CHAPTER 38

# The Regular Expression and RegExp Objects

**In This Chapter**

What regular expressions are

How to use regular expressions for text search-and-replace

How to apply regular expressions to string object methods

Web programmers who have worked in Perl (and other Web application programming languages) know the power of regular expressions for processing incoming data and formatting data for readability in an HTML page or for accurate storage in a server database. Any task that requires extensive search and replacement of text can greatly benefit from the flexibility and conciseness of regular expressions. Navigator 4 and Internet Explorer 4 (more fully fleshed out in IE5.5) bring that power to JavaScript.

Most of the benefit of JavaScript regular expressions accrues to those who script their CGI programs on servers that support a JavaScript version that contains regular expressions. But that's not to exclude the client-side from application of this "language within a language." If your scripts perform client-side data validations or any other extensive text entry parsing, then consider using regular expressions, rather than cobbling together comparatively complex JavaScript functions to perform the same tasks.

## Regular Expressions and Patterns

In several chapters earlier in this book, I describe expressions as any sequence of identifiers, keywords, and/or operators that evaluate to some value. A regular expression follows that description, but has much more power behind it. In essence, a regular expression uses a sequence of characters and symbols to define a pattern of text. Such a pattern is used to locate a chunk of text in a string by matching up the pattern against the characters in the string.

An experienced JavaScript writer may point out the availability of the `*string*.indexOf()` and `*string*.lastIndexOf()`

methods that can instantly reveal whether a string contains a substring and even where in the string that substring begins. These methods work perfectly well when the match is exact, character for character. But if you want to do more sophisticated matching (for example, does the string contain a five-digit ZIP code?), you'd have to cast aside those handy string methods and write some parsing functions. That's the beauty of a regular expression: It lets you define a matching substring that has some intelligence about it and can follow guidelines you set as to what should or should not match.

The simplest kind of regular expression pattern is the same kind you use in the `string.indexOf()` method. Such a pattern is nothing more than the text that you want to match. In JavaScript, one way to create a regular expression is to surround the expression by forward slashes. For example, consider the string

```
Oh, hello, do you want to play Othello in the school play?
```

This string and others may be examined by a script whose job it is to turn formal terms into informal ones. Therefore, one of its tasks is to replace the word "hello" with "hi." A typical brute force search-and-replace function starts with a simple pattern of the search string. In JavaScript, you define a pattern (a regular expression) by surrounding it with forward slashes. For convenience and readability, I usually assign the regular expression to a variable, as in the following example:

```
var myRegExpression = /hello/
```

In concert with some regular expression or string object methods, this pattern matches the string "hello" wherever that series of letters appears. The problem is that this simple pattern causes problems during the loop that searches and replaces the strings in the example string: It finds not only the standalone word "hello," but also the "hello" in "Othello."

Trying to write another brute force routine for this search-and-replace operation that looks only for standalone words would be a nightmare. You can't merely extend the simple pattern to include spaces on either or both sides of "hello," because there could be punctuation — a comma, a dash, a colon, or whatever — before or after the letters. Fortunately, regular expressions provide a shortcut way to specify general characteristics, including a feature known as a word boundary. The symbol for a word boundary is `\b` (backslash, lowercase b). If you redefine the pattern to include these specifications on both ends of the text to match, the regular expression creation statement looks like

```
var myRegExpression = /\bhello\b/
```

When JavaScript uses this regular expression as a parameter in a special string object method that performs search-and-replace operations, it changes only the standalone word "hello" to "hi," and passes over "Othello" entirely.

If you are still learning JavaScript and don't have experience with regular expressions in other languages, you have a price to pay for this power: Learning the regular expression lingo filled with so many symbols means that expressions sometimes look like cartoon substitutions for swear words. The goal of this chapter is to introduce you to regular expression syntax as implemented in JavaScript rather than engage in lengthy tutorials for this language. Of more importance in the long run is understanding how JavaScript treats regular expressions as objects and distinctions between instances of regular expression objects and the `RegExp` static object. I hope the examples in the following sections begin to reveal the powers of regular expressions. An in-depth treatment of the possibilities and idiosyncrasies of regular expressions can be found in *Mastering Regular Expressions* by Jeffrey E.F. Friedl (1997, O'Reilly & Associates, Inc.).

# Language Basics

To cover the depth of the regular expression syntax, I divide the subject into three sections. The first covers simple expressions (some of which you've already seen). Then I get into the wide range of special characters used to define specifications for search strings. Last comes an introduction to the usage of parentheses in the language, and how they not only help in grouping expressions for influencing calculation precedence (as they do for regular math expressions), but also how they temporarily store intermediate results of more complex expressions for use in reconstructing strings after their dissection by the regular expression.

## Simple patterns

A simple regular expression uses no special characters for defining the string to be used in a search. Therefore, if you wanted to replace every space in a string with an underscore character, the simple pattern to match the space character is

```
var re = / /
```

A space appears between the regular expression start-end forward slashes. The problem with this expression, however, is that it knows only how to find a single instance of a space in a long string. Regular expressions can be instructed to apply the matching string on a global basis by appending the `g` modifier:

```
var re = / /g
```

When this `re` value is supplied as a parameter to the `replace()` method that uses regular expressions (described later in this chapter), the replacement is performed throughout the entire string, rather than just once on the first match found. Notice that the modifier appears after the final forward slash of the regular expression creation statement.

