The String Object

26 CHAPTER

In This Chapter

How to parse and work with text

Performing search-and-replace operations

Scripted alternatives to text formatting

Chapter 6's tutorial introduced you to the concepts of values and the types of values that JavaScript works with — things such as strings, numbers, and Boolean values. In this chapter, you look more closely at the very important String data type, as well as its relationship to the Number data type. Along the way, you encounter the many ways in which JavaScript enables scripters to manipulate strings.

Much of the syntax you see in this chapter is identical to that of the Java programming language. Because the scope of JavaScript activity is narrower than that of Java, there isn't nearly as much to learn for JavaScript as for Java. At the same time, certain string object language features apply to scripting but not to Java programming. Improvements to the string object's methods in Navigator 4 greatly simplify a number of string manipulation tasks. If you must script for a lower common denominator of browser, however, you may need some of the same kind of string micro-management skills that a C programmer needs. I'll soften the blow by providing some general purpose functions that you can plug into your scripts to make those jobs easier.

String and Number Data Types

Although JavaScript is not what is known as a "strongly typed" language, you still need to be aware of several data types because of their impact on the way you work with the information in those forms. In this section, I focus on strings and two types of numbers.

Simple strings

A string consists of one or more standard text characters between matching quote marks. JavaScript is forgiving in one regard: You can use single or double quotes, as long as you match two single quotes or two double quotes around a string. Another benefit to this scheme becomes apparent when you try to include a quoted string inside a string. For example, say that you're assembling a line of HTML code in a variable that you will eventually write to a new window

completely controlled by JavaScript. The line of text you want to assign to a variable is this:

```
<INPUT TYPE="checkbox" NAME="candy">Chocolate
```

To assign this entire line of text to a variable, you have to surround the line in quotes. But because quotes appear inside the string, JavaScript (or any language) has problems deciphering where the string begins or ends. By carefully placing the other kind of quote pairs, however, you can make the assignment work. Here are two equally valid ways:

```
result = '<INPUT TYPE="checkbox" NAME="candy">Chocolate'
result = "<INPUT TYPE='checkbox' NAME='candy'>Chocolate"
```

Notice in both cases, the entire string is surrounded by the same unique pair of quotes. Inside the string, two quoted strings appear that will be treated as such by JavaScript. I recommend that you settle on one form or the other and then use it consistently throughout your scripts.

Building long string variables

The act of joining strings together — *concatenation* — enables you to assemble long strings out of several little pieces. This feature is very important for some of your scripting — for example, when you need to build an HTML page's specifications entirely within a variable before writing the page to another frame with one `document.write()` statement.

One tactic I use keeps the length of each statement in this building process short enough so it's easily readable in your text editor. This method uses the add-by-value assignment operator (+=) that appends the right-hand side of the equation to the left-hand side. Here is a simple example, which begins by initializing a variable as an empty string:

```
var newDocument = ""
newDocument += "<HTML><HEAD><TITLE>Life and Times</TITLE></HEAD>"
newDocument += "<BODY><H1>My Life and Welcome to It</H1>"
newDocument += "by Sidney Finortny<HR>"
```

Starting with the second line, each statement adds more data to the string being stored in `newDocument`. You can continue appending string data until the entire page's specification is contained in the `newDocument` variable.

Joining string literals and variables

In some cases, you need to create a string out of literal strings (characters with quote marks around them) and string variable values. The methodology for concatenating these types of strings is no different from that of multiple string literals. The plus-sign operator does the job. Therefore, in the following example, a variable contains a name. That variable value is made a part of a larger string whose other parts are string literals:

```
yourName = prompt("Please enter your name:","")
var msg = "Good afternoon, " + yourName + "."
alert(msg)
```

Some common problems that you may encounter while attempting this kind of concatenation include the following:

✦ Accidentally omitting one of the quotes around a literal string

✦ Failing to insert blank spaces in the string literals to accommodate word spaces

✦ Forgetting to concatenate punctuation after a variable value

Also, don't forget that what I show here as variable values can be any expression that evaluates to a string, including property references and the results of some methods. For example

```
var msg = "The name of this document is " + document.title + "."
alert(msg)
```

Special inline characters

The way string literals are created in JavaScript makes adding certain characters to strings difficult. I'm talking primarily about adding quotes, apostrophes, carriage returns, and tab characters to strings. Fortunately, JavaScript provides a mechanism for entering such characters into string literals. A backslash symbol, followed by the character you want to appear as inline, makes that task happen. For the "invisible" characters, a special set of letters following the backslash tells JavaScript what to do.

The most common backslash pairs are as follows:

`\"`	Double quote
`\'`	Single quote (apostrophe)
`\\`	Backslash
`\b`	Backspace
`\t`	Tab
`\n`	New line
`\r`	Carriage return
`\f`	Form feed

Use these "inline characters" (also known as "escaped characters," but this terminology has a different connotation for Internet strings) inside quoted string literals to make JavaScript recognize them. When assembling a block of text that needs a new paragraph, insert the `\n` character pair. Here are some examples of syntax using these special characters:

```
msg = "You\'re doing fine."
msg = "This is the first line.\nThis is the second line."
msg = document.title + "\n" + document.links.length + " links present."
```

Technically speaking, a complete carriage return, as known from typewriting days, is both a line feed (advance the line by one) and a carriage return (move the

carriage all the way to the left margin). Although JavaScript strings treat a line feed (\n new line) as a full carriage return, you may have to construct \r\n breaks when assembling strings that go back to a CGI script on a server. The format that you use all depends on the string-parsing capabilities of the CGI program. (Also see the special requirements for the textarea object in Chapter 22.)

It's easy to confuse the strings assembled for display in textarea objects or alert boxes with strings to be written as HTML. For HTML strings, make sure that you use the standard HTML tags for line breaks (<BR>) and paragraph breaks (<P>) rather than the inline return or line feed symbols.

