



# JavaScript: Objects



## OBJECTIVES

In this chapter you'll:

- Learn object-based programming terminology and concepts.
- Learn the concepts of encapsulation and data hiding.
- Learn the value of object orientation.
- Use the methods of the JavaScript objects `Math`, `String`, `Date`, `Boolean` and `Number`.
- Use HTML5 web storage to create a web application that stores user data locally.
- Represent objects simply using JSON.



# 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.



## 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**
- ▶ In parentheses following the method name are arguments to the method



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.**



Method	Description	Examples
<code>abs( x )</code>	Absolute value of x.	<code>abs( 7.2 )</code> is <code>7.2</code> <code>abs( 0 )</code> is <code>0</code> <code>abs( -5.6 )</code> is <code>5.6</code>
<code>ceil( x )</code>	Rounds x to the smallest integer not less than x.	<code>ceil( 9.2 )</code> is <code>10</code> <code>ceil( -9.8 )</code> is <code>-9.0</code>
<code>cos( x )</code>	Trigonometric cosine of x (x in radians).	<code>cos( 0 )</code> is <code>1</code>
<code>exp( x )</code>	Exponential method $e^x$ .	<code>exp( 1 )</code> is <code>2.71828</code> <code>exp( 2 )</code> is <code>7.38906</code>
<code>floor( x )</code>	Rounds x to the largest integer not greater than x.	<code>floor( 9.2 )</code> is <code>9</code> <code>floor( -9.8 )</code> is <code>-10.0</code>
<code>log( x )</code>	Natural logarithm of x (base e).	<code>log( 2.718282 )</code> is <code>1</code> <code>log( 7.389056 )</code> is <code>2</code>
<code>max( x, y )</code>	Larger value of x and y.	<code>max( 2.3, 12.7 )</code> is <code>12.7</code> <code>max( -2.3, -12.7 )</code> is <code>-2.3</code>

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



Method	Description	Examples
<code>min( x, y )</code>	Smaller value of x and y.	<code>min( 2.3, 12.7 )</code> is <code>2.3</code> <code>min( -2.3, -12.7 )</code> is <code>-12.7</code>
<code>pow( x, y )</code>	$x$ raised to power $y$ ( $x^y$ ).	<code>pow( 2, 7 )</code> is <code>128</code> <code>pow( 9, .5 )</code> is <code>3.0</code>
<code>round( x )</code>	Rounds $x$ to the closest integer.	<code>round( 9.75 )</code> is <code>10</code> <code>round( 9.25 )</code> is <code>9</code>
<code>sin( x )</code>	Trigonometric sine of $x$ ( $x$ in radians).	<code>sin( 0 )</code> is <code>0</code>
<code>sqrt( x )</code>	Square root of $x$ .	<code>sqrt( 900 )</code> is <code>30</code> <code>sqrt( 9 )</code> is <code>3</code>
<code>tan( x )</code>	Trigonometric tangent of $x$ ( $x$ in radians).	<code>tan( 0 )</code> is <code>0</code>

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

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	$\pi$ —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 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 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**



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

**Fig. 11.3 | Some String-object methods. (Part 2 of 3.)**



Method	Description
<code>split( string )</code>	Splits the source string into an array of strings (tokens), where its <i>string</i> argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).
<code>substr( start, length )</code>	Returns a string containing <i>length</i> characters starting from index <i>start</i> in the source string. If <i>length</i> is not specified, a string containing characters from <i>start</i> to the end of the source string is returned.
<code>substring( start, end )</code>	Returns a string containing the characters from index <i>start</i> up to but not including index <i>end</i> in the source string.
<code>toLowerCase()</code>	Returns a string in which all uppercase letters are converted to lowercase letters. Non-letter characters are not changed.
<code>toUpperCase()</code>	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.**

