

have the same set of properties and methods, at least initially. These objects can diverge from each other through user code changes. Furthermore, there is no convenient way to determine in the script whether two objects have the same set of properties and methods.

4.12 Pattern Matching by Using Regular Expressions

JavaScript has powerful pattern-matching capabilities based on regular expressions. There are two approaches to pattern matching in JavaScript: one that is based on the methods of the `RegExp` object and one that is based on methods of the `String` object. The regular expressions used by these two approaches are the same and based on the regular expressions of the Perl programming language. This book covers only the `String` methods for pattern matching.

As stated previously, patterns are specified in a form that is based on regular expressions, which originally were developed to define members of a simple class of formal languages. Elaborate and complex patterns can be used to describe specific strings or categories of strings. Patterns, which are sent as parameters to the pattern-matching methods, are delimited with slashes.

The simplest pattern-matching method is `search`, which takes a pattern as a parameter. The `search` method returns the position in the `String` object (through which it is called) at which the pattern matched. If there is no match, `search` returns `-1`. Most characters are normal, which means that, in a pattern, they match themselves. The position of the first character in the string is 0. As an example, consider the following statements:

```
var str = "Rabbits are furry";
var position = str.search(/bits/);
if (position >= 0)
    document.write("'bits' appears in position", position,
                    "<br />");
else
    document.write("'bits' does not appear in str <br />");
```

These statements produce the following output:

```
'bits' appears in position 3
```

4.12.1 Character and Character-Class Patterns

The *normal* characters are those that are not metacharacters. Metacharacters are characters that have special meanings in some contexts in patterns. The following are the pattern metacharacters:

```
\ | ( ) [ ] { } ^ $ * + ? .
```

Metacharacters can themselves be placed in a pattern by being immediately preceded by a backslash.

A period matches any character except newline. So, the following pattern matches "snowy", "snowe", and "snowd", among others:

```
/snow./
```

To match a period in a string, the period must be preceded by a backslash in the pattern. For example, the pattern `/3\.4/` matches `3.4`. The pattern `/3.4/` matches `3.4` and `374`, among others.

It is often convenient to be able to specify classes of characters rather than individual characters. Such classes are defined by placing the desired characters in brackets. Dashes can appear in character class definitions, making it easy to specify sequences of characters. For example, the following character class matches 'a', 'b', or 'c':

```
[abc]
```

The following character class matches any lowercase letter from 'a' to 'h':

```
[a-h]
```

If a circumflex character (^) is the first character in a class, it inverts the specified set. For example, the following character class matches any character except the letters 'a', 'e', 'i', 'o', and 'u':

```
[^aeiou]
```

Because they are frequently used, some character classes are predefined and named and can be specified by their names. These are shown in Table 4.8, which gives the names of the classes, their literal definitions as character classes, and descriptions of what they match.

Table 4.8 Predefined character classes

Name	Equivalent Pattern	Matches
<code>\d</code>	<code>[0-9]</code>	A digit
<code>\D</code>	<code>[^0-9]</code>	Not a digit
<code>\w</code>	<code>[A-Za-z_0-9]</code>	A word character (alphanumeric)
<code>\W</code>	<code>[^A-Za-z_0-9]</code>	Not a word character
<code>\s</code>	<code>[\r\t\n\f]</code>	A white-space character
<code>\S</code>	<code>[^\r\t\n\f]</code>	Not a white-space character

The following examples show patterns that use predefined character classes:

```
// \d\.\d\d/    // Matches a digit, followed by a period,
                // followed by two digits
// \D\d\D/      // Matches a single digit
// \w\w\w/      // Matches three adjacent word characters
```

In many cases, it is convenient to be able to repeat a part of a pattern, often a character or character class. To repeat a pattern, a numeric quantifier, delimited by braces, is attached. For example, the following pattern matches `xyyyyz`:

```
/xy{4}z/
```

There are also three symbolic quantifiers: asterisk (*), plus (+), and question mark (?). An asterisk means zero or more repetitions, a plus sign means one or more repetitions, and a question mark means one or none. For example, the following pattern matches strings that begin with any number of `x`'s (including zero), followed by one or more `y`'s, possibly followed by `z`:

```
/x*y+z?/
```

The quantifiers are often used with the predefined character-class names, as in the following pattern, which matches a string of one or more digits followed by a decimal point and possibly more digits:

```
/\d+\.\d*/
```

As another example, the pattern

```
/[A-Za-z]\w*/
```

matches the identifiers (a letter, followed by zero or more letters, digits, or underscores) in some programming languages.

There is one additional named pattern that is often useful: `\b` (boundary), which matches the boundary position between a word character (`\w`) and a non-word character (`\W`), in either order. For example, the following pattern matches "A tulip is a flower" but not "A frog isn't":

```
/\bis\b/
```

The pattern does not match the second string because the "is" is followed by another word character (`n`).

The boundary pattern is different from the named character classes in that it does not match a character; instead, it matches a position between two characters.

