig. 12.2(f).

This section introduced the basics of DOM tree traversal and manipulation. Next, we introduce the concept of collections, which give you access to multiple elements in a page.

12.4 DOM Collections

Included in the Document Object Model is the notion of **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** and **anchors collection**. These collections contain all the elements of the corresponding type on the page. Figure 12.3 gives an example that uses the links collection to extract all of the links on a page and display them together at the bottom of the page.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml11/DTD/xhtml11-strict.dtd">

<!-- Fig. 12.3: collections.html -->
<!-- Using the links collection. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using Links Collection</title>
      <style type = "text/css">
         body           { font-family: arial, helvetica, sans-serif }
         h1             { font-family: tahoma, geneva, sans-serif;
                          text-align: center }
         p              { margin: 5% }
         p a            { color: #aa0000 }
         .links         { font-size: 14px;
                          text-align: justify;
                          margin-left: 10%;
                          margin-right: 10% }
         .link a        { text-decoration: none }
         .link a:hover  { text-decoration: underline }
      </style>
      <script type = "text/javascript">
         <!--
         function processlinks()
         {
            var linkslist = document.links; // get the document's links
            var contents = "Links in this page:\n<br />| ";

            // concatenate each link to contents
            for ( var i = 0; i < linkslist.length; i++ )
            {
```

Fig. 12.3 | Using the links collection. (Part 1 of 2.)