Regular expression matching — like a lot of other aspects of JavaScript — is case-sensitive. But you can override this behavior by using one other modifier that lets you specify a case-insensitive match. Therefore, the following expression

```
var re = /web/i
```

finds a match for “web,” “Web,” or any combination of uppercase and lowercase letters in the word. You can combine the two modifiers together at the end of a regular expression. For example, the following expression is both case-insensitive and global in scope:

```
var re = /web/gi
```

In compliance with the ECMA-262 Edition 3 standard, IE5.5 and NN6 also allow a flag to force the regular expression to operate across multiple lines (meaning a carriage-return-delimited string) of a larger string. That modifier is the letter `m`.

## Special characters

The regular expression in JavaScript borrows most of its vocabulary from the Perl regular expression. In a few instances, JavaScript offers alternatives to simplify the syntax, but also accepts the Perl version for those with experience in that arena.

Significant programming power comes from the way regular expressions allow you to include terse specifications about such facets as types of characters to accept in a match, how the characters are surrounded within a string, and how often a type of character can appear in the matching string. A series of escaped one-character commands (that is, letters preceded by the backslash) handle most of the character issues; punctuation and grouping symbols help define issues of frequency and range.

You saw an example earlier how `\b` specified a word boundary on one side of a search string. Table 38-1 lists the escaped character specifiers in JavaScript regular expressions. The vocabulary forms part of what are known as metacharacters — characters in expressions that are not matchable characters themselves, but act more as commands or guidelines of the regular expression language.

**Table 38-1 JavaScript Regular Expression Matching Metacharacters**

| *Character* | *Matches* | *Example* |
|---|---|---|
| `\b` | Word boundary | `/\bor/` matches “origami” and “or” but not “normal” |
| | | `/or\b/` matches “traitor” and “or” but not “perform” |
| | | `/\bor\b/` matches full word “or” and nothing else |
| `\B` | Word non-boundary | `/\Bor/` matches “normal” but not “origami” |
| | | `/or\B/` matches “normal” and “origami” but not “traitor” |
| | | `/\Bor\B/` matches “normal” but not “origami” or “traitor” |

| *Character* | *Matches* | *Example* |
|---|---|---|
| `\d` | Numeral 0 through 9 | `/\d\d\d/` matches "212" and "415" but not "B17" |
| `\D` | Non-numeral | `/\D\D\D/` matches "ABC" but not "212" or "B17" |
| `\s` | Single white space | `/over\sbite/` matches "over bite" but not "overbite" or "over   bite" |
| `\S` | Single non-white space | `/over\Sbite/` matches "over-bite" but not "overbite" or "over bite" |
| `\w` | Letter, numeral, or underscore | `/A\w/` matches "A1" and "AA" but not "A+" |
| `\W` | Not letter, numeral, or underscore | `/A\W/` matches "A+" but not "A1" and "AA" |
| `.` | Any character except newline | `/.../` matches "ABC", "1+3", "A 3", or any three characters |
| `[...]` | Character set | `/[AN]BC/` matches "ABC" and "NBC" but not "BBC" |
| `[^...]` | Negated character set | `/[^AN]BC/` matches "BBC" and "CBC" but not "ABC" or "NBC" |

Not to be confused with the metacharacters listed in Table 38-1 are the escaped string characters for tab (`\t`), newline (`\n`), carriage return (`\r`), formfeed (`\f`), and vertical tab (`\v`).

Let me further clarify about the `[...]` and `[^...]` metacharacters. You can specify either individual characters between the brackets (as shown in Table 38-1) or a contiguous range of characters or both. For example, the `\d` metacharacter can also be defined by `[0-9]`, meaning any numeral from zero through nine. If you only want to accept a value of 2 and a range from 6 through 8, the specification would be `[26-8]`. Similarly, the accommodating `\w` metacharacter is defined as `[A-Za-z0-9_]`, reminding you of the case-sensitivity of regular expression matches not otherwise modified.

All but the bracketed character set items listed in Table 38-1 apply to a single character in the regular expression. In most cases, however, you cannot predict how incoming data will be formatted — the length of a word or the number of digits in a number. A batch of extra metacharacters lets you set the frequency of the occurrence of either a specific character or a type of character (specified like the ones in Table 38-1). If you have experience in command-line operating systems, you can see some of the same ideas that apply to wildcards also apply to regular expressions. Table 38-2 lists the counting metacharacters in JavaScript regular expressions.

**Table 38-2 JavaScript Regular Expression Counting Metacharacters**

| *Character* | *Matches Last Character* | *Example* |
|---|---|---|
| `*` | Zero or more times | `/Ja*vaScript/` matches "JvaScript", "JavaScript", and "JaaavaScript" but not "JovaScript" |
| `?` | Zero or one time | `/Ja?vaScript/` matches "JvaScript" or "JavaScript" but not "JaaavaScript" |
| `+` | One or more times | `/Ja+vaScript/` matches "JavaScript" or "JaavaScript" but not "JvaScript" |
| `{n}` | Exactly n times | `/Ja{2}vaScript/` matches "JaavaScript" but not "JvaScript" or "JavaScript" |
| `{n,}` | n or more times | `/Ja{2,}vaScript/` matches "JaavaScript" or "JaaavaScript" but not "JavaScript" |
| `{n,m}` | At least n, at most m times | `/Ja{2,3}vaScript/` matches "JaavaScript" or "JaaavaScript" but not "JavaScript" |

Every metacharacter in Table 38-2 applies to the character immediately preceding it in the regular expression. Preceding characters may also be matching metacharacters from Table 38-1. For example, a match occurs for the following expression if the string contains two digits separated by one or more vowels:

```
/\d[aeiouy]+\d/
```

The last major contribution of metacharacters is helping the regular expression search a particular position in a string. By position, I don't mean something such as an offset — the matching functionality of regular expressions can tell me that. But, rather, whether the string to look for should be at the beginning or end of a line (if that is important) or whatever string is offered as the main string to search. Table 38-3 shows the positional metacharacters for JavaScript's regular expressions.

