11

JavaScript: Objects

Outline

11.1 Introduction

- This 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
- JavaScript uses objects to perform many tasks
  - It is referred to as an object-based programming language
- Objects have attributes and exhibit behaviors

11.2 Introduction to Object Technology (Cont.)

- **Objects have the property of information hiding**
  - **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**
- **Web browsers**
  - **Contain a set of objects that encapsulate an XHTML document's elements**
  - **The objects expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects)**

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

Fig. 11.1 | `Math` object methods.

Method	Description	Examples

`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`
`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`

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

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

Method	Description
`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.

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

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

```
<?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. 11.4: CharacterProcessing.html -->
<!-- String methods charAt, charCodeAt, fromCharCode, toLowercase and
   toUpperCase. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Character Processing Methods</title>
      <script type = "text/javascript">
         <!--
         var s = "ZEBRA";
         var s2 = "AbCdEfG";

         document.writeln( "<p>Character at index 0 in '" +
            s + "' is " + s.charAt( 0 ) );
         document.writeln( "<br />Character code at index 0 in '"
            + s + "' is " + s.charCodeAt( 0 ) + "</p>" );

         document.writeln( "<p>'" +
            String.fromCharCode( 87, 79, 82, 68 ) +
            "' contains character codes 87, 79, 82 and 68</p>" )

```

Returns the character at index 0 of string s

Returns the Unicode value of the character at index 0 of string s

Creates a string from the characters with the Unicode values 87, 79, 82 and 68

```
25          document.writeln( "<p>'" + s2 + "' in lowercase is '" +
26             s2.toLowerCase() + "'" );
27          document.writeln( "<br />'" + s2 + "' in uppercase is '"
28             + s2.toUpperCase() + "'</p>" );
29           // -->
30       </script>
31    </head><body></body>
32 </html>
```

Converts s2 to lowercase

Converts s2 to uppercase

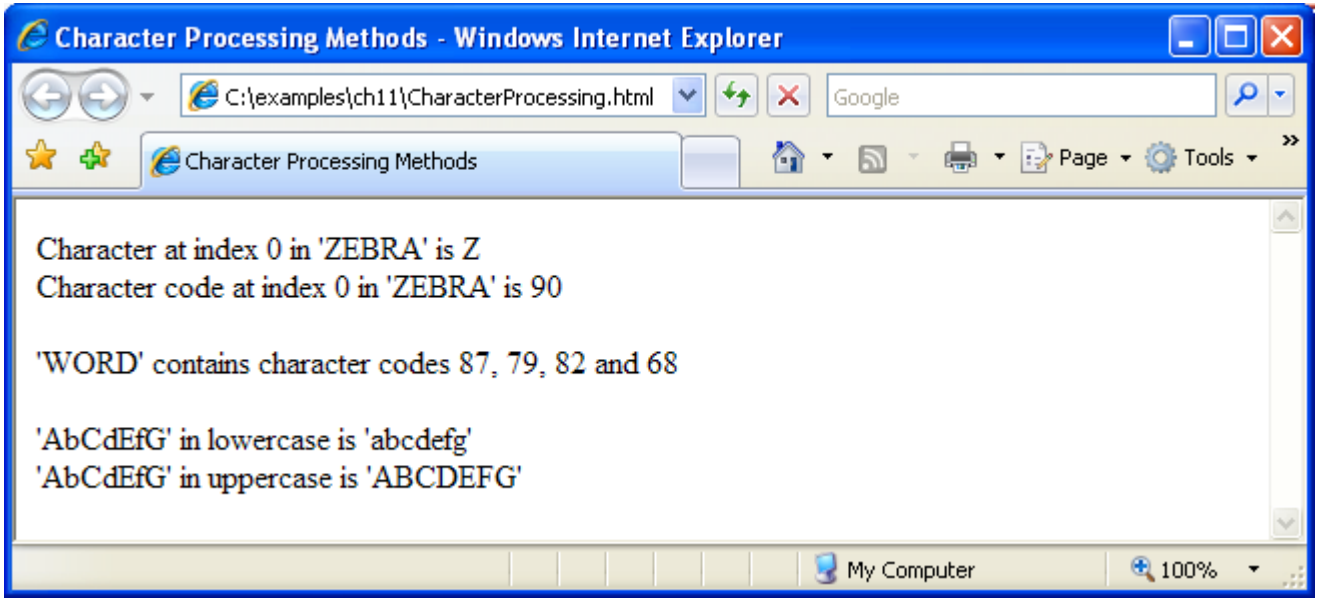

Fig. 11.4 | String methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase (Part 2 of 2).

11.4 `String` Object (Cont.)

- **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**
  - **Argument is the delimiter string**
  - **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**

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

```
<?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. 11.6: SplitAndSubString.html -->
<!-- String object methods split and substring. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>String Methods split and substring</title>
      <script type = "text/javascript">
         <!--
         function splitButtonPressed()
         {
            var inputString = document.getElementById( "inputVal" ).value;
            var tokens = inputString.split( " " );
            document.getElementById( "output" ).value =
               tokens.join( "\n" );

            document.getElementById( "outputSubstring" ).value =
               inputString.substring( 0, 10 );
         } // end function splitButtonPressed
         // -->
      </script>
   </head>
   <body>
      <form action = "">
         <p>Enter a sentence to split into words<br />
         <input id = "inputVal" type = "text" size = "40" />
         <input type = "button" value = "Split"
            onclick = "splitButtonPressed()" /></p>

```

Splits `inputString` into new strings at each space and stores them in array `tokens`

Creates a string from the elements in `tokens`, inserting a newline character between each element

Inserts the first 10 characters of `inputString` into the `outputSubstring` text field

```
32          <p>The sentence split into words is<br />
33          <textarea id = "output" rows = "8" cols = "34">
34          </textarea></p>
35
36          <p>The first 10 characters of the input string are
37          <input id = "outputSubstring" type = "text"
38                size = "15" /></p>
39       </form>
40    </body>
41 </html>
```

