JavaScript: Objects

11.1 Introduction

- This chapter presents a more formal treatment of objects
- We use HTML5's new web storage capabilities to create a web application that stores a user's favorite Twitter searches on the computer for easy access at a later time
- We also provide a brief introduction to JSON
  - Means for creating JavaScript objects
  - Transferring data over the Internet between client-side and server-side programs

11.2 Math Object

- Math object methods enable you to conveniently 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
  - **var result = Math.sqrt(900);**
- In parentheses following the method name is are arguments to the method

Software Engineering Observation 11.1

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

Method	Description	Examples

`abs( x )`	Absolute value of `x`.	`abs( 7.2 )` is `7.2`
		`abs( 0 )` is `0`
		`abs( -5.6 )` is `5.6`

`ceil( x )`	Rounds `x` to the smallest integer not less than `x`.	`ceil( 9.2 )` is `10` `ceil( -9.8 )` is `-9.0`
`cos( x )`	Trigonometric cosine of `x` (`x` in radians).	`cos( 0 )` is `1`
`exp( x )`	Exponential method e^x.	`exp( 1 )` is `2.71828` `exp( 2 )` is `7.38906`
`floor( x )`	Rounds `x` to the largest integer not greater than `x`.	`floor( 9.2 )` is `9` `floor( -9.8 )` is `-10.0`
`log( x )`	Natural logarithm of `x` (base *e*).	`log( 2.718282 )` is `1` `log( 7.389056 )` is `2`
`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`

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

Method	Description	Examples
`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, 7 )` is `128` `pow( 9, .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 )` is `0`
`sqrt( x )`	Square root of x.	`sqrt( 900 )` is `30` `sqrt( 9 )` is `3`
`tan( x )`	Trigonometric tangent of x (x in radians).	`tan( 0 )` is `0`

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

The Math object defines several properties that represent commonly used mathematical constants

Constant	Description	Value
`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
`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.

11.3 String Object

- Characters are the building blocks of JavaScript programs
- Every program is composed of a sequence of characters grouped together meaningfully
  - 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 are written as a sequence of characters in double or single quotation marks

11.3.2 Methods of the String Object

- Combining strings is called concatenation
  - **"John Q. Doe" (a name)**
  - **'9999 Main Street' (a street address)**
  - **"Waltham, Massachusetts" (a city and state)**
  - **'(201) 555-1212' (a telephone number)**
  - **var color = "blue";**

Method	Description
`charAt( index )`	Returns a string containing the character at the specified *index*. If there's 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's no character at that *index*.
`concat( string )`	Concatenates its argument to the end of the string on which the method is invoked. The original string 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.

Fig. 11.3 | Some `String`-object methods. (Part 1 of 3.)

Method	Description
`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*, replaces the first occurrence with *replaceString* and returns the modified string, or returns 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 2 of 3.)

Method	Description
`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. Non-letter characters are not changed.
`toUpperCase()`	Returns a string in which all lowercase letters are converted to uppercase letters. Non-letter characters are not changed.

Fig. 11.3 | Some `String`-object methods. (Part 3 of 3.)

11.3.3 Character Processing Methods

- String method **charAt**
  - Returns the character at a specific position
  - 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
  - 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 position
  - If the index is outside the bounds of the string, the method returns NaN

11.3.3 Character Processing Methods (Cont.)

- String method **fromCharCode**
  - Returns a string created from a series of Unicode values
- String method **toLowerCase**
  - Returns the lowercase version of a string
- String method **toUpperCase**
  - Returns the uppercase version of a string

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 11.4: CharacterProcessing.html -->
4  <!-- HTML5 document to demonstrate String methods charAt, charCodeAt,
5     fromCharCode, toLowercase and toUpperCase. -->
6  <html>
7     <head>
8        <meta charset = "utf-8">
9        <title>Character Processing</title>
10       <link rel = "stylesheet" type = "text/css" href = "style.css">
11       <script src = "CharacterProcessing.js"></script>
12    </head>
13    <body>
14       <div id = "results"></div>
15    </body>
16 </html>
```

Fig. 11.4 | HTML5 document to demonstrate methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 1 of 2.)

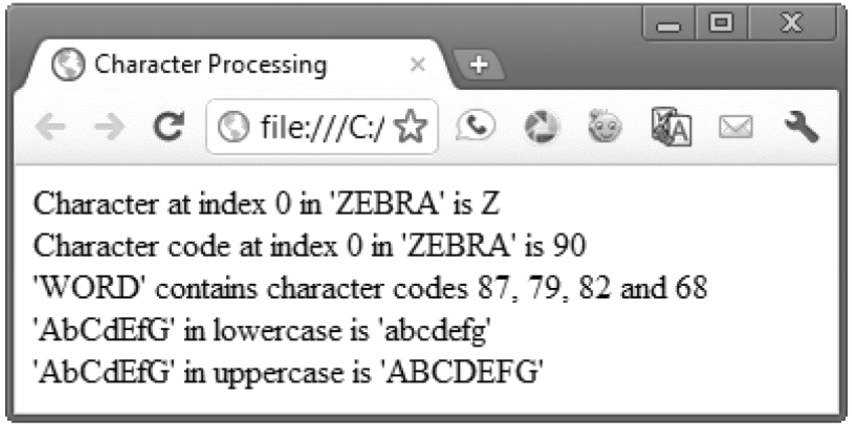

Fig. 11.4 | HTML5 document to demonstrate methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 2 of 2.)

```
1  // Fig. 11.5: CharacterProcessing.js
2  // String methods charAt, charCodeAt, fromCharCode,
3  // toLowercase and toUpperCase.
4  function start()
5  {
6     var s = "ZEBRA";
7     var s2 = "AbCdEfG";
8     var result = "";
9
10    result = "<p>Character at index 0 in '" + s + "' is " +
11       s.charAt( 0 ) + "</p>";
12    result += "<p>Character code at index 0 in '" + s + "' is " +
13       s.charCodeAt( 0 ) + "</p>";
14
15    result += "<p>'" + String.fromCharCode( 87, 79, 82, 68 ) +
16       "' contains character codes 87, 79, 82 and 68</p>";
17
```

Fig. 11.5 | String methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 1 of 2.)

