# 11. JavaScript: Objects

***My object all sublime I shall achieve in time.***

**—W. S. Gilbert**

***Is it a world to hide virtues in?***

**—William Shakespeare**

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.

Outline

## 11.1. Introduction

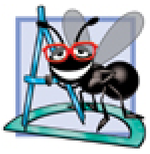
This chapter presents a more formal treatment of **objects**. We presented a brief introduction to object-oriented programming concepts in Chapter 1. This chapter overviews—and serves as a reference for—several of JavaScript's built-in objects and demonstrates many of their capabilities. We use HTML5's 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, a means for creating JavaScript objects—typically for transferring data over the Internet between client-side and server-side programs (a technique we discuss in Chapter 16). In subsequent chapters on the Document Object Model and JavaScript Events, you'll work with many objects provided by the browser that enable scripts to manipulate the elements of an HTML5 document.

## 11.2. `Math` Object

The **`Math`** object's methods enable you to conveniently perform many common mathematical calculations. As shown previously, an object's methods are called by writing the name of the object followed by a dot ( `.` ) and the name of the method. In parentheses following the method name are arguments to the method. For example, to calculate the square root of `900` you might write

```
var result = Math.sqrt( 900 );
```

which first calls method `Math.sqrt` to calculate the square root of the number contained in the parentheses ( `900` ), then assigns the result to a variable. The number `900` is the argument of the `Math.sqrt` method. The above statement would return `30` . Some `Math` -object methods are summarized in Fig. 11.1.



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.*

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

| Method | Description | Examples |
|---|---|---|
| `abs( x )` | Absolute value of x. | `abs( 7.2 )` is 7.2<br>`abs( 0 )` is 0<br>`abs( -5.6 )` is 5.6 |
| `ceil( x )` | Rounds x to the smallest integer not less than x. | `ceil( 9.2 )` is 10<br>`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<br>`exp( 2 )` is 7.38906 |
| `floor( x )` | Rounds x to the largest integer not greater than x. | `floor( 9.2 )` is 9<br>`floor( -9.8 )` is -10.0 |
| `log( x )` | Natural logarithm of x (base *e*). | `log( 2.718282 )` is 1<br>`log( 7.389056 )` is 2 |
| `max( x, y )` | Larger value of x and y. | `max( 2.3, 12.7 )` is 12.7<br>`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<br>`min( -2.3, -12.7 )` is -12.7 |
| `pow( x, y )` | x raised to power y ($x^y$). | `pow( 2, 7 )` is 128<br>`pow( 9, .5 )` is 3.0 |
| `round( x )` | Rounds x to the closest integer. | `round( 9.75 )` is 10<br>`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<br>`sqrt( 9 )` is 3 |
| `tan( x )` | Trigonometric tangent of x (x in radians). | `tan( 0 )` is 0 |

Fig. 11.1. `Math` object methods.

| 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

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

### 11.3.1. Fundamentals of Characters and Strings

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

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

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

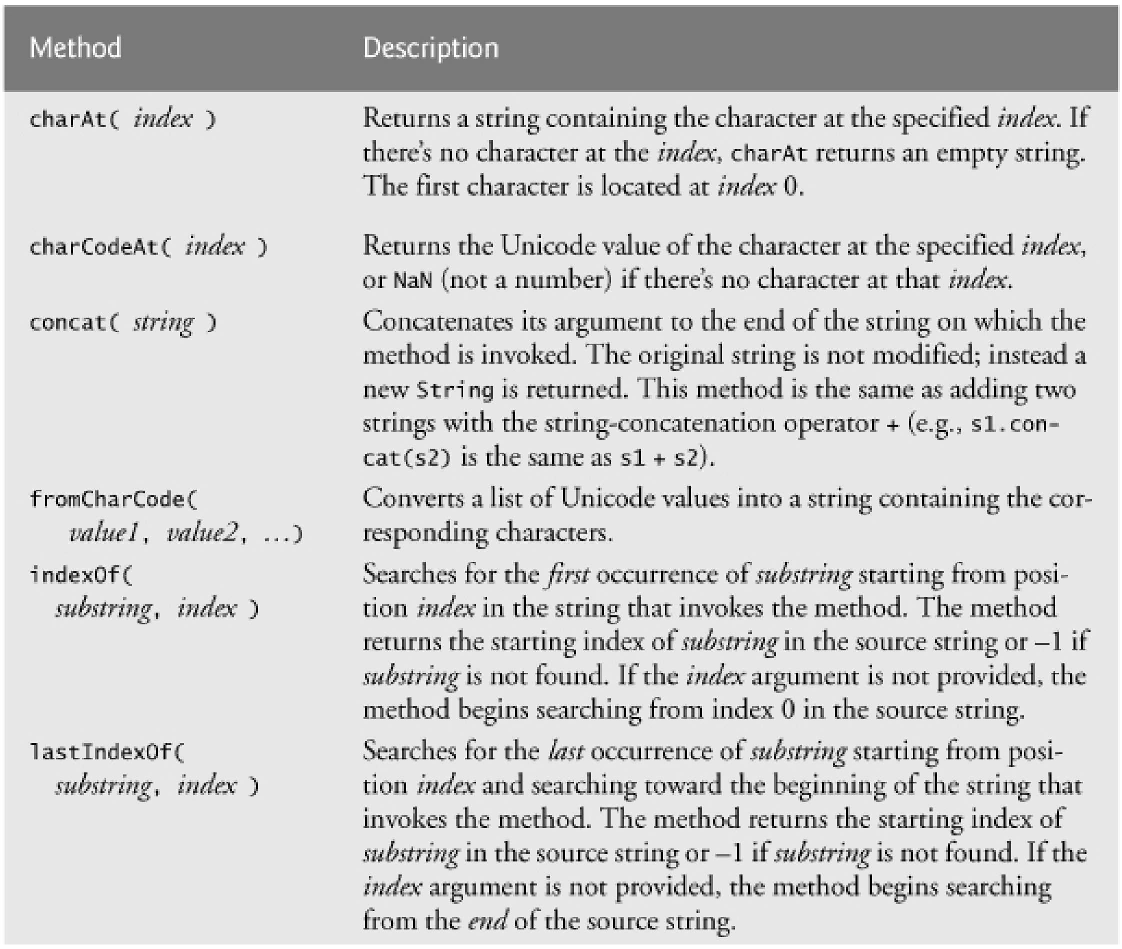
A `String` may be assigned to a variable in a declaration. The declaration

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

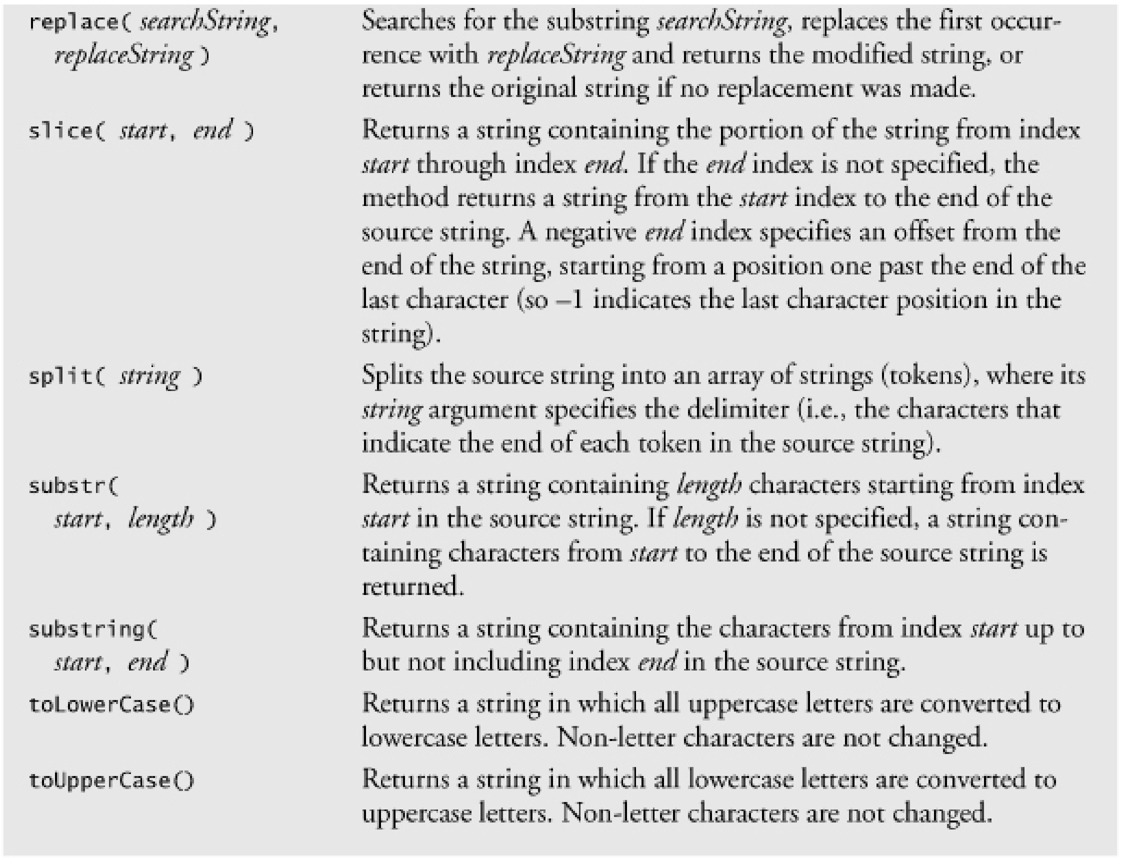
initializes variable `color` with the `String` object containing the string `"blue"`. `String`s can be compared via the relational (`<`, `<=`, `>` and `>=`) and equality operators (`==`, `===`, `!=` and `!==`). The comparisons are based on the Unicode values of the corresponding characters. For example,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

### 11.3.2. Methods of the `String` Object

The `String` object encapsulates the attributes and behaviors of a string of characters. It provides many methods (behaviors) that accomplish useful tasks such as selecting characters from a string, combining strings (called **concatenation**), obtaining *substrings* (portions) of a string, searching for substrings within a string, *tokenizing strings* (i.e., splitting strings into individual words) and converting strings to all uppercase or lowercase letters. The `String` object also provides several methods that generate HTML5 tags. Figure 11.3 summarizes many `String` methods. Figures 11.4–11.9 demonstrate some of these methods.



| 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. |
| `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). |
| `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.