4.12.2 Anchors

Frequently, it is useful to be able to specify that a pattern must match at a particular position in a string. The most common example of this type of specification is requiring a pattern to match at one specific end of the string. A pattern is tied to a position at one of the ends of a string with an anchor. It can be specified to match only at the beginning of the string by preceding it with a circumflex (^) anchor. For example, the following pattern matches "pearls are pretty" but does not match "My pearls are pretty":

```
/^pearl/
```

A pattern can be specified to match at the end of a string only by following the pattern with a dollar sign anchor. For example, the following pattern matches "I like gold" but does not match "golden":

```
/gold$/
```

Anchor characters are like boundary-named patterns: They do not match specific characters in the string; rather, they match positions before, between, or after characters. When a circumflex appears in a pattern at a position other than the beginning of the pattern or at the beginning of a character class, it has no special meaning. (It matches itself.) Likewise, if a dollar sign appears in a pattern at a position other than the end of the pattern, it has no special meaning.

4.12.3 Pattern Modifiers

Modifiers can be attached to patterns to change how they are used, thereby increasing their flexibility. The modifiers are specified as letters just after the right delimiter of the pattern. The *i* modifier makes the letters in the pattern match either uppercase or lowercase letters in the string. For example, the pattern `/Apple/i` matches 'APPLE', 'apple', 'APPlE', and any other combination of uppercase and lowercase spellings of the word "apple."

The *x* modifier allows white space to appear in the pattern. Because comments are considered white space, this provides a way to include explanatory comments in the pattern. For example, the pattern

```
/\d+          # The street number
\s           # The space before the street name
[A-Z] [a-z]+ # The street name
/x
```

is equivalent to

```
/\d+\s[A-Z] [a-z]+/
```

4.12.4 Other Pattern-Matching Methods of String

The `replace` method is used to replace substrings of the `String` object that match the given pattern. The `replace` method takes two parameters: the pattern and the replacement string. The *g* modifier can be attached to the pattern if the replacement is to be global in the string, in which case the replacement is done for every match in the string. The matched substrings of the string are made available through the predefined variables `$1`, `$2`, and so on. For example, consider the following statements:

```
var str = "Fred, Freddie, and Frederica were siblings";
str.replace(/Fre/g, "Boy");
```

In this example, `str` is set to "Boyd, Boyddie, and Boyderica were siblings", and `$1`, `$2`, and `$3` are all set to "Fre".

The `match` method is the most general of the `String` pattern-matching methods. The `match` method takes a single parameter: a pattern. It returns an array of the results of the pattern-matching operation. If the pattern has the `g` modifier, the returned array has all the substrings of the string that matched. If the pattern does not include the `g` modifier, the returned array has the match as its first element, and the remainder of the array has the matches of parenthesized parts of the pattern if there are any:

```
var str =
  "Having 4 apples is better than having 3 oranges";
var matches = str.match(/\d/g);
```

In this example, `matches` is set to `[4, 3]`.

Now consider a pattern that has parenthesized subexpressions:

```
var str = "I have 428 dollars, but I need 500";
var matches = str.match(/(\d+) ([^\d]+) (\d+)/);
document.write(matches, "<br />");
```

The following is the value of the `matches` array after this code is interpreted:

```
["428 dollars, but I need 500", "428",
 " dollars, but I need ", "500"]
```

In this result array, the first element, `"428 dollars, but I need 500"`, is the match; the second, third, and fourth elements are the parts of the string that matched the parenthesized parts of the pattern, `(\d+)`, `([^\d]+)`, and `(\d+)`.

The `split` method of `String` splits its object string into substrings on the basis of a given string or pattern. The substrings are returned in an array. For example, consider the following code:

```
var str = "grapes:apples:oranges";
var fruit = str.split(":");
```

In this example, `fruit` is set to `[grapes, apples, oranges]`.

4.13 Another Example

One of the common uses for JavaScript is to check the format of input from HTML forms, which is discussed in detail in Chapter 5. The following example presents a simple function that uses pattern matching to check a given string that is supposed to contain a phone number, in order to determine whether the format of the phone number is correct:

```
// forms_check.js
// A function tst_phone_num is defined and tested.
// This function checks the validity of phone
// number input from a form
```

```
// Function tst_phone_num
//   Parameter: A string
//   Result: Returns true if the parameter has the form of a valid
//           seven-digit phone number (3 digits, a dash, 4 digits)

function tst_phone_num(num) {

// Use a simple pattern to check the number of digits and the dash
  var ok = num.search(/^\\d{3}-\\d{4}$/);

  if (ok == 0)
    return true;
  else
    return false;

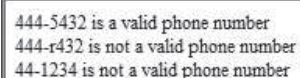
} // end of function tst_phone_num

// A script to test tst_phone_num
var tst = tst_phone_num("444-5432");
if (tst)
  document.write("444-5432 is a valid phone number <br />");
else
  document.write("Error in tst_phone_num <br />");

tst = tst_phone_num("444-r432");
if (tst)
  document.write("Program error <br />");
else
  document.write(
    "444-r432 is not a valid phone number <br />");

tst = tst_phone_num("44-1234");
if (tst)
  document.write("Program error <br />");
else
  document.write("44-1234 is not a valid phone number <br />");
```

Figure 4.12 shows a browser display of `forms_check.js`.



```
444-5432 is a valid phone number
444-r432 is not a valid phone number
44-1234 is not a valid phone number
```

Figure 4.12 Display of `forms_check.js`