String Object

Properties	*Methods*	*Event Handlers*
`length`	`anchor()`	(None)
`prototype`	`big()`	
	`blink()`	
	`bold()`	
	`charAt()`	
	`charCodeAt()`	
	`concat()`	
	`fixed()`	
	`fontcolor()`	
	`fontsize()`	
	`fromCharCode()`	
	`indexOf()`	
	`italics()`	
	`lastIndexOf()`	
	`link()`	
	`match()`	
	`replace()`	
	`search()`	
	`slice()`	
	`small()`	
	`split()`	
	`strike()`	
	`sub()`	
	`substr()`	
	`substring()`	

Properties	Methods	Event Handlers
`prototype`	`sup()`	
	`toLowerCase()`	
	`toUpperCase()`	

Syntax

Creating a string object:

```
var myString = new String("characters")
```

Accessing select object properties and methods:

```
string.property | method
```

About this object

JavaScript draws a fine line between a string value and a string object. Both let you use the same methods on their contents, so by and large, you do not have to create a string object (with the `new String()` constructor) every time you want to assign a string value to a variable. A simple assignment operation (`var myString = "fred"`) is all you need to create a string value that behaves on the surface very much like a full-fledged string object.

Where the difference comes into play is when you wish to exploit the "object-ness" of a genuine string object, which I explain further in the discussion of the `string.prototype` property later in this chapter.

With string data often comes the need to massage that text in scripts. In addition to concatenating strings, you at times need to extract segments of strings, delete parts of strings, and replace one part of a string with some other text. Unlike many plain-language scripting languages, JavaScript is fairly low-level in its built-in facilities for string manipulation. This means that unless you can take advantage of the regular expression powers of Navigator 4, you must fashion your own string handling routines out of very elemental powers built into JavaScript. Later in this chapter, I provide several functions that you can use in your own scripts for common string handling.

As you work with string values, visualize every string value as an object with properties and methods like other JavaScript objects. JavaScript defines one property and a slew of methods for any string value (and one extra property for a true string object). The syntax is the same for string methods as it is for any other object method:

```
stringObject.method()
```

What may seem odd at first is that the `stringObject` part of this reference can be any expression that evaluates to a string, including string literals, variables containing strings, or other object properties. Therefore, the following examples of calling the `toUpperCase()` method are all valid:

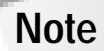
```
"george burns".toUpperCase()
```

```
yourName.toUpperCase() // yourName is a variable containing a string
document.forms[0].entry.value.toUpperCase() // entry is a text field
object
```

An important concept to remember is that invoking a string method does not change the string object that is part of the reference. Rather, the method returns a value, which can be used as a parameter to another method or function call, or assigned to a variable value.

Therefore, to change the contents of a string variable to the results of a method, you must use an assignment operator, as in

```
yourName = yourName.toUpperCase() // variable is now all uppercase
```



Note

In Navigator 2, avoid nesting method calls for the same string object when the methods modify the string. The evaluation does not work as you might expect. Instead, break out each call as a separate JavaScript statement.

Properties

length

Value: Integer **Gettable:** Yes **Settable:** No

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The most frequently used property of a string is `length`. To derive the length of a string, extract its property as you would extract the `length` property of any object:

```
string.length
```

The `length` value represents an integer count of the number of characters within the string. Spaces and punctuation symbols count as characters. Any backslash special characters embedded in a string count as one character, including such characters as newline and tab. Here are some examples:

```
"Lincoln".length // result = 7
"Four score".length // result = 10
"One\ntwo".length // result = 7
"".length // result = 0
```

The `length` property is commonly summoned when dealing with detailed string manipulation in repeat loops.

prototype

Value: Object **Gettable:** Yes **Settable:** Yes

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔			✔

String objects defined with the `new String("stringValue")` constructor are robust objects compared to plain old variables that are assigned string values. You certainly don't have to create this kind of string object for every string in your scripts, but these objects do come in handy when you find that strings in variables go awry. This happens occasionally while trying to preserve string information as script variables in other frames or windows. By using the string object constructor, you can be relatively assured that the string value will be available in the distant frame when needed.

Another byproduct of true string objects is that you can assign prototype properties and methods to all string objects in the document. A prototype is a property or method that becomes a part of every new object created after the prototype items have been added. For strings, as an example, you may want to define a new method for converting a string into a new type of HTML font tag not already defined by JavaScript's string object. Listing 26-1 shows how to create and use such a prototype.

Listing 26-1: **A String Object Prototype**

```
<HTML>
<HEAD>
<TITLE>String Object Prototype</TITLE>
<SCRIPT LANGUAGE="JavaScript1.1">
function makeItHot() {
       return "<FONT COLOR='red'>" + this.toString() + "</FONT>"
}
String.prototype.hot = makeItHot
</SCRIPT>
<BODY>
<SCRIPT LANGUAGE="JavaScript1.1">
document.write("<H1>This site is on " + "FIRE".hot() + "!!</H1>")
</SCRIPT>
</BODY>
</HTML>
```

A function definition (`makeItHot()`) accumulates string data to be returned to the object when the function is invoked as the object's method. The `this` keyword extracts the object making the call, which you convert to a string for concatenation with the rest of the strings to be returned. In the page's Body, I call upon that prototype method in the same way one calls upon existing String methods that turn strings into HTML tags (discussed later in this chapter).

In the next sections, I divide the string object methods into two distinct categories. The first, parsing methods, focuses on string analysis and character manipulation within strings. The second group, formatting methods, are devoted

entirely to assembling strings in HTML syntax for those scripts that assemble the text to be written into new documents or other frames.

Parsing methods

string.charAt(*index*)

Returns: Character in *string* at the *index* count.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

Use the `string.charAt()` method to extract a single character from a string when you know the position of that character. For this method, you specify an index value in the string as a parameter to the method. The index value of the first character of the string is 0. To grab the last character of a string, mix string methods:

```
myString.charAt(myString.length - 1)
```

If your script needs to get a range of characters, use the `string.substring()` method. It is a common mistake to use `string.substring()` to extract a character from inside a string, when the `string.charAt()` method is more efficient.