**Table 38-3 JavaScript Regular Expression Positional Metacharacters**

| *Character* | *Matches Located* | *Example* |
|---|---|---|
| `^` | At beginning of a string or line | `/^Fred/` matches "Fred is OK" but not "I'm with Fred" or "Is Fred here?" |
| `$` | At end of a string or line | `/Fred$/` matches "I'm with Fred" but not "Fred is OK" or "Is Fred here?" |

For example, you may want to make sure that a match for a roman numeral is found only when it is at the start of a line, rather than when it is used inline somewhere else. If the document contains roman numerals in an outline, you can match all the top-level items that are flush left with the document with a regular expression, such as the following:

```
/^[IVXMDCL]+\./
```

This expression matches any combination of roman numeral characters followed by a period (the period is a special character in regular expressions, as shown in Table 38-1, so that you have to escape the period to offer it as a character), provided the roman numeral is at the beginning of a line and has no tabs or spaces before it. There would also not be a match in a line that contains, for example, the phrase "see Part IV" because the roman numeral is not at the beginning of a line.

Speaking of lines, a line of text is a contiguous string of characters delimited by a newline and/or carriage return (depending on the operating system platform). Word wrapping in TEXTAREA elements does not affect the starts and ends of true lines of text.

## Grouping and backreferencing

Regular expressions obey most of the JavaScript operator precedence laws with regard to grouping by parentheses and the logical Or operator. One difference is that the regular expression Or operator is a single pipe character (|) rather than JavaScript's double pipe.

Parentheses have additional powers that go beyond influencing the precedence of calculation. Any set of parentheses (that is, a matched pair of left and right) stores the results of a found match of the expression within those parentheses. Parentheses can be nested inside one another. Storage is accomplished automatically, with the data stored in an indexed array accessible to your scripts and to your regular expressions (although through different syntax). Access to these storage bins is known as *backreferencing,* because a regular expression can point backward to the result of an expression component earlier in the overall expression. These stored subcomponents come in handy for replace operations, as demonstrated later in this chapter.

# Object Relationships

JavaScript has a lot going on behind the scenes when you create a regular expression and perform the simplest operation with it. As important as the regular expression language described earlier in this chapter is to applying regular expressions in your scripts, the JavaScript object interrelationships are perhaps even more important if you want to exploit regular expressions to the fullest.

The first concept to master is that two entities are involved: a regular expression instance object and the `RegExp` static object. Both objects are core objects of JavaScript and are not part of the document object model. Both objects work together, but have entirely different sets of properties that may be useful to your application.

When you create a regular expression (even via the `/.../` syntax), JavaScript invokes the `new RegExp()` constructor, much the way a `new Date()` constructor creates a date object around one specific date. The regular expression instance object returned by the constructor is endowed with several properties containing details of its data. At the same time, the single, static `RegExp` object maintains its own properties that monitor regular expression activity in the current window (or frame).

To help you see the typically unseen operations, I step you through the creation and application of a regular expression. In the process, I show you what happens to all of the related object properties when you use one of the regular expression methods to search for a match.

**Note** Several properties of both the regular expression instance object and the static `RegExp` object shown in the following "walk-through" are not available in IE until version 5.5. All are available in NN4+. See the individual property listings later in this chapter for compatibility ratings.

The starting text that I use to search through is the beginning of Hamlet's soliloquy (assigned to an arbitrary variable named `mainString`):

```
var mainString = "To be, or not to be: That is the question:"
```

If my ultimate goal is to locate each instance of the word "be," I must first create a regular expression that matches the word "be." I set the regular expression up to perform a global search when eventually called upon to replace itself (assigning the expression to an arbitrary variable named `re`):

```
var re = /\bbe\b/g
```

To guarantee that only complete words "be" are matched, I surround the letters with the word boundary metacharacters. The final "g" is the global modifier. The variable to which the expression is assigned, `re`, represents a regular expression object whose properties and values are as follows:

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `re.source` | `"\bbe\bg"` |
| `re.global` | `true` |
| `re.ignoreCase` | `false` |
| `re.lastIndex` | `0` |

A regular expression's `source` property is the string consisting of the regular expression syntax (less the literal forward slashes). Each of the two possible modifiers, `g` and `i`, have their own properties, `global` and `ignoreCase`, whose values are Booleans indicating whether the modifiers are part of the source expression. The final property, `lastIndex`, indicates the index value within the main string at which the next search for a match should start. The default value for this property in a newly hatched regular expression is zero so that the search starts with the first character of the string. This property is read/write, so your scripts may want to adjust the value if they must have special control over the search process. As you see in a moment, JavaScript modifies this value over time if a global search is indicated for the object.

The `RegExp` constructor does more than just create regular expression objects. Like the `Math` object, the `RegExp` object is always "around"—one `RegExp` per window or frame—and tracks regular expression activity in a script. Its properties reveal what, if any, regular expression pattern matching has just taken place in the window. At this stage of the regular expression creation process, the `RegExp` object has only one of its properties set:

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `RegExp.input` | |
| `RegExp.multiline` | `false` |
| `RegExp.lastMatch` | |
| `RegExp.lastParen` | |
| `RegExp.leftContext` | |
| `RegExp.rightContext` | |
| `RegExp.$1` | |
| `...` | |
| `RegExp.$9` | |

The last group of properties (`$1` through `$9`) is for storage of backreferences. But because the regular expression I define above doesn't have any parentheses in it, these properties are empty for the duration of this examination and omitted from future listings in this "walk-through" section.