```
33                 var currentLink = linkslist[ i ];
34                 contents += "<span class = 'link'>" +
35                    currentLink.innerHTML.link( currentLink.href ) +
36                    "</span> | ";
37              } // end for
38
39              document.getElementById( "links" ).innerHTML = contents;
40           } // end function processlinks
41           // -->
42        </script>
43     </head>
44     <body onload = "processlinks()">
45        <h1>Deitel Resource Centers</h1>
46        <p><a href = "http://www.deitel.com/">Deitel's website</a> contains
47           a rapidly growing
48           <a href = "http://www.deitel.com/ResourceCenters.html">list of
49           Resource Centers</a> on a wide range of topics. Many Resource
50           centers related to topics covered in this book,
51           <a href = "http://www.deitel.com/iw3htp4">Internet and World Wide
52           Web How to Program, 4th Edition</a>. We have Resouce Centers on
53           <a href = "http://www.deitel.com/Web2.0">Web 2.0</a>,
54           <a href = "http://www.deitel.com/Firefox">Firefox</a> and
55           <a href = "http://www.deitel.com/IE7">Internet Explorer 7</a>,
56           <a href = "http://www.deitel.com/XHTML">XHTML</a>, and
57           <a href = "http://www.deitel.com/JavaScript">JavaScript</a>.
58           Watch the list of Deitel Resource Centers for related new
59           Resource Centers.</p>
60        <div id = "links" class = "links"></div>
61     </body>
62  </html>
```

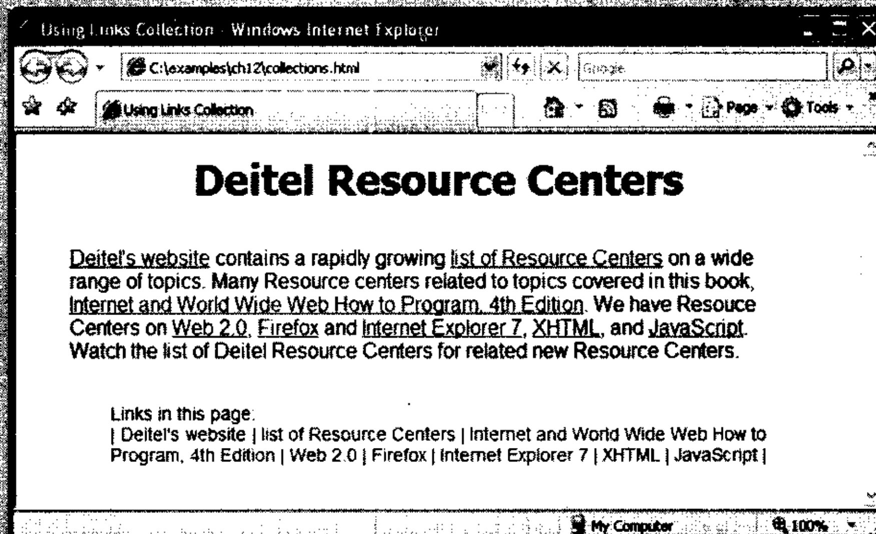


Fig. 12.3 | Using the `links` collection. (Part 2 of 2.)

The XHTML body contains a paragraph (lines 46–59) with links at various places in the text and an empty `div` (line 60) with `id links`. The body's `onload` attribute specifies that the `processlinks` method is called when the `body` finishes loading.

Method `processlinks` declares variable `linkslist` (line 27) to store the `document`'s `links` collection, which is accessed as the `links` property of the `document` object. Line 28 creates the string (`contents`) that will contain all the document's links, to be inserted into the `links div` later. Line 31 begins a `for` statement to iterate through each link. To find the number of elements in the collection, we use the collection's **`length` property**.

Line 33 inside the `for` statement creates a variable (`currentlink`) that stores the current link. Note that we can access the collection stored in `linkslist` using indices in square brackets, just as we did with arrays. DOM collections are stored in objects which have only 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—can be used to access specific elements in a collection by taking an index as an argument. The `namedItem` method takes a name as a parameter and finds the element in the collection, if any, whose `id` attribute or `name` attribute matches it.

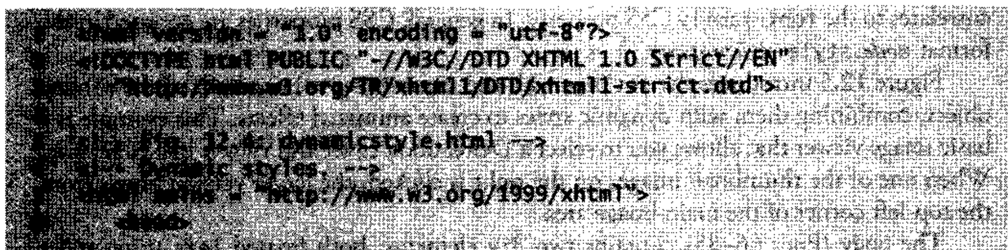
Lines 34–36 add a `span` element to the `contents` string containing the current link. Recall that the `link` method of a string object returns the string as a link to the URL passed to the method. Line 35 uses the `link` method to create an a (anchor) element containing the proper text and `href` attribute.

Notice that variable `currentLink` (a DOM node representing an a element) has a specialized **`href` property** to refer to the link's `href` attribute. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node. Line 39 inserts the contents into the empty `div` with `id "links"` (line 60) in order 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 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. The next section discusses how to dynamically modify CSS styles using JavaScript and DOM nodes.

12.5 Dynamic Styles

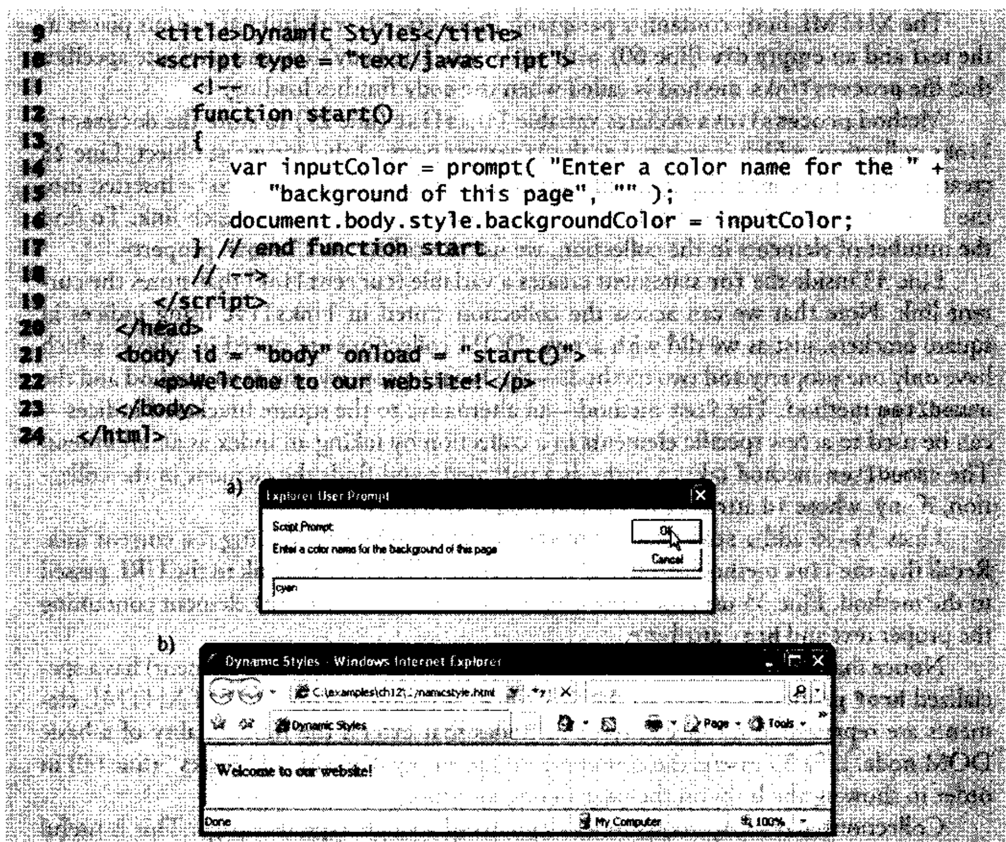
An element's style can be changed dynamically. Often such a change is made in response to user events, which we discuss in Chapter 13. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations. Figure 12.4 is a simple example that changes the `background-color` style property in response to user input.