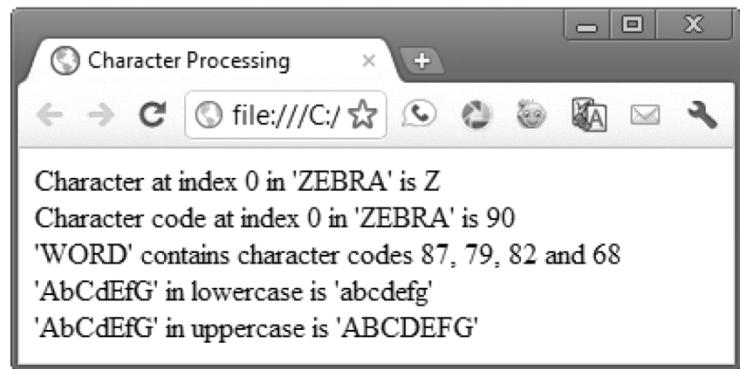


## 11.3.3 Character Processing Methods (Cont.)

- ▶ 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 I 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**

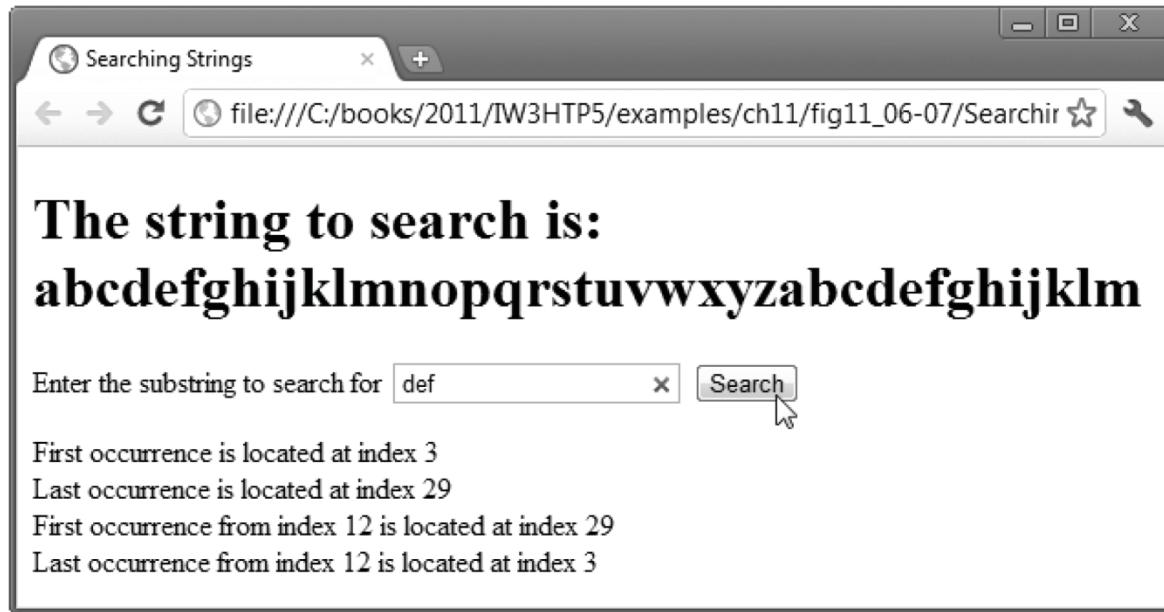
### ▶ String method `lastIndexOf`

- 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

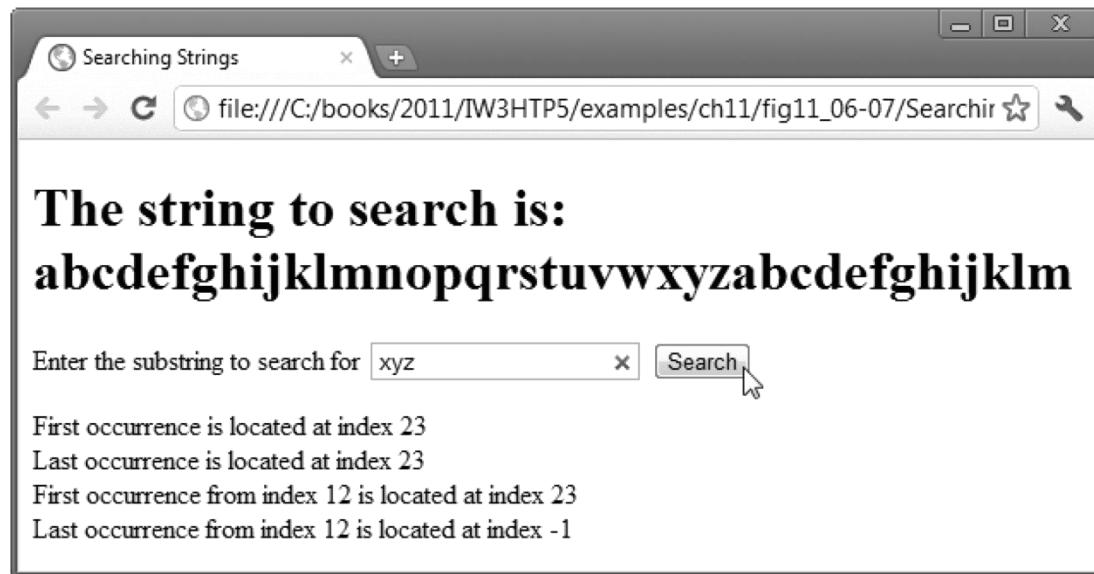


```
1 <!DOCTYPE html>
2
3 <!-- Fig. 11.6: SearchingStrings.html -->
4 <!-- HTML document to demonstrate methods indexOf and lastIndexOf. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Searching Strings</title>
9     <link rel = "stylesheet" type = "text/css" href = "style.css">
10    <script src = "SearchingStrings.js"></script>
11  </head>
12  <body>
13    <form id = "searchForm" action = "#">
14      <h1>The string to search is:
15        abcdefghijklmnopqrstuvwxyzabcdefghijklm</h1>
16      <p>Enter the substring to search for
17        <input id = "inputField" type = "search">
18        <input id = "searchButton" type = "button" value = "Search"></p>
19        <div id = "results"></div>
20    </form>
21  </body>
22 </html>
```

**Fig. 11.6** | HTML document to demonstrate methods `indexOf` and `lastIndexOf`. (Part I 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     var inputField = document.getElementById( "inputField" );
7
8     document.getElementById( "results" ).innerHTML =
9         "<p>First occurrence is located at index " +
10        letters.indexOf( inputField.value ) + "</p>" +
11        "<p>Last occurrence is located at index " +
12        letters.lastIndexOf( inputField.value ) + "</p>" +
13        "<p>First occurrence from index 12 is located at index " +
14        letters.indexOf( inputField.value, 12 ) + "</p>" +
15        "<p>Last occurrence from index 12 is located at index " +
16        letters.lastIndexOf( inputField.value, 12 ) + "</p>";
17
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.)