```
18      result += "<p>'" + s2 + "' in lowercase is '" +
19         s2.toLowerCase() + "'</p>";
20      result += "<p>'" + s2 + "' in uppercase is '" +
21         s2.toUpperCase() + "'</p>";
22
23      document.getElementById( "results" ).innerHTML = result;
24   } // end function start
25
26   window.addEventListener( "load", start, false );
```

Fig. 11.5 | String methods charAt, charCodeAt, fromCharCode, toLowercase and toUpperCase. (Part 2 of 2.)

11.3.4. Searching Methods

- 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
  - 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
  - Receives an optional second argument specifying the index from which to begin the search

```
<!DOCTYPE html>

<!-- Fig. 11.6: SearchingStrings.html -->
<!-- HTML document to demonstrate methods indexOf and lastIndexOf. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Searching Strings</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "SearchingStrings.js"></script>
   </head>
   <body>
      <form id = "searchForm" action = "#">
         <h1>The string to search is:
             abcdefghijklmnopqrstuvwxyzabcdefghijklm</h1>
         <p>Enter the substring to search for
         <input id = "inputField" type = "search">
         <input id = "searchButton" type = "button" value = "Search"></p>
         <div id = "results"></div>
      </form>
   </body>
</html>
```

Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 1 of 3.)

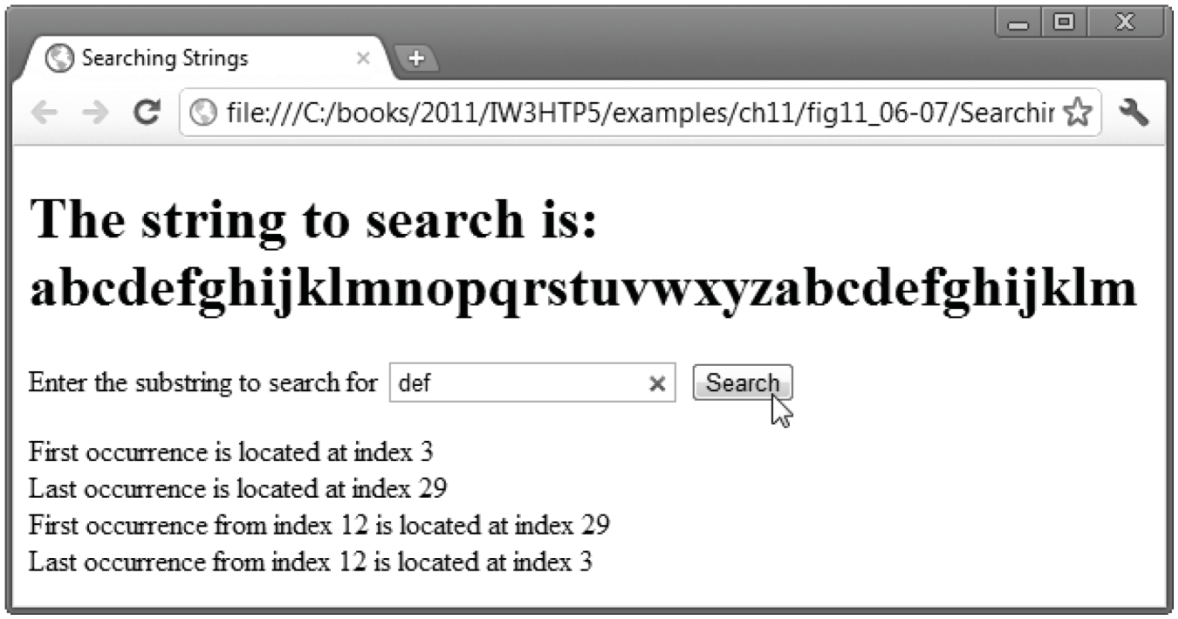

Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 2 of 3.)

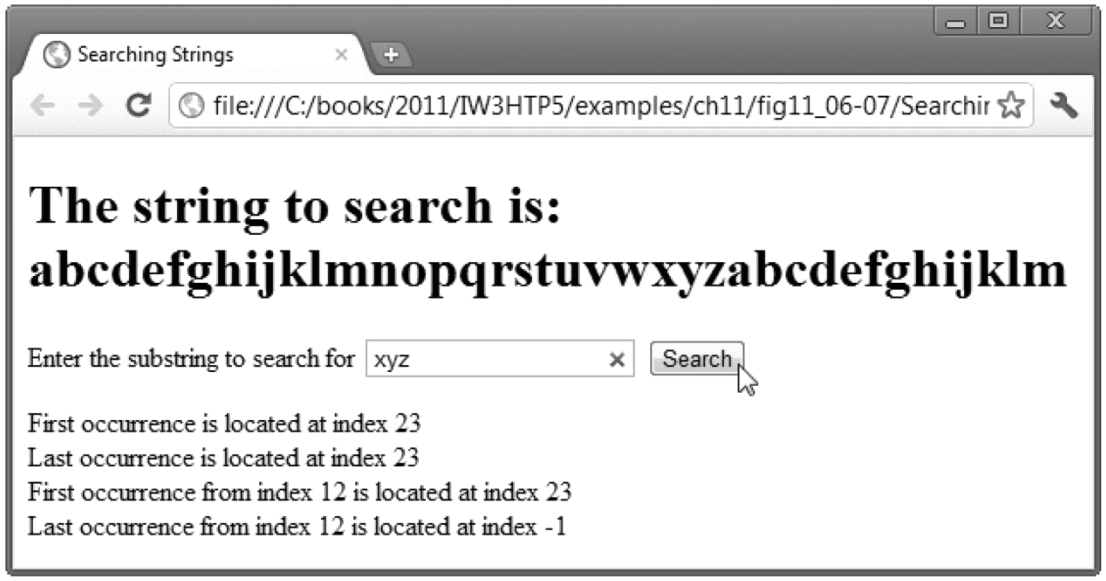

Fig. 11.6 | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part 3 of 3.)