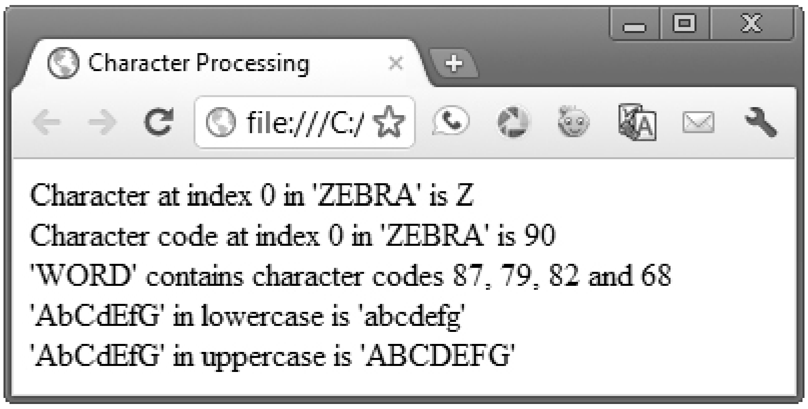
### 11.3.3. Character-Processing Methods

The example in Figs. 11.4–11.5 demonstrates some of the `String` object's character-processing methods, including:

- **`charAt`** —returns the character at a specific position
- **`charCodeAt`** —returns the Unicode value of the character at a specific position
- **`fromCharCode`** —returns a string created from a series of Unicode values
- **`toLowerCase`** —returns the lowercase version of a string
- **`toUpperCase`** —returns the uppercase version of a string

The HTML document (Fig. 11.4) calls the script's `start` function to display the results in the results `div` . [*Note:* Throughout this chapter, we show the CSS style sheets only if there are new features to discuss. You can view each example's style-sheet contents by opening the style sheet in a text editor.]

```
<!DOCTYPE html>

<!-- Fig. 11.4: CharacterProcessing.html -->
<!-- HTML5 document to demonstrate String methods charAt, char-
CodeAt,
  fromCharCode, toLowercase and toUpperCase. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Character Processing</title>
     <link rel = "stylesheet" type = "text/css" href = "style.css">
     <script src = "CharacterProcessing.js"></script>
   </head>
   <body>
     <div id = "results"></div>
   </body>
 </html>
```



```
Character at index 0 in 'ZEBRA' is Z
Character code at index 0 in 'ZEBRA' is 90
'WORD' contains character codes 87, 79, 82 and 68
'AbCdEfG' in lowercase is 'abcdefg'
'AbCdEfG' in uppercase is 'ABCDEFG'
```

Fig. 11.4. HTML5 document to demonstrate methods `charAt` , `char-CodeAt` , `fromCharCode` , `toLowercase` and `toUpperCase` .

In the script (Fig. 11.5), lines 10–11 get the first character in `String s` ( `"ZEBRA"` ) using `String` method `charAt` and append it to the `result` string. Method **`charAt`** returns a string containing the character at the specified index ( `0` in this example). Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (e.g., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

```
// Fig. 11.5: CharacterProcessing.js
// String methods charAt, charCodeAt, fromCharCode,
// toLowercase and toUpperCase.
function start()
{
  var s = "ZEBRA";
  var s2 = "AbCdEfG";
  var result = "";

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

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

   result += "<p>'" + s2 + "' in lowercase is '" +
     s2.toLowerCase() + "'</p>";
   result += "<p>'" + s2 + "' in uppercase is '" +
     s2.toUpperCase() + "'</p>";

   document.getElementById( "results" ).innerHTML = result;
 } // end function start

 window.addEventListener( "load", start, false );
```

Fig. 11.5. `String` methods `charAt` , `charCodeAt` , `fromCharCode` , `toLowercase` and `toUpperCase` .

Lines 12–13 get the character code for the first character in `String` `s` (`"ZEBRA"`) by calling `String` method `charCodeAt`. Method `charCodeAt` returns the Unicode value of the character at the specified index (`0` in this example). If the index is outside the bounds of the string, the method returns `NaN`.

`String` method `fromCharCode` receives as its argument a comma-separated list of Unicode values and builds a string containing the character representations of those Unicode values. Lines 15–16 create the string `"WORD"`, which consists of the character codes 87, 79, 82 and 68. Note that we use the `String` object to call method `fromCharCode`, rather than a specific `String` variable. Appendix D, ASCII Character Set, contains the character codes for the ASCII character set—a subset of the Unicode character set (Appendix F) that contains only Western characters.

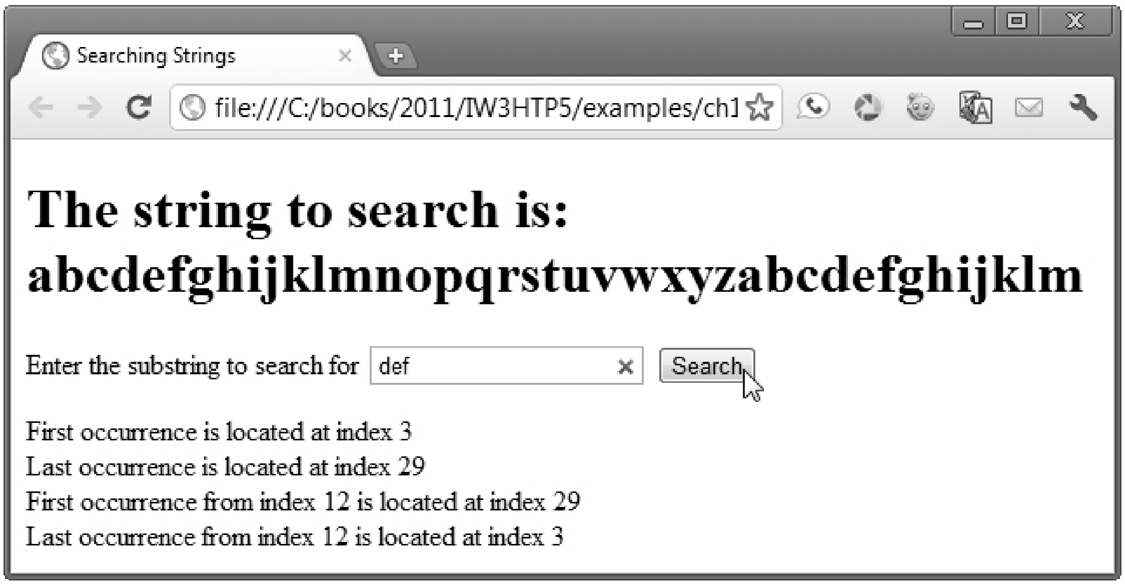
Lines 18–21 use `String` methods **`toLowerCase`** and **`toUpperCase`** to get versions of `String` `s2` (`"AbCdEfG"`) in all lowercase letters and all uppercase letters, respectively.

### 11.3.4. Searching Methods

The example in Figs. 11.6–11.7 demonstrates the `String`-object methods **`indexOf`** and **`lastIndexOf`** that *search* for a specified substring in a string. All the searches in this example are performed on a global string named `letters` in the script (Fig. 11.7, line 3). The user types a substring in the HTML5 form `searchForm`'s `inputField` and presses the **Search** button to search for the substring in `letters`. Clicking the **Search** button calls function `buttonPressed` (lines 5–18) to respond to the `click` event and perform the searches. The results of each search are displayed in the `div` named `results`.

In the script (Fig. 11.7), lines 10–11 use `String` method `indexOf` to determine the location of the *first* occurrence in string `letters` of the string `inputField.value` (i.e., the string the user typed in the `inputField` text field). If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, –1 is returned.

