Regular Expressions

Made Easy With Examples

Regular Expressions

Made Easy With Examples

Disclaimer of Warranty

Contents

Introduction

CONTENTS

A Regular Expression (regex) is a sequence of characters, numbers, and symbols that define a pattern that allows you to search, match, replace, manipulate, and manage text. Regular expressions are frequently used to configure and administer networks and computer systems. They are used to validate email addresses, email addresses and other data. They are used for searching databases, and uncountable other purposes.

Regular expressions are very powerful, but also difficult to understand and complicated to use. Many programmers prefer to use String object methods. Almost anything that can be done with a regular expression can be done in a more understandable way using String object methods. But sometimes a regular expression is a the most optimal, or the only solution for a task.

Regular Expressions is a programming language within a programming language. You can use regular expression with almost all programming languages: JavaScript, Java, VB, C, C++, C#, Python, and many others. Regular Expressions work similar in most languages with slight syntax variations and different support for advanced features in different languages.

My goal in this book is to make you proficient in the use of regular expressions without the burden of ups and extras and the complicated structures of programming languages. Since JavaScript is by far the most popular programming language, and because JavaScript is an interpreted language that can be executed in any web browser, that's the language I use here.

You can find plenty of regular expressions reference sheets online, but simple examples are scarce. This book provides over 50 simple examples that you can easily copy and experiment with. If you are not comfortable with regular expressions after reading this book, then I haven't done my job, and I want to hear from you. You can reach me through the contact form on bucarotechelp.com or through amazon's review process.

Your First Regular Expression

The first and last characters of a regular expression are the delimiters. The delimiters mark the boundaries of the regular expression pattern. JavaScript uses the forward slash (/) as a delimiter. An example of a most basic regular expression is shown below.

/efgh/

The four letters inside the regular expression delimiters are character literals. Shown below is an example of how this regular expression could be used.

```
<script>
var strTarget = "abcdefghijkl";
var loc = strTarget.search(/efgh/);
alert(loc);
</script>
```

In case you are not familiar with how to execute code like that shown above. Open a new text file (use a basic ASCII text editor like Windows Notepad, not a word processor. Word processors add formatting characters to the text). Paste the code shown above into the text file. Save the text file with a name that has the file extension .htm. (say test.htm) Double-click on the file to open it in your web browser.

When you execute the above code, a message box will open displaying the number 4. The code creates a text string "abcdefghijkl" and then passes the regular expression /efgh/ to the JavaScript String Object method *search*. The search method returns the number 4, storing it in the variable loc, which is the character position in the target string where it found the regular expression literal efgh.

The code then uses the JavaScript *alert* method to display the contents of the variable loc. 4 is the 0 based location of the first character of efgh in the string "abcdefghijkl". Had the search the method not found the literal efgh in the target string, it would have returned -1.

Pattern Matching

CONTENTS

One of the most useful functions in regular expression pattern matching is the String.match() method. The .match() method returns the result of matching a string against a regular expression.

The example below matches the characters "ain" in the string "It's a pain on a train in the rain."

```
<script>
let strTarget = "It's a pain on a train in the rain.";
let regexp = /ain/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The regular expression /ain/ matches the letters "ain" in the word "pain". In this example, the regular expression finds the first match.

This example displays "ain" in a message box.

Matches All Instances

The example below matches all instances of the characters "ain" in the string.

```
<script>
let strTarget = "It's a pain on a train in the rain.";
let regexp = /ain/g;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The addition of the control character g (global) in the regular expression causes the .match() method to return all matches in the string in an array.

This example displays "ain,ain,ain" in a message box.

Case Sensitivity

By default regular expression matches are case sensitive. The example below matches all instances of the characters "ain" in the string, regardless of the case of the letters in the character group.

```
<script>
let strTarget = "It's a pAin on a traIn in the raiN.";
let regexp = /ain/gi;
let res = strTarget.match(regexp);
alert(res);
</script>
```

In addition to the control character g, we've added the control character i (case insensitivity) in the regular expression. This causes the .match() method to return all matches in the string regardless of case.

This example displays "Ain,aIn,aiN" in a message box.

Matching Whitespace

The example below ignores "trainintherain" (no spaces) in the string "trainintherain train in the rain." and matches "train in the rain".

```
<script>
let strTarget = "trainintherain train in the rain.";
let regexp = /train\sin\sthe\srain/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

Note the placement of the \s escape characters in the regular expression pattern. The backslash in the escape character tells the expression interpreter that the s is not a regular s character, but has special meaning. In this case the special meaning is to match a space character.

This example displays "train in the rain" in a message box.

Matching Special Characters

The example below matches the pattern "(train)" in the string "the rain (train) in the rain."

```
<script>
let strTarget = "the rain