```
1  // Fig. 11.7: SearchingStrings.js
2  // Searching strings with indexOf and lastIndexOf.
3  var letters = "abcdefghijklmnopqrstuvwxyzabcdefghijklm";
4
5  function buttonPressed()
6  {
7     var inputField = document.getElementById( "inputField" );
8
9     document.getElementById( "results" ).innerHTML =
10       "<p>First occurrence is located at index " +
11          letters.indexOf( inputField.value ) + "</p>" +
12       "<p>Last occurrence is located at index " +
13          letters.lastIndexOf( inputField.value ) + "</p>" +
14       "<p>First occurrence from index 12 is located at index " +
15          letters.indexOf( inputField.value, 12 ) + "</p>" +
16       "<p>Last occurrence from index 12 is located at index " +
17          letters.lastIndexOf( inputField.value, 12 ) + "</p>";
18 } // end function buttonPressed
```

Fig. 11.7 | Searching strings with `indexOf` and `lastIndexOf`. (Part 1 of 2.)

```
19
20   // register click event handler for searchButton
21   function start()
22   {
23      var searchButton = document.getElementById( "searchButton" );
24      searchButton.addEventListener( "click", buttonPressed, false );
25   } // end function start
26
27   window.addEventListener( "load", start, false );
```

Fig. 11.7 | Searching strings with `indexOf` and `lastIndexOf`. (Part 2 of 2.)

11.3.5 Splitting Strings and Obtaining Substrings

- Breaking a string into tokens is called tokenization
- Tokens are separated from one another by delimiters
  - Characters such as blank, tab, newline and carriage return
  - Other characters may also be used as delimiters
- String method **split**
  - Breaks a string into its component tokens
  - Argument is the delimiter string
  - Returns an array of strings containing the tokens
  - **var tokens = inputString.split(" ");**

11.3.5 Splitting Strings and Obtaining Substrings

- 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
  - **inputString.substring(0, 10);**

```
<!DOCTYPE html>

<!-- Fig. 11.8: SplitAndSubString.html -->
<!-- HTML document demonstrating String methods split and substring. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>split and substring</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "SplitAndSubString.js"></script>
   </head>
   <body>
      <form action = "#">
         <p>Enter a sentence to split into words:</p>
         <p><input id = "inputField" type = "text">
            <input id = "splitButton" type = "button" value = "Split"></p>
         <div id = "results"></p>
      </form>
   </body>
</html>
```

Fig. 11.8 | HTML document demonstrating `String` methods `split` and `substring`. (Part 1 of 2.)

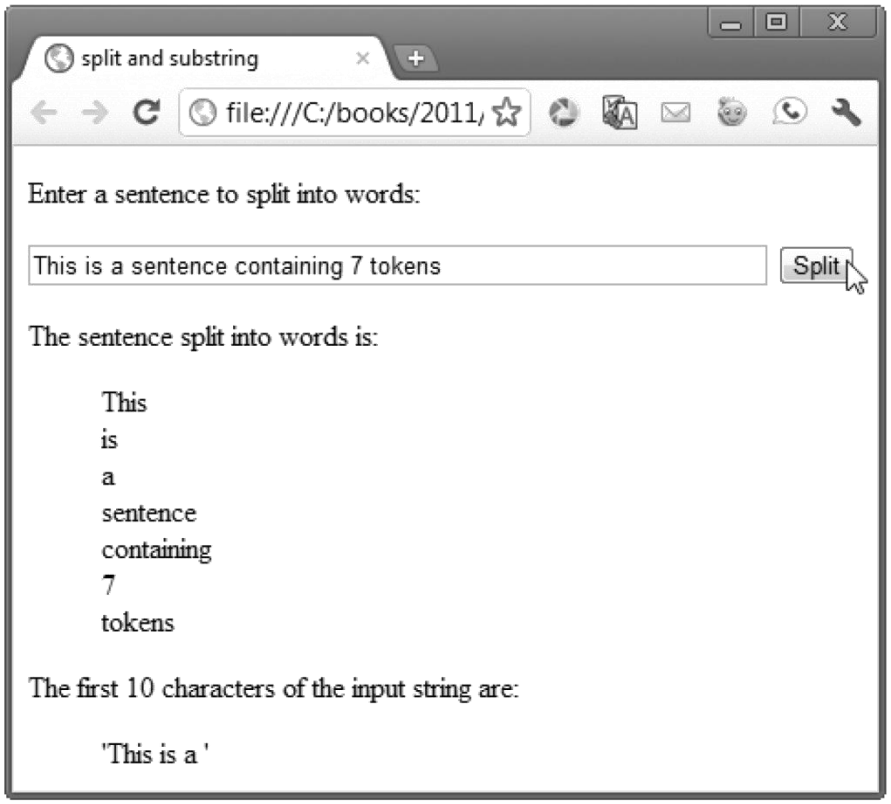

Fig. 11.8 | HTML document demonstrating `String` methods `split` and `substring`. (Part 2 of 2.)