```
"1.0" encoding = "utf-8"?>
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

12.4: dynamicstyle.html -->
styles. -->
= "http://www.w3.org/1999/xhtml">
```

Fig. 12.4 | Dynamic styles. (Part 1 of 2.)



```
 9        <title>Dynamic Styles</title>
10        <script type = "text/javascript">
11           <!--
12           function start()
13           {
14              var inputColor = prompt( "Enter a color name for the " +
15                 "background of this page", "" );
16              document.body.style.backgroundColor = inputColor;
17           } // end function start
18           // -->
19        </script>
20     </head>
21     <body id = "body" onload = "start()">
22        <p>Welcome to our website!</p>
23     </body>
24  </html>
```


Fig. 12.4 | Dynamic styles. (Part 2 of 2.)

Function `start` (lines 12–17) prompts the user to enter a color name, then sets the background color to that value. [*Note:* An error occurs if the value entered is not a valid color. See Appendix B, XHTML Colors, for further information.] We refer to the background color as `document.body.style.backgroundColor`—the **body property** of the `document` object refers to the `body` element. We then use the `style` property (a property of most XHTML elements) to set the `background-color` CSS property. This is referred to as `backgroundColor` in JavaScript—the hyphen is removed to avoid confusion with the subtraction (-) operator. This naming convention is consistent for most CSS properties. For example, `borderWidth` correlates to the `border-width` CSS property, and `fontFamily` correlates to the `font-family` CSS property. In general, CSS properties are accessed in the format *node*.`style`.*styleproperty*.

Figure 12.5 introduces the `setInterval` and `clearInterval` methods of the `window` object, combining them with dynamic styles to create animated effects. This example is a basic image viewer that allows you to select a Deitel book cover and view it in a larger size. When one of the thumbnail images on the right is clicked, the larger version grows from the top-left corner of the main image area.

The `body` (lines 66–85) contains two `div` elements, both floated `left` using styles defined in lines 14 and 17 in order to present them side by side. The left `div` contains the

full-size image iw3htp4.jpg, the cover of this book, which appears when the page loads. The right div contains six thumbnail images which respond to the click event by calling the display method and passing it the filename of the corresponding full-size image.

The display function (lines 46–62) dynamically updates the image in the left div to the one corresponding to the user's click. Lines 48–49 prevent the rest of the function from executing if interval is defined (i.e., an animation is in progress.) Line 51 gets the left div by its id, imgCover. Line 52 creates a new img element. Lines 53–55 set its id to imgCover, set its src to the correct image file in the fullsize directory, and set its required alt attribute. Lines 56–59 do some additional initialization before beginning the animation in line 61. To create the growing animation effect, lines 57–58 set the image width and height to 0. Line 59 replaces the current bigImage node with newNode (created in line 52), and line 60 sets count, the variable that controls the animation, to 0.

Line 61 introduces the window object's **setInterval method**, which starts the 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

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 12.5: coverviewer.html -->
 6  <!-- Dynamic styles used for animation. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Deitel Book Cover Viewer</title>
10        <style type = "text/css">
11           .thumbs   { width: 192px;
12                       height: 370px;
13                       padding: 5px;
14                       float: left }
15           .mainimg  { width: 289px;
16                       padding: 5px;
17                       float: left }
18           .imgCover { height: 373px }
19           img       { border: 1px solid black }
20        </style>
21        <script type = "text/javascript">
22           <!--
23           var interval = null; // keeps track of the interval
24           var speed = 6; // determines the speed of the animation
25           var count = 0; // size of the image during the animation
26
27           // called repeatedly to animate the book cover
28           function run()
29           {
30              count += speed;
31
32              // stop the animation when the image is large enough
33              if ( count >= 375 )
34              {
```