With the regular expression object ready to go, I invoke the `exec()` regular expression method, which looks through a string for a match defined by the regular expression. If the method is successful in finding a match, it returns a third object whose properties reveal a great deal about the item it found (I arbitrarily assign the variable `foundArray` to this returned object):

```
var foundArray = re.exec(mainString)
```

JavaScript includes a shortcut for the `exec()` method if you turn the regular expression object into a method:

```
var foundArray = re(mainString)
```

Normally, a script would check whether `foundArray` is `null` (meaning that there was no match) before proceeding to inspect the rest of the related objects. Because this is a controlled experiment, I know at least one match exists, so I first look into some other results. Running this simple method has not only generated the `foundArray` data, but also altered several properties of the `RegExp` and regular expression objects. The following shows you the current stage of the regular expression object:

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `re.source` | `"\bbe\bg"` |
| `re.global` | `true` |
| `re.ignoreCase` | `false` |
| `re.lastIndex` | `5` |

The only change is an important one: The `lastIndex` value has bumped up to `5`. In other words, this one invocation of the `exec()` method must have found a match whose offset plus length of matching string shifts the starting point of any successive searches with this regular expression to character index 5. That's exactly where the comma after the first "be" word is in the main string. If the global (`g`) modifier had not been appended to the regular expression, the `lastIndex` value would have remained at zero, because no subsequent search would be anticipated.

As the result of the `exec()` method, the `RegExp` object has had a number of its properties filled with results of the search:

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `RegExp.input` | |
| `RegExp.multiline` | `false` |
| `RegExp.lastMatch` | `"be"` |
| `RegExp.lastParen` | |
| `RegExp.leftContext` | `"To "` |
| `RegExp.rightContext` | `", or not to be: That is the question:"` |

From this object you can extract the string segment that was found to match the regular expression definition. The main string segments before and after the matching text are also available individually (in this example, the `leftContext` property has a space after "To"). Finally, looking into the array returned from the `exec()` method, some additional data is readily accessible:

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `foundArray[0]` | `"be"` |
| `foundArray.index` | `3` |
| `foundArray.input` | `"To be, or not to be: That is the question:"` |

The first element in the array, indexed as the zeroth element, is the string segment found to match the regular expression, which is the same as the `RegExp.lastMatch` value. The complete main string value is available as the input property. A potentially valuable piece of information to a script is the index for the start of the matched string found in the main string. From this last bit of data, you can extract from the found data array the same values as `RegExp.leftContext` (with `foundArray.input.substring(0, foundArray.index)`) and `RegExp. rightContext` (with `foundArray.input.substring(foundArray.index, foundArray[0].length)`).

Because the regular expression suggested a multiple execution sequence to fulfill the global flag, I can run the `exec()` method again without any change. While the JavaScript statement may not be any different, the search starts from the new `re.lastIndex` value. The effects of this second time through ripple through the resulting values of all three objects associated with this method:

```
var foundArray = re.exec(mainString)
```

Results of this execution are as follows (changes are in boldface).

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `re.source` | `"\bbe\bg"` |
| `re.global` | `true` |
| `re.ignoreCase` | `false` |
| `re.lastIndex` | **19** |
| `RegExp.input` | |
| `RegExp.multiline` | `false` |

*Continued*

| ***Object.PropertyName*** | ***Value*** |
|---|---|
| `RegExp.lastMatch` | `"be"` |
| `RegExp.lastParen` | |
| `RegExp.leftContext` | `", or not to "` |
| `RegExp.rightContext` | `": That is the question:"` |
| `foundArray[0]` | `"be"` |
| `foundArray.index` | `17` |
| `foundArray.input` | `"To be, or not to be: That is the question:"` |

Because there was a second match, `foundArray` comes back again with data. Its index property now points to the location of the second instance of the string matching the regular expression definition. The regular expression object's `lastIndex` value points to where the next search would begin (after the second "be"). And the `RegExp` properties that store the left and right contexts have adjusted accordingly.

If the regular expression were looking for something less stringent than a hard-coded word, some other properties may also be different. For example, if the regular expression defined a format for a ZIP code, the `RegExp.lastMatch` and `foundArray[0]` values would contain the actual found ZIP codes, which would likely be different from one match to the next.

Running the same `exec()` method once more does not find a third match in my original `mainString` value, but the impact of that lack of a match is worth noting. First of all, the `foundArray` value is `null`—a signal to our script that no more matches are available. The regular expression object's `lastIndex` property reverts to zero, ready to start its search from the beginning of another string. Most importantly, however, the `RegExp` object's properties maintain the same values from the last successful match. Therefore, if you put the `exec()` method invocations in a repeat loop that exits after no more matches are found, the `RegExp` object still has the data from the last successful match, ready for further processing by your scripts.

# Using Regular Expressions

Despite the seemingly complex hidden workings of regular expressions, JavaScript provides a series of methods that make common tasks involving regular expressions quite simple to use (assuming you figure out the regular expression syntax to create good specifications). In this section, I present examples of syntax for specific kinds of tasks for which regular expressions can be beneficial in your pages.

## Is there a match?

I said earlier that you can use `string.indexOf()` or `string.lastIndexOf()` to look for the presence of simple substrings within larger strings. But if you need the matching power of regular expression, you have two other methods to choose from:

```
regexObject.test(string)
string.search(regexObject)
```

The first is a regular expression object method, the second a string object method. Both perform the same task and influence the same related objects, but they return different values: a Boolean value for `test()` and a character offset value for `search()` (or `-1` if no match is found). Which method you choose depends on whether you need only a true/false verdict on a match or the location within the main string of the start of the substring.