```
1  // Fig. 11.9: SplitAndSubString.js
2  // String object methods split and substring.
3  function splitButtonPressed()
4  {
5     var inputString = document.getElementById( "inputField" ).value;
6     var tokens = inputString.split( " " );
7
8     var results = document.getElementById( "results" );
9     results.innerHTML = "<p>The sentence split into words is: </p>" +
10       "<p class = 'indent'>" +
11       tokens.join( "</p><p class = 'indent'>" ) + "</p>" +
12       "<p>The first 10 characters of the input string are: </p>" +
13       "<p class = 'indent'>'" + inputString.substring( 0, 10 ) + "'</p>";
14 } // end function splitButtonPressed
15
16 // register click event handler for searchButton
17 function start()
18 {
19    var splitButton = document.getElementById( "splitButton" );
20    splitButton.addEventListener( "click", splitButtonPressed, false );
21 } // end function start
22
23 window.addEventListener( "load", start, false );
```

Fig. 11.9 | String-object methods `split` and `substring`.

11.3.5 Splitting Strings and Obtaining Substrings (Cont.)

- **delimiter** string
  - The string that determines the end of each token in the original string
- 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 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

11.4 Date Object

- 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

11.4 Date Object

- 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

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.

Fig. 11.10 | Date-object methods. (Part 1 of 4.)

Method	Description
`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. 11.10 | `Date`-object methods. (Part 2 of 4.)

Method	Description
`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 seconds in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current milliseconds 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.

Fig. 11.10 | `Date`-object methods. (Part 3 of 4.)

Method	Description
`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. 11.10 | `Date`-object methods. (Part 4 of 4.)

11.4 Date Object (Cont.)

- 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
  - Arguments include the required year, month and date, and the optional hours, minutes, seconds and milliseconds
  - If an argument is not specified, a 0 is supplied in its place
  - For hours, minutes and seconds, if the argument to the right of any of these arguments is specified, that argument must also be specified

```
<!DOCTYPE html>

<!-- Fig. 11.11: DateTime.html -->
<!-- HTML document to demonstrate Date-object methods. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Date and Time Methods</title>
      <link rel = "stylesheet" type = "text/css" href = "style.css">
      <script src = "DateTime.js"></script>
   </head>
   <body>
      <h1>String representations and valueOf</h1>
      <section id = "strings"></section>
      <h1>Get methods for local time zone</h1>
      <section id = "getMethods"></section>
      <h1>Specifying arguments for a new Date</h1>
      <section id = "newArguments"></section>
      <h1>Set methods for local time zone</h1>
      <section id = "setMethods"></section>
   </body>
</html>
```

Fig. 11.11 | HTML document to demonstrate `Date`-object methods. (Part 1 of 3.)

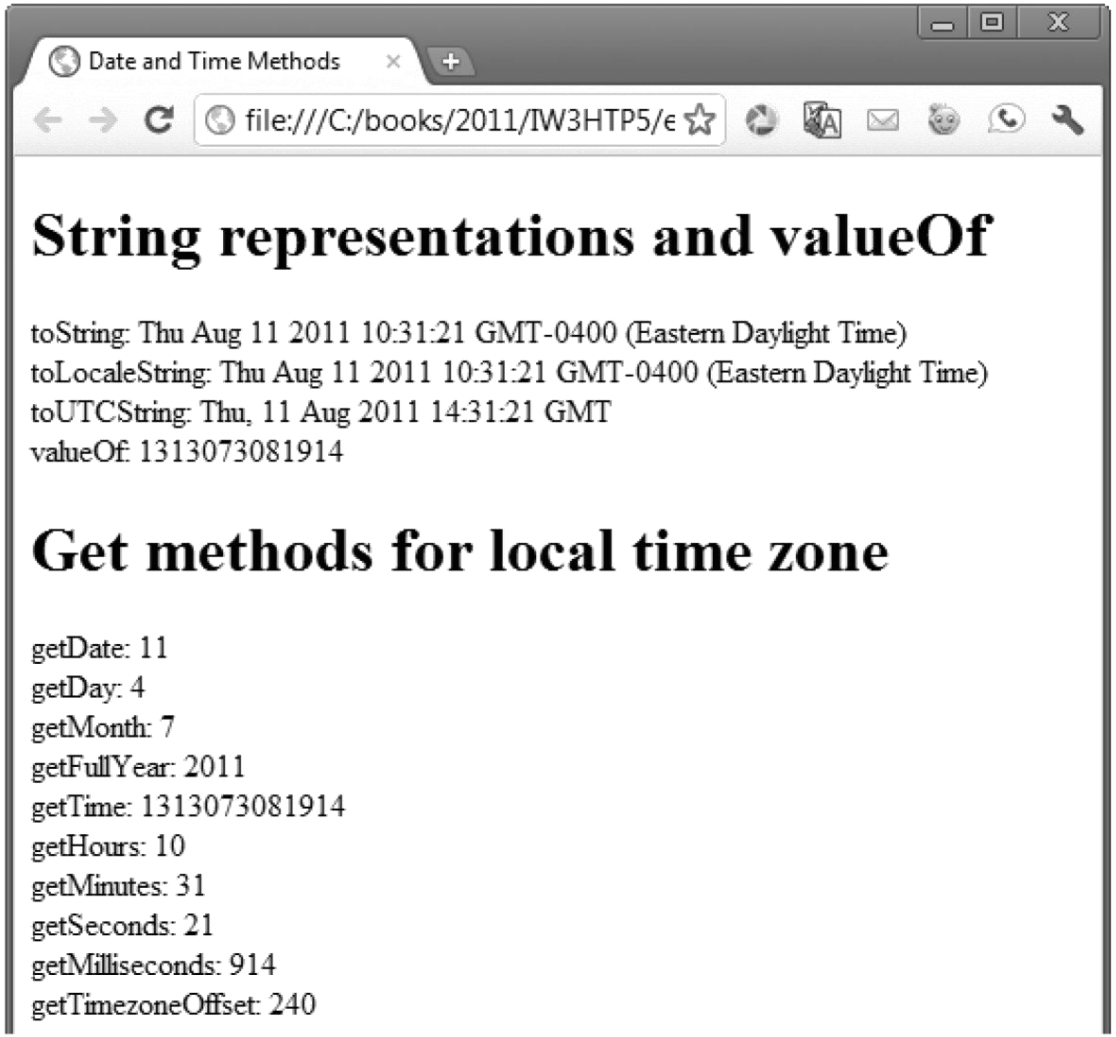

Fig. 11.11 | HTML document to demonstrate `Date`-object methods. (Part 2 of 3.)

Fig. 11.11 | HTML document to demonstrate `Date`-object methods. (Part 3 of 3.)