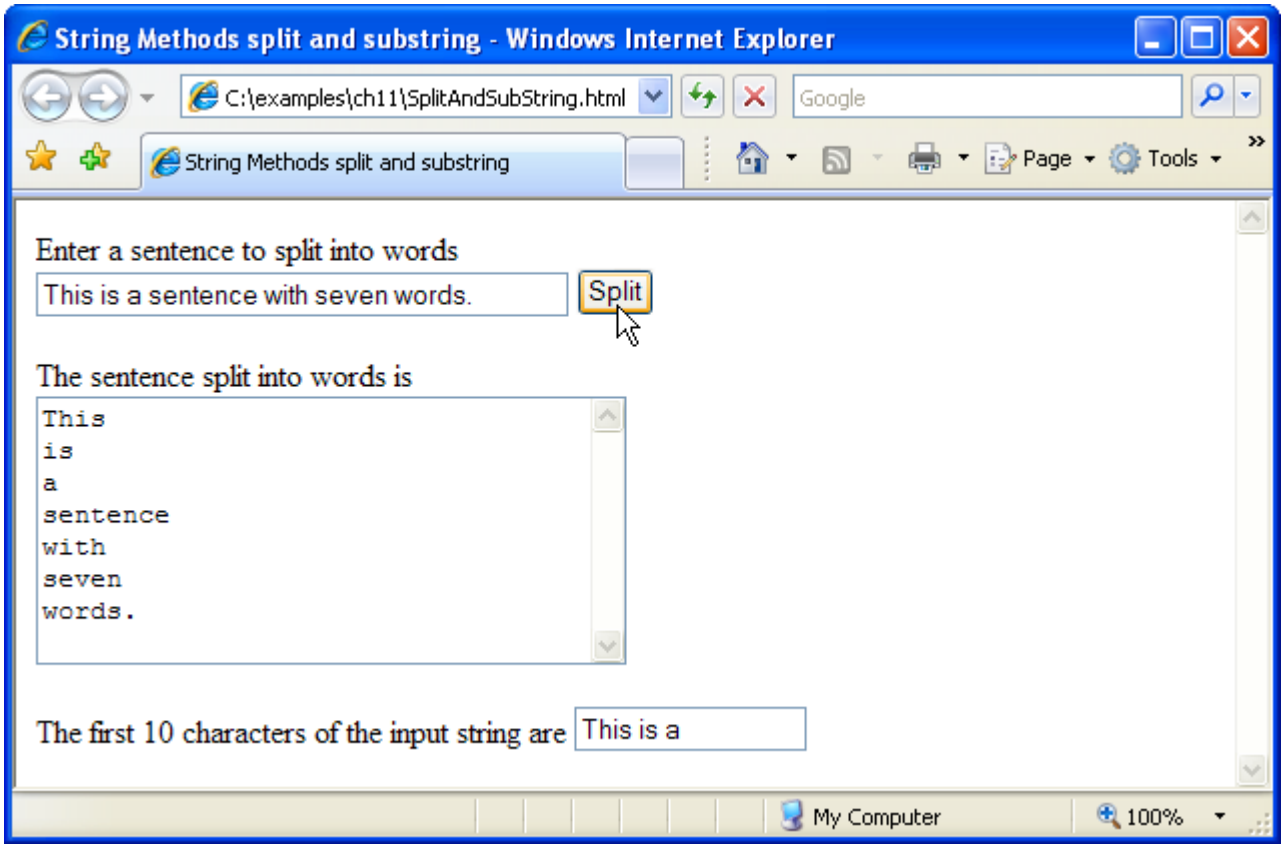

Fig. 11.6 | `String` object methods `split` and `substring` (Part 2 of 2).

11.5 `Date` Object

- `Date` object provides methods for date and time manipulations
  - Based either on the computer's local time zone or on World Time Standard's Coordinated Universal Time (abbreviated UTC)
- Most methods have a local time zone and a UTC version
- Empty parentheses after an object name indicate a call to the object's constructor with no arguments
  - A constructor is an initializer method for an object
  - 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
  - A new `Date` object can be initialized by passing the number of milliseconds since midnight, January 1, 1970, to the `Date` constructor
  - Can also create a new `Date` object by supplying arguments to the `Date` constructor for year, month, date, hours, minutes, seconds and milliseconds.
    - 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
    - If an argument is specified, all arguments to its left must be specified

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

Method	Description
`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.

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

`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.
`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. 11.9 | Date and time methods of the `Date` object (Part 1 of 3).

```
<?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. 11.9: DateTime.html -->
<!-- Date and time methods of the Date object. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Date and Time Methods</title>
      <script type = "text/javascript">
         <!--
         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() );

```

Initializes `current` as a new `Date` object with the local computer's time and date

Converts `current` to a string representation of the date

Converts `current` to the locale's string representation of the date

Converts `current` to a string representation of the UTC time

Calculates the number of milliseconds since midnight, January 1, 1970

Fig. 11.9 | Date and time methods of the `Date` object (Part 2 of 3).

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

         document.writeln(
            "<h1>Specifying arguments for a new Date</h1>" );
         var anotherDate = new Date( 2007, 2, 18, 1, 5, 0, 0 );
         document.writeln( "Date: " + anotherDate );

         document.writeln( "<h1>Set methods for local time zone</h1>" );
         anotherDate.setDate( 31 );
         anotherDate.setMonth( 11 );
         anotherDate.setFullYear( 2007 );
         anotherDate.setHours( 23 );
         anotherDate.setMinutes( 59 );
         anotherDate.setSeconds( 59 );
         document.writeln( "Modified date: " + anotherDate );
         // -->
      </script>
   </head><body></body>
</html>
```

Returns the date, day, month, year, milliseconds since 1/1/1970, hours

Creates a new `Date` object by passing the year, month, date, hours, minutes, seconds and milliseconds to the `Date` constructor

Sets the date, month, year, hours, minutes, and seconds of a new `Date` object

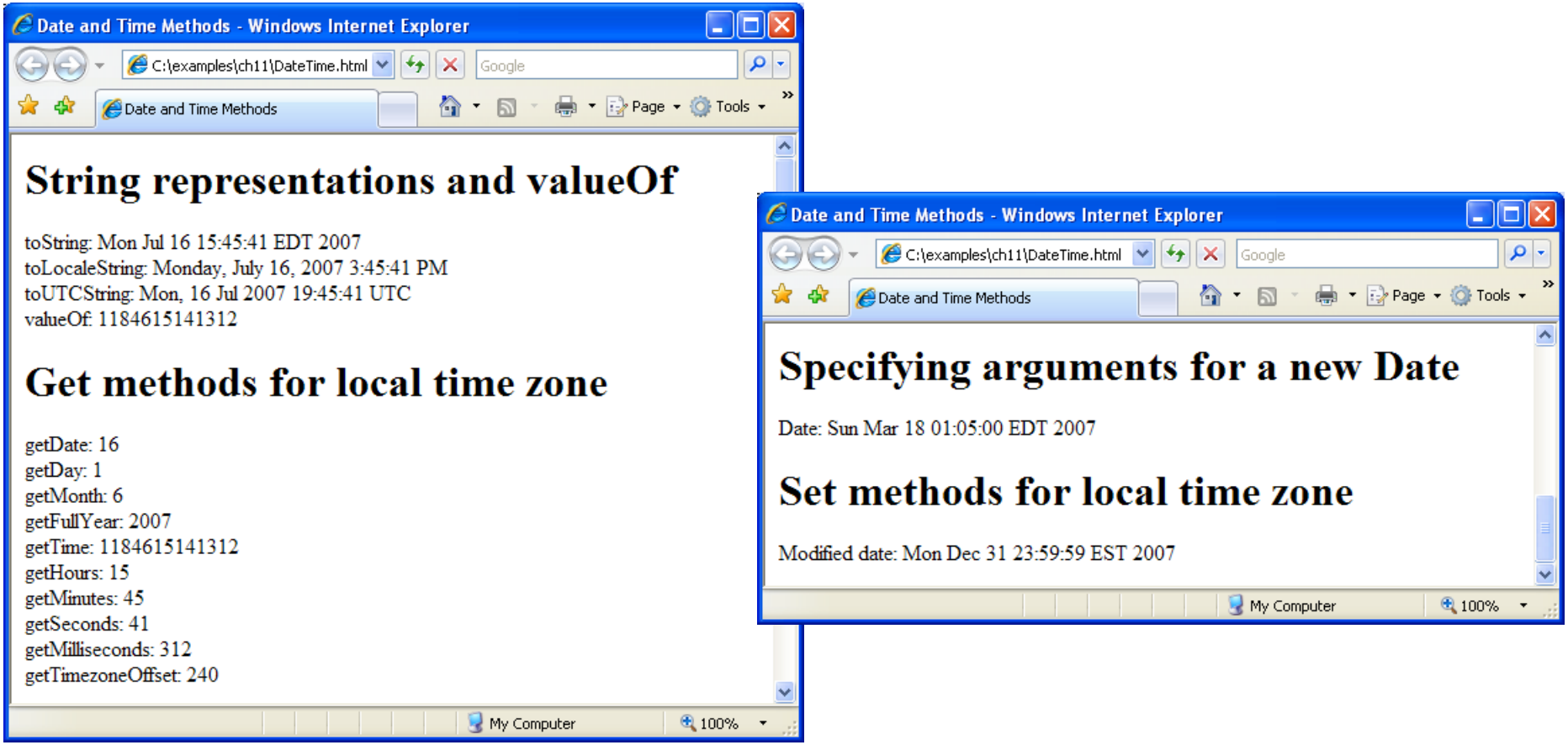

Fig. 11.9 | Date and time methods of the `Date` object (Part 3 of 3).

11.6 `Boolean` and `Number` Objects

- The `Boolean` and `Number` objects are 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

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

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`

11.6 `Boolean` and `Number` Objects (Cont.)

- **JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program**
- **Can create a `Number` object with the statement**

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

***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**

Fig. 11.10 | `Boolean` object methods.

Method	Description
`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. 11.11 | `Number` object methods and properties.

Method or property	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. See Appendix E, Number Systems, for a review of the binary, octal, decimal and hexadecimal number systems.
`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`.

11.7 `document` Object

- `document` object
  - For manipulating the document that is currently visible in the browser window

Fig. 11.12 | Important **`document`** object methods and properties.

Method or property	Description
`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.

11.8 `window` Object

- `window` object provides methods for manipulating browser windows
- `window` object `open` method
  - Creates a window
  - Three parameters—the URL of the page to open in the new window, the name of the window, 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.

11.8 `window` Object (Cont.)

- `window` object `closed` property
  - Contains a boolean value that is `true` if the window is closed and `false` if the window is open
- `close` method
  - Closes the current window and deletes its object from memory
- `location` property
  - contains a string representation of the URL displayed in the current window