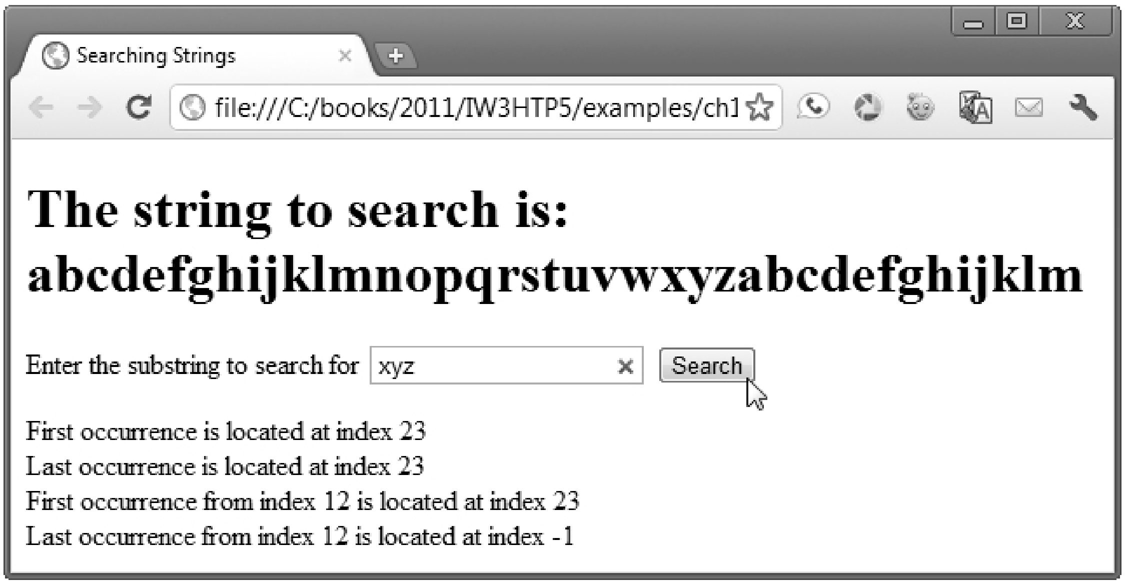
```
<!DOCTYPE html>

<!-- Fig. 11.6: SearchingStrings.html -->
<!-- HTML document to demonstrate methods indexOf and lastIn-
dexOf. -->
<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>
```



The string to search is: abcdefghijklmnopqrstuvwxyzabcdefghijklm

Enter the substring to search for def Search

```
First occurrence is located at index 3
Last occurrence is located at index 29
First occurrence from index 12 is located at index 29
Last occurrence from index 12 is located at index 3
```



```
The string to search is:
abcdefghijklmnopqrstuvwxyzabcdefghijklm

Enter the substring to search for xyz

First occurrence is located at index 23
Last occurrence is located at index 23
First occurrence from index 12 is located at index 23
Last occurrence from index 12 is located at index -1
```

Fig. 11.6. HTML document to demonstrate methods `indexOf` and `lastIndexOf`.

Lines 12–13 use `String` method `lastIndexOf` to determine the location of the *last* occurrence in `letters` of the string in `inputField`. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, –1 is returned.

Lines 14–15 use `String` method `indexOf` to determine the location of the *first* occurrence in string `letters` of the string in the `inputField` text field, starting from index `12` in `letters`. If the substring is found, the index at which the first occurrence of the substring (starting from index `12`) begins is returned; otherwise, –1 is returned.

Lines 16–17 use `String` method `lastIndexOf` to determine the location of the *last* occurrence in `letters` of the string in the `inputField` text field, starting from index `12` in `letters` and moving toward the beginning of the input. If the substring is found, the index at which the first occurrence of the substring (if one appears before index `12`) begins is returned; otherwise, –1 is returned.

```
// Fig. 11.7: SearchingStrings.js
// Searching strings with indexOf and lastIndexOf.
var letters = "abcdefghijklmnopqrstuvwxyzabcdefghijklm";
```

```

function buttonPressed()
{
  var inputField = document.getElementById( "inputField" );

  document.getElementById( "results" ).innerHTML =
     "<p>First occurrence is located at index " +
       letters.indexOf( inputField.value ) + "</p>" +
     "<p>Last occurrence is located at index " +
       letters.lastIndexOf( inputField.value ) + "</p>" +
     "<p>First occurrence from index 12 is located at index " +
       letters.indexOf( inputField.value, 12 ) + "</p>" +
     "<p>Last occurrence from index 12 is located at index " +
       letters.lastIndexOf( inputField.value, 12 ) + "</p>";
} // end function buttonPressed

// register click event handler for searchButton
function start()
{
  var searchButton = document.getElementById( "searchButton" );
  searchButton.addEventListener( "click", buttonPressed, false );
} // end function start

window.addEventListener( "load", start, false );
```

Fig. 11.7. Searching strings with `indexOf` and `lastIndexOf`.

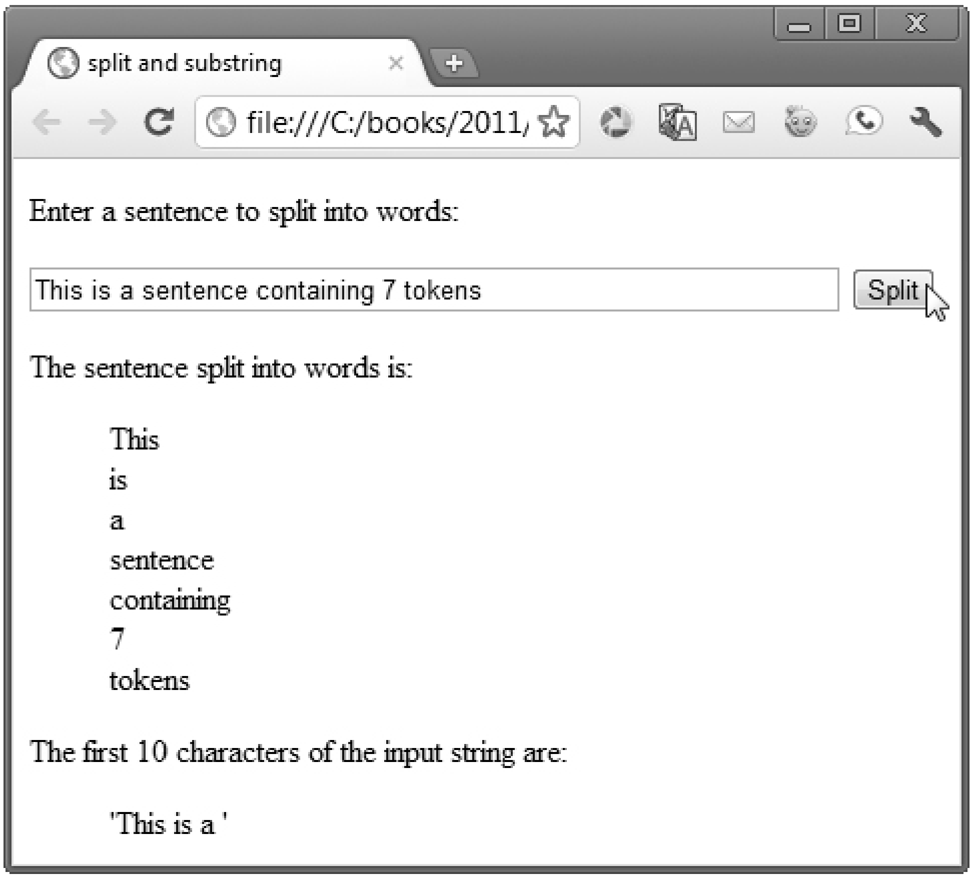
### 11.3.5. Splitting Strings and Obtaining Substrings

When you read a sentence, your mind breaks it into individual words, or **tokens**, each of which conveys meaning to you. The process of breaking a string into tokens is called **tokenization**. Interpreters also perform tokenization. They break up statements into such individual pieces as keywords, identifiers, operators and other elements of a programming language. The example in Figs. 11.8–11.9 demonstrates `String` method **`split`**, which breaks a string into its component tokens. Tokens are separated from one another by **delimiters**, typically white-space characters such as blanks, tabs, newlines and carriage returns. Other characters may also be used as delimiters to separate tokens. The HTML5 document displays a form containing a text field where the user types a sentence to tokenize. The results of the tokenization process are displayed in a `div`. The script also demonstrates `String` method **`substring`**, which returns a portion of a string.

The user types a sentence into the text field with `id inputField` and presses the **Split** button to tokenize the string. Function `splitButton-Pressed` (Fig. 11.9) is called in respons to the button's `click` event.

---

```
<!DOCTYPE html>

<!-- Fig. 11.8: SplitAndSubString.html -->
<!-- HTML document demonstrating String methods split and sub-
string. -->
<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>
```



Enter a sentence to split into words:

This is a sentence containing 7 tokens

The sentence split into words is:

This
is
a
sentence
containing
7
tokens

The first 10 characters of the input string are:

'This is a '

Fig. 11.8. HTML document demonstrating `String` methods `split` and `substring`.

In the script (Fig. 11.9), line 5 gets the value of the input field and stores it in variable `inputString`. Line 6 calls `String` method `split` to tokenize `inputString`. The argument to method `split` is the **delimiter string**—the string that determines the end of each token in the original string. In this example, the space character delimits the tokens. The delimiter string can contain multiple characters to be used as delimiters. Method `split` returns an array of strings containing the tokens. Line 11 uses `Array` method `join` to combine the tokens in array `tokens` and separate each token with `</p><p class = 'indent'>` to end one paragraph element and start a new one. Line 13 uses `String` method `substring` to obtain a string containing the first 10 characters of the string the user entered (still stored in `inputString`). The method returns the substring from the **starting index** (`0` in this example) up to but not including the **ending index** (`10` in this example). If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string. The result of the string concatenations in lines 9–13 is displayed in the document's `results div`.

```
// Fig. 11.9: SplitAndSubString.js
// String object methods split and substring.
function splitButtonPressed()
{
  var inputString = document.getElementById( "inputField" ).value;
  var tokens = inputString.split( " " );

  var results = document.getElementById( "results" );
  results.innerHTML = "<p>The sentence split into words is: </p>" +
     "<p class = 'indent'>" +
     tokens.join( "</p><p class = 'indent'>" ) + "</p>" +
     "<p>The first 10 characters of the input string are: </p>" +
     "<p class = 'indent'>'" + inputString.substring( 0, 10 ) + "'</p>";
} // end function splitButtonPressed

// register click event handler for searchButton
function start()
{
  var splitButton = document.getElementById( "splitButton" );
  splitButton.addEventListener( "click", splitButtonPressed, false );
} // end function start

window.addEventListener( "load", start, false );
```

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

## 11.4. `Date` Object

JavaScript's **`Date`** object provides methods for date and time manipulations. These can be performed based on the computer's **local time zone** or based on World Time Standard's **Coordinated Universal Time** (abbreviated **UTC**)—formerly called **Greenwich Mean Time (GMT)**. Most methods of the `Date` object have a local time zone and a UTC version. `Date`-object methods are summarized in Fig. 11.10.

| Method | Description |
|---|---|
| `getDate()`<br>`getUTCDate()` | Returns a number from 1 to 31 representing the day of the month in local time or UTC. |
| `getDay()`<br>`getUTCDay()` | Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC. |
| `getFullYear()`<br>`getUTCFullYear()` | Returns the year as a four-digit number in local time or UTC. |
| `getHours()`<br>`getUTCHours()` | Returns a number from 0 to 23 representing hours since midnight in local time or UTC. |
| `getMilliseconds()`<br>`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()`<br>`getUTCMinutes()` | Returns a number from 0 to 59 representing the minutes for the time in local time or UTC. |
| `getMonth()`<br>`getUTCMonth()` | Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC. |
| `getSeconds()`<br>`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* `)`<br>`setUTCDate(` *val* `)` | Sets the day of the month (1 to 31) in local time or UTC. |
| `setFullYear(` *y*, *m*, *d* `)`<br>`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 )`<br>`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 )`<br>`setUTCMilliseconds( ms )` | Sets the number of milliseconds in local time or UTC. |
| `setMinutes( m, s, ms )`<br>`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 )`<br>`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 )`<br>`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. |
| `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.