```
1  // Fig. 11.12: DateTime.js
2  // Date and time methods of the Date object.
3  function start()
4  {
5     var current = new Date();
6
7     // string-formatting methods and valueOf
8     document.getElementById( "strings" ).innerHTML =
9        "<p>toString: " + current.toString() + "</p>" +
10       "<p>toLocaleString: " + current.toLocaleString() + "</p>" +
11       "<p>toUTCString: " + current.toUTCString() + "</p>" +
12       "<p>valueOf: " + current.valueOf() + "</p>";
13
14    // get methods
15    document.getElementById( "getMethods" ).innerHTML =
16       "<p>getDate: " + current.getDate() + "</p>" +
17       "<p>getDay: " + current.getDay() + "</p>" +
18       "<p>getMonth: " + current.getMonth() + "</p>" +
19       "<p>getFullYear: " + current.getFullYear() + "</p>" +
20       "<p>getTime: " + current.getTime() + "</p>" +
21       "<p>getHours: " + current.getHours() + "</p>" +
22       "<p>getMinutes: " + current.getMinutes() + "</p>" +
23       "<p>getSeconds: " + current.getSeconds() + "</p>" +
24       "<p>getMilliseconds: " + current.getMilliseconds() + "</p>" +
25       "<p>getTimezoneOffset: " + current.getTimezoneOffset() + "</p>";
```

Fig. 11.12 | Date and time methods of the `Date` object. (Part 1 of 2.)