```
<?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. 11.13: window.html -->
<!-- Using the window object to create and modify child windows. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Using the Window Object</title>
<script type = "text/javascript">
   <!--
   var childWindow; // variable to control the child window

   function createChildWindow()
   {
      // these variables all contain either "yes" or "no"
      // to enable or disable a feature in the child window
      var toolBar;
      var menuBar;
      var scrollBars;

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

```

Fig. 11.13 | Using the **window** object to create and modify child windows (Part 1 of 6).

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

      // determine whether the Scroll Bar checkbox is checked
      if ( document.getElementById( "scrollBarsCheckBox" ).checked )
         scrollBars = "yes";
      else
         scrollBars = "no";

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

      // disable buttons
      document.getElementById( "closeButton" ).disabled = false;
      document.getElementById( "modifyButton" ).disabled = false;
      document.getElementById( "setURLButton" ).disabled = false;
   } // end function createChildWindow

   // insert text from the textbox in the chil
   function modifyChildWindow()
   {
      if ( childWindow.closed )
         alert( "You attempted to interact with
      else
         childWindow.document.write(
            document.getElementById( "textForChild" ).value );
```

Creates a new window with an empty URL and name, and the options specified by the user

Checks `childwindow`'s `closed` property

Writes text in `textForChild` to the child window

Fig. 11.13 | Using the **`window`** object to create and modify child windows (Part 2 of 6).

```
   // close the child window
   function closeChildWindow()
   {
      if ( childWindow.closed )
         alert( "You attempted to interact with a closed window" );
      else
         childWindow.close();

      document.getElementById( "closeButton" ).disabled = true;
      document.getElementById( "modifyButton" ).disabled = true;
      document.getElementById( "setURLButton" ).disabled = true;
   } // end function closeChildWindow

   // set the URL of the child window to the URL
   // in the parent window's myChildURL
   function setChildWindowURL()
   {
      if ( childWindow.closed )
         alert( "You attempted to interact with a closed window" );
      else
         childWindow.location =
            document.getElementById( "myChildURL" ).value;
   } // end function setChildWindowURL
   //-->
</script>
</head>
```

Checks `childwindow`'s `closed` property

Closes `childWindow` and deletes the object from memory

Checks `childwindow`'s `closed` property

Sets the `location` of `childWindow` to the string in the `myChildURL` textbox, changing the child window's URL

Fig. 11.13 | Using the `window` object to create and modify child windows (Part 3 of 6).

Fig. 11.13 | Using the **window** object to create and modify child windows (Part 4 of 6).

```
<body>
   <h1>Hello, this is the main window</h1>
   <p>Please check the features to enable for the child window<br/>
   <input id = "toolBarCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Tool Bar</label>
   <input id = "menuBarCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Menu Bar</label>
   <input id = "scrollBarsCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Scroll Bars</label></p>

   <p>Please enter the text that you would like to display
   in the child window<br/>
   <input id = "textForChild" type = "text"
      value = "<h1>Hello, I am a child window.</h1> " />
   <input id = "createButton" type = "button"
      value = "Create Child Window" onclick = "createChildWindow()" />
   <input id= "modifyButton" type = "button" value = "Modify Child Window"
      onclick = "modifyChildWindow()" disabled = "disabled" />
   <input id = "closeButton" type = "button" value = "Close Child Window"
      onclick = "closeChildWindow()" disabled = "disabled" />

   <p>The other window's URL is: <br/>
   <input id = "myChildURL" type = "text" value = "./" />
   <input id = "setURLButton" type = "button" value = "Set Child URL"
      onclick = "setChildWindowURL()" disabled = "disabled" /></p>
</body>
</html>
```

When the `Create Child Window` button is clicked, call function `createChildWindow`

When the `Modify Child Window` button is clicked, call function `modifyChildWindow`

When the `Close Child Window` button is clicked, call function `closeChildWindow`