Fig. 12.5 | Dynamic styles used for animation. (Part 1 of 4.)

```
35             window.clearInterval( interval );
36             interval = null;
37          } // end if
38
39          var bigImage = document.getElementById( "imgCover" );
40          bigImage.style.width = .7656 * count + "px";
41          bigImage.style.height = count + "px";
42       } // end function run
43
44       // inserts the proper image into the main image area and
45       // begins the animation
46       function display( imgfile )
47       {
48          if ( interval )
49             return;
50
51          var bigImage = document.getElementById( "imgCover" );
52          var newNode = document.createElement( "img" );
53          newNode.id = "imgCover";
54          newNode.src = "fullsize/" + imgfile;
55          newNode.alt = "Large image";
56          newNode.className = "imgCover";
57          newNode.style.width = "0px";
58          newNode.style.height = "0px";
59          bigImage.parentNode.replaceChild( newNode, bigImage );
60          count = 0; // start the image at size 0
61          interval = window.setInterval( "run()", 10 );
62       } // end function display
63       // -->
64    </script>
65 </head>
66 <body>
67    <div id = "mainimg" class = "mainimg">
68       <img id = "imgCover" src = "fullsize/iw3htp4.jpg"
69          alt = "Full cover image" class = "imgCover" />
70    </div>
71    <div id = "thumbs" class = "thumbs">
72       <img src = "thumbs/iw3htp4.jpg" alt = "iw3htp4"
73          onclick = "display( 'iw3htp4.jpg' )" />
74       <img src = "thumbs/chtp5.jpg" alt = "chtp5"
75          onclick = "display( 'chtp5.jpg' )" />
76       <img src = "thumbs/cpphtp6.jpg" alt = "cpphtp6"
77          onclick = "display( 'cpphtp6.jpg' )" />
78       <img src = "thumbs/jhtp7.jpg" alt = "jhtp7"
79          onclick = "display( 'jhtp7.jpg' )" />
80       <img src = "thumbs/vbhtp3.jpg" alt = "vbhtp3"
81          onclick = "display( 'vbhtp3.jpg' )" />
82       <img src = "thumbs/vcsharphtp2.jpg" alt = "vcsharphtp2"
83          onclick = "display( 'vcsharphtp2.jpg' )" />
84    </div>
85 </body>
86 </html>
```

Fig. 12.5 | Dynamic styles used for animation. (Part 2 of 4.)