```
26
27      // creating a Date
28      var anotherDate = new Date( 2011, 2, 18, 1, 5, 0, 0 );
29      document.getElementById( "newArguments" ).innerHTML =
30         "<p>Date: " + anotherDate + "</p>";
31
32      // set methods
33      anotherDate.setDate( 31 );
34      anotherDate.setMonth( 11 );
35      anotherDate.setFullYear( 2011 );
36      anotherDate.setHours( 23 );
37      anotherDate.setMinutes( 59 );
38      anotherDate.setSeconds( 59 );
39      document.getElementById( "setMethods" ).innerHTML =
40         "<p>Modified date: " + anotherDate + "</p>";
41   } // end function start
42
43   window.addEventListener( "load", start, false );
```

Fig. 11.12 | Date and time methods of the `Date` object. (Part 2 of 2.)

Common Programming Error 11.1

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

11.5 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);**
  - 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

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. 11.13 | `Boolean`-object methods.

11.5 Boolean and Number Objects (Cont.)

- JavaScript automatically creates Number objects to store numeric values in a script
- You can create a Number object with the statement
  - **var n = new Number(numericValue);**
- Although you can explicitly create Number objects, normally they are created when needed by the JavaScript interpreter

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. Radix 2 results in the *binary* representation, 8 in the *octal* representation, 10 in the *decimal* representation and 16 in the *hexadecimal* representation. See Appendix E, Number Systems, for an explanation of the binary, octal, decimal and hexadecimal number systems.
`valueOf()`	Returns the numeric value.
`Number.MAX_VALUE`	The largest value that can be stored in a JavaScript program.
`Number.MIN_VALUE`	The smallest value that can be stored in a JavaScript program.
`Number.NaN`	*Not a number*—a value returned from an arithmetic expression that doesn't result in a number (e.g., `parseInt("hello")` cannot convert the string `"hello"` to 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`	A value less than `-Number.MAX_VALUE`.
`Number.POSITIVE_INFINITY`	A value greater than `Number.MAX_VALUE`.

Fig. 11.14 | Number-object methods and properties.

11.6 document Object

- document object
  - Provided by the browser and allows JavaScript code to manipulate the current document in the browser
  - http://www.w3schools.com/jsref/dom_obj_document.asp
  - http://www.w3schools.com/jsref/met_document_getelementsbytagname.asp

Method	Description
`getElementById( id )`	Returns the HTML5 element whose `id` attribute matches *id*.
`getElementByTagName( tagName )`	Returns an array of the HTML5 elements with the specified *tagName*.

Fig. 11.15 | document-object methods.

11.7 Favorite Twitter Searches

- Before HTML5, websites could store only small amounts of text-based information on a user's computer using cookies
- A cookie is a key/value pair in which each key has a corresponding value
  - The key and value are both strings
- Cookies are stored by the browser on the user's computer to maintain client-specific information during and between browser sessions
- A website might use a cookie to record user preferences or other information that it can retrieve during the client's subsequent visits

11.7 Favorite Twitter Searches (Cont.)

- When a user visits a website, the browser locates any cookies written by that website and sends them to the server
- Cookies may be accessed only by the web server and scripts of the website from which the cookies originated
- Problems with Cookies
  - They're extremely limited in size
    - http://www.ietf.org/rfc/rfc2109.txt
  - Cookies cannot store entire documents
  - If the user browses the same site from multiple tabs, all of the site's cookies are shared by the pages in each tab
    - This could be problematic in web applications that allow the user to purchase items

11.7 Favorite Twitter Searches (Cont.)

- As of HTML5, there are two new mechanisms for storing key/value pairs that help eliminate some of the problems with cookies
  - Web applications can use the window object's **localStorage** property to store up to several megabytes of key/value-pair string data on the user's computer and can access that data across browsing sessions and browser tabs
  - Web applications that need access to data for only a browsing session and that must keep that data separate among multiple tabs can use the window object's **sessionStorage** property
    - There's a separate sessionStorage object for every browsing session, including separate tabs that are accessing the same website

11.7 Favorite Twitter Searches (Cont.)

- Favorite Twitter Searches App Using **localStorage** and **sessionStorage**
- This app allows users to save their favorite (possibly lengthy) Twitter search strings with easy-to-remember, user-chosen, short tag names
- Users can then conveniently follow the tweets on their favorite topics by visiting this web page and clicking the link for a saved search

11.7 Favorite Twitter Searches (Cont.)

- The user's favorite searches are saved using **localStorage**
  - Immediately available each time the user browses the app's web page
  - **window.localStorage**
    - Stores data with no expiration date
- The app uses **sessionStorage** to determine whether the user has visited the page previously during the current browsing session
  - If not, the app displays a welcome message
  - **window.sessionStorage**
    - Stores data for one session (data is lost when the browser tab is closed)
- http://www.w3schools.com/html/html5_webstorage.asp

a) **Favorite Twitter Searches** app when it's loaded for the first time in this browsing session and there are no tagged searches

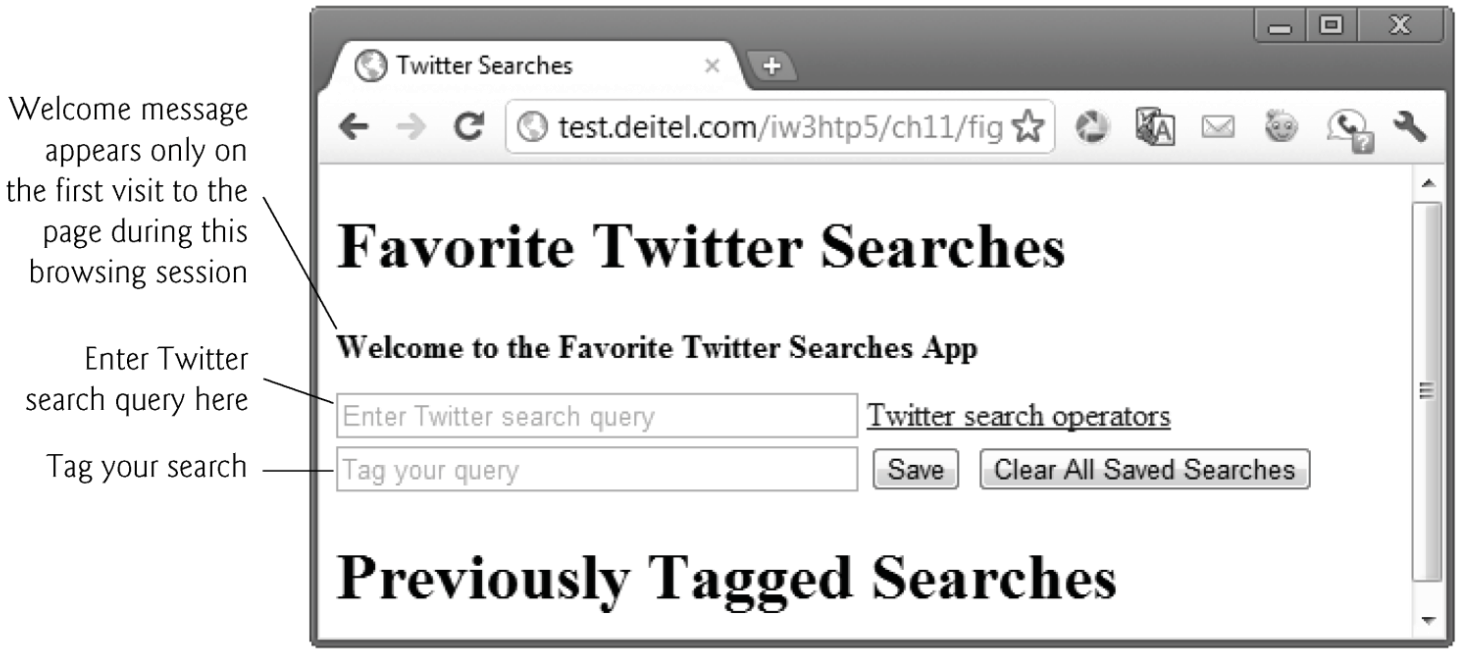

Fig. 11.16 | Sample outputs from the Favorite Twitter Searches web application. (Part 1 of 4.)

b) App with several saved searches and the user saving a new search

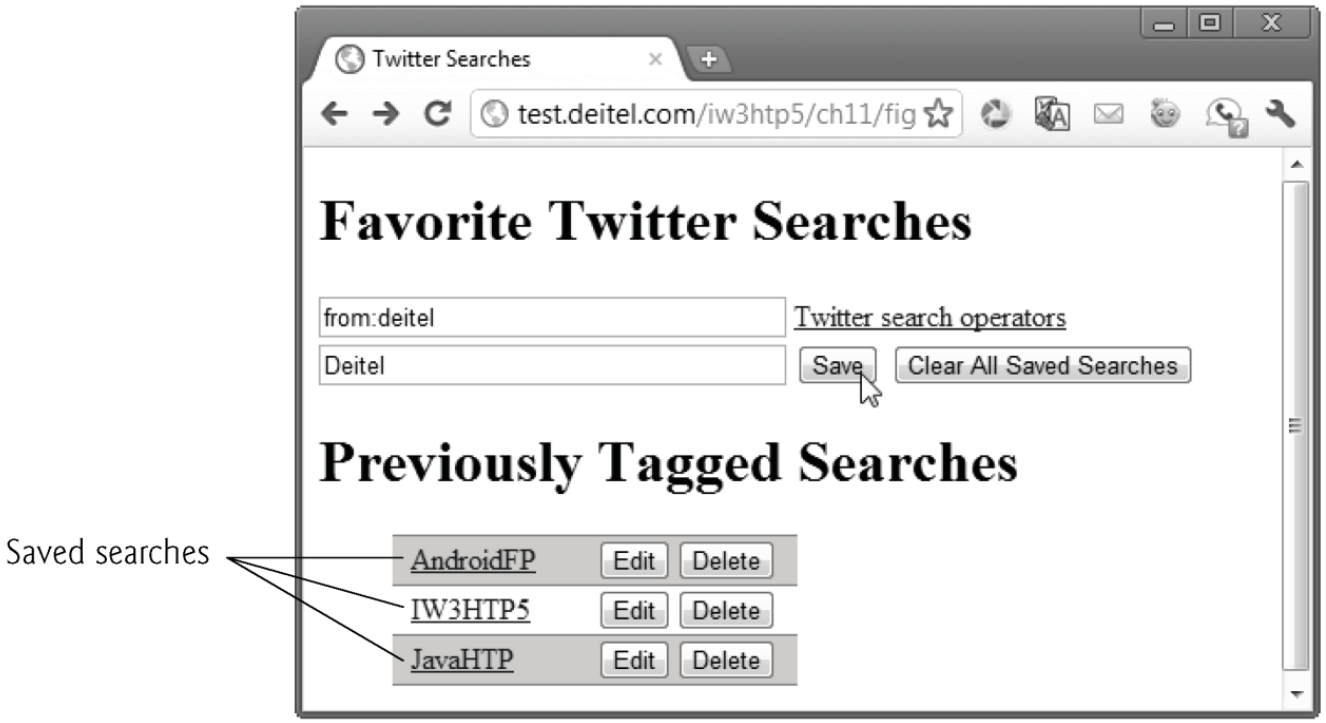

Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 2 of 4.)

c) App after new search is saved—the user is about to click the Deitel search

Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 3 of 4.)

d) Results of touching the **Deitel** link

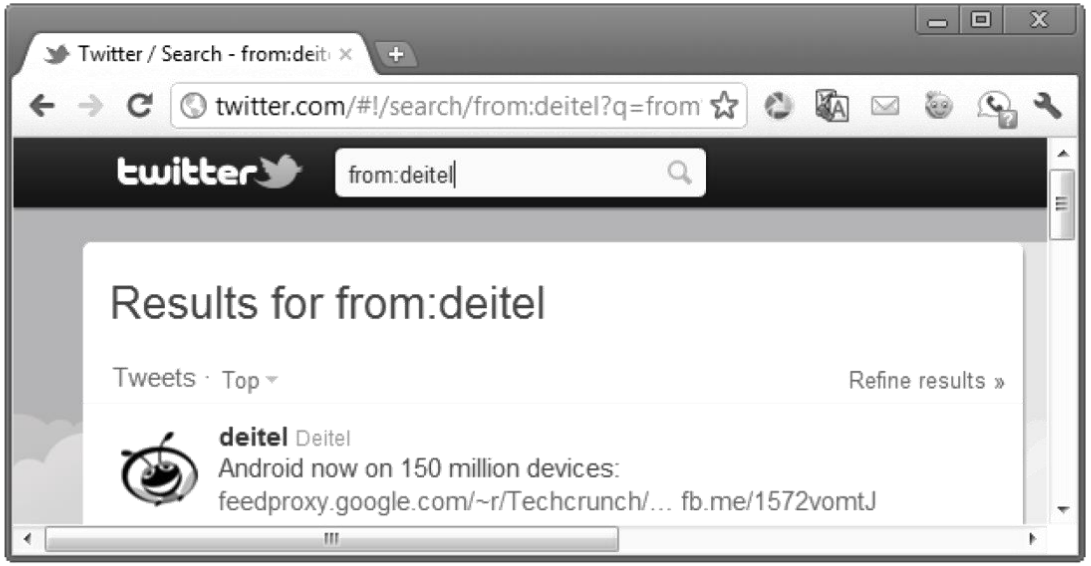

Fig. 11.16 | Sample outputs from the **Favorite Twitter Searches** web application. (Part 4 of 4.)

11.7 Favorite Twitter Searches (Cont.)

- Favorite Twitter Searches HTML5 Document
- The Favorite Twitter Searches application contains three files
  - FavoriteTwitterSearches.html
  - styles.css
  - FavoriteTwitterSearches.js
- The HTML5 document provides a form that allows the user to enter new searches
- Previously tagged searches are displayed in the div named searches

