

## 16. Regular Expressions

### Introduction

The `match()` instance method of `String` has the syntax

```
str.match( regex )
```

where

- *regex* is a regular expression and
- `str` is the string against which the regular expression is matched.

It returns `null` (which counts as `false` in a comparison) if the pattern specified by *regex* is not found in `str`.

A regular expression specifies a pattern, an abstract representation of a set of strings.

A pattern matches when the string with which it is compared contains as a substring a member of that set.

A pattern is delimited by `/ ... /`.

The simplest patterns are strings of individual characters.

Such a pattern matches strings that contain it as a substring.

#### *Example*

The following displays “Yes”.

```
var str = "payday",
    reg = /day/;

if ( str.match( reg ) )
    document.write( "<p>Yes</p>" );
```

We could write this more compactly as

```
if ( "payday".match( /day/ ) )
    document.write( "<p>Yes</p>" );
```

There is nothing special about a space in a pattern.

For example,

```
/a day/
```

matches the indicated five-element string.

An escape sequence (such as `\t`, a tab) in a pattern matches itself.

A *metacharacter* is a character with special meaning.

To match a metacharacter, we precede it by a `\` in the pattern.

The metacharacters (some of which have special meaning only in certain contexts) are

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

The period ( `.` ) matches any character but newline.

*Example*

The regular expression here matches any dollar quantity between \$10.00 and \$99.99.

This fragment displays “Yes”.

```
var str = "$52.47",
    reg = /\$..\.../;

if ( str.match( reg ) )
    document.write( "<p>Yes</p>" );
```

## Character Classes

A character class is a pattern specified as a collection of characters.

Any character in the class can match one character in the target string.

A character class is defined by putting the characters within [ ... ].

For example, [ >a.b! ] matches any of the characters '>', 'a', 'b', or '!'.  
[ >a.b! ]

A dash ( - ) may be used to specify a range.

For example:

[ a-z ] matches any lowercase letter.

[ A-Za-z ] matches any letter.

[ 0-9a-zA-Z ] matches any alphanumeric character.

A dash at the end of a character class is considered a literal dash.

For example, [ 0-2- ] matches any of the characters '0', '1', '2', or '-'.  
[ 0-2- ]

By placing a ^ at the left end of the characters, we specify the characters not in the class.

For example, [ ^A-Za-z ] matches any non-letter.

*Example*

The following displays “Yes”.

```
var str = "XY23?ab",
    reg = /[A-Za-z_][0-9][0-9][^A-Za-z_]/;

if ( str.match( reg ) )
    document.write( "<p>Yes</p>" );
```

There are abbreviations for commonly used character classes.

<b>Abbreviation</b>	<b>Equivalent Pattern</b>	<b>Matches</b>
<code>\d</code>	<code>[0-9]</code>	a digit
<code>\D</code>	<code>[^0-9]</code>	a nondigit
<code>\w</code>	<code>[A-Za-z_]</code>	a word character
<code>\W</code>	<code>[^A-Za-z_]</code>	a nonword character
<code>\s</code>	<code>[ \r\t\n\f ]</code>	a white-space character
<code>\S</code>	<code>[^ \r\t\n\f ]</code>	a non-white-space character

For example, the pattern in the last example can be specified as

```
/\w\d\d\W/
```

## Quantifiers

Quantifiers allow patterns to be repeated a specified number of times.

The most general quantifier has the form

$$\{m, n\}$$

where  $m$  and  $n$  are natural numbers.

<b>Quantifier</b>	<b>Meaning</b>
$\{m\}$	exactly $m$ repetitions
$\{m, \}$	at least $m$ repetitions
$\{m, n\}$	at least $m$ but no more than $n$ repetitions

When attached to a string, a quantifier modifies only the last character.

Part of a pattern can be grouped with parentheses so that a quantifier may apply to what's within the parentheses.

### *Examples*

<code>/ab{3}c/</code>	matches <code>abbbc</code>
<code>/ab{2,}c/</code>	matches <code>abbc</code> , <code>abbbc</code> , ...
<code>/a{1,3}b/</code>	matches <code>ab</code> , <code>aab</code> , <code>aaab</code>
<code>/(cats){2} z{3}/</code>	matches <code>catcats</code> <code>zzz</code>
<code>/\d{3}-\d{2}-\d{4}/</code>	matches Social Security numbers

The curly braces, `{ ... }`, used where they can be interpreted as part of a quantifier, are metacharacters.

Elsewhere they match themselves.

There are metacharacter quantifiers that specialize the general form  $\{m, n\}$ .

Quantifier	Equivalent	Description
*	$\{0, \}$	zero or more repetitions
+	$\{1, \}$	one or more repetitions
?	$\{0, 1\}$	zero or one – an optional item

When a \*- or ?-quantified pattern matches zero of what it quantifies, it matches a position in the string, not a character.

### Examples

`/\${?}\d+\.\d{2}/` matches a price, with or without the \$.

`/\d+\.\d+/` matches numbers with a decimal point and at least one digit on each side.

`/cat.*fish/` matches `cat` and `fish` separated by zero or more characters except a newline.