Listing 38-1 demonstrates the `search()` method on a page that lets you get the Boolean and offset values for a match. Some default text and regular expression is provided (it looks for a five-digit number). You can experiment with other strings and regular expressions. Because this script creates a regular expression object with the new `RegExp()` constructor method, you do not include the literal forward slashes around the regular expression.

### Listing 38-1: **Looking for a Match**

```
<HTML>
<HEAD>
<TITLE>Got a Match?</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function findIt(form) {
    var re = new RegExp(form.regexp.value)
    var input = form.main.value
    if (input.search(re) != -1) {
        form.output[0].checked = true
    } else {
        form.output[1].checked = true
    }
}
function locateIt(form) {
    var re = new RegExp(form.regexp.value)
    var input = form.main.value
    form.offset.value = input.search(re)
}
</SCRIPT>
</HEAD>
<BODY>
<B>Use a regular expression to test for the existence of a string:</B>
<HR>
```

*Continued*

Listing 38-1 *(continued)*

```
<FORM>
Enter some text to be searched:<BR>
<TEXTAREA NAME="main" COLS=40 ROWS=4 WRAP="virtual">
The most famous ZIP code on Earth may be 90210.
</TEXTAREA><BR>
Enter a regular expression to search:<BR>
<INPUT TYPE="text" NAME="regexp" SIZE=30 VALUE="\b\d\d\d\d\d\b"><P>
<INPUT TYPE="button" VALUE="Is There a Match?" onClick="findIt(this.form)">
<INPUT TYPE="radio" NAME="output">Yes
<INPUT TYPE="radio" NAME="output">No <P>
<INPUT TYPE="button" VALUE="Where is it?" onClick="locateIt(this.form)">
<INPUT TYPE="text" NAME="offset" SIZE=4><P>
<INPUT TYPE="reset">
</FORM>
</BODY>
</HTML>
```

## Getting information about a match

For the next application example, the task is not only to verify that a one-field date entry is in the desired format, but also to extract match components of the entry and use those values to perform further calculations in determining the day of the week. The regular expression in the example that follows is a fairly complex one, because it performs some rudimentary range checking to make sure the user doesn't enter a month over 12 or a date over 31. What it does not take into account is the variety of lengths of each month. But the regular expression and method invoked with it extract each date object component in such a way that you can perform additional validation on the range to make sure the user doesn't try to give September 31 days. Also be aware that this is not the only way to perform date validations in forms. Chapter 43 offers additional thoughts on the matter that work without regular expressions for backward compatibility.

Listing 38-2 contains a page that has a field for date entry, a button to process the date, and an output field for display of a long version of the date, including the day of the week. At the start of the function that does all the work, I create two arrays (using the JavaScript 1.2 literal array creation syntax) to hold the plain language names of the months and days. These arrays are used only if the user enters a valid date.

Next comes the regular expression to be matched against the user entry. If you can decipher all the symbols, you see that three components are separated by potential hyphen or forward slash entries (`[\-\/]`). These symbols must be escaped in the regular expression. Importantly, each of the three component definitions is surrounded by parentheses, which are essential for the various objects created with the regular expression to remember their values for extraction later.

Here is a brief rundown of what the regular expression is looking for:

- ✦ A string beginning after a word break
- ✦ A string value for the month that contains a 1 plus a 0 through 2; or an optional 0 plus a 1 through 9
- ✦ A hyphen or forward slash
- ✦ A string value for the date that starts with a 0 plus a 1 through 9; or starts with a 1 or 2 and ends with a 0 through 9; or starts with a 3 and ends with 0 or 1
- ✦ Another hyphen or forward slash
- ✦ A string value for the year that begins with 19 or 20, followed by two digits

An extra pair of parentheses must surround the `19|20` segment to make sure that either one of the matching values is attached to the two succeeding digits. Without the parentheses, the logic of the expression attaches the digits only to 20.

For invoking the regular expression action, I select the `exec()` method, assigning the returned object to the variable `matchArray`. I can also use the *`string`*`.match()` method here. Only if the match is successful (that is, all conditions of the regular expression specification are met) does the major processing continue in the script.

The parentheses around the segments of the regular expression instruct JavaScript to assign each found value to a slot in the `matchArray` object. The month segment is assigned to `matchArray[1]`, the date to `matchArray[2]`, and the year to `matchArray[3]` (`matchArray[0]` contains the entire matched string). Therefore, the script can extract each component to build a plain-language date string with the help of the arrays defined at the start of the function. I even use the values to create a new date object that calculates the day of the week for me. After I have all pieces, I concatenate them and assign the result to the value of the output field. If the regular expression `exec()` method doesn't match the typed entry with the expression, the script provides an error message in the field.

## Listing 38-2: **Extracting Data from a Match**

```
<HTML>
<HEAD>
<TITLE>Got a Match?</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function extractIt(form) {
    var months = ["January","February","March","April","May","June","July",
                  "August","September","October","November","December"]
```

*Continued*

## Listing 38-2 *(continued)*

```
    var days = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday",
               "Saturday"]
    var re = /\b(1[0-2]|0?[1-9])[\-\/](0?[1-9]|[12][0-9]|3[01])[\-
\/]((19|20)\d{2})/
    var input = form.entry.value
    var matchArray = re.exec(input)
    if (matchArray) {
        var theMonth = months[matchArray[1] - 1] + " "
        var theDate = matchArray[2] + ", "
        var theYear = matchArray[3]
        var dateObj = new Date(matchArray[3],matchArray[1]-1,matchArray[2])
        var theDay = days[dateObj.getDay()] + " "
        form.output.value = theDay + theMonth + theDate + theYear
    } else {
        form.output.value = "An invalid date."
    }
}
</SCRIPT>
</HEAD>
<BODY>
<B>Use a regular expression to extract data from a string:</B>
<HR>
<FORM>
Enter a date in the format mm/dd/yyyy or mm-dd-yyyy:<BR>
<INPUT TYPE="text" NAME="entry" SIZE=12><P>
<INPUT TYPE="button" VALUE="Extract Date Components"
onClick="extractIt(this.form)"><P>
The date you entered was:<BR>
<INPUT TYPE="text" NAME="output" SIZE=40><P>
<INPUT TYPE="reset">
</FORM>
</BODY>
</HTML>
```