The example in Figs. 11.11–11.12 demonstrates many of the local-time-zone methods in Fig. 11.10. The HTML document (Fig. 11.11) provides several `section` s in which the results are displayed.

### `Date` -Object Constructor with No Arguments

In the script (Fig. 11.12), line 5 creates a new `Date` object. The `new` operator creates the `Date` object. The empty parentheses indicate a call to the `Date` object's **constructor** with no arguments. A constructor is an *initializer* method for an object. *Constructors are called automatically when an object is allocated with* `new`. The `Date` constructor with no arguments initializes the `Date` object with the local computer's current date and time.

### Methods `toString`, `toLocaleString`, `toUTCString` and `valueOf`

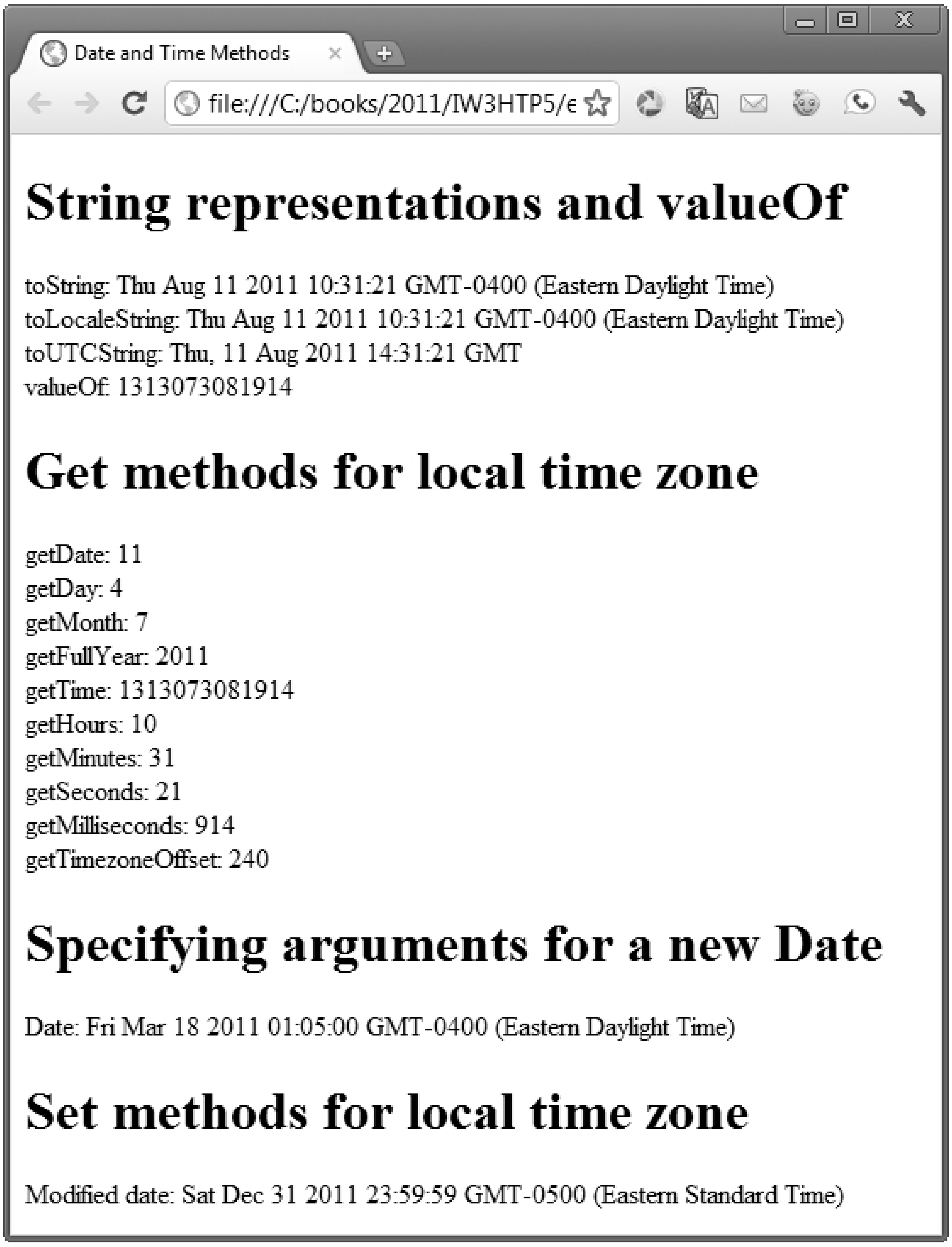
Lines 9–12 demonstrate the methods `toString`, `toLocaleString`, `toUTCString` and `valueOf`. Method `valueOf` returns a large integer value representing the total number of milliseconds between midnight, January 1, 1970, and the date and time stored in `Date` object `current`.

### `Date`-Object *get* Methods

Lines 16–25 demonstrate the `Date` object's *get* methods for the local time zone. The method `getFullYear` returns the year as a four-digit number. The method `getTimeZoneOffset` returns the difference in minutes between the local time zone and UTC time (i.e., a difference of four hours in our time zone when this example was executed).

---

```
<!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>
```



# 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
```

# 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.

```
// Fig. 11.12: DateTime.js
// Date and time methods of the Date object.
function start()
```

```
{
  var current = new Date();

  // string-formatting methods and valueOf
  document.getElementById( "strings" ).innerHTML =
    "<p>toString: " + current.toString() + "</p>" +
     "<p>toLocaleString: " + current.toLocaleString() + "</p>" +
     "<p>toUTCString: " + current.toUTCString() + "</p>" +
     "<p>valueOf: " + current.valueOf() + "</p>";

   // get methods
   document.getElementById( "getMethods" ).innerHTML =
     "<p>getDate: " + current.getDate() + "</p>" +
     "<p>getDay: " + current.getDay() + "</p>" +
     "<p>getMonth: " + current.getMonth() + "</p>" +
     "<p>getFullYear: " + current.getFullYear() + "</p>" +
     "<p>getTime: " + current.getTime() + "</p>" +
     "<p>getHours: " + current.getHours() + "</p>" +
     "<p>getMinutes: " + current.getMinutes() + "</p>" +
     "<p>getSeconds: " + current.getSeconds() + "</p>" +
     "<p>getMilliseconds: " + current.getMilliseconds() + "</p>" +
     "<p>getTimezoneOffset: " + current.getTimezoneOffset() + "</p>";

   // creating a Date
   var anotherDate = new Date( 2011, 2, 18, 1, 5, 0, 0 );
   document.getElementById( "newArguments" ).innerHTML =
     "<p>Date: " + anotherDate + "</p>";

   // set methods
   anotherDate.setDate( 31 );
   anotherDate.setMonth( 11 );
   anotherDate.setFullYear( 2011 );
   anotherDate.setHours( 23 );
   anotherDate.setMinutes( 59 );
   anotherDate.setSeconds( 59 );
   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.

### `Date`-Object Constructor with Arguments

Line 28 creates a new `Date` object and supplies arguments to the `Date` constructor for *year*, *month*, *date*, *hours*, *minutes*, *seconds* and *milliseconds*. The *hours*, *minutes*, *seconds* and *milliseconds* arguments are all optional. If an argument is not specified, 0 is supplied in its place. For *hours*, *minutes* and *seconds*, if the argument to the right of any of these is specified, it too must be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified).

### `Date`-Object *set* Methods

Lines 33–38 demonstrate the `Date`-object *set* methods for the local time zone. `Date` objects represent the month internally as an integer from 0 to 11. These values are off by one from what you might expect (i.e., 1 for January, 2 for February, ..., and 12 for December). When creating a `Date` object, you must specify 0 to indicate January, 1 to indicate February, ..., and 11 to indicate December.

**COMMON PROGRAMMING ERROR 11.1**

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

### `Date`-Object `parse` and `UTC` Methods

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

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

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

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

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

## 11.5. `Boolean` and `Number` Objects

JavaScript provides the **`Boolean`** and **`Number`** objects as **object wrappers** for boolean `true`/`false` values and numbers, respectively. These wrappers define methods and properties useful in manipulating boolean values and numbers.

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

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

The *booleanValue* specifies whether the `Boolean` object should contain `true` or `false`. If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or an empty string (`""`), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`. Figure 11.13 summarizes the methods of the `Boolean` object.