```
<!DOCTYPE html>

<!-- Fig. 11.17: FavoriteTwitterSearchs.html -->
<!-- Favorite Twitter Searches web application. -->
<html>
<head>
   <title>Twitter Searches</title>
   <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "FavoriteTwitterSearches.js"></script>
</head>
<body>
   <h1>Favorite Twitter Searches</h1>
   <p id = "welcomeMessage"></p>
   <form action = "#">
      <p><input id = "query" type = "text"
         placeholder = "Entery Twitter search query">
         <a href = "https://dev.twitter.com/docs/using-search">
            Twitter search operators</a></p>
      <p><input id = "tag" type = "text" placeholder = "Tag your query">
         <input type = "button" value = "Save"
            id = "saveButton">
         <input type = "button" value = "Clear All Saved Searches"
            id = "clearButton"></p>
   </form>
```

Fig. 11.17 | Favorite Twitter Searches web application. (Part 1 of 2.)

```
25      <h1>Previously Tagged Searches</h1>
26      <div id = "searches"></div>
27  </body>
28  </html>
```

Fig. 11.17 | Favorite Twitter Searches web application. (Part 2 of 2.)

11.7 Favorite Twitter Searches (Cont.)

- CSS for Favorite Twitter Searches
- styles.css contains the CSS styles for this app