Examples

```
char = "banana daiquiri".charAt(0) // result = "b"
char = "banana daiquiri".charAt(5) // result = "a" (third "a" in
"banana")

char = "banana daiquiri".charAt(6) // result = " " (a space character)

char = "banana daiquiri".charAt(20) // result = "" (empty string)
```

Related Items: `string.lastIndexOf()` method; `string.IndexOf()` method; `string.substring()` method.

string.charCodeAt([index])
String.fromCharCode(num1 [, num2 [, ... numn]])

Returns: Integer code number for a character; concatenated string value of code numbers supplied as parameters.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

Conversions from plain language characters to their numeric equivalents have a long tradition in computer programming. For a long time, the most common numbering scheme was the ASCII standard, which covers the basic English alphanumeric characters and punctuation within 128 values (numbered 0 through 127). An extended version with a total of 256 characters, with some variations depending on the operating system, accounts for other roman characters in other languages, particularly vowels with umlauts and other pronunciation marks. To bring all languages, including pictographic languages and other nonroman alphabets, into the computer age, a world standard called Unicode provides space for thousands of characters.

In JavaScript, the character conversions are string methods. Acceptable values depend on the browser you are using. Navigator works only with the 256 ISO-Latin-I values; Internet Explorer works with the Unicode system.

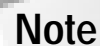
The two methods that perform these conversions work in very different ways syntactically. The first, *string*.charCodeAt(), converts a single string character to its numerical equivalent. The string being converted is the one to the left of the method name — and it may be a literal string or any other expression that evaluates to a string value. If no parameter is passed, the character being converted is by default the first character of the string. However, you can also specify a different character as an index value into the string (first character is 0), as demonstrated here:

```
"abc".charCodeAt()  // result = 97
"abc".charCodeAt(0)  // result = 97
"abc".charCodeAt(1)  // result = 98
```

If the string value is an empty string, the result is NaN.

To convert numeric values to their characters, use the String.fromCharCode() method. Notice that the object beginning the method call is the generic string object, not a string value. Then, as parameters, you can include one or more integers separated by commas. In the conversion process, the method combines the characters for all of the parameters into one string, an example of which is shown here:

```
String.fromCharCode(97, 98, 99)  // result "abc"
```



Note

The *string*.charCodeAt() method is broken on the Macintosh version of Navigator 4, and always returns NaN.

Example

Listing 26-2 provides examples of both methods on one page. Moreover, because one of the demonstrations relies on the automatic capture of selected text on the page, the scripts include code to accommodate the different handling of selection events and capture of the selected text in Navigator and Internet Explorer 4.

After you load the page, select part of the body text anywhere on the page. If you start the selection with the lowercase letter “a,” the character code displays as 97. If you select no text, the result is NaN.

Try entering numeric values in the three fields at the bottom of the page. Values below 32 are ASCII control characters that most fonts represent as hollow squares. But try all other values to see what you get. Notice that the script passes all three values as a group to the String.fromCharCode() method, and the result is a combined string.

Listing 26-2: **Character Conversions**

```
<HTML>
<HEAD>
<TITLE>Character Codes</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var isNav = (navigator.appName == "Netscape")
function showProps(objName,obj) {
        var msg = ""
        for (var i in obj) {
           objName + "." + i + "=" + obj[i]
        }
        alert(msg)
}
function showCharCode() {
        if (isNav) {
           var theText = document.getSelection()
        } else {
           var theText = document.selection.createRange().text
        }
        document.forms[0].charCodeDisplay.value = theText.charCodeAt()
}
function showString(form) {
        form.result.value =
String.fromCharCode(form.entry1.value,form.entry2.value,form.entry3.value)
}
if (isNav) {
        document.captureEvents(Event.MOUSEUP)
}
document.onmouseup = showCharCode
</SCRIPT>
</HEAD>
<BODY>
<B>Capturing Character Codes</B>
<FORM>
Select any of this text, and see the character code of the first
character.<P>
Character Code:<INPUT TYPE="text" NAME="charCodeDisplay" SIZE=3><BR>
<HR>
<B>Converting Codes to Characters</B><BR>
Enter a value  0-255:<INPUT TYPE="text" NAME="entry1" SIZE=4><BR>
Enter a value  0-255:<INPUT TYPE="text" NAME="entry2" SIZE=4><BR>
Enter a value  0-255:<INPUT TYPE="text" NAME="entry3" SIZE=4><BR>
<INPUT TYPE="button" VALUE="Show String" onClick="showString(this.form)">
Result:<INPUT TYPE="text" NAME="result" SIZE=5>
</FORM>
</BODY>
</HTML>
```

Related Items: None.

string.concat(*string2*)

Returns: Combined string.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

JavaScript's add-by-value operator (+=) provides a convenient way to concatenate strings. Navigator 4, however, introduces a string object method that performs the same task. The base string to which more text is appended is the object or value to the left of the period. The string to be appended is the parameter of the method, as the following example demonstrates:

```
"abc".concat("def")  // result: "abcdef"
```

Like the add-by-value operator, the `concat()` method doesn't know about word endings. You are responsible for including the necessary space between words if the two strings require a space between them in the result.

Related Items: Add-by-value (+=) operator.

string.indexOf(*searchString* [, *startIndex*])

Returns: Index value of the character within *string* where *searchString* begins.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

Like some languages' offset string function, JavaScript's `indexOf()` method enables your script to obtain the number of the character in the main string where a search string begins. Optionally, you can specify where in the main string the search should begin — but the returned value is always relative to the very first character of the main string. Like all string object methods, index values start their count with 0. If no match occurs within the main string, the returned value is -1. Thus, this method is a convenient way to determine whether one string contains another.

A bug exists in some versions of Navigator 2 and 3 that can trip up your scripts if you don't guard against it. If the string being searched is empty, the `indexOf()` method returns an empty string rather than the expected -1 value. Therefore, you may want to test to make sure the string is not empty before applying this method. A look at the following examples tells you more about this method than a long description. In all examples, you assign the result of the method to a variable named `offset`.