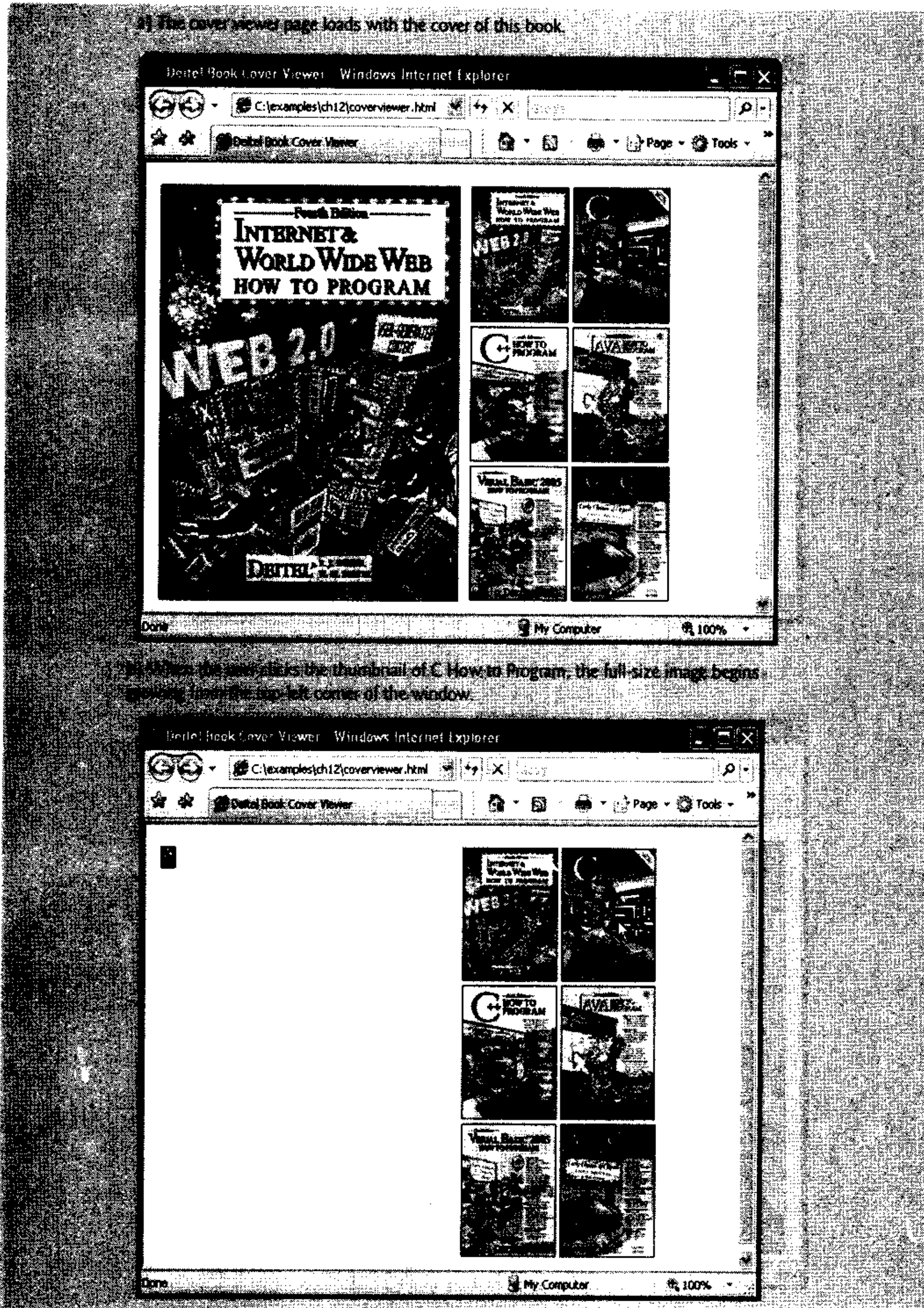


Fig. 12.5 | Dynamic styles used for animation. (Part 3 of 4.)

c) The cover continues to grow.

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

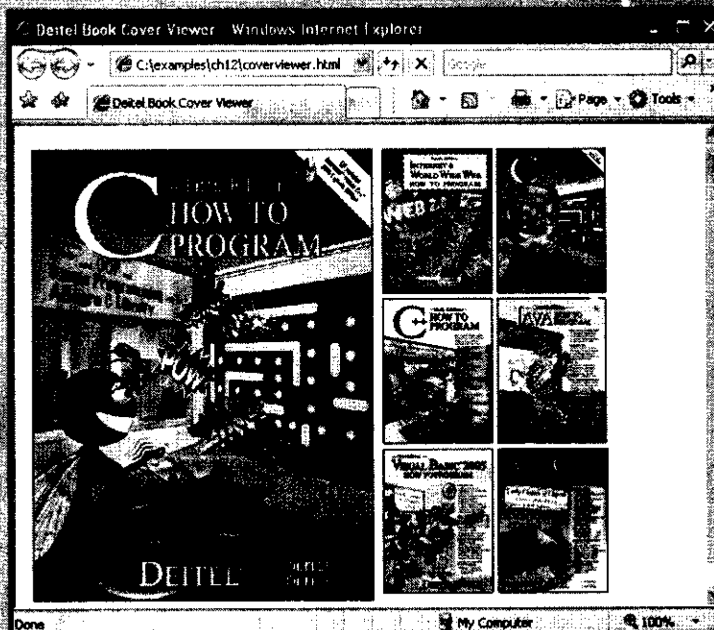


Fig. 12.5 | Dynamic styles used for animation. (Part 4 of 4.)

function run every 10 milliseconds. The setInterval method returns a unique identifier to keep track of that particular interval—we assign this identifier to the variable interval. We use this identifier to stop the animation when the image has finished growing.