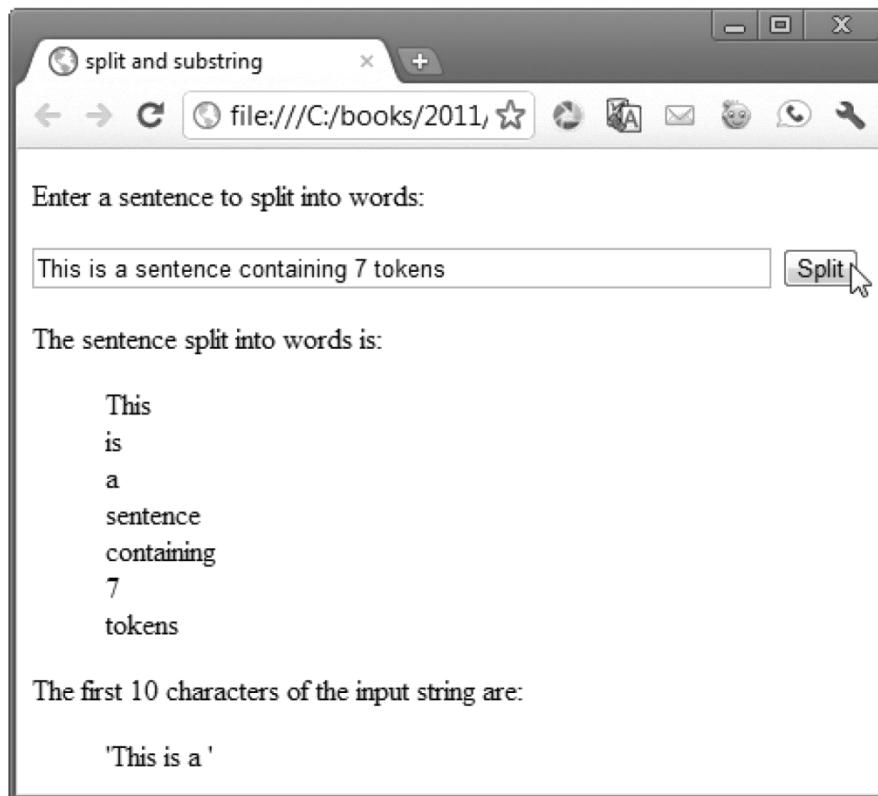
## 11.3.5 Splitting Strings and Obtaining Substrings

- ▶ Breaking a string into tokens is called **tokenization**
- ▶ Tokens are separated from one another by delimiters, 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



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 11.8: SplitAndSubString.html -->
4 <!-- HTML document demonstrating String methods split and substring. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>split and substring</title>
9     <link rel = "stylesheet" type = "text/css" href = "style.css">
10    <script src = "SplitAndSubString.js"></script>
11  </head>
12  <body>
13    <form action = "#">
14      <p>Enter a sentence to split into words:</p>
15      <p><input id = "inputField" type = "text">
16        <input id = "splitButton" type = "button" value = "Split"></p>
17      <div id = "results"></p>
18    </form>
19  </body>
20 </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.



## 11.4 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)

<http://pansci.asia/archives/84978>



# 11.4 Date Object

- ▶ 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



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
<code>getTime()</code>	Returns the number of milliseconds between January 1, 1970, and the time in the <code>Date</code> object.
<code>getTimezoneOffset()</code>	Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).
<code> setDate( val )</code> <code>setUTCDate( val )</code>	Sets the day of the month (1 to 31) in local time or UTC.
<code>setFullYear( y, m, d )</code> <code>setUTCFullYear( y, m, d )</code>	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 <code>Date</code> object is used.
<code>setHours( h, m, s, ms )</code> <code>setUTCHours( h, m, s, ms )</code>	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 <code>Date</code> object is used.
<code>setMilliseconds( ms )</code> <code>setUTCMilliseconds( ms )</code>	Sets the number of milliseconds in local time or UTC.

**Fig. 11.10** | Date-object methods. (Part 2 of 4.)



Method	Description
<code>setMinutes( m, s, ms )</code> <code>setUTCMilliseconds( m, s, ms )</code>	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 <code>Date</code> object is used.
<code>setMonth( m, d )</code> <code>setUTCMonth( m, d )</code>	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 <code>Date</code> object is used.
<code>setSeconds( s, ms )</code> <code>setUTCSeconds( s, ms )</code>	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 <code>Date</code> object is used.
<code> setTime( ms )</code>	Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.
<code>toLocaleString()</code>	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 <i>09/13/07 15:47:22</i> in the United States and <i>13/09/07 15:47:22</i> in Europe.