Examples

```
offset = "bananas".indexOf("b") // result = 0 (index of first letter is
zero)
offset = "bananas".indexOf("a") // result = 1
offset = "bananas".indexOf("a",1) // result = 1 (start from second
letter)
offset = "bananas".indexOf("a",2) // result = 3 (start from third
letter)
offset = "bananas".indexOf("a",4) // result = 5 (start from fifth
letter)
offset = "bananas".indexOf("nan") // result = 2
offset = "bananas".indexOf("nas") // result = 4
offset = "bananas".indexOf("s") // result = 6
offset = "bananas".indexOf("z") // result = -1 (no "z" in string)
```

Related Items: `string.lastIndexOf()`; `string.charAt()`; `string.substring()`.

string.lastIndexOf(*searchString* [, *startIndex*])

Returns: Index value of the last character within *string* where *searchString* begins.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The `string.lastIndexOf()` method is closely related to the `string.IndexOf()` method. The only difference is that this method starts its search for a match from the end of the string (`string.length - 1`) and works its way backward through the string. All index values are still counted, starting with 0, from the front of the string. In the examples that follow, I use the same values as in the examples for `string.IndexOf` so that you can compare the results. In cases where only one instance of the search string is found, the results are the same; but when multiple instances of the search string exist, the results can vary widely — hence the need for this method.

Note

This string method has experienced numerous bugs, particularly in Navigator 2, and in later versions for UNIX. Scripts using this method should be tested exhaustively.

Examples

```
offset = "bananas".lastIndexOf("b") // result = 0 (index of first
letter is zero)
```

```
offset = "bananas".lastIndexOf ("a") // result = 5

offset = "bananas".lastIndexOf ("a",1) // result = 1 (start from
second letter working toward the front)

offset = "bananas".lastIndexOf ("a",2) // result = 1 (start from third
letter working toward front)

offset = "bananas".lastIndexOf ("a",4) // result = 3 (start from fifth
letter)

offset = "bananas".lastIndexOf ("nan") // result = 2 [ except for -1
Nav 2.0 bug]

offset = "bananas".lastIndexOf ("nas") // result = 4

offset = "bananas".lastIndexOf ("s") // result = 6

offset = "bananas".lastIndexOf ("z") // result = -1 (no "z" in string)
```

Related Items: `string.lastIndexOf()`; `string.charAt()`; `string.substring()`.

string.match(*regExpression*)

Returns: Array of matching strings.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

The `string.match()` method relies on the RegExp (regular expression) object introduced to JavaScript in Navigator 4. The string value under scrutiny is to the left of the dot, while the regular expression to be used by the method is passed as a parameter. The parameter must be a regular expression object, created according to the two ways these objects can be generated.

This method returns an array value when at least one match turns up; otherwise the returned value is null. Each entry in the array is a copy of the string segment that matches the specifications of the regular expression. You can use this method to uncover how many times a substring or sequence of characters appears in a larger string. Finding the offset locations of the matches requires other string parsing.

Example

To help you understand the `string.match()` method, Listing 26-3 provides a workshop area for experimentation. Two fields occur for data entry: the first is for the long string to be examined by the method; the second is for a regular expression. Some default values are provided in case you're not yet familiar with the syntax of regular expressions. A checkbox lets you specify whether the search through the string for matches should be case-sensitive. When you click the "Execute Match()" button, the script creates a regular expression object out of your input, performs the `string.match()` method on the big string, and reports

two kinds of results to the page. The primary result is a string version of the array returned by the method; the other is a count of items returned.

Listing 26-3: Regular Expression Match Workshop

```
<HTML>
<HEAD>
<TITLE>Regular Expression Match</TITLE>
<SCRIPT LANGUAGE="JavaScript">
function doMatch(form) {
       var str = form.entry.value
       var delim = (form.caseSens.checked) ? "/g" : "/gi"
       var regexp = eval("/" + form.regexp.value + delim)
       var resultArray = str.match(regexp)
       if (resultArray) {
          form.result.value = resultArray.toString()
          form.count.value = resultArray.length
       } else {
          form.result.value = "<no matches>"
          form.count.value = ""
       }
}
</SCRIPT>
</HEAD>
<BODY>
<B>String Match with Regular Expressions</B>
<HR>
<FORM>
Enter a main string:<INPUT TYPE="text" NAME="entry" SIZE=60
  VALUE="Many a maN and womAN have meant to visit GerMAny."><BR>
Enter a regular expression to match:<INPUT TYPE="text" NAME="regexp"
SIZE=25
  VALUE="\wa\w">
<INPUT TYPE="checkbox" NAME="caseSens">Case-sensitive<P>
<INPUT TYPE="button" VALUE="Execute Match()"
onClick="doMatch(this.form)">
<INPUT TYPE="reset"><P>
Result:<INPUT TYPE="text" NAME="result" SIZE=40><BR>
Count:<INPUT TYPE="text" NAME="count" SIZE=3><BR>
</FORM>
</BODY>
</HTML>
```

The default value for the main string has unusual capitalization intentionally. It lets you see more clearly where some of the matches come from. For example, the default regular expression looks for any three-character string that has the letter “a” in the middle. Six string segments match that expression. With the help of capitalization, you can see where each of the four strings containing “man” are extracted from the main string. The following table lists some other regular expressions to try with the default main string:

RegExp	Description
man	Both case-sensitive and not
man\b	Where "man" is at the end of a word
\bman	Where "man" is at the start of a word
me*an	Where zero or more "e" letters occur between "m" and "a"
.a.	Where "a" is surrounded by any one character (including space)
\sa\s	Where "a" is surrounded by a space on both sides
z	Where a "z" occurs (none in the default string)

In the scripts for Listing 26-3, if the `string.match()` method returns null, you are informed politely, and the count field is emptied.