The run function, defined in lines 28–42, increases the height of the image by the value of speed and updates its width accordingly to keep the aspect ratio consistent. Because the run function is called every 10 milliseconds, this increase happens repeatedly to create an animated growing effect. Line 30 adds the value of speed (declared and initialized to 6 in line 24) to count, which keeps track of the animation's progress and dictates the current size of the image. If the image has grown to its full height (375), line 35 uses the window's **clearInterval method** to stop the repetitive calls of the run method. We pass to clearInterval the interval identifier (stored in interval) that setInterval created in line 61. Although it seems unnecessary in this script, this identifier allows the script to keep track of multiple intervals running at the same time and to choose which interval to stop when calling clearInterval.

Line 39 gets the image and lines 40–41 set its width and height CSS properties. Note that line 40 multiplies count by a scaling factor of .7656 in order to keep the ratio of the image's dimensions consistent with the actual dimensions of the image. Run the code example and click on a thumbnail image to see the full animation effect.

This section demonstrated the concept of dynamically changing CSS styles using JavaScript and the DOM. We also discussed the basics of how to create scripted animations using setInterval and clearInterval.

12.6 Summary of the DOM Objects and Collections

As you have seen in the preceding sections, the objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page. We have shown how to access the objects in a page, how to access the objects in a collection, and how to change element styles dynamically.

The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object. The diagram in Fig. 12.6 shows many of the important objects and collections provided by the W3C DOM. Figure 12.7 provides a brief description of each object and collection in Fig. 12.6.

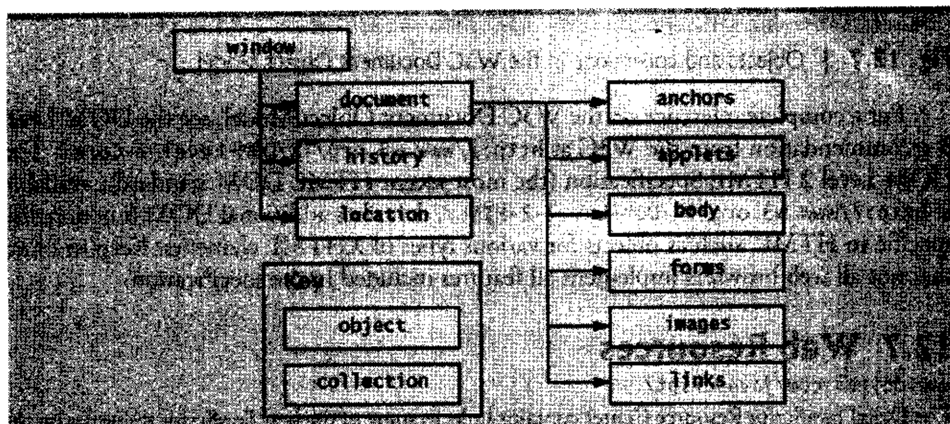


Fig. 12.6 | W3C Document Object Model.

Objects	
`window`	Represents the browser window and provides access to the document object contained in the window. Also contains `history` and `location` objects.
`document`	Represents the XHTML document rendered in a `window`. The `document` object provides access to every element in the XHTML document and allows dynamic modification of the XHTML document. Contains several collections for accessing all elements of a given type.
`body`	Provides access to the `body` element of an XHTML document.
`history`	Keeps track of the sites visited by the browser user. The object provides a script programmer with the ability to move forward and backward through the visited sites.
`location`	Contains the URL of the rendered document. When this object is set to a new URL, the browser immediately navigates to the new location.
Collections	
`anchors`	Collection contains all the anchor elements (`a`) that have a `name` or `id` attribute. The elements appear in the collection in the order in which they were defined in the XHTML document.
`forms`	Contains all the `form` elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.
`images`	Contains all the `img` elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.
`links`	Contains all the anchor elements (`a`) with an `href` property. The elements appear in the collection in the order in which they were defined in the XHTML document.

Fig. 12.7 | Objects and collections in the W3C Document Object Model.

For a complete reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at `http://www.w3.org/TR/DOM-Level-3-Core/`. The DOM Level 2 HTML Specification (the most recent HTML DOM standard), available at `http://www.w3.org/TR/DOM-Level-2-HTML/`, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements. Keep in mind that not all web browsers implement all features included in the specification.