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

Method	Description
<code>toUTCString()</code>	Returns a string representation of the date and time in the form: <i>15 Sep 2007 15:47:22 UTC</i> .
<code>toString()</code>	Returns a string representation of the date and time in a form specific to the locale of the computer ( <i>Mon Sep 17 15:47:22 EDT 2007</i> in the United States).
<code>valueOf()</code>	The time in number of milliseconds since midnight, January 1, 1970. (Same as <code>getTime()</code> .)

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

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 11.11: DateTime.html -->
4 <!-- HTML document to demonstrate Date-object methods. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Date and Time Methods</title>
9     <link rel = "stylesheet" type = "text/css" href = "style.css">
10    <script src = "DateTime.js"></script>
11  </head>
12  <body>
13    <h1>String representations and valueOf</h1>
14    <section id = "strings"></section>
15    <h1>Get methods for local time zone</h1>
16    <section id = "getMethods"></section>
17    <h1>Specifying arguments for a new Date</h1>
18    <section id = "newArguments"></section>
19    <h1>Set methods for local time zone</h1>
20    <section id = "setMethods"></section>
21  </body>
22 </html>
```

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

Date and Time Methods

file:///C:/books/2011/IW3HTP5/e

## String representations and valueOf

toString: Thu Aug 11 2011 10:31:21 GMT-0400 (Eastern Daylight Time)  
toLocaleString: Thu Aug 11 2011 10:31:21 GMT-0400 (Eastern Daylight Time)  
toUTCString: Thu, 11 Aug 2011 14:31:21 GMT  
valueOf: 1313073081914

## Get methods for local time zone

getDate: 11  
getDay: 4  
getMonth: 7  
getFullYear: 2011  
getTime: 1313073081914  
getHours: 10  
getMinutes: 31  
getSeconds: 21  
getMilliseconds: 914  
getTimezoneOffset: 240

Fig. 11.11 | HTML document to demonstrate Date-object methods.

(Page 2 of 2)

## Specifying arguments for a new Date

Date: Fri Mar 18 2011 01:05:00 GMT-0400 (Eastern Daylight Time)

## Set methods for local time zone

Modified date: Sat Dec 31 2011 23:59:59 GMT-0500 (Eastern Standard Time)

**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 I 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.)

# JavaScript Objects

- ▶ All JavaScript values, except primitives, are objects.
  - JavaScript defines multiple types of primitive data types:
    - string
    - number
    - boolean
- ▶ A JavaScript object is a collection of named values (*properties*)
  - `var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};`



# JSON (JavaScript Object Notation)

- ▶ JSON is text, written with JavaScript object notation.
- ▶ JSON is for storing and exchanging data.
  - When exchanging data between a browser and a server, the data can only be text.
- ▶ We can convert any JSON received from the server into JavaScript objects.

```
{  
    "name": "John",  
    "age": 31,  
    "city": "New York",  
    "Habits": ["Swimming", "Reading"]  
}
```

# JSON Syntax

- ▶ JSON syntax is derived from JavaScript object notation syntax:
  - Data is in name/value pairs
  - Data is separated by commas
  - Curly braces hold objects
  - Square brackets hold arrays

```
{  
    "name": "John",  
    "age": 31,  
    "city": "New York",  
    "Habits": ["Swimming", "Reading"]}
```

# Double Quotes in JSON

- ▶ Properties in JSON have to be strings
  - a sequence of zero or more Unicode characters, wrapped **in double quotes**.
- ▶ In JSON, string values must be written with double quotes:

```
{ "name": "John" }
```

- ▶ In JavaScript, you can write string values with double or single quotes:

```
{ name: 'John' }
```

# From Object to JSON

- ▶ If you have data stored in a JavaScript object, you can convert the object into JSON.
- ▶ **JSON.stringify():**
  - [https://www.w3schools.com/js/js\\_json\\_stringify.asp](https://www.w3schools.com/js/js_json_stringify.asp)
  - When sending data to a web server (or to the localStorage), the data has to be a string.
  - Convert a JavaScript object into a string with **JSON.stringify()**.

<https://www.w3schools.com/code/tryit.asp?filename=FX7AFRM45M1>

# From JSON to Object

- ▶ If you receive data in JSON format, you can convert it into a JavaScript object.
- ▶ **JSON.parse()**
  - [https://www.w3schools.com/js/js\\_json\\_parse.asp](https://www.w3schools.com/js/js_json_parse.asp)
  - When receiving data from a web server, the data is always a string.
  - Parse the data with `JSON.parse()`, and the data becomes a JavaScript object.

[https://www.w3schools.com/js/tryit.asp?filename=tryjson\\_receive](https://www.w3schools.com/js/tryit.asp?filename=tryjson_receive)



# Declare an Object Constructor and Instantiate an Object

```
function Person(firstName, lastName, age, eyeColor) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.eyeColor = eyeColor;  
    this.changeName = function (name) {  
        this.firstName = name;  
    };  
}
```

*// JavaScript methods are the actions that can be performed on objects.*

*// "**this**" is the object that "owns" the JavaScript code.*

*// constructor invocation*

```
var p = new Person("Stephen", "Curry", 28, "brown");  
p.changeName("Wardell Stephen");
```



# Object Prototype

- ▶ Every JavaScript object has a prototype (原型).
- ▶ All JavaScript objects inherit their properties and methods from their prototype.
  - 透過prototype才能讓多個new出來的物件共享method。

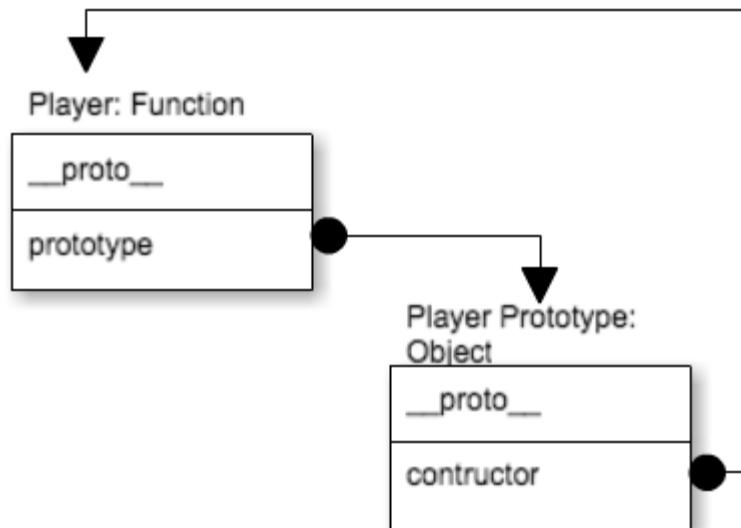
```
function Player(first, last, age, power) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.powerRating = power;  
}
```

<https://www.w3schools.com/code/tryit.asp?filename=GA84EANWV3J7>

```
Player.prototype.name = function () {  
    return this.firstName + " " + this.lastName  
};
```

[https://www.w3schools.com/js/js\\_object\\_prototypes.asp](https://www.w3schools.com/js/js_object_prototypes.asp)

# Object Prototype<sub>2</sub>



<https://eyesofkids.gitbooks.io/javascript-start-from-es6/content/part4/prototype.html>

# “this” Keyword

- ▶ In a function definition, this refers to the "owner" of the function.