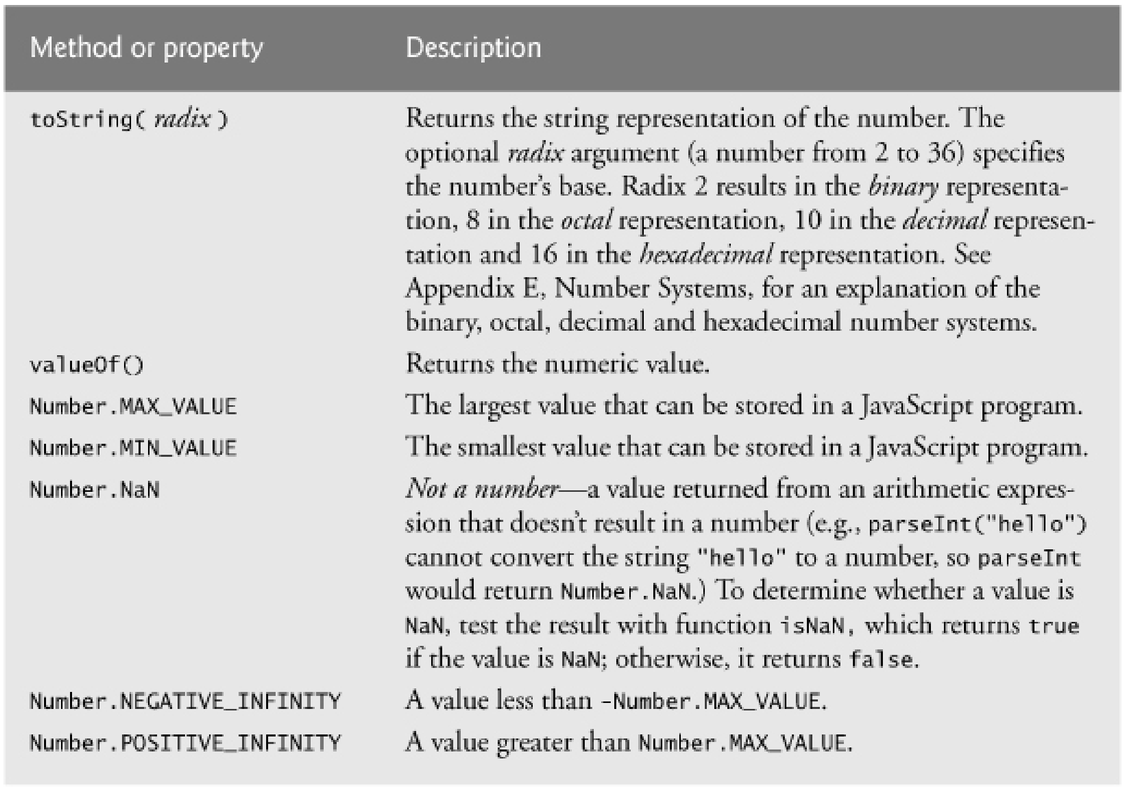
| Method | Description |
|---|---|
| `toString()` | Returns the string `"true"` if the value of the `Boolean` object is `true`; otherwise, returns the string `"false"`. |
| `valueOf()` | Returns the value `true` if the `Boolean` object is `true`; otherwise, returns `false`. |

Fig. 11.13. `Boolean`-object methods.

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* );

The constructor argument *numericValue* is the number to store in the object. Although you can explicitly create `Number` objects, normally the JavaScript interpreter creates them as needed. Figure 11.14 summarizes the methods and properties of the `Number` object.

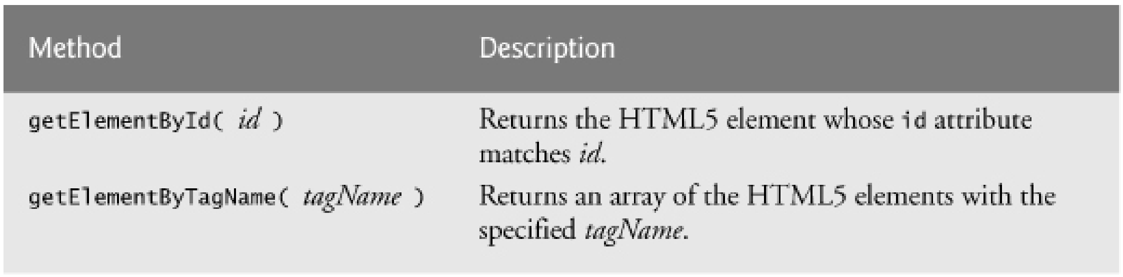


| Method or property | Description |
|---|---|
| `toString(` *radix* `)` | Returns the string representation of the number. The optional *radix* argument (a number from 2 to 36) specifies the number's base. Radix 2 results in the *binary* representation, 8 in the *octal* representation, 10 in the *decimal* 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

The **`document`** object, which we've used extensively, is provided by the browser and allows JavaScript code to manipulate the current document in the browser. The `document` object has several properties and methods, such as method `document.getElementByID` , which has been used in many examples. Figure 11.15 shows the methods of the `document` object that are used in this chapter. We'll cover several more in Chapter 12.



| 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: HTML5 Web Storage

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. For example, a website can retrieve the user's name from a cookie and use it to display a personalized greeting. Similarly, many websites used cookies during a browsing session to track user-specific information, such as the contents of an online shopping cart.

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* (i.e., a cookie set by a script on `amazon.com` can be read only by `amazon.com` servers and scripts). The browser sends these cookies with *every* request to the server.

### Problems with Cookies

There are several problems with cookies. One is that they're extremely limited in size. Today's web apps often allow users to manipulate large amounts of data, such as documents or thousands of emails. Some web applications allow so-called *offline access*—for example, a word-processing web application might allow a user to access documents locally, even when the computer is not connected to the Internet. Cookies cannot store entire documents.

Another problem is that a user often opens many tabs in the same browser window. 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. For example, if the user is purchasing different items in each tab, with cookies it's possible that the user could accidentally purchase the same item twice.

### 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. Unlike cookies, data in the `localStorage` object is not sent to the web server with each request. Each website domain (such as `deitel.com` or `google.com`) has a separate `localStorage` object—all the pages from a given domain share one `localStorage` object. Typically, 5MB are reserved for each `localStorage` object, but a web browser can ask the user if more space should be allocated when the space is full.

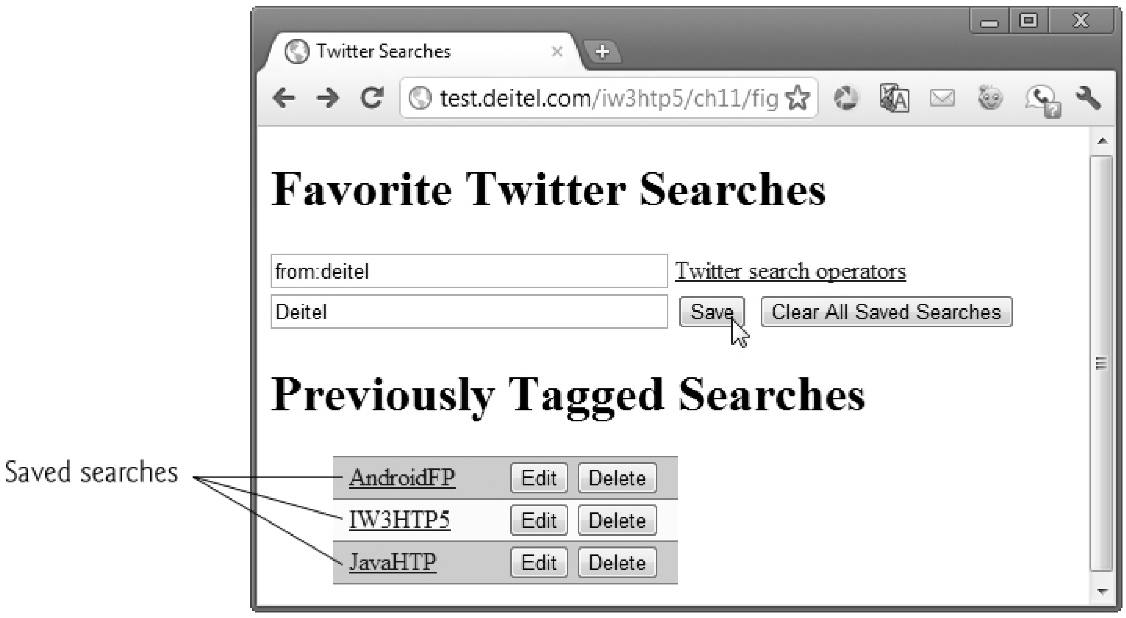
Web applications that need access to data for *only* a browsing session and that must keep that data *separate* among multiple tabs can use the `window` object's **`sessionStorage` property**. There's a separate `sessionStorage` object for every browsing session, including separate tabs that are accessing the same website.