12.7 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries,

frameworks, tutorials and more. Check out the section specifically dedicated to the Document Object Model. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 12.1 Introduction

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

Section 12.2 Modeling a Document: DOM Nodes and Trees

- The `getElementById` method returns objects called DOM nodes. Every element in an XHTML page 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 XHTML element inside another element is said to be a child of the containing element. The containing element is known as the parent. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as siblings.
- Firefox's DOM Inspector and the IE Web Developer Toolbar allow you to see a visual representation of a document's DOM tree and information about each node.
- The document node in a DOM tree is called the root node, because it has no parent.

Section 12.3 Traversing and Modifying a DOM Tree

- The `className` property of a DOM node allows you to change an XHTML element's `class` attribute.
- The `id` property of a DOM node controls an element's `id` attribute.
- The `document` object's `createElement` method 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 creates a DOM node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node.
- Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.
- The `parentNode` property of any DOM node contains the node's parent.
- The `insertBefore` method is called on a parent with 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 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 is called on a parent with a child to be removed as an argument.

Section 12.4 DOM Collections

- Included in the Document Object Model is the notion of 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 and anchors collection. 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.
- To access items in a collection, use square brackets the same way you would with an array, or use the item method. The item method of a DOM collection is used to access specific elements in a collection, taking an index as an argument. The namedItem method 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. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node.
- 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 across an entire page.

Section 12.5 Dynamic Styles

- An element's style can be changed dynamically. Often such a change is made in response to user events, which are discussed in the next chapter. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations.
- The body property of the document object refers to the body element in the XHTML page.
- The style property can access a CSS property in the format *node*.style.*styleproperty*.
- A CSS property with a hyphen (-), such as background-color, is referred to as backgroundColor in JavaScript, to avoid confusion with the subtraction (-) operator. Removing the hyphen and capitalizing the first letter of the following word is the convention for most CSS properties.
- The setInterval method 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 stops the repetitive calls of object's setInterval method. We pass to clearInterval the interval identifier that setInterval returned.

Section 12.6 Summary of the DOM Objects and Collections

- The objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page.
- The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object.
- For a reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at http://www.w3.org/TR/DOM-Level-3-Core/. The DOM Level 2 HTML Specification, available at http://www.w3.org/TR/DOM-Level-2-HTML/, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements.
- Not all web browsers implement all features included in the DOM specification.

Terminology

anchors collection of the document object
body property of the document object
appendChild method of a DOM node
child
className property of a DOM node
clearInterval method of the window object

collection
createElement method of the document object
createTextNode method of the document object
document object
Document Object Model
DOM collection
DOM Inspector
DOM node
DOM tree
dynamic style
forms collection of the document object
href property of an a (anchor) node
Internet Explorer Web Developer Toolbar
id property of a DOM node
images collection
innerHTML property of a DOM node
insertBefore method of a DOM node
item method of a DOM collection
length property of a DOM collection
links collection of the document object
namedItem method of a DOM collection
object hierarchy
parent
removeChild method of a DOM node
replaceChild method of a DOM node
root node
setInterval method of the window object
sibling
style property of a DOM node
W3C Document Object Model

Self-Review Exercises

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

a) Every XHTML 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 with the style property.
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 name or id attributes.

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

a) The ______ property refers to the text inside an element, including XHTML tags.
b) A document's DOM ______ represents all of 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.
f) The ______ object contains information about the sites that a user previously visited.
g) CSS properties may be accessed using the ______ object.

Exercises

12.3 Use the Firefox DOM Inspector or the IE Web Developer Toolbar to view the DOM tree of the document in Fig. 12.2. Look at the document tree of your favorite website. Notice the information these tools give you in the right panel(s) about an element when you click it.

12.4 Write a script that contains a button and a counter in a div. The button should increment the counter each time it is clicked.

12.5 *(15 Puzzle)* Write a web page that enables the user to play the game of 15. There is a 4-by-4 board (implemented as an XHTML 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 onclick event, write a script that allows the user to swap the positions of the open position and an adjacent tile. [*Hint:* The onclick event should be specified for each table cell.]

12.6 Modify your solution to Exercise 12.5 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.

12.7 Modify your solution to Exercise 12.6 to use an image that is split into 16 equally sized pieces. Discard one of the pieces and randomly place the other 15 pieces in the XHTML table.