```
var person = {  
    firstName: "John",  
    lastName : "Doe",  
    id       : 5566,  
    fullName : function() {  
        return this.firstName + " " + this.lastName;  
    }  
};
```

- In the example above, this refers to the person object.
- The person object "owns" the fullName method.



# “this” Keyword:

## Default Binding and Object Method Binding

- ▶ When used alone, “this” refers to the Global object.
  - In a browser the Global object is [object Window].

[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_this](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this)

- ▶ When using object, “this” refers to the "owner" object of the function.

[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_this\\_object](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_object)

<https://www.w3schools.com/code/tryit.asp?filename=FS807C22B4KI>



# “this” Keyword in Event Handlers

- ▶ In HTML event handlers, this refers to the HTML element that received the event:

[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_this\\_event](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_event)



## 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.
- ▶ 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*



# 11.7 Favorite Twitter Searches (Cont.)

## *Problems with Cookies*

- ▶ They're extremely **limited in size**.
- ▶ 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.)

## *Introducing localStorage and sessionStorage*

- ▶ 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`*

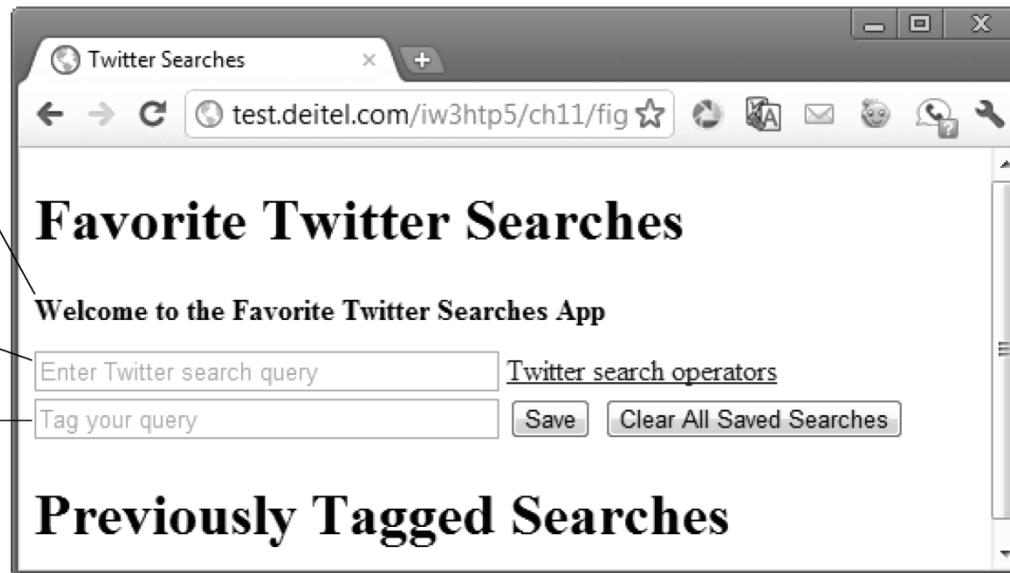
- ▶ This app allows users to save their favorite (possibly lengthy) Twitter search strings with easy-to-remember, user-chosen, short tag names. Users can then conveniently follow the tweets on their favorite topics by visiting this web page and clicking the link for a saved search.
- ▶ The user's favorite searches are saved using `localStorage`, so they're immediately available each time the user browses the app's web page.
- ▶ 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.

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

Welcome message  
appears only on  
the first visit to the  
page during this  
browsing session

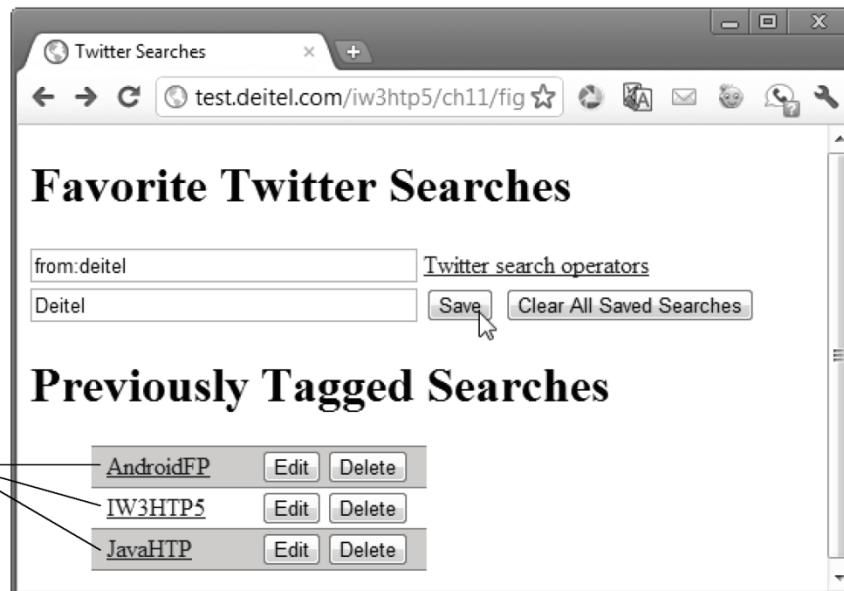
Enter Twitter  
search query here

Tag your search



**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



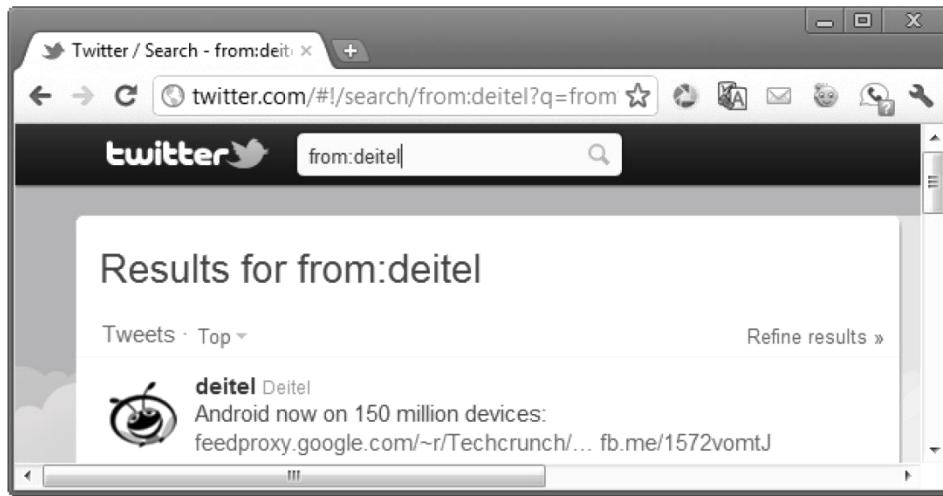
**Fig. 11.16** | Sample outputs from the Favorite Twitter Searches web application. (Part 2 of 4.)

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.