```
1   p { margin: 0px; }
2   #welcomeMessage { margin-bottom: 10px; font-weight: bold; }
3   input[type = "text"] { width: 250px; }
4
5   /* list item styles */
6   span { margin-left: 10px; display: inline-block; width: 100px; }
7   li { list-style-type: none; width: 220px;}
8   li:first-child { border-top: 1px solid grey; }
9   li:nth-child(even) { background-color: lightyellow;
10     border-bottom: 1px solid grey; }
11  li:nth-child(odd) { background-color: lightblue;
12     border-bottom: 1px solid grey; }
```

Fig. 11.18 | Styles used in the **Favorite Twitter Searches** app.

11.7 Favorite Twitter Searches (Cont.)

- Script for Favorite Twitter Searches
- FavoriteTwitterSearches.js presents the JavaScript for the app
- When the HTML5 document loads, function start is called to register event handlers and call function **loadSearches**
- The **sessionStorage** object is used to determine whether the user has already visited the page during this browsing session
- The **getItem** method receives a name of a key as an argument
  - If the key exists, the method returns the corresponding string value
  - Otherwise, it returns null

11.7 Favorite Twitter Searches (Cont.)

- If this is the user's first visit to the page during this browsing session
  - The **setItem** method is used to set the key "herePreviously" to the string "true", then the app displays a welcome message
- The **localStorage** object's length represents the number of key/value pairs stored
- Method key receives an index as an argument and returns the corresponding key
- For simplicity, we use the **onclick** attributes of the dynamically generated Edit and Delete buttons to set the buttons' event handlers
  - This is an older mechanism for registering event handlers

11.7 Favorite Twitter Searches (Cont.)

- To register these with the elements' **addEventListener** method
  - We'd have to dynamically locate the buttons in the page after we've created them
  - Then register the event handlers, which would require significant additional code
- Separately, notice that each event handler is receiving the button input element's id as an argument
  - Enables the event handler to use the id value when handling the event

11.7 Favorite Twitter Searches (Cont.)

- Function **clearAllSearches** is called when the user clicks the Clear All Saved Searches button
- The **clear** method of the localStorage object removes all key/value pairs from the object
- **loadSearches** is called to refresh the list of saved searches in the web page
- Function **saveSearch** is called when the user clicks Save to save a search

11.7 Favorite Twitter Searches (Cont.)

- The **setItem** method stores a key/value pair in the localStorage object
  - If the key already exits, **setItem** replaces the corresponding value
  - Otherwise, it creates a new key/value pair
- loadSearches is called to refresh the list of saved searches in the web page
- **removeItem** method is called to remove a key/value pair from the localStorage object

```
1  // Fig. 11.19: FavoriteTwitterSearchs.js
2  // Storing and retrieving key/value pairs using
3  // HTML5 localStorage and sessionStorage
4  var tags; // array of tags for queries
5
6  // loads previously saved searches and displays them in the page
7  function loadSearches()
8  {
9     if ( !sessionStorage.getItem( "herePreviously" ) )
10    {
11       sessionStorage.setItem( "herePreviously", "true" );
12       document.getElementById( "welcomeMessage" ).innerHTML =
13          "Welcome to the Favorite Twitter Searches App";
14    } // end if
15
16    var length = localStorage.length; // number of key/value pairs
17    tags = []; // create empty array
18
19    // load all keys
20    for (var i = 0; i < length; ++i)
21    {
22       tags[i] = localStorage.key(i);
23    } // end for
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 1 of 4.)

```
24
25      tags.sort(); // sort the keys
26
27      var markup = "<ul>"; // used to store search link markup
28      var url = "http://search.twitter.com/search?q=";
29
30      // build list of links
31      for (var tag in tags)
32      {
33         var query = url + localStorage.getItem(tags[tag]);
34         markup += "<li><span><a href = '" + query + "'>" + tags[tag] +
35            "</a></span>" +
36            "<input id = '" + tags[tag] + "' type = 'button' " +
37               "value = 'Edit' onclick = 'editTag(id)'>" +
38            "<input id = '" + tags[tag] + "' type = 'button' " +
39               "value = 'Delete' onclick = 'deleteTag(id)'>";
40      } // end for
41
42      markup += "</ul>";
43      document.getElementById("searches").innerHTML = markup;
44   } // end function loadSearches
45
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 2 of 4.)

```
46  // deletes all key/value pairs from localStorage
47  function clearAllSearches()
48  {
49     localStorage.clear();
50     loadSearches(); // reload searches
51  } // end function clearAllSearches
52
53  // saves a newly tagged search into localStorage
54  function saveSearch()
55  {
56     var query = document.getElementById("query");
57     var tag = document.getElementById("tag");
58     localStorage.setItem(tag.value, query.value);
59     tag.value = ""; // clear tag input
60     query.value = ""; // clear query input
61     loadSearches(); // reload searches
62  } // end function saveSearch
63
64  // deletes a specific key/value pair from localStorage
65  function deleteTag( tag )
66  {
67     localStorage.removeItem( tag );
68     loadSearches(); // reload searches
69  } // end function deleteTag
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 3 of 4.)

```
70
71  // display existing tagged query for editing
72  function editTag( tag )
73  {
74     document.getElementById("query").value = localStorage[ tag ];
75     document.getElementById("tag").value = tag;
76     loadSearches(); // reload searches
77  } // end function editTag
78
79  // register event handlers then load searches
80  function start()
81  {
82     var saveButton = document.getElementById( "saveButton" );
83     saveButton.addEventListener( "click", saveSearch, false );
84     var clearButton = document.getElementById( "clearButton" );
85     clearButton.addEventListener( "click", clearAllSearches, false );
86     loadSearches(); // load the previously saved searches
87  } // end function start
88
89  window.addEventListener( "load", start, false );
```

Fig. 11.19 | Storing and retrieving key/value pairs using HTML5 `localStorage` and `sessionStorage`. (Part 4 of 4.)

11.8 Using JSON to Represent Objects

- JavaScript Object Notation (JSON)
  - A format for storing and transporting data
  - 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
  - **[value0, value1, value2]**

11.8 Using JSON to Represent Objects (Cont.)

- 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
  - **var text = '{"employees":[' + '{"firstName":"John","lastName":"Doe","gender":" M" },' + '{"firstName":"Anna","lastName":"Smith", "gender":"F" },' + '{"firstName":"Peter","lastName":"Jones", "gender":"M" }]}';**

11.8 Using JSON to Represent Objects (Cont.)

- 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 easily extract data and efficiently transmit it across the Internet
  - http://www.w3schools.com/js/js_json.asp