Related Items: `window.setTimeout()`.

string.replace(*regExpression*, *replaceString*)

Returns: Changed string.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

Regular expressions are commonly used to perform search-and-replace operations. JavaScript's `string.replace()` method provides a simple framework in which to perform this kind of operation on any string.

Searching and replacing requires three components. The first is the main string that is the target of the operation. Second is the regular expression to search for. And third is the string to replace each instance of the text found by the operation. For the `string.replace()` method, the main string is the string value or object referenced to the left of the period. This string can also be a literal string (that is, text surrounded by quotes). The regular expression to search for is the first parameter, while the replacement string is the second parameter.

As long as you know how to generate a regular expression, you don't have to be a whiz to use the `string.replace()` method to perform simple replacement operations. But using regular expressions can make the operation more powerful. Consider these soliloquy lines by Hamlet:

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

If you wanted to replace both instances of "be" with "exist," you could do it in this case by specifying

```
var regexp = /be/
soliloquy.replace(regexp, "exist")
```

But you can't always be assured that the letters "b" and "e" will be standing alone as a word. What happens if the main string contains the word "being" or "saber"? The above example would replace the "be" letters in them as well.

The regular expression help comes from the special characters to better define what to search for. In the example here, the search is for the word "be." Therefore, the regular expression should surround the search text with word boundaries (the `\b` special character), as in

```
var regexp = /\bbe\b/
soliloquy.replace(regexp, "exist")
```

This syntax also takes care of the fact that the first two "be" words are followed by punctuation, rather than a space, as you might expect for a free-standing word. For more about regular expression syntax, see Chapter 30.

Example

The page in Listing 26-4 lets you practice with the `string.replace()` and `string.search()` methods and regular expressions in a protected environment. The source text is a five-line excerpt from *Hamlet*. You can enter the regular expression to search for as well as the replacement text. Note that the script completes the job of creating the regular expression object, so you can focus on the other special characters used to define the matching string.

Default values in the fields replace the contraction `'tis` with `it is` when you click the Execute Replace() button. As described in the section on the `string.search()` method, the button connected to that method returns the offset character number of the matching string (or `-1` if no match occurs).

Listing 26-4: Lab for string.replace() and string.search()

```
<HTML>
<HEAD>
<TITLE>Regular Expression Replace and Search</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var mainString = "To be, or not to be: that is the question:\n"
mainString += "Whether 'tis nobler in the mind to suffer\n"
mainString += "The slings and arrows of outrageous fortune,\n"
mainString += "Or to take arms against a sea of troubles,\n"
mainString += "And by opposing end them."

function doReplace(form) {
        var replaceStr = form.replaceEntry.value
        var delim = (form.caseSens.checked) ? "/g" : "/gi"
        var regexp = eval("/" + form.regexp.value + delim)
        form.result.value = mainString.replace(regexp, replaceStr)
}
function doSearch(form) {
        var replaceStr = form.replaceEntry.value
        var delim = (form.caseSens.checked) ? "/g" : "/gi"
        var regexp = eval("/" + form.regexp.value + delim)
        form.result.value = mainString.search(regexp)
}
```

```
</SCRIPT>
</HEAD>
<BODY>
<B>String Replace and Search with Regular Expressions</B>
<HR>
Text used for string.replace() and string.search() methods:<BR>
<B>To be, or not to be: that is the question:<BR>
Whether 'tis nobler in the mind to suffer<BR>
The slings and arrows of outrageous fortune,<BR>
Or to take arms against a sea of troubles,<BR>
And by opposing end them.</B>

<FORM>
Enter a regular expression to match:<INPUT TYPE="text" NAME="regexp"
SIZE=25 VALUE="\B't">
<INPUT TYPE="checkbox" NAME="caseSens">Case-sensitive<BR>
Enter a string to replace the matching strings:<INPUT TYPE="text"
NAME="replaceEntry" SIZE=30 VALUE="it "><P>
<INPUT TYPE="button" VALUE="Execute Replace()"
onClick="doReplace(this.form)">
<INPUT TYPE="reset">
<INPUT TYPE="button" VALUE="Execute Search()"
onClick="doSearch(this.form)"><P>
Result:<BR>
<TEXTAREA NAME="result" COLS=60 ROWS=5 WRAP="virtual"></TEXTAREA>
</FORM>
</BODY>
</HTML>
```

Related Items: `string.match()` method; regular expression object.

string.search(*regExpression*)

Returns: Offset Integer.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

The results of the `string.search()` method should remind you of the `string.indexOf()` method. In both cases, the returned value is the character number where the matching string first appears in the main string, or `-1` if no match occurs. The big difference, of course, is that the matching string for `string.search()` is a regular expression.

Example

Listing 26-4, in the preceding section, provides a laboratory to experiment with the `string.search()` method.

Related Items: `string.match()` method; regular expression object.

string.slice(*startIndex* [, *endIndex*])

Returns: String.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

The `string.slice()` method (new in Navigator 4) resembles the `string.substring()` method in that both let you extract a portion of one string and create a new string as a result (without modifying the original string). A helpful improvement in `string.slice()`, however, is that it is easier to specify an ending index value relative to the end of the main string.

Using `string.substring()` to extract a substring that ends before the end of the string requires machinations such as this:

```
string.substring(4, (string.length-2))
```

Instead, you can assign a negative number to the second parameter of `string.slice()` to indicate an offset from the end of the string:

```
string.slice(4, -2)
```

The second parameter is optional. If you omit it, the returned value is a string from the starting offset to the end of the main string.

Note

In Windows 95, you receive an "out of memory" error if you assign a positive integer to the second parameter that is smaller than the first parameter integer. This is a bug.

Example

With Listing 26-5, you can try several combinations of parameters with the `string.slice()` method (see Figure 26-1). A base string is provided (along with character measurements). Select from the different choices available for parameters, and study the outcome of the slice.