```
1 <!DOCTYPE html>
2
3 <!-- Fig. 11.17: FavoriteTwitterSearchs.html -->
4 <!-- Favorite Twitter Searches web application. -->
5 <html>
6 <head>
7   <title>Twitter Searches</title>
8   <link rel = "stylesheet" type = "text/css" href = "style.css">
9   <script src = "FavoriteTwitterSearches.js"></script>
10 </head>
11 <body>
12   <h1>Favorite Twitter Searches</h1>
13   <p id = "welcomeMessage"></p>
14   <form action = "#">
15     <p><input id = "query" type = "text"
16       placeholder = "Enter Twitter search query">
17       <a href = "https://dev.twitter.com/docs/using-search">
18         Twitter search operators</a></p>
19     <p><input id = "tag" type = "text" placeholder = "Tag your query">
20       <input type = "button" value = "Save"
21         id = "saveButton">
22       <input type = "button" value = "Clear All Saved Searches"
23         id = "clearButton"></p>
24   </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.
- ▶ 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.



## 11.7 Favorite Twitter Searches (Cont.)

- ▶ 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.
- ▶ 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**—this enables the event handler to use the id value when handling the event.
- ▶ [Note: The `localStorage` and `sessionStorage` properties and methods we discuss throughout this section apply to both objects.]



## 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.
- ▶ The `setItem` method stores a key/value pair in the `localStorage` object.
  - If the key already exists, `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.)