When the `Set Child URL` button is clicked, call function `setChildWindowURL`

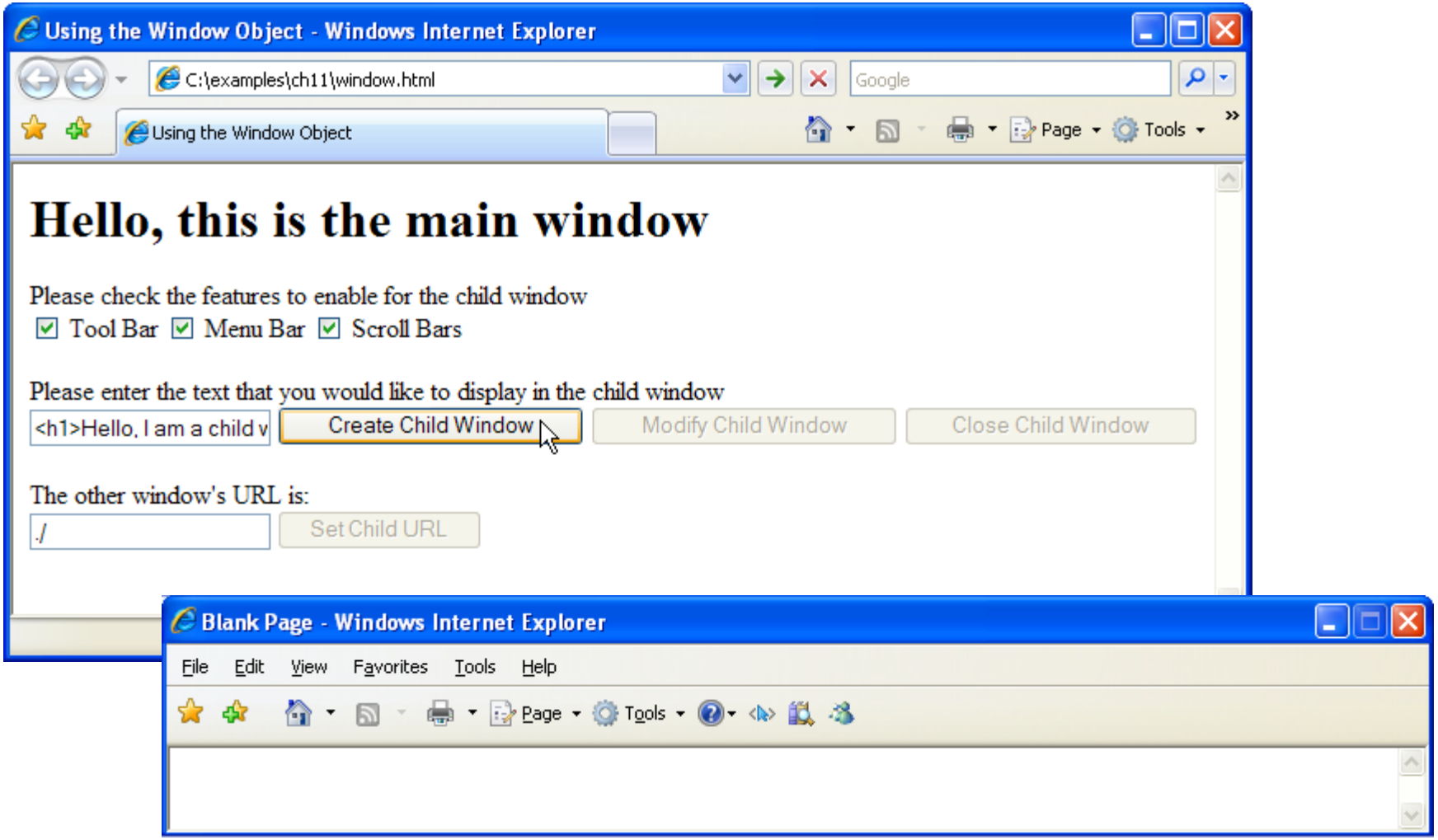

Fig. 11.13 | Using the `window` object to create and modify child windows (Part 5 of 6).

Fig. 11.13 | Using the `window` object to create and modify child windows (Part 6 of 6).

Fig. 11.14 | Important `window` object methods and properties.

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

11.9 Using Cookies

- **Cookie—a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions**
  - **Accessible in JavaScript through the document object's `cookie` property**
  - **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**
  - **`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**
- **Assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it**
- **Function `escape` converts any non-alphanumeric characters, such as spaces and semicolons, in a string to their equivalent hexadecimal escape sequences**
- **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**

Fig. 11.15 | Using cookies to store user identification data (Part 1 of 4).

```
<?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. 11.15: cookie.html -->
<!-- Using cookies to store user identification data. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using Cookies</title>
      <script type = "text/javascript">
         <!--
         var now = new Date(); // current date and time
         var hour = now.getHours(); // current hour (0-23)
         var name;

         if ( hour < 12 ) // determine whether it is morning
            document.write( "<h1>Good Morning, " );
         else
         {
            hour = hour - 12; // convert from 24-hour clock to PM time

            // determine whether it is afternoon or evening
            if ( hour < 6 )
               document.write( "<h1>Good Afternoon, " );
            else
               document.write( "<h1>Good Evening, " );
         } // end else

```

Outputs a time-appropriate greeting message

Fig. 11.15 | Using cookies to [illegible] data (Part 2 of 4).

```
29      // determine whether there is a cookie
30      if ( document.cookie )
31      {
32         // convert escape characters in the cookie string to their
33         // English notation
34         var myCookie = unescape( document.cookie );
35
36         // split the cookie into tokens using = as delimiter
37         var cookieTokens = myCookie.split( "=" );
38
39         // set name to the part of the cookie that follows the = sign
40         name = cookieTokens[ 1 ];
41      } // end if
42      else
43      {
44         // if there was no cookie, ask the user to input a name
45         name = window.prompt( "Please enter your name", "Paul" );
46
47         // escape special characters in the name string
48         // and add name to the cookie
49         document.cookie = "name=" + escape( name );
50      } // end else
51
```

Determines if a cookie exists on the client computer

Variable `myCookie` holds the unescaped cookie value

Breaks `myCookie` into identifier and value tokens in the `cookieTokens` array

Assigns global variable `name`'s value to the value of the original cookie

Assigns the cookie's `name` to an escaped value

```
52          document.writeln(
53             name + ", welcome to JavaScript programming!</h1>" );
54          document.writeln( "<a href = 'javascript:wrongPerson()'> " +
55             "Click here if you are not " + name + "</a>" );
56
57          // reset the document's cookie if wrong person
58          function wrongPerson()
59          {
60             // reset the cookie
61             document.cookie= "name=null;" +
62                " expires=Thu, 01-Jan-95 00:00:01 GMT";
63
64             // reload the page to get a new name after removing the cookie
65             location.reload();
66          } // end function wrongPerson
67
68          // -->
69       </script>
70    </head>
71    <body>
72       <p>Click Refresh (or Reload) to run the script again</p>
73    </body>
74 </html>
```

Resets the cookie with a new one that immediately expires

Forces the page to refresh, and will prompt the user to enter a name, as there will be no cookie

Fig. 11.15 | Using cookies to store user identification data (Part 3 of 4).

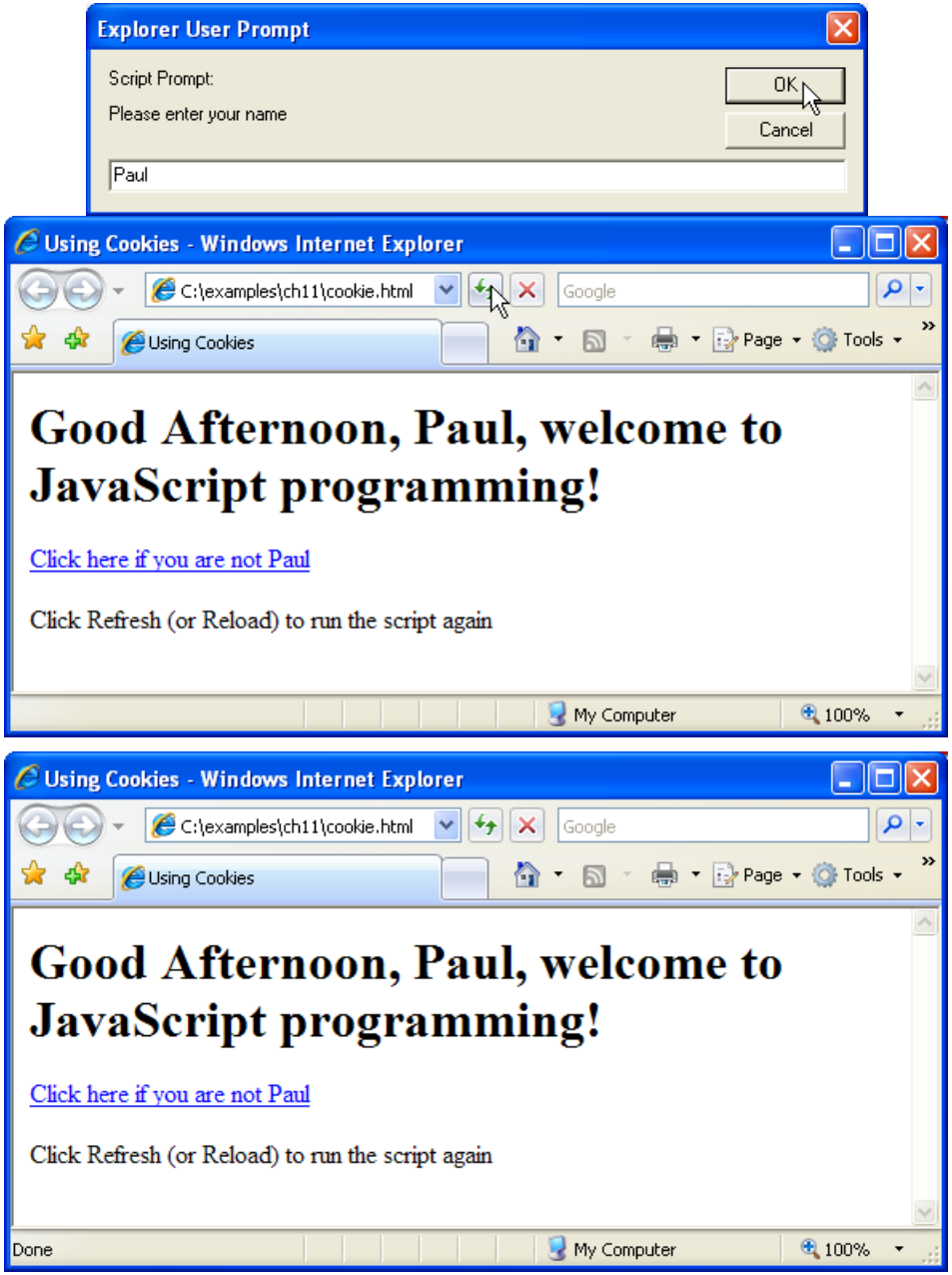

Fig. 11.15 | Using cookies to store user identification data (Part 4 of 4).