# String replacement

To demonstrate using regular expressions for performing search-and-replace operations, I choose an application that may be of value to many page authors who have to display and format large numbers. Databases typically store large integers without commas. After five or six digits, however, such numbers are difficult for users to read. Conversely, if the user needs to enter a large number, commas help ensure accuracy.

Helping the procedure in JavaScript regular expressions is the `string.replace()` method (see Chapter 34). The method requires two parameters, a regular expression to search the string and a string to replace any match found in the string. The replacement string can be properties of the `RegExp` object as it stands after the most recent `exec()` method.

Listing 38-3 demonstrates how only a handful of script lines can do a lot of work when regular expressions handle the dirty work. The page contains three fields. Enter any number you want in the first one. A click of the Insert Commas button invokes the `commafy()` function in the page. The result is displayed in the second field. You can also enter a comma-filled number in the second field and click the Remove Commas button to see the inverse operation executed through the `decommafy()` function.

Specifications for the regular expression accept any positive or negative string of numbers. The keys to the action of this script are the parentheses around two segments of the regular expression. One set encompasses all characters not included in the second group: a required set of three digits. In other words, the regular expression is essentially working from the rear of the string, chomping off three-character segments and inserting a comma each time a set is found.

A `while` repeat loop cycles through the string and modifies the string (in truth, the string object is not being modified, but, rather, a new string is generated and assigned to the old variable name). I use the `test()` method because I don't need the returned value of the `exec()` method. The `test()` method impacts the regular expression and `RegExp` object properties the same way as the `exec()` method, but more efficiently. The first time the `test()` method runs, the part of the string that meets the first segment is assigned to the `RegExp.$1` property; the second segment, if any, is assigned to the `RegExp.$2` property. Notice that I'm not assigning the results of the `exec()` method to any variable, because for this application I don't need the array object generated by that method.

Next comes the tricky part. I invoke the *`string`*`.replace()` method, using the current value of the string (`num`) as the starting string. The pattern to search for is the regular expression defined at the head of the function. But the replacement string may look strange to you. The replacement string is replacing whatever the regular expression matches with the value of `RegExp.$1`, a comma, and the value of `RegExp.$2`. The `RegExp` object should not be part of the references used in the `replace()` method parameter. Because the regular expression matches the entire `num` string, the `replace()` method is essentially rebuilding the string from its components, plus adding a comma before the second component (the last free-standing three-digit section). Each `replace()` method invocation sets the value of `num` for the next time through the `while` loop and the `test()` method.

Looping continues until no matches occur — meaning that no more freestanding sets of three digits appear in the string. Then the results are written to the second field on the page.

### Listing 38-3: **Replacing Strings via Regular Expressions**

```
<HTML>
<HEAD>
<TITLE>Got a Match?</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function commafy(form) {
    var re = /(-?\d+)(\d{3})/
    var num = form.entry.value
    while (re.test(num)) {
        num = num.replace(re, "$1,$2")
    }
    form.commaOutput.value = num
}
function decommafy(form) {
    var re = /,/g
    form.plainOutput.value = form.commaOutput.value.replace(re,"")
}
</SCRIPT>
</HEAD>
<BODY>
<B>Use a regular expression to add/delete commas from numbers:</B>
<HR>
<FORM>
Enter a large number without any commas:<BR>
<INPUT TYPE="text" NAME="entry" SIZE=15><P>
<INPUT TYPE="button" VALUE="Insert commas" onClick="commafy(this.form)"><P>
The comma version is:<BR>
<INPUT TYPE="text" NAME="commaOutput" SIZE=20><P>
<INPUT TYPE="button" VALUE="Remove commas" onClick="decommafy(this.form)"><P>
The un-comma version is:<BR>
<INPUT TYPE="text" NAME="plainOutput" SIZE=15><P>
<INPUT TYPE="reset">
</FORM>
</BODY>
</HTML>
```

Removing the commas is an even easier process. The regular expression is a comma with the global flag set. The `replace()` method reacts to the global flag by repeating the process until all matches are replaced. In this case, the replacement string is an empty string. For further examples of using regular expressions with string objects, see the discussions of the *`string`*`.match()`, *`string`*`.replace()`, and *`string`*`.split()` methods in Chapter 34.

# Regular Expression Object

| *Properties* | *Methods* |
|---|---|
| `constructor` | `compile()` |
| `global` | `exec()` |
| `ignoreCase` | `test()` |
| `lastIndex` | |
| `multilane` | |
| `source` | |

## Syntax

Accessing regular expression properties or methods:

*regularExpressionObject.property* | *method*([*parameters*])

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

## About this object

The regular expression object is created on the fly by your scripts. Each regular expression object contains its own pattern and other properties. Deciding which object creation style to use depends on the way the regular expression will be used in your scripts.