### Favorite Twitter Searches App Using `localStorage` and `sessionStorage`

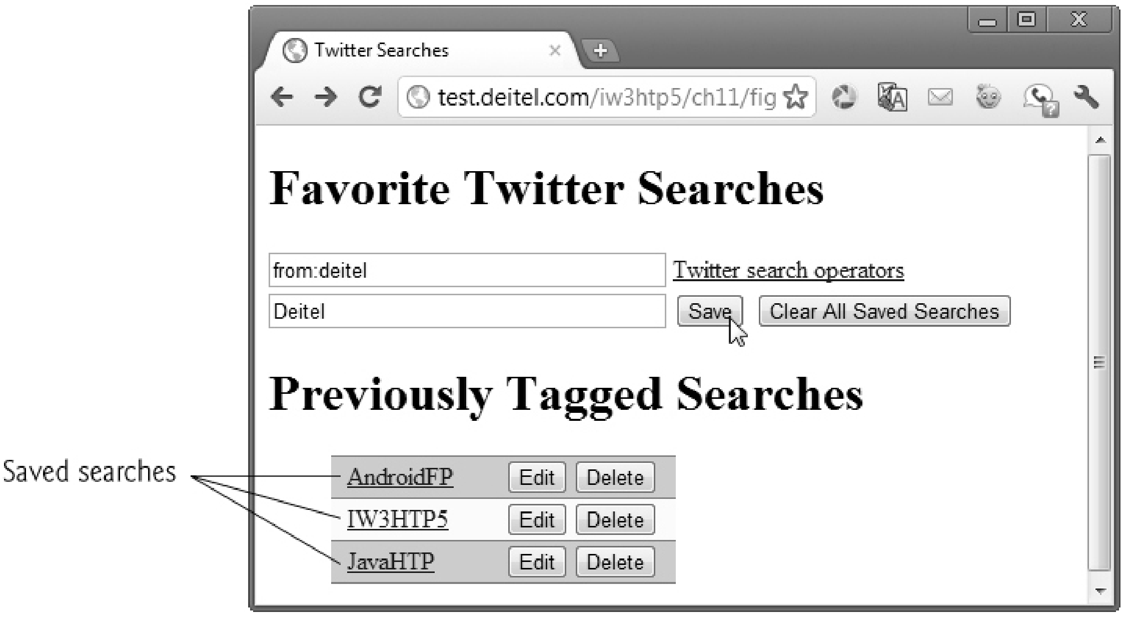
To demonstrate these new HTML5 storage capabilities, we'll implement a **Favorite Twitter Searches** app. Twitter's search feature is a great way to follow trends and see what people are saying about specific topics. The app we present here allows users to save their favorite (possibly lengthy) Twitter 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. Twitter search queries can be finely tuned using Twitter's search operators ( `dev.twitter.com/docs/using-search` )—but more complex queries are lengthy, time consuming and error prone to type. The user's favorite searches are saved using `localStorage` , so they're immediately available each time the user browses the app's web page.

Figure 11.16(a) shows the app when it's loaded for the first time. 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. The user can save many searches and view them in alphabetical order. Search queries and their corresponding tags are entered in the `text` `input` s at the top of the page. Clicking the **Save** `button` adds the new search to the favorites list. Clicking a the link for a saved search requests the search page from Twitter's website, passing the user's saved search as an argument, and displays the search results in the web browser.

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



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



# Favorite Twitter Searches

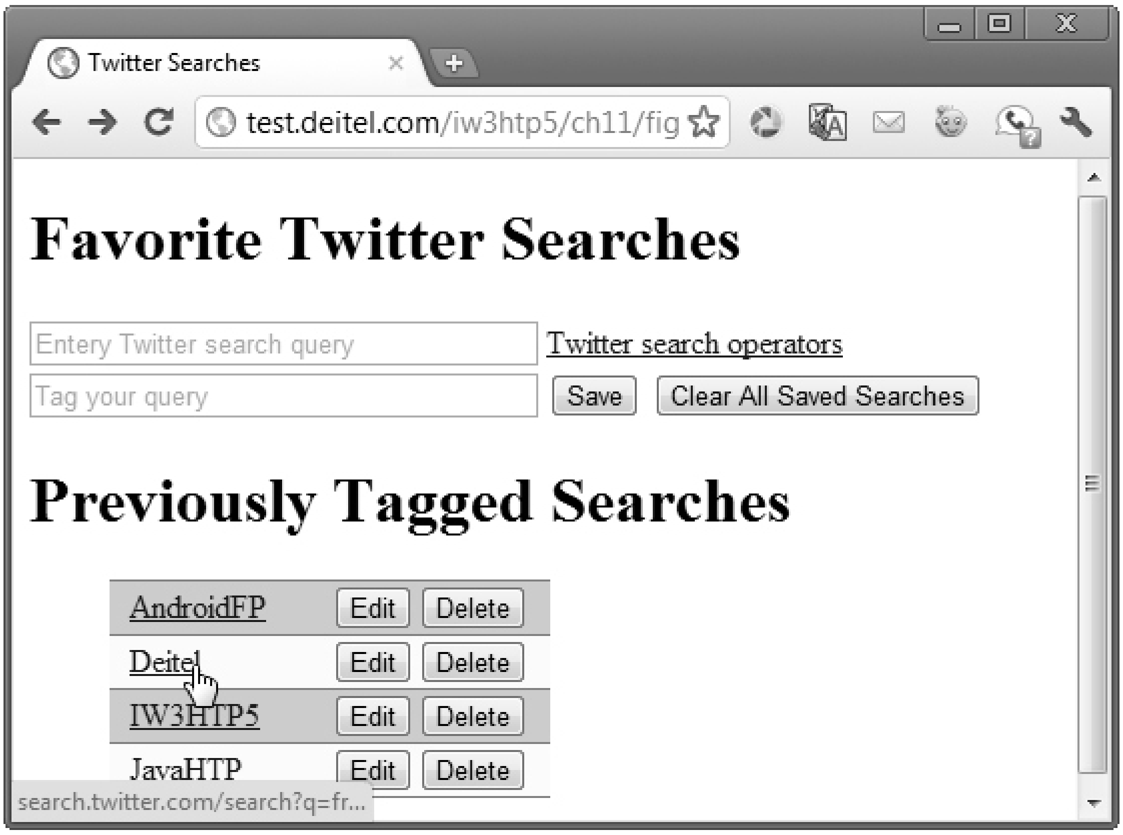
from:deitel

Deitel

## Previously Tagged Searches

|  |  |  |
|---|---|---|
| AndroidFP | Edit | Delete |
| IW3HTP5 | Edit | Delete |
| JavaHTP | Edit | Delete |

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



# Favorite Twitter Searches

Entery Twitter search query

Tag your query

## Previously Tagged Searches

|  |  |  |
|---|---|---|
| AndroidFP | Edit | Delete |
| Deitel | Edit | Delete |
| IW3HTP5 | Edit | Delete |
| JavaHTP | Edit | Delete |

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

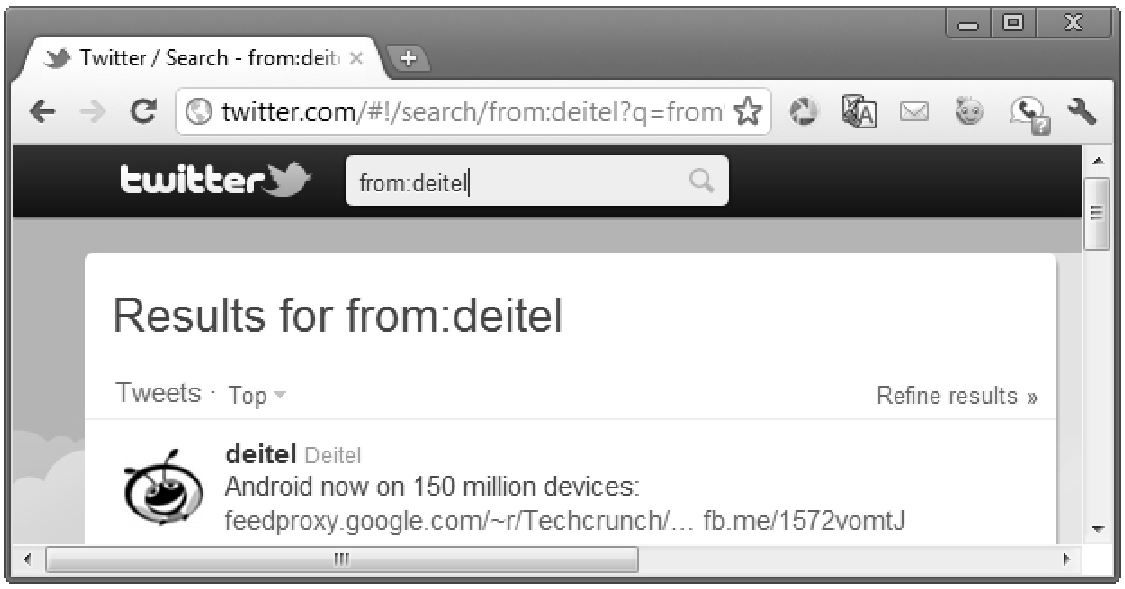

Twitter / Search - from:deit
twitter.com/#!/search/from:deitel?q=from
twitter
from:deitel
Results for from:deitel
Tweets · Top
Refine results »
deitel Deitel
Android now on 150 million devices:
feedproxy.google.com/~r/Techcrunch/... fb.me/1572vomtJ

Fig. 11.16. Sample outputs from the **Favorite Twitter Searches** web application.

Figure 11.16(b) shows the app with several previously saved searches. Figure 11.16(c) shows the user entering a new search. Figure 11.16(d) shows the result of touching the **Deitel** link, which searches for tweets from Deitel—specified in Fig. 11.16(c) with the Twitter search `from:Deitel`. You can edit the searches using the **Edit** `button` s to the right of each search link. This enables you to tweak your searches for better results after you save them as favorites. Touching the **Clear All Saved Searches** `button` removes all the searches from the favorites list. Some browsers support `localStorage` and `sessionStorage` only for web pages that are downloaded from a web server, not for web pages that are loaded directly from the local file system. So, we've posted the app online for testing at:

http://test.deitel.com/iw3htp5/ch11/fig11_20-22/
FavoriteTwitterSearches.html

## Favorite Twitter Searches HTML5 Document

The **Favorite Twitter Searches** application contains three files—`FavoriteTwitterSearches.html` (Fig. 11.17), `styles.css` (Fig. 11.18) and `FavoriteTwitterSearches.js` (Fig. 11.18). The HTML5 document provides a `form` (lines 14–24) that allows the user to enter new searches.

Previously tagged searches are displayed in the `div` named `searches` (line 26).

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

Fig. 11.17. Favorite Twitter Searches web application.