11.10 Final JavaScript Example

- `window.opener` always contains a reference to the window that opened the current window
- 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
- `window` object `close` method closes the browser window represented by the `window` object

```
<?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. 11.16: final.html -->
<!-- Rich welcome page using several JavaScript concepts. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Putting It All Together</title>
      <script type = "text/javascript">
         <!--
         var now = new Date(); // current date and time
         var hour = now.getHours(); // current hour

         // array with names of the images that will be randomly selected
         var pictures =
            [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];

         // array with the quotes that will be randomly selected
         var quotes = [ "Form ever follows function.<br/>" +
            " Louis Henri Sullivan", "E pluribus unum." +
            " (One composed of many.) <br/> Virgil", "Is it a" +
            " world to hide virtues in?<br/> William Shakespeare" ];

         // write the current date and time to the web page
         document.write( "<p>" + now.toLocaleString() + "<br/></p>" );

```

Outputs the current date as a string in the locale's format

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 1 of 8).

```
         // determine whether it is morning
         if ( hour < 12 )
            document.write( "<h2>Good Morning, " );
         else
         {
            hour = hour - 12; // convert from 24-hour clock to PM time

            // determine whether it is afternoon or evening
            if ( hour < 6 )
               document.write( "<h2>Good Afternoon, " );
            else
               document.write( "<h2>Good Evening, " );
         } // end else

         // determine whether there is a cookie
         if ( document.cookie )
         {
            // convert escape characters in the cookie string to their
            // English notation
            var myCookie = unescape( document.cookie );

            // split the cookie into tokens using = as delimiter
            var cookieTokens = myCookie.split( "=" );

            // set name to the part of the cookie that follows the = sign
            name = cookieTokens[ 1 ];
         } // end if
         else
         {
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 2 of 8).

```
      // if there was no cookie, ask the user to input a name
      name = window.prompt( "Please enter your name", "Paul" );

      // escape special characters in the name string
      // and add name to the cookie
      document.cookie = "name =" + escape( name );
   } // end else

   // write the greeting to the page
   document.writeln(
      name + ", welcome to JavaScript programming!</h2>" );

   // write the link for deleting the cookie to the page
   document.writeln( "<a href = \"javascript:wrongPerson()\" > " +
      "Click here if you are not " + name + "</a><br/>" );

   // write the random image to the page
   document.write ( "<img src = \"" +
      pictures[ Math.floor( Math.random() * 7 ) ] +
      ".gif\" /> <br/>" );

   // write the random quote to the page
   document.write ( quotes[ Math.floor( Math.random() * 3 ) ] );

   // create a window with all the quotes in it
   function allQuotes()
   {
      // create the child window for the quotes
      var quoteWindow = window.open( "", "", "resizable=yes, " +
         "toolbar=no, menubar=no, status=no, location=no," +
         " scrollBars=yes" );
      quoteWindow.document.write( "<p>" )

```

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 3 of 8).

```
            // loop through all quotes and write them in the new window
            for ( var i = 0; i < quotes.length; i++ )
               quoteWindow.document.write( ( i + 1 ) + ".) " +
                  quotes[ i ] + "<br/><br/>");

            // write a close link to the new window
            quoteWindow.document.write( "</p><br/><a href = " +
               "\"javascript:window.close()\">Close this window</a>" );
         } // end function allQuotes

         // reset the document's cookie if wrong person
         function wrongPerson()
         {
            // reset the cookie
            document.cookie= "name=null;" +
               " expires=Thu, 01-Jan-95 00:00:01 GMT";

            // reload the page to get a new name after removing the cookie
            location.reload();
         } // end function wrongPerson

         // open a new window with the quiz2.html file in it
         function openQuiz()
         {
            window.open( "quiz2.html", "", "toolbar = no, " +
                "menubar = no, scrollBars = no" );
         } // end function openQuiz
      // -->
      </script>
   </head>
   <body>
      <p><a href = "javascript:allQuotes()">View all quotes</a></p>
```

Iterates through all quotes in the `quotes` array and writes them in the new window

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 4 of 8).