When you create a regular expression with the literal notation (that is, with the two forward slashes), the expression is automatically compiled for efficient processing as the assignment statement executes. The same is true when you use the new `RegExp()` constructor and specify a pattern (and optional modifier flags) as a parameter. Whenever the regular expression is fixed in the script, use the literal notation; when some or all of the regular expression is derived from an external source (for example, user input from a text field), assemble the expression as a parameter to the new `RegExp()` constructor. A compiled regular expression should be used at whatever stage the expression is ready to be applied and reused within the script. Compiled regular expressions are not saved to disk or given any more permanence beyond the life of a document's script (that is, it dies when the page unloads).

However, there may be times in which the specification for the regular expression changes with each iteration through a loop construction. For example, if statements in a `while` loop modify the content of a regular expression, compile the expression inside the `while` loop, as shown in the following skeletal script fragment:

```
var srchText = form.search.value
var re = new RegExp()  // empty constructor
while (someCondition) {
    re.compile("\\s+" + srchText + "\\s+", "gi")
    statements that change srchText
}
```

Each time through the loop, the regular expression object is both given a new expression (concatenated with metacharacters for one or more white spaces on both sides of some search text whose content changes constantly) and compiled into an efficient object for use with any associated methods.

# Properties

## constructor

See `string.constructor` (Chapter 34).

## global
## ignoreCase

**Value:** Booleans Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

These two properties reflect the regular expression `g` and `i` modifier flags, if any, associated with a regular expression. Settings are read-only and are determined as the object is created. Each property is independent of the other.

**Related Items:** None.

## lastIndex

**Value:** Integer Read/Write

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

The `lastIndex` property indicates the index counter of the main string to be searched against the current regular expression object. When a regular expression object is created, this value is zero, meaning that there have been no searches with this object, and the default behavior of the first search is to start at the beginning of the string.

If the regular expression has the global modifier specified, the `lastIndex` property value advances to some higher value after the object is used in a method to match within a main string. The value is the position in the main string immediately after the previous matched string (and not including any character of the matched string). After locating the final match in a string, the method resets the `lastIndex` property to zero for the next time. You can also influence the behavior of matches by setting this value on the fly. For example, if you want the expression to begin its search at the fourth character of a target string, you change the setting immediately after creating the object, as follows:

```
var re = /somePattern/
re.lastIndex = 3  // fourth character in zero-based index system
```

**Related Items:** Match result object index property.

## multiline

**Value:** Boolean Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | | ✓ | | | | | ✓ |

The multiline property reveals whether searches extend across multiple lines of a target string, as directed by the optional `m` modifier flag for a regular expression. NN4+ also includes the same-named property for the `RegExp` object (see the following section).

**Related Items:** `RegExp.multiline` property.

## `source`

**Value:** String Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

The `source` property is simply the string representation of the regular expression used to define the object. This property is read-only.

**Related Items:** None.

# Methods

## `compile("`*`pattern`*`", ["g" | "i" | "m"])`

**Returns:** Regular expression object.

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

Use the `compile()` method to compile on the fly a regular expression whose content changes continually during the execution of a script. See the discussion earlier about this object for an example. Other regular expression creation statements (the literal notation and the new `RegExp()` constructor that passes a regular expression) automatically compile their expressions. The `m` pattern modifier is available in IE5.5+ and NN6+.

**Related Items:** None.

## `exec("`*`string`*`")`

**Returns:** Match array object or `null`.

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

The `exec()` method examines the string passed as its parameter for at least one match of the specification defined for the regular expression object. The behavior of this method is similar to that of the *string*`.match()` method (although the `match()` method is more powerful in completing global matches). Typically, a call to the `exec()` method is made immediately after the creation of a regular expression object, as in the following example.

```
var re = /somePattern/
var matchArray = re.exec("someString")
```

Much happens as a result of the `exec()` method. Properties of both the regular expression object and window's `RegExp` object are updated based on the success of the match. The method also returns an object that conveys additional data about the operation. Table 38-4 shows the properties of this returned object.

**Table 38-4 Match Found Array Object Properties**

| *Property* | *Description* |
|---|---|
| `index` | Zero-based index counter of the start of the match inside the string |
| `input` | Entire text of original string |
| `[0]` | String of most recent matched characters |
| `[1],...[n]` | Parenthesized component matches |

Some of the properties in this returned object echo properties in the `RegExp` object. The value of having them in the regular expression object is that their contents are safely stowed in the object while the `RegExp` object and its properties may be modified soon by another call to a regular expression method. Items the two objects have in common are the `[0]` property (mapped to the `RegExp.lastMatch` property) and the `[1],. . .[n]` properties (the first nine of which map to `RegExp.$1. . .RegExp.$9`). While the `RegExp` object stores only nine parenthesized subcomponents, the returned array object stores as many as are needed to accommodate parenthesis pairs in the regular expression.

If no match turns up between the regular expression specification and the string, the returned value is `null`. See Listing 38-2 for an example of how this method can be applied. An alternate shortcut syntax may be used for the `exec()` method. Turn the regular expression into a function, as in

```
var re = /somePattern/
var matchArray = re("someString")
```

**Related Items:** `string.match()` method.

## `test("`*`string`*`")`

**Returns:** Boolean.

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

The most efficient way to find out if a regular expression has a match in a string is to use the `test()` method. Returned values are `true` if a match exists and `false` if not. In case you need more information, a companion method, *`string`*`.search()`, returns the starting index value of the matching string. See Listing 38-1 for an example of this method in action.

**Related Items:** `string.search()` method.

# RegExp Object

| *Properties* | *Methods* |
|---|---|
| `input` | |
| `lastMatch` | |
| `lastParen` | |
| `leftContext` | |
| `multilane` | |
| `prototype` | |
| `rightContext` | |
| `$1, ... $9` | |

## Syntax

Accessing `RegExp` properties:

```
RegExp.property
```

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

## About this object

Beginning with Navigator 4 and Internet Explorer 4, the browser maintains a single instance of a `RegExp` object for each window or frame. The object oversees the action of all methods that involve regular expressions (including the few related string object methods). Properties of this object are exposed not only to JavaScript in the traditional manner, but also to a parameter of the method *`string`*`.replace()` for some shortcut access (see Listing 38-3).