### CSS for Favorite Twitter Searches

Figure 11.18 contains the CSS styles for this app. Line 3 uses a CSS3 attribute selector to select all `input` elements that have the `type` `"text"` and sets their width to `250px`. Each link that represents a saved search is displayed in a `span` that has a fixed width (line 6). To specify the width, we set the `display` property of the `span`s to `inline-block`. Line 8 specifies a **`:first-child` selector** that's used to select the first list item in the unordered list of saved searches that's displayed at the bottom of the web page. Lines 9–10 and 11–12 use **`:nth-child` selectors** to specify the styles of the odd (first, third, fifth, etc.) and even (second, fourth, sixth, etc.) list items, respectively. We use these selectors or alternate the background colors of the saved searches.

```
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.

## Script for Favorite Twitter Searches

Figure 11.19 presents the JavaScript for the **Favorite Twitter Searches** app. When the HTML5 document in Fig. 11.17 loads, function `start` (lines 80–87) is called to register event handlers and call function `loadSearches` (lines 7–44). Line 9 uses the `sessionStorage` object 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, line 11 uses the **`setItem` method** to set the key `"herePreviously"` to the string `"true"`, then lines 12–13 display a welcome message in the `welcomeMessage` paragraph element. Next, line 16 gets the `localStorage` object's **`length`**, which represents the number of key/value pairs stored. Line 17 creates an array and assigns it to the script variable `tags`, then lines 20–23 get the keys from the `localStorage` object and store them in the `tags` array. Method **`key`** (line 22) receives an index as an argument and returns the corresponding key. Line 25 sorts the `tags` array, so that we can display the searches in alphabetical order by tag name (i.e., key). Lines 27–42 build the unordered list of links representing the saved searches. Line 33 calls the `localStorage` object's `getItem` method to obtain the search string for a given tag and appends the search string to the Twitter search URL (line 28). Notice that, for simplicity, lines 37 and 38 use the `onclick` attributes of the dynamically generated **Edit** and **Delete** `button`s to set the `button`s' 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.]

```
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" ) )
```

```
   {
     sessionStorage.setItem( "herePreviously", "true" );
     document.getElementById( "welcomeMessage" ).innerHTML =
       "Welcome to the Favorite Twitter Searches App";
   } // end if

   var length = localStorage.length; // number of key/value pairs
   tags = []; // create empty array

   // load all keys
   for (var i = 0; i < length; ++i)
   {
     tags[i] = localStorage.key(i);
   } // end for

   tags.sort(); // sort the keys

   var markup = "<ul>"; // used to store search link markup
   var url = "http://search.twitter.com/search?q=";

   // build list of links
   for (var tag in tags)
   {
     var query = url + localStorage.getItem(tags[tag]);
     markup += "<li><span><a href = '" + query + "'>" + tags[tag] +
       "</a></span>" +
       "<input id = '" + tags[tag] + "' type = 'button' " +
         "value = 'Edit' onclick = 'editTag(id)'>" +
       "<input id = '" + tags[tag] + "' type = 'button' " +
         "value = 'Delete' onclick = 'deleteTag(id)'>";
   } // end for

   markup += "</ul>";
   document.getElementById("searches").innerHTML = markup;
 } // end function loadSearches

```

```
// deletes all key/value pairs from localStorage
function clearAllSearches()
{
  localStorage.clear();
  loadSearches(); // reload searches
} // end function clearAllSearches

// saves a newly tagged search into localStorage
function saveSearch()
{
  var query = document.getElementById("query");
  var tag = document.getElementById("tag");
  localStorage.setItem(tag.value, query.value);
  tag.value = ""; // clear tag input
  query.value = ""; // clear query input
  loadSearches(); // reload searches
} // end function saveSearch

// deletes a specific key/value pair from localStorage
function deleteTag( tag )
{
  localStorage.removeItem( tag );
  loadSearches(); // reload searches
} // end function deleteTag

// display existing tagged query for editing
function editTag( tag )
{
  document.getElementById("query").value = localStorage[ tag ];
  document.getElementById("tag").value = tag;
  loadSearches(); // reload searches
} // end function editTag

// register event handlers then load searches
function start()
{
```

```
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`.

Function `clearAllSearches` (lines 47–51) is called when the user clicks the **Clear All Saved Searches** button. The **`clear` method** of the `localStorage` object (line 49) removes all key/value pairs from the object. We then call `loadSearches` to refresh the list of saved searches in the web page.

Function `saveSearch` (lines 54–62) is called when the user clicks **Save** to save a search. Line 58 uses the `setItem` method to store 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. We then call `loadSearches` to refresh the list of saved searches in the web page.

Function `deleteTag` (lines 65–69) is called when the user clicks the **Delete** button next to a particular search. The function receives the tag representing the key/value pair to delete, which we set in line 38 as the `button`'s `id`. Line 67 uses the **`removeItem` method** to remove a key/value pair from the `localStorage` object. We then call `loadSearches` to refresh the list of saved searches in the web page.

Function `editTag` (lines 72–77) is called when the user clicks the **Edit** button next to a particular search. The function receives the tag representing the key/value pair to edit, which we set in line 36 as the `button`'s `id`. In this case, we display the corresponding key/value pair's contents in the `input` elements with the `id` s `"tag"` and `"query"` , respectively, so the user can edit them. Line 74 uses the `[]` operator to access the value for a specified key ( `tag` )—this performs the same task as calling `getItem` on the `localStorage` object. We then call `loadSearches` to refresh the list of saved searches in the web page.

## 11.8. Using JSON to Represent Objects

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

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

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

[ *value0*, *value1*, *value2* ]

Each value can be a string, a number, a JSON object, `true, false` or `null` . To appreciate the simplicity of JSON data, examine this representation of an array of address–book entries that we'll use in Chapter 16:

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

JSON provides a straightforward way to manipulate objects in JavaScript, and many other programming languages now support this format. In addition to simplifying object creation, JSON allows programs to easily extract data and efficiently transmit it across the Internet. JSON integrates especially well with Ajax applications, discussed in Chapter 16. See Section 16.6 for a more detailed discussion of JSON, as well as an Ajax-specific example. For more information on JSON, visit our JSON Resource Center at www.deitel.com/json.

## Summary

### Section 11.2 `Math` Object

- `Math` -object methods (p. 361) enable you to perform many common mathematical calculations.
- An object's methods are called by writing the name of the object followed by a dot ( `.` ) and the name of the method. In parentheses following the method name are arguments to the method.

### Section 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's 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 (p. 363), which represents a large portion of the world's languages.
- String literals or string constants (p. 363) are written as a sequence of characters in double or single quotation marks.
- Combining strings is called concatenation (p. 363).
- String method `charAt` (p. 365) returns the character at a specific index in a string. Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

- `String` method `charCodeAt` (p. 365) returns the Unicode value of the character at a specific index in a string. If the index is outside the bounds of the string, the method returns `NaN`. `String` method `fromCharCode` (p. 365) creates a string from a list of Unicode values.

- `String` method `toLowerCase` (p. 365) returns the lowercase version of a string. `String` method `toUpperCase` (p. 365) returns the uppercase version of a string.

- `String` method `indexOf` (p. 366) determines the location of the first occurrence of its argument in the string used to call the method. If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.

- `String` method `lastIndexOf` (p. 366) determines the location of the last occurrence of its argument in the string used to call the method. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.

- The process of breaking a string into tokens (p. 369) is called tokenization (p. 369). 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` (p. 369) breaks a string into its component tokens. The argument to method `split` is the delimiter string (p. 370)—the string that determines the end of each token in the original string. Method `split` returns an array of strings containing the tokens.

- `String` method `substring` returns the substring from the starting index (its first argument, p. 370) up to but not including the ending index (its second argument, p. 370). 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.

**Section 11.4 `Date` Object**

- JavaScript's `Date` object (p. 371) provides methods for date and time manipulations.

- Date and time processing can be performed based either on the computer's local time zone (p. 371) or on World Time Standard's Coordinated Universal Time (abbreviated UTC, p. 371)—formerly called Greenwich Mean Time (GMT, p. 371).

- Most methods of the `Date` object have a local time zone and a UTC version.

- `Date` method `parse` receives as its argument a string representing a date and time and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.

- `Date` method `UTC` (p. 375) returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required year, month and date, and the optional hours, minutes, seconds and milliseconds. If any of the hours, minutes, seconds or milliseconds arguments is not specified, a zero is supplied in its place. For the hours, minutes and seconds arguments, if the argument to the right of any of these arguments is specified, that argument must also be specified (e.g., if the minutes argument is specified, the hours argument must be specified; if the milliseconds argument is specified, all the arguments must be specified).

**Section 11.5 `Boolean` and `Number` Objects**

- JavaScript provides the `Boolean` (p. 376) and `Number` (p. 376) objects as object wrappers for boolean `true/false` values and numbers, respectively.

- When a boolean value is required in a JavaScript program, JavaScript automatically creates a `Boolean` object to store the value.

- JavaScript programmers can create `Boolean` objects explicitly with the statement

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

  The argument *booleanValue* specifies the value of the `Boolean` object (`true` or `false`). If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or the empty string (`""`), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`.

- JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program.

- JavaScript programmers can create a `Number` object with the statement

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

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

**Section 11.6 `document` Object**

- JavaScript provides the `document` object (p. 377) for manipulating the document that's currently visible in the browser window.

**Section 11.7 Favorite Twitter Searches: HTML5 Web Storage**

- Before HTML5, websites could store only small amounts of text-based information on a user's computer using cookies. A cookie (p. 378) 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.

- 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.

- Web applications can use the `window` object's `localStorage` property (p. 378) 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.

- Unlike cookies, data in the `localStorage` object is not sent to the web server with each request.