Listing 26-5: **Slicing a String**

```
<HTML>
<HEAD>
<TITLE>String Slicing and Dicing, Part I</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var mainString = "Electroencephalograph"
function showResults() {
        var form = document.forms[0]
        var param1 =
parseInt(form.param1.options[form.param1.selectedIndex].value)
        var param2 =
parseInt(form.param2.options[form.param2.selectedIndex].value)
```

```
        if (!param2) {
            form.result1.value = mainString.slice(param1)
        } else {
            form.result1.value = mainString.slice(param1, param2)
        }
}
</SCRIPT>
</HEAD>
<BODY onLoad="showResults()">
<B>String slice() Method</B>
<HR>
Text used for the methods:<BR>
<FONT SIZE=+1><TT><B>Electroencephalograph<BR>
----5----5----5----5-</B></TT></FONT>
<TABLE BORDER=1>
<FORM>
<TR><TH>String Method</TH><TH>Method
Parameters</TH><TH>Results</TH></TR>
<TR>
<TD>string.slice()</TD><TD ROWSPAN=3 VALIGN=middle>
( <SELECT NAME="param1" onChange="showResults()">
        <OPTION VALUE=0>0
        <OPTION VALUE=1>1
        <OPTION VALUE=2>2
        <OPTION VALUE=3>3
        <OPTION VALUE=5>5
</SELECT>,
<SELECT NAME="param2" onChange="showResults()">
        <OPTION >(None)
        <OPTION VALUE=5>5
        <OPTION VALUE=10>10
        <OPTION VALUE=-1>-1
        <OPTION VALUE=-5>-5
        <OPTION VALUE=-10>-10
</SELECT> ) </TD>
<TD><INPUT TYPE="text" NAME="result1" SIZE=25></TD>
</TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Related Items: `string.substr()` method; `string.substring()` method.

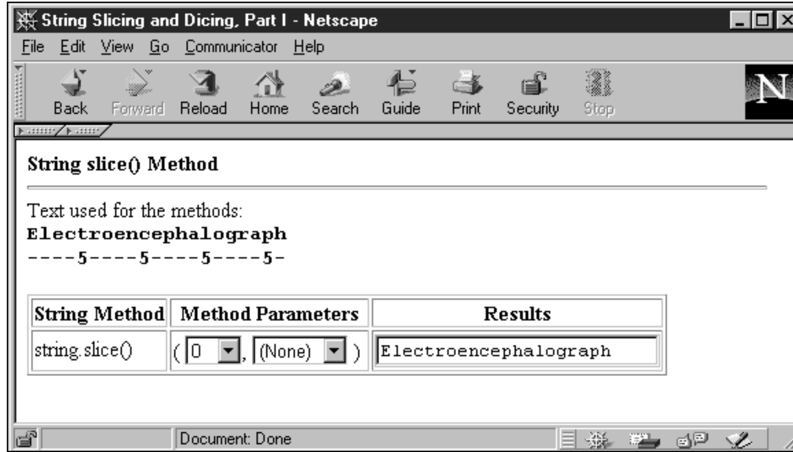


Figure 26-1: Lab for exploring the `string.slice()` method

string.split("*delimiterCharacter*" [, *limitInteger*])

Returns: Array of delimited items.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility		✔	✔			✔

The `split()` method is the functional opposite of the `array.join()` method (see Chapter 29). From the string object point of view, JavaScript splits a long string into pieces delimited by a specific character and then creates a dense array with those pieces. You do not need to initialize the array via the `new Array()` constructor. Given the powers of array object methods such as `array.sort()`, you may want to convert a series of string items to an array to take advantage of those powers. Also, if your goal is to divide a string into an array of single characters, you can still use the `split()` method, but specify an empty string as a parameter. For Navigator 3 and Internet Explorer 4, only the first parameter is observed.

Navigator 4 loads additional functionality onto the `string.split()` method. For one, you can use a regular expression object for the first parameter, enhancing the powers of finding delimiters in strings. For example, consider the following string:

```
var nameList  = "1.Fred,2.Jane,3.Steve"
```

To convert that string into a three-element array of only the names would take a lot of parsing without regular expressions before you could even use `string.split()`. However, with a regular expression as a parameter,

```
var regexp = /,*\d.\b/
var newArray = nameList.split(regexp)
      // result = an array "Fred", "Jane", "Steve"
```

the new array entries hold only the names and not the leading numbers or periods. A second addition is an optional second parameter. This integer value allows you to specify a limit to the number of array elements generated by the method.

And finally, if you use the `string.split()` method inside a `<SCRIPT LANGUAGE="JavaScript1.2">` tag, an empty space as a single parameter, such as `string.split(" ")`, is interpreted to mean any white space (spaces, tabs, carriage returns, line feeds) between runs of characters. Even if the number of spaces between elements is not uniform, they are treated all the same.

Examples

```
var myString = "Anderson,Smith,Johnson,Washington"
var myArray = myString.split(",")
var itemCount = myArray.length  // result: 4
```

Related Items: `Array.join()`.

string.substr(*start* [, *length*])

Returns: Nothing.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility			✔			✔

Navigator 4's new `string.substr()` method offers a variation of the `string.substring()` method that has been in the language since the beginning. The distinction is that the `string.substr()` method's parameters specify the starting index and a number of characters to be included from that start point. In contrast, the `string.substring()` method parameters specify index points for the start and end characters within the main string.

As with all string methods requiring an index value, the `string.substr()` first parameter is zero-based. If you do not specify a second parameter, the returned substring starts at the indexed point and extends to the end of the string. A second parameter value that exceeds the end point of the string means that the method returns a substring to the end of the string.

Note

Macintosh users should avoid setting the second parameter to a negative number, to prevent a crash.

Example

Listing 26-6 lets you experiment with a variety of values to see how the `string.substr()` method works.

Listing 26-6: **Extracting from a String**