With one `RegExp` object serving all regular expression-related methods in your document's scripts, you must exercise care in accessing or modifying this object's properties. You must make sure that the `RegExp` object has not been affected by another method. Most properties are subject to change as the result of any method involving a regular expression. This may be reason enough to use the properties of the array object returned by most regular expression methods instead of the `RegExp` properties. The former stick with a specific regular expression object even after other regular expression objects are used in the same script. The `RegExp` properties reflect the most recent activity, irrespective of the regular expression object involved.

In the following listings, I supply the long, JavaScript-like property names. But each property also has an abbreviated, Perl-like manner to refer to the same properties. You can use these shortcut property names in the *`string`*`.replace()` method if you need the values.

## Properties

### `input`

**Value:** String Read/Write

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

The `RegExp.input` property is the main string against which a regular expression is compared in search of a match. In all of the example listings earlier in this chapter, the property was `null`. Such is the case when the main string is supplied as a parameter to the regular expression-related method.

But many text-related document objects have an unseen relationship with the `RegExp` object. If a text, TEXTAREA, SELECT, or link object contains an event handler that invokes a function containing a regular expression, the `RegExp.input` property is set to the relevant textual data from the object. You don't have to specify any parameters for the event handler call or in the function called by the event handler. For text and TEXTAREA objects, the `input` property value becomes the content of the object; for the SELECT object, it is the text (not the value) of the selected option; and for a link, it is the text highlighted in the browser associated with the link (and reflected in the link's `text` property).

Having JavaScript set the `RegExp.input` property for you may simplify your script. You can invoke either of the regular expression methods without having to specify the main string parameter. When that parameter is empty, JavaScript applies the `RegExp.input` property to the task. You can also set this property on the fly if you want. The short version of this property is `$_` (dollar sign underscore).

**Related Items:** Matching array object `input` property.

## multiline

**Value:** Boolean　　　　Read/Write

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | |

The `RegExp.multiline` property determines whether searches extend across multiple lines of a target string. This property is automatically set to `true` as an event handler of a TEXTAREA triggers a function containing a regular expression. You can also set this property on the fly if you want. The short version of this property is `$*`. This version of the property (as distinct from the `multiline` property of an instance of a regular expression) is not defined in the ECMA-262 specification and is supported only in NN4+.

**Related Items:** Regular expression instance object `multiline` property.

## lastMatch

**Value:** String Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

After execution of a regular expression-related method, any text in the main string that matches the regular expression specification is automatically assigned to the `RegExp.lastMatch` property. This value is also assigned to the `[0]` property of the object array returned after the `exec()` and `string.match()` methods find a match. The short version of this property is `$&`.

**Related Items:** Matching array object [0] property.

## lastParen

**Value:** String Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

When a regular expression contains many parenthesized subcomponents, the `RegExp` object maintains a list of the resulting strings in the `$1,...$9` properties. You can also extract the value of the last matching parenthesized subcomponent with the `RegExp.lastParen` property, which is a read-only property. The short version of this property is `$+`.

**Related Items:** `RegExp.$1,...$9` properties.

## leftContext
## rightContext

**Value:** String Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | | | ✓ |

After a match is found in the course of one of the regular expression methods, the `RegExp` object is informed of some key contextual information about the match. The `leftContext` property contains the part of the main string to the left of (up to but not including) the matched string. Be aware that the `leftContext` starts its string from the point at which the most recent search began. Therefore, for second or subsequent times through the same string with the same regular expression, the `leftContext` substring varies widely from the first time through.

The `rightContext` consists of a string starting immediately after the current match and extending to the end of the main string. As subsequent method calls work on the same string and regular expression, this value obviously shrinks in length until no more matches are found. At this point, both properties revert to `null`. The short versions of these properties are `` $` `` and `$'` for `leftContext` and `rightContext`, respectively.

**Related Items:** None.

## prototype

See `String.prototype` (Chapter 34).

## $1...$9

**Value:** String Read-Only

| | NN2 | NN3 | NN4 | NN6 | IE3/J1 | IE3/J2 | IE4 | IE5 | IE5.5 |
|---|---|---|---|---|---|---|---|---|---|
| **Compatibility** | | | ✓ | ✓ | | | ✓ | ✓ | ✓ |

As a regular expression method executes, any parenthesized result is stored in `RegExp`'s nine properties reserved for just that purpose (called backreferences).

The same values (and any beyond the nine that RegExp has space for) are stored in the array object returned with the exec() and *string*.match() methods. Values are stored in the order in which the left parenthesis of a pair appears in the regular expression, regardless of nesting of other components.

You can use these backreferences directly in the second parameter of the *string*.replace() method, without using the RegExp part of their address. The ideal situation is to encapsulate components that need to be rearranged or recombined with replacement characters. For example, the following script function turns a name that is last name first into first name last:

```
function swapEm() {
    var re = /(\w+),\s*(\w+)/
    var input = "Lincoln, Abraham"
    return input.replace(re,"$2 $1")
}
```

In the replace() method, the second parenthesized component (just the first name) is placed first, followed by a space and the first component. The original comma is discarded. You are free to combine these shortcut references as you like, including multiple times per replacement, if it makes sense to your application.

**Related Items:** Matching array object [1]. . .[n] properties.

✦ ✦ ✦