## When More than One Substring Matches a Pattern

Consider a case where there is no quantifier:

```
"Freddie".match( /d/ )
```

Matches are made at the leftmost position where the entire pattern may match.

So here the left `d` matches.

In

```
"Freddie".match( /d*/ )
```

both `d` and `dd` would match if the matcher didn't try left matches first

But the pattern actually matches the empty string just before the 'F'.

(I.e., it matches a position, not a character.)

In

```
"Freddie".match( /d*i/ )
```

the match is with "ddi" since it is the leftmost substring fitting the pattern.

By default, the matcher handles quantifiers in *greedy* mode.

Thus, `.*` or `.+` matches the maximum number of non-newline characters.

In

```
"Tom saw the Tomcat".match( /. *Tom/ )
```

the match against Tom in the pattern is the Tom in Tomcat.

By matching the rightmost occurrence of Tom, `.*` matches the maximum number of characters.

We can specify *minimal* mode for a quantifier by following it with a `?`.

Thus, `. *?` or `. +?` matches the minimum number of characters.

In

```
"Tom saw the Tomcat".match( /. *?Tom/ )
```

the match against Tom in the pattern is the Tom at the beginning of the string.

When a pattern has two quantified subpatterns, the leftmost is greediest.

For example, in

```
"Tom saw a Tomcat and heard Tom-toms".match(
                                         /Tom.*Tom.*toms/ )
```

the first `. *` matches

```
“ saw a Tomcat and heard “
```

## Alternation

| is the alternation (OR) operation.

### Examples

/a|b|c/ – the same as [a-c]

/Jack|Jill/

Parentheses may be used to group alternative – e.g.,

/(Bob|Tom)cat/

The alternatives are tried from left to right.

So /Bob|Bobby/ never makes a match with the “Bobby” alternative.

## Precedence of Operators

parentheses	<b>highest</b>
quantifiers	↑
character sequence	
alternative	<b>lowest</b>

### Examples

/a|b+/ matches an a or one of more b's.

/(a|b)+/ matches one or more a's or b's.

## **Anchors**

Anchors match positions between characters, not characters.

`^` at the beginning of a pattern requires the pattern to match at the beginning of the string.

`/^one/` matches “one more” but not “and one”.

`$` at the end of a pattern requires the pattern to match at the end of the string.

`/one$/` matches “and one” but not “one more”.

`\b` matches the position between a word character (`\w`) and a non-word character (`\W`).

It’s used on both sides of a word pattern to match a string that’s not part of a bigger word.

`/\band\b/` matches “one and two” but not “one hand”.

## **Pattern Modifiers**

Pattern modifiers are placed after the regular expression.

`i` causes the case of the letters in the string to be ignored.

`/CAT/i` matches “CAT”, “cat”, “Cat”, caT”, ....

## Remembering Matches

Sometimes we want to reference the part of the string that matched an earlier part of the pattern.

Put parentheses around the part of the pattern whose match is to be remembered.

Then the *implicit variable* `\1` in the pattern contains the part of the string remembered.

Example

`/(\d) . *\1/` matches “121” but not “123”.

The part matching the second parenthesized part (if there is one) is in `\2`.

Implicit variables go up to `\9`.

*Example*

`/(. ) . *( . ) . *\1 . *\2/` matches “church” because of the ‘c’, ‘h’ then ‘c’, ‘h’ again.

The implicit variables `RegExp.$1` to `RegExp.$9` are used outside the pattern instead of `\1` to `\9`.

*Example*

The pattern

`/\((\d{3})\) (\d{3})-(\d{4})/`

matches a phone number such as

(336) 334-7245

After the match,

`RegExp.$1` contains “336”,

`RegExp.$2` contains “334”, and

`RegExp.$3` contains “7245”.

The implicit variables `\1` to `\9` and `RegExp.$1` to `RegExp.$9` are called *backreferences*.

## The `split()` Method Revisited

We've previously used the `split()` instance method of `String` with a string as its argument, giving a delimiter to split the string into substrings.

For example,

```
var str1 = "John, Bill, Fred",
    names = str1.split( ", " ),
```

sets `names` to the array [ "John", "Bill", "Fred" ].

The parameter of `split()` can be a regular expression.

Then anything matching the regular expression is a delimiter.

For example,

```
str2 = "(336) 334-7245",
codes = str2.split( /\(|\)|-/ );
```

sets `codes` to the array [ "336", "334", "7245" ].

## The `replace()` Method

The `replace()` instance method of `String` is used to replace the (first) part of a string that matches a given pattern with a given string.

It has the syntax

```
str.replace( regexp, ReplacementString )
```

where

- `str` is the string before the replacement,
- `regexp` is a regular expression matched against `str`, and
- `ReplacementString` is the string that replaces the part of `str` matched by `regexp`.

The `replace()` method returns a new string.

It does not change the string invoking it – replacement is not “in place”.

For example,

```
var
  str1 = "Dogs are pets.  "
        + "Some dogs are big, some dogs are small.",
  str2 = str1.replace( /dog/, "cat"),
```

sets `str2` to

```
"Dogs are pets.  Some cats are big, some dogs are small."
```

Only the first substring in `str1` matching `/dog/` is replaced.