```
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>
```

Links to the quiz

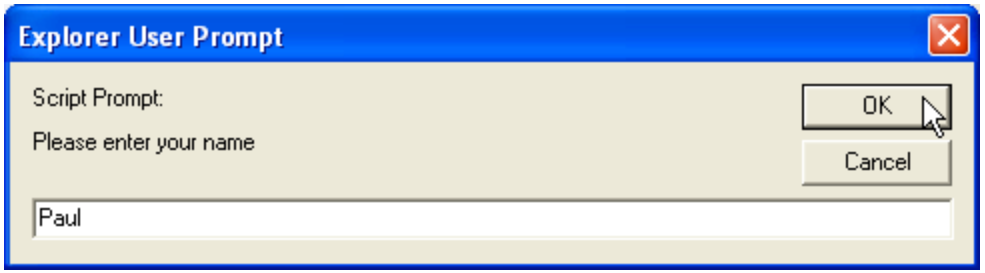

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 5 of 8).

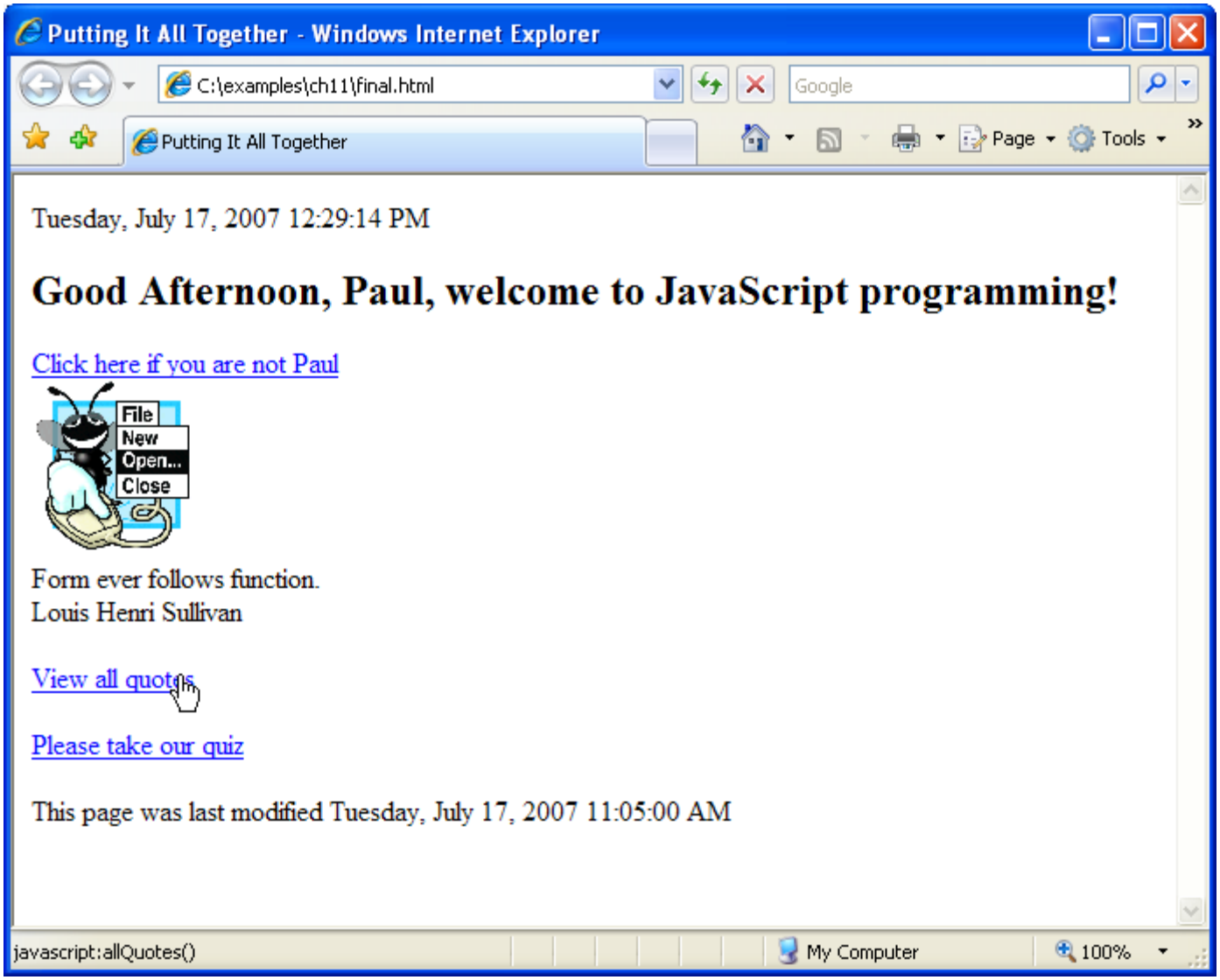

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 6 of 8).

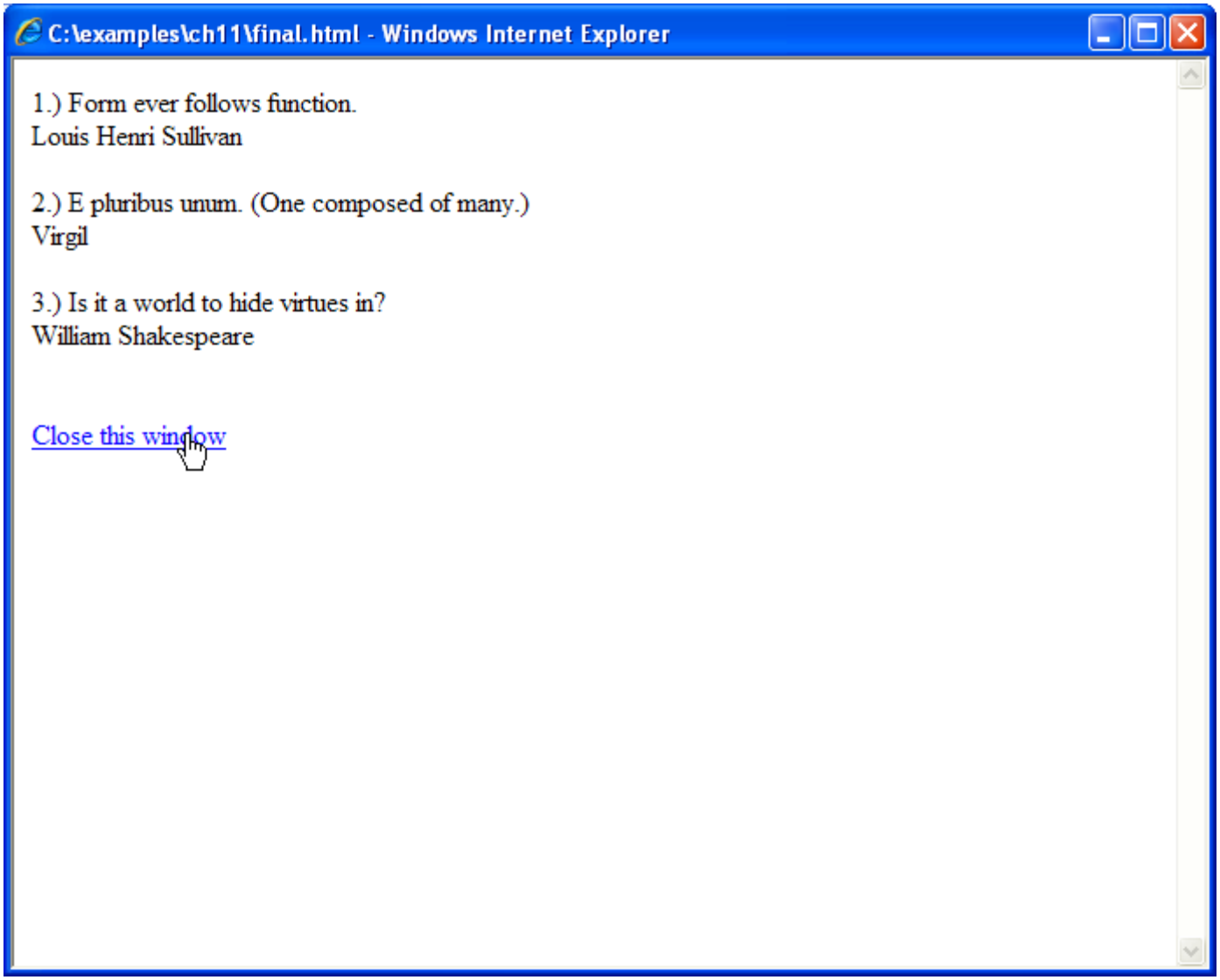

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 7 of 8).

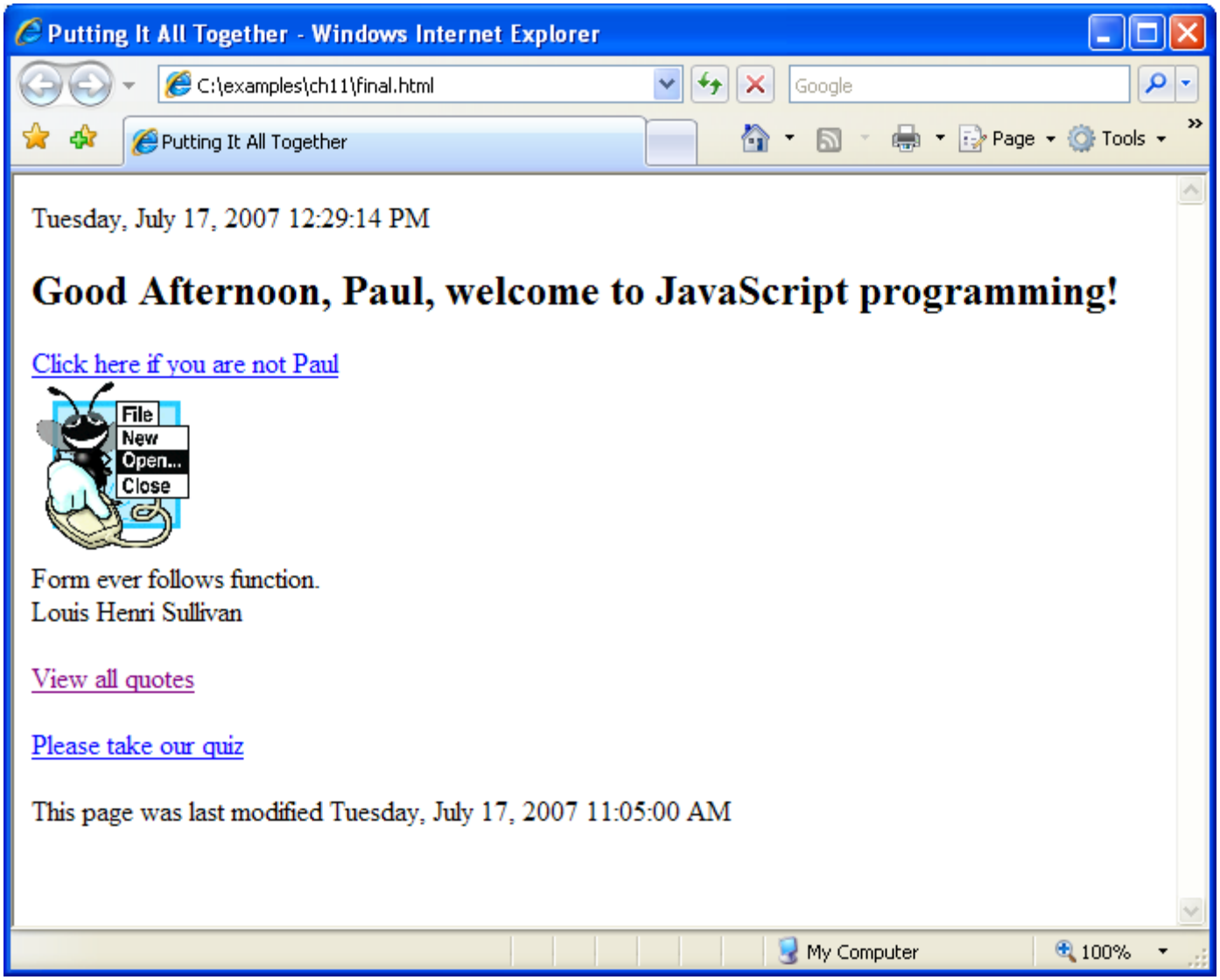

Fig. 11.16 | Rich welcome page using several JavaScript concepts (Part 8 of 8).

Fig. 11.17 | Online quiz in a child window (Part 1 of 5).