```
<HTML>
<HEAD>
<TITLE>String Slicing and Dicing, Part II</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var mainString = "Electroencephalograph"
function showResults() {
       var form = document.forms[0]
       var param1 =
parseInt(form.param1.options[form.param1.selectedIndex].value)
       var param2 =
parseInt(form.param2.options[form.param2.selectedIndex].value)
       if (!param2) {
          form.result1.value = mainString.substr(param1)
       } else {
          form.result1.value = mainString.substr(param1, param2)
       }
}
</SCRIPT>
</HEAD>
<BODY onLoad="showResults()">
<B>String substr() Method</B>
<HR>
Text used for the methods:<BR>
<FONT SIZE=+1><TT><B>Electroencephalograph<BR>
----5----5----5----5-</B></TT></FONT>
<TABLE BORDER=1>
<FORM>
<TR><TH>String Method</TH><TH>Method
Parameters</TH><TH>Results</TH></TR>
<TR>
<TD>string.substr()</TD><TD ROWSPAN=3 VALIGN=middle>
( <SELECT NAME="param1" onChange="showResults()">
       <OPTION VALUE=0>0
       <OPTION VALUE=1>1
       <OPTION VALUE=2>2
       <OPTION VALUE=3>3
       <OPTION VALUE=5>5
</SELECT>,
<SELECT NAME="param2" onChange="showResults()">
       <OPTION >(None)
       <OPTION VALUE=5>5
       <OPTION VALUE=10>10
       <OPTION VALUE=20>20
</SELECT> ) </TD>
<TD><INPUT TYPE="text" NAME="result1" SIZE=25></TD>
</TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Related Items: `string.substring()` method.

string.substring(*indexA*, *indexB*)

Returns: Characters of *string* between index values *indexA* and *indexB*.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

The `string.substring()` method enables your scripts to extract a contiguous range of characters from any string. The parameters to this method are the starting and ending index values (first character of the string object is index value 0) of the main string from which the excerpt should be taken. An important item to note is that the excerpt goes up to, ***but does not include***, the character pointed to by the higher index value.

It makes no difference which index value in the parameters is larger than the other: The method starts the excerpt from the lowest value and continues to (but does not include) the highest value. If both index values are the same, the method returns an empty string; and if you omit the second parameter, the end of the string is assumed to be the endpoint.

Behavior of this method in Navigator 2 and 3 called for parameter values that were switched around (that is, the second value was smaller than the first) to be automatically reversed by JavaScript. If you use this method in a `<SCRIPT LANGUAGE="JavaScript1.2">` tag, the values are not automatically switched.

Example

Listing 26-7 lets you experiment with a variety of values to see how the `string.substring()` method works. If you are using Navigator 4, try changing the `LANGUAGE` attribute of the script to `JavaScript1.2` and see the different behavior when you set the parameters to 5 and 3. The parameters switch themselves, essentially letting the second index value become the beginning of the extracted substring.

Listing 26-7: Extracting from a String

```
<HTML>
<HEAD>
<TITLE>String Slicing and Dicing, Part II</TITLE>
<SCRIPT LANGUAGE="JavaScript">
var mainString = "Electroencephalograph"
function showResults() {
        var form = document.forms[0]
        var param1 =
parseInt(form.param1.options[form.param1.selectedIndex].value)
        var param2 =
parseInt(form.param2.options[form.param2.selectedIndex].value)
```

(continued)

Listing 26-7 *(continued)*

```
        if (!param2) {
           form.result1.value = mainString.substring(param1)
        } else {
           form.result1.value = mainString.substring(param1, param2)
        }
}
</SCRIPT>
</HEAD>
<BODY onLoad="showResults()">
<B>String substr() Method</B>
<HR>
Text used for the methods:<BR>
<FONT SIZE=+1><TT><B>Electroencephalograph<BR>
----5----5----5----5-</B></TT></FONT>
<TABLE BORDER=1>
<FORM>
<TR><TH>String Method</TH><TH>Method
Parameters</TH><TH>Results</TH></TR>
<TR>
<TD>string.substring()</TD><TD>
( <SELECT NAME="param1" onChange="showResults()">
        <OPTION VALUE=0>0
        <OPTION VALUE=1>1
        <OPTION VALUE=2>2
        <OPTION VALUE=3>3
        <OPTION VALUE=5>5
</SELECT>,
<SELECT NAME="param2" onChange="showResults()">
        <OPTION >(None)
        <OPTION VALUE=3>3
        <OPTION VALUE=5>5
        <OPTION VALUE=10>10
</SELECT> ) </TD>
<TD><INPUT TYPE="text" NAME="result1" SIZE=25></TD>
</TR>
</TABLE></FORM>
</BODY>
</HTML>
```

Related Items: `string.substr()` **method;** `string.slice()` **method.**

string.toLowerCase()
string.toUpperCase()

Returns: The string in all lower- or uppercase, depending on which method you invoke.

	Nav2	Nav3	Nav4	IE3/J1	IE3/J2	IE4/J3
Compatibility	✔	✔	✔	✔	✔	✔

A great deal of what takes place on the Internet (and in JavaScript) is case-sensitive. URLs on some servers, for instance, are case-sensitive for directory names and filenames. These two methods, the simplest of the string methods, convert any string to either all lowercase or all uppercase. Any mixed-case strings get converted to a uniform case. If you want to compare user input from a field against some coded string without worrying about matching case, you should convert both strings to the same case for the comparison.

Examples

```
newString = "HTTP://www.Netscape.COM".toLowerCase()
       // result = "http://www.netscape.com"

if (guess.toUpperCase() == answer.toUpperCase()) {...
       // comparing strings without case sensitivity
```

Related Items: None.

String utility functions

Figuring out how to apply the various string object methods to a string manipulation challenge is not always an easy task, especially if you need backward compatibility to older scriptable browsers. I also find it difficult to anticipate every possible way you may need to massage strings in your scripts. But to help you get started, Listing 26-8 contains a library of string functions for inserting, deleting, and replacing chunks of text in a string. If your audience uses browsers capable of including external .js library files, that would be an excellent way to make these functions available to your scripts.