- Each website domain has a separate `localStorage` object—all the pages from a given domain share it. Typically, 5MB are reserved for each `localStorage` object, but a web browser can ask the user whether more space should be allocated when the space is full.

- Web applications that need access to key/value pair data for only a browsing session and that must keep that data separate among multiple tabs can use the window object's `sessionStorage` property (p. 379). There's a separate `sessionStorage` object for every browsing session, including separate tabs that are accessing the same website.

- A CSS3 `:first-child` selector (p. 382) selects the first child of an element.

- A CSS3 `:nth-child` selector (p. 382) with the argument `"odd"` selects the odd child elements, and one with the argument `"even"` selects the even child elements.

- The `localStorage` and `sessionStorage` method `getItem` (p. 382) receives a name of a key as an argument. If the key exists, the method returns the corresponding string value; otherwise, it returns `null`. Method `setItem` (p. 382) sets a key/value pair. If the key already exits, `setItem` replaces the value for the specified key; otherwise, it creates a new key/value pair.

- The `localStorage` and `sessionStorage` `length` property (p. 382) returns the number of key/value pairs stored in the corresponding object.

- The `localStorage` and `sessionStorage` method `key` (p. 382) receives an index as an argument and returns the corresponding key.
- The `localStorage` and `sessionStorage` method `clear` (p. 384) removes all key/value pairs from the corresponding object.
- The `localStorage` and `sessionStorage` method `removeItem` (p. 385) removes a key/value pair from the corresponding object.
- In addition to `getItem`, you can use the `[]` operator to access the value for a specified key in a `localStorage` or `sessionStorage` object.

**Section 11.8 Using JSON to Represent Objects**

- JSON (JavaScript Object Notation, p. 385) 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:

  [ *value0*, *value1*, *value2* ]

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

## Self-Review Exercise

**11.1** Fill in the blanks in each of the following statements:

**a.** Because JavaScript uses objects to perform many tasks, JavaScript is commonly referred to as a(n) ________.

**b.** All objects have ________ and exhibit ________.

**c.** The methods of the ________ object allow you to perform many common mathematical calculations.

**d.** Invoking (or calling) a method of an object is referred to as ________.

**e.** String literals or string constants are written as a sequence of characters in ________ or ________.

**f.** Indices for the characters in a string start at ________.

**g.** `String` methods ________ and ________ search for the first and last occurrences of a substring in a `String`, respectively.

**h.** The process of breaking a string into tokens is called ________.

**i.** Date and time processing can be performed based on the ________ or on World Time Standard's ________.

**j.** `Date` method ________ receives as its argument a string representing a date and time and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.

**k.** Web applications can use the `window` object's ________ 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.

**l.** Web applications that need access to key/value pair data for only a browsing session and that must keep that data separate among multiple tabs can use the window object's ________ property.

**m.** A CSS3 ________ selector selects the first child of an element.

**n.** A CSS3 ________ selector with the argument `"odd"` selects the odd child elements, and one with the argument `"even"` selects the even child elements.

## Answers to Self-Review Exercise

**11.1**

**a.** object-based programming language.

**b.** attributes, behaviors.

**c.** `Math`.

**d.** sending a message to the object.

**e.** double quotation marks, single quotation marks.

**f.** 0.

**g.** `indexOf`, `lastIndexOf`.

**h.** tokenization.

**i.** computer’s local time zone, Coordinated Universal Time (UTC).

**j.** `parse`.

**k.** `localStorage`.

**l.** `sessionStorage`.

**m.** `:first-child`.

**n.** `:nth-child`.

## Exercises

**11.2** Create a web page that contains four buttons. Each button, when clicked, should cause an alert dialog to display a different time or date in relation to the current time. Create a `Now` button that alerts the current time and date and a `Yesterday` button that alerts the time and date 24 hours ago.

The other two buttons should alert the time and date ten years ago and one week from today.

**11.3** Write a script that tests as many of the `Math` library functions in Fig. 11.1 as you can. Exercise each of these functions by having your program display tables of return values for several argument values in an HTML5 `textarea`.

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

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

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

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

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

**a.** `roundToInteger( number )`

**b.** `roundToTenths( number )`

**c.** `roundToHundredths( number )`

**d.** `roundToThousandths( number )`

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

**11.5** Modify the solution to Exercise 11.4 to use `Math` method `round` instead of method `floor`.

**11.6** Write a script that uses relational and equality operators to compare two `String` s input by the user through an HTML5 form. Display whether the first string is less than, equal to or greater than the second.

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

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

The program should generate 20 sentences to form a short story and output the result to an HTML5 `textarea` . The story should begin with a line reading `"Once upon a time..."` and end with a line reading `"THE END"` .

**11.8** ***(Limericks)*** A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 11.7, write a script that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

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

To form a pig Latin phrase from an English-language phrase, tokenize the phrase into an array of words using `String` method `split`. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters “ `ay`.” Thus the word “ `jump` ” becomes “ `umpjay`,” the word “ `the` ” becomes “ `hetay` ” and the word “ `computer` ” becomes “ `omputercay`.” Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. Each token (i.e., word in the sentence) is passed to method `printLatinWord` to print the pig Latin word. Enable the user to input the sentence through an HTML5 form. Keep a running display of all the converted sentences in an HTML5 `textarea`.

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

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

**11.12** Write a script that inputs text from an HTML5 form and outputs it in uppercase and lowercase letters.

**11.13** Write a script that inputs several lines of text and a search character and uses `String` method `indexOf` to determine the number of occurrences of the character in the text.

**11.14** Write a script based on the program in Exercise 11.13 that inputs several lines of text and uses `String` method `indexOf` to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format in an HTML5 `textarea` after the totals have been determined.

**11.15** Write a script that reads a series of strings and outputs in an HTML5 `textarea` only those strings beginning with the character “ `b` .”

**11.16** Write a script that reads a series of strings and outputs in an HTML5 `textarea` only those strings ending with the characters “ `ed` .”

**11.17** Write a script that inputs an integer code for a character and displays the corresponding character.

**11.18** Modify your solution to Exercise 11.17 so that it generates all possible three-digit codes in the range 000 to 255 and attempts to display the corresponding characters. Display the results in an HTML5 `textarea` .

**11.19** Write your own version of the `String` method `indexOf` and use it in a script.

**11.20** Write your own version of the `String` method `lastIndexOf` and use it in a script.

**11.21** Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word “bathe” include the commonly used words “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.” Output the results in an HTML5 `textarea` .

**11.22** ***(Printing Dates in Various Formats)*** Dates are printed in several common formats. Write a script that reads a date from an HTML5 form and creates a `Date` object in which to store it. Then use the various methods of the `Date` object that convert `Date` s into strings to display the date in several formats.

## Special Section: Challenging String-Manipulation Projects

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

**11.23** ***(Text Analysis)*** The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare really wrote the works attributed to him. Some scholars believe there's substantial evidence indicating that Christopher Marlowe actually penned these masterpieces. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

**a.** Write a script that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

To be, or not to be: that is the question:

contains one "a," two "b's," no "c's," etc.

**b.** Write a script that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

Whether 'tis nobler in the mind to suffer

contains

| Word length | Occurrences |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 1 |
| 4 | 2 (including 'tis) |
| 5 | 0 |
| 6 | 2 |
| 7 | 1 |

**c.** Write a script that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer

contain the word "to" three times, the word "be" twice, and the word "or" once. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

**11.24** ***(Check Protection)*** Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Incorrect amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example:

```
1,230.60 (check amount)
--------
12345678 (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces will ordinarily be left blank. For example,

```
   99.87
--------
12345678
```

contains three blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87
--------
12345678
```

Write a script that inputs a dollar amount to be printed on a check, then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing the amount.

**11.25** ***(Writing the Word Equivalent of a Check Amount)*** Continuing the discussion in the preceding exercise, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be both written in numbers and spelled out in words. Even if someone is able to alter the numerical amount of the check, it's extremely difficult to change the amount in words.

Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.

Write a script that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100

**11.26** ***(Metric Conversion Program)*** Write a script that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (e.g., centimeters, liters, grams, for the metric system and inches, quarts, pounds, for the English system) and should respond to simple questions such as

"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"

Your program should recognize invalid conversions. For example, the question

"How many feet are in 5 kilograms?"

is not a meaningful question because `"feet"` is a unit of length whereas `"kilograms"` is a unit of mass.

**11.27** ***(Project: A Spell Checker)*** Many popular word-processing software packages have built-in spell checkers.

In this project, you’re asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. Use a computerized dictionary (if you have access to one) as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it’s because we simply do not know the correct spelling, so we make a best guess. In some cases, it’s because we transpose two letters (e.g., “defualt” instead of “default”). Sometimes we double-type a letter accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"). And so on.

Design and implement a spell-checker application in JavaScript. Your program should maintain an array `wordList` of strings. Enable the user to enter these strings.

Your program should ask a user to enter a word. The program should then look up the word in the `wordList` array. If the word is present in the array, your program should print "`Word is spelled correctly`."

If the word is not present in the array, your program should print "`word is not spelled correctly`." Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word "default" is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as "edfault," "dfeault," "deafult," "defalut" and "defautl." When you find a new word that matches one in `wordList`, print that word in a message, such as "`Did you mean "default?"`"

Implement any other tests you can develop, such as replacing each double letter with a single letter, to improve the value of your spell checker.

**11.28** ***(Project: Crossword Puzzle Generator)*** Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is suggested here as a string-manipulation project requiring substantial sophistication and effort.

You must resolve many issues to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle in the computer? Should one use a series of strings, or use double-subscripted arrays?

You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program?

The really ambitious reader will want to generate the clues portion of the puzzle, in which the brief hints for each across word and each down word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.