```
<?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
         //-->
      </script>
   </head>
```

Writes a congratulatory message to replace the link in the `quizSpot` element in the window that opened the quiz

Replaces the `quizSpot` element in the window that opened the quiz with a “try again” message and a new copy of the link

Gives the main window focus so that the user can see the response to the quiz input

```
   <body>
      <form id = "myQuiz" action = "javascript:checkAnswers()">
         <p>Select the name of the tip that goes with the
            image shown:<br />
            <img src = "EPT.gif" alt = "mystery tip"/>
            <br />

            <input type = "radio" name = "radiobutton" value = "CPE" />
            <label>Common Programming Error</label>

            <input type = "radio" name = "radiobutton" value = "EPT" />
            <label>Error-Prevention Tip</label>

            <input type = "radio" name = "radiobutton" value = "PERF" />
            <label>Performance Tip</label>

            <input type = "radio" name = "radiobutton" value = "PORT" />
            <label>Portability Tip</label><br />

            <input type = "submit" name = "Submit" value = "Submit" />
            <input type = "reset" name = "reset" value = "Reset" />
         </p>
      </form>
   </body>
</html>
```

Fig. 11.17 | Online quiz in a child window (Part 2 of 5).

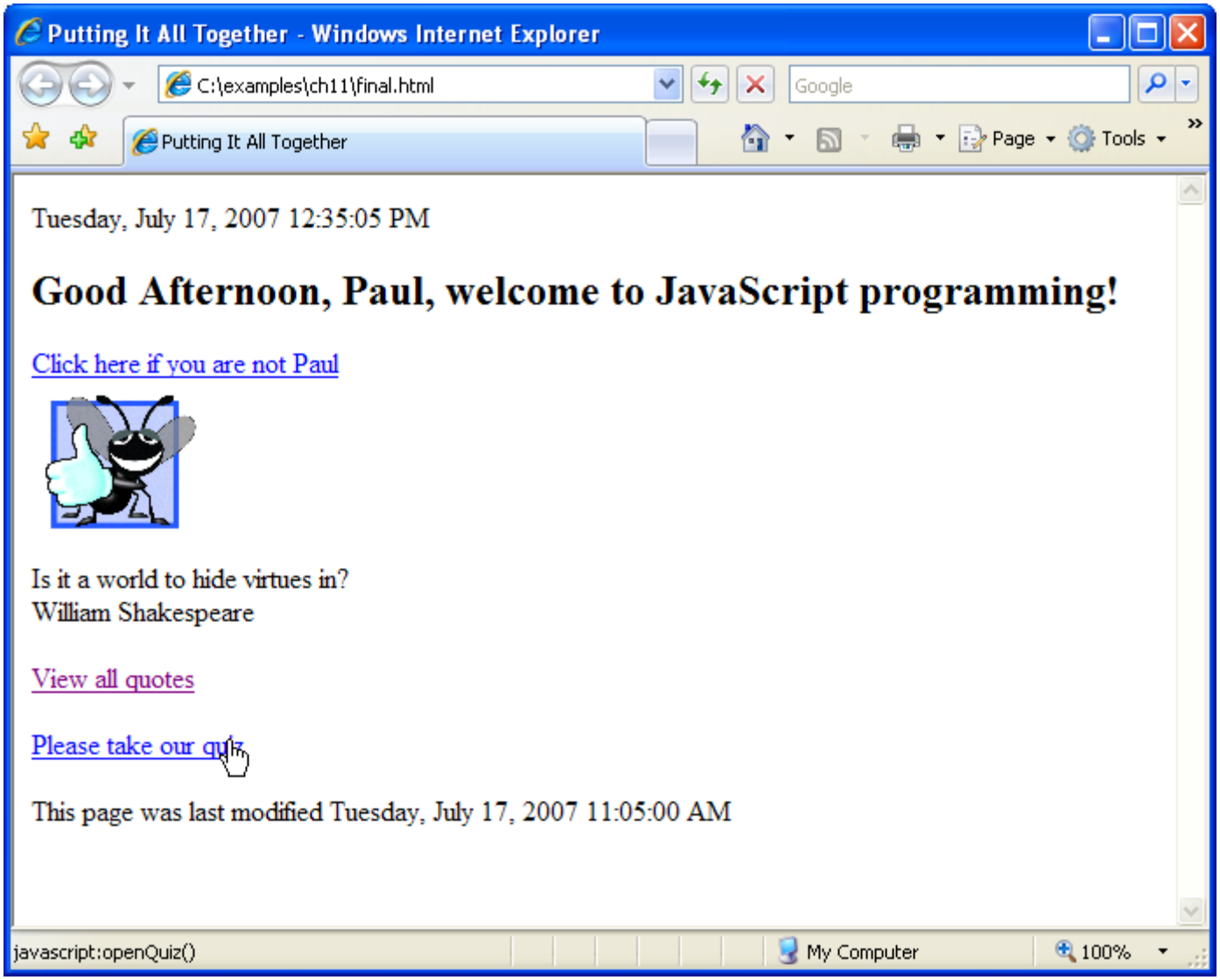

Fig. 11.17 | Online quiz in a child window (Part 3 of 5).

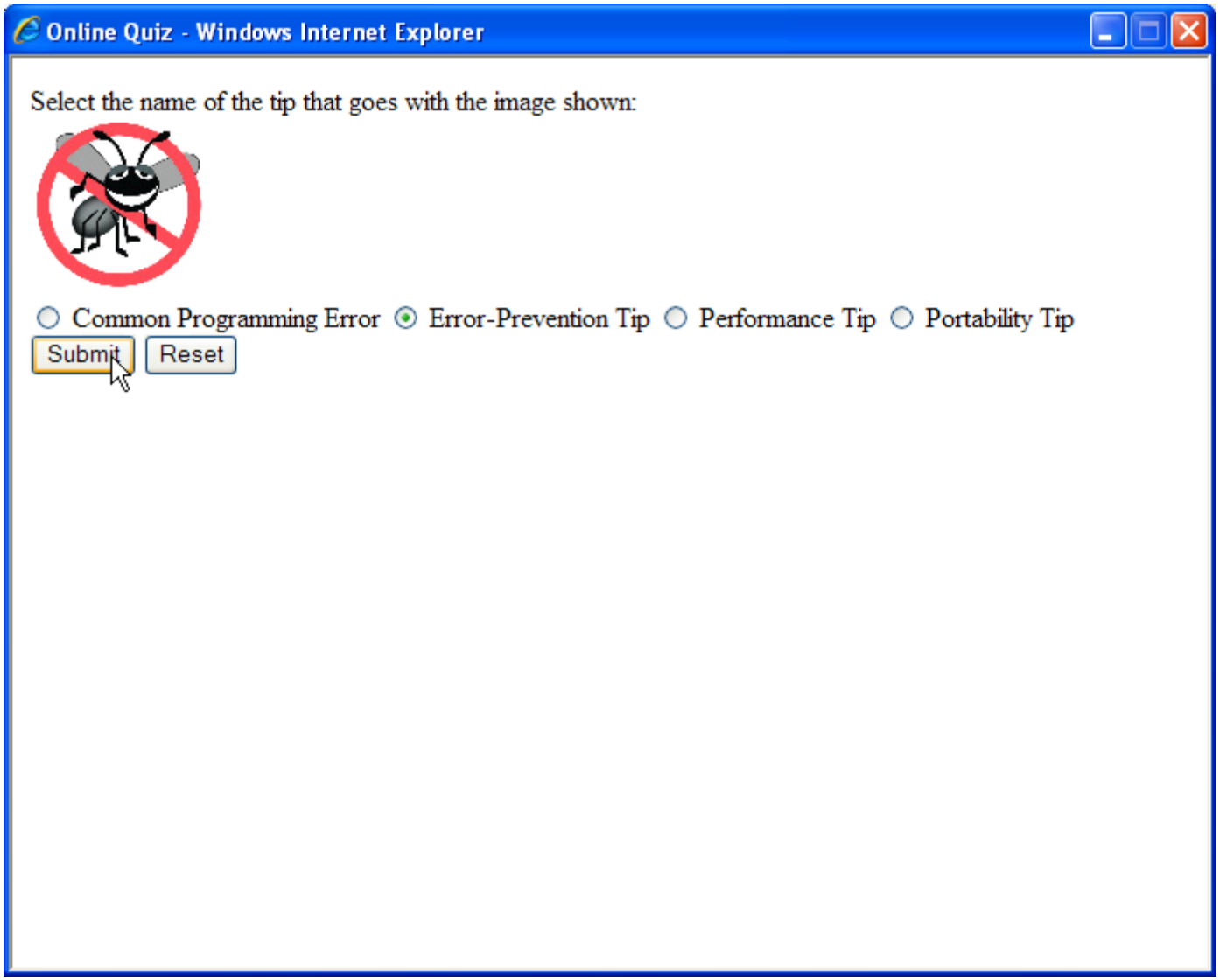

Fig. 11.17 | Online quiz in a child window (Part 4 of 5).

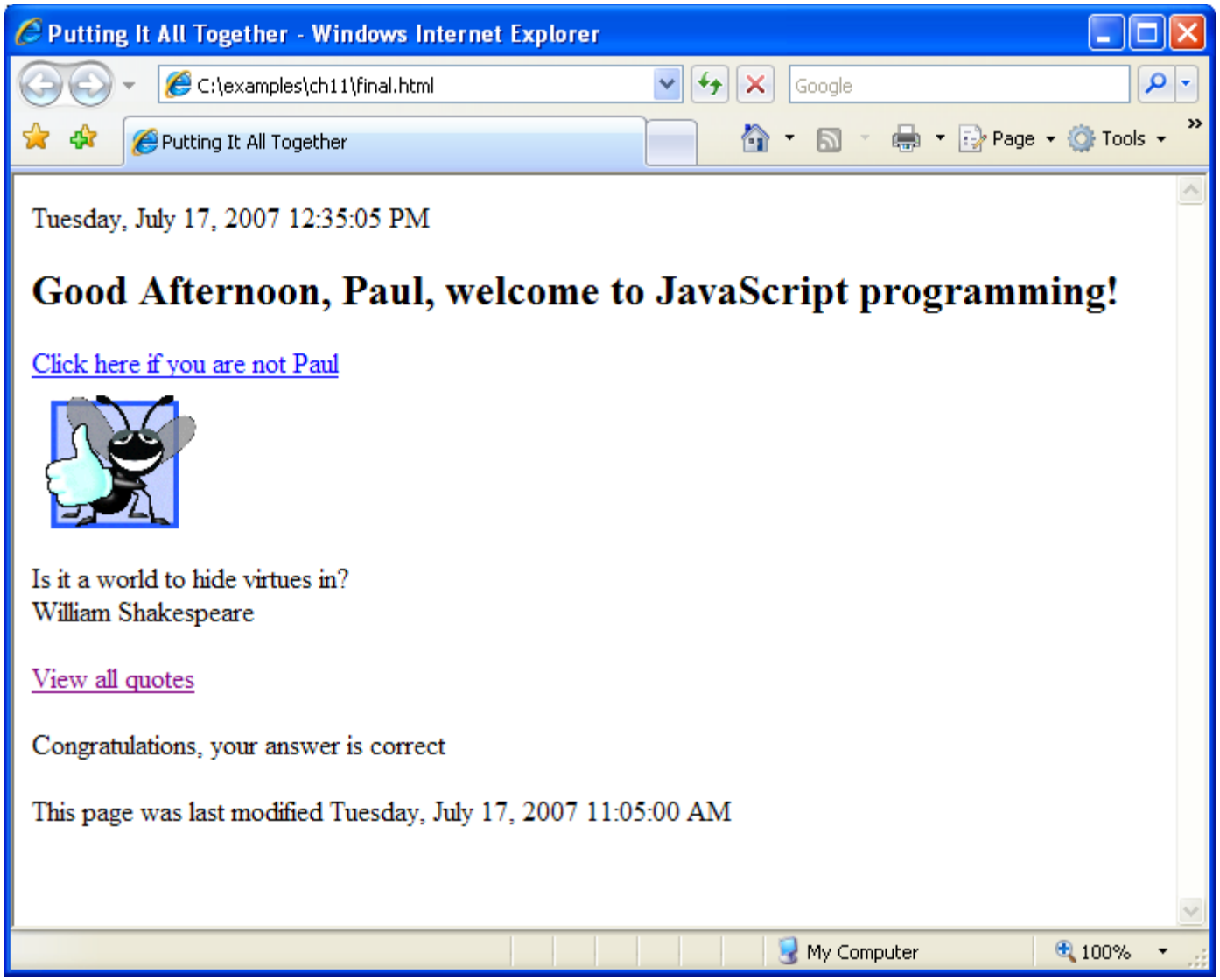

Fig. 11.17 | Online quiz in a child window (Part 5 of 5).

11.9 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`.**