Listing 26-8: Utility String Handlers

```
// extract front part of string prior to searchString
function getFront(mainStr,searchStr){
       foundOffset = mainStr.indexOf(searchStr)
       if (foundOffset == -1) {
          return null
       }
       return mainStr.substring(0,foundOffset)
}

// extract back end of string after searchString
function getEnd(mainStr,searchStr) {
       foundOffset = mainStr.indexOf(searchStr)
       if (foundOffset == -1) {
          return null
       }
```

(continued)

Listing 26-8 *(continued)*

```
        return
mainStr.substring(foundOffset+searchStr.length,mainStr.length)
}

// insert insertString immediately before searchString
function insertString(mainStr,searchStr,insertStr) {
        var front = getFront(mainStr,searchStr)
        var end = getEnd(mainStr,searchStr)
        if (front != null && end != null) {
           return front + insertStr + searchStr + end
        }
        return null
}

// remove deleteString
function deleteString(mainStr,deleteStr) {
        return replaceString(mainStr,deleteStr,"")
}

// replace searchString with replaceString
function replaceString(mainStr,searchStr,replaceStr) {
        var front = getFront(mainStr,searchStr)
        var end = getEnd(mainStr,searchStr)
        if (front != null && end != null) {
           return front + replaceStr + end
        }
        return null
}
```

The first two functions extract the front or end components of strings as needed for some of the other functions in this suite. The final three functions are the core of these string-handling functions. If you plan to use these functions in your scripts, be sure to notice the dependence that some functions have on others. Including all five functions as a group ensures that they work as designed.

Formatting methods

Now we come to the other group of string object methods, which ease the process of creating the numerous string display characteristics when you use JavaScript to assemble HTML code. The following is a list of these methods:

```
string.anchor("anchorName")
string.blink()
string.bold()
string.fixed()
string.fontcolor (colorValue)
string.fontsize(integer1to7)
string.italics()
string.link(locationOrURL)
string.big()
string.small()
string.strike()
string.sub()
string.sup()
```

Let's first examine the methods that don't require any parameters. You probably see a pattern: All of these methods are font-style attributes that have settings of on or off. To turn on these attributes in an HTML document, you surround the text in the appropriate tag pairs, such as `<B>…</B>` for boldface text. These methods take the string object, attach those tags, and return the resulting text, which is ready to be put into any HTML that your scripts are building. Therefore, the expression

```
"Good morning!".bold()
```

evaluates to

```
<B>Good morning!</B>
```

Of course, nothing is preventing you from building your HTML by embedding real tags instead of by calling the string methods. The choice is up to you. One advantage to the string methods is that they never forget the ending tag of a tag pair. Listing 26-9 shows an example of incorporating a few simple string methods in a string variable that is eventually written to the page as it loads. Internet Explorer does not support the `<BLINK>` tag, and therefore ignores the `string.blink()` method.

Listing 26-9: **Using Simple String Methods**

```
<HTML>
<HEAD>
<TITLE>HTML by JavaScript</TITLE>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JavaScript">
var page = ""
page += "JavaScript can create HTML on the fly.<P>Numerous string
object methods facilitate creating text that is " + "boldfaced".bold()
+ ", " + "italicized".italics() + ", or even the terribly annoying " +
"blinking text".blink() + "."
document.write(page)
</SCRIPT>
</BODY>
</HTML>
```

Of the remaining string methods, two more (`string.fontsize()` and `string.fontcolor()`) also affect the font characteristics of strings displayed in the HTML page. The parameters for these items are pretty straightforward — an integer between 1 and 7 corresponding to the seven browser font sizes and a color value (as either a hexadecimal triplet or color constant name) for the designated text. Listing 26-10 adds a line of text to the string of Listing 26-9. This line of text not only adjusts the font size of some parts of the string, but also nests multiple attributes inside one another to set the color of one word in a large-font-size string. Because these string methods do not change the content of the string, you can safely nest methods here.

Listing 26-10: Nested String Methods

```
<HTML>
<HEAD>
<TITLE>HTML by JavaScript</TITLE>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JavaScript">
var page = ""
page += "JavaScript can create HTML on the fly.<P>Numerous string
object methods facilitate creating text that is " + "boldfaced".bold()
+ ", " + "italicized".italics() + ", or even the terribly annoying " +
"blinking text".blink() + ".<P>"
page += "We can make " + "some words big".fontsize(5) + " and some
words both " + ("big and " + "colorful".fontcolor('coral')).fontsize(5)
+ " at the same time."
document.write(page)
</SCRIPT>
</BODY>
</HTML>
```

The final two string methods let you create an anchor and a link out of a string. The `string.anchor()` method uses its parameter to create a name for the anchor. Thus, the following expression

```
"Table of Contents".anchor("toc")
```

evaluates to

```
<A NAME="toc">Table of Contents</A>
```

In a similar fashion, the `string.link()` method expects a valid location or URL as its parameter, creating a genuine HTML link out of the string:

```
"Back to Home".link("index.html")
```

This evaluates to the following:

```
<A HREF="index.html">Back to Home</A>
```

Again, the choice of whether you use string methods to build HTML anchors and links over assembling the actual HTML is up to you. The methods may be a bit easier to work with if the values for the string and the parameters are variables whose content may change based on user input elsewhere in your Web site.

URL String Encoding and Decoding

When browsers and servers communicate, some nonalphanumeric characters that we take for granted (such as a space) cannot make the journey in their native form. Only a narrower set of letters, numbers, and punctuation is allowed. To accommodate the rest, the characters must be encoded with a special symbol (%)

and their hexadecimal ASCII values. For example, the space character is hex 20 (ASCII decimal 32). When encoded, it looks like `%20`. You may have seen this symbol in browser history lists or URLs.

JavaScript includes two functions, `escape()` and `unescape()`, that offer instant conversion of whole strings. To convert a plain string to one with these escape codes, use the escape function, as in

```
escape("Howdy Pardner") // result = "Howdy%20Pardner"
```

The `unescape()` function converts the escape codes into human-readable form.