And the match is case sensitive.

To have every occurrence of a pattern replaced by the replacement string, we must use the `g` (global) modifier for the pattern.

For example, given `str1` as above,

```
str3 = str1.replace( /dog/g, "cat" ),  
sets str3 to
```

```
"Dogs are pets. Some cats are big, some cats are small."
```

We can use the `g` and `i` modifiers together.

The order in which they appear is immaterial.

For example, given `str1` as above,

```
str4 = str1.replace( /dog/gi, "cat" );  
sets str4 to
```

```
"cats are pets. Some cats are big, some cats are small."
```

If a sequence of replacements is needed, we can make a replacement to the result of a replacement.

For example,

```
str5 =  
str1.replace( /dog/g, "cat" ).replace( /Dog/g, "Cat" );  
sets str5 to
```

```
"Cats are pets. Some cats are big, some cats are small."
```

The entire regular expression syntax can be used for the pattern that is the first argument of `replace`.

For example, the pattern

```
/\w+\W+\w+/
```

matches two words.

So

```
var str1 = "One Two Three",
    reg = /\w+\W+\w+/,
    str2 = str1.replace( reg, "Zero" );
```

sets `str2` to

```
"Zero Three"
```

The backreference implicit variables `RegExp.$1` to `RegExp.$9` are defined for `replace()` just as they are for `match()`.

For example, if we execute

```
var str1 = "One Two Three",
    reg = /(\w+)\W+(\w+)/,
    str2 = str1.replace( reg, "Zero" );
```

then

```
RegExp.$1 contains "One" and
```

```
RegExp.$2 contains "Two".
```

The implicit variables used for backreference in the replacement string are `$1` to `$9`.

These are not properties of `RegExp` (like `RegExp.$1` to `RegExp.$9`).

They must occur within the quoted (single or double quotes) string.

For example,

```
var str1 = "One Two Three",
    reg = /(\w+)\W+(\w+)/,
    str2 = str1.replace( reg, "$2 $1");
```

sets `str2` to

*“Two One Three”*

This is *variable interpolation*: the value of the variable is substituted for the variable within a string.

The string where variable interpolation takes place may contain any number of other characters.

For example, with `str1` and `reg` as in the last example,

```
str3 = str1.replace( reg, "$2 and $1")
```

sets `str3` to

*“Two and One Three”*

Variable interpolation works only for the implicit variables `$1` to `$9`.

To involve normal variables in the replacement string, you must concatenate them in.

For example,

```
var str1 = "One Two Three",
    reg = /(\w+)\W+(\w+)/,
    str2 = "Zero",
    str4 = str1.replace( reg, str2 + " $2 $1");
```

sets `str4` to

*“Zero Two One Three”*

## More on `match()`

When

```
str.match( reg )
```

does not find a match, it returns `null`.

When it finds a match, it returns an object.

Suppose we execute

```
ar = str.match( reg );
```

Then

`ar.index` is the index of the beginning of the match in the string.

(The first position in the string is 0.)

`ar.input` is the original target string.

`ar[0]` is the part of the string that matched.

`ar[1]` to `ar[9]` are the same as `RegExp.$1` to `RegExp.$9`  
(the parenthesized substring matches, if any).

For example,

```
var str = " !?% One Two Three",
    reg = /(\w+)\W+(\w+)/,
    ar = str.match( reg );

document.writeln( ar.index + "<br>"
                  + ar[0],    "<br>",
                  + ar[1] + "<br>" + ar[2] );
```

outputs

```
4
One Two
One
Two
```

## **The `search()` Method**

The `search()` instance method of `String` is just like `match()` except that it returns just a non-negative integer if it finds a match.

This integer is the position of the match in the string.

Also, if it doesn't find a match, it returns `-1`.

## The RegExp Prototype Object

We've already seen the \$1 to \$9 properties of the RegExp object.

These are the implicit variables RegExp.\$1 to RegExp.\$9.

The RegExp() constructor function takes two arguments, the second being optional.

The first argument is the pattern without the /.../ delimiters and enclosed in quotes (i.e., a string).

The second argument (if present) is a string containing the one or two modifiers for the regular expression – either "i", "g", "ig", or "gi".

For example,

```
var reg1 = RegExp( "(cat|dog)fish", "i" ),
    reg2 = RegExp( "(\\w+)\\W+(\\w+)" );
```

sets reg1 to

```
/(cat|dog)fish/i
```

and reg2 to

```
/(\\w+)\\W+(\\w+)/
```

This constructor function is useful when the pattern is unknown when the program is written.

This commonly happens when the pattern is constructed at runtime or is taken from user input.

There are `RegExp` instance methods that correspond to the instance methods `match()` and `search()` of `String`.

The positions of the target string and the pattern are swapped.

Where `regexp` is a regular expression and `str` is the target string,

```
regexp.exec( str )
```

is the same as

```
str.match( regexp )
```

and

```
regexp.test( str )
```

is the same as

```
str.search( regexp )
```

except it returns `true` (not the position of the match) or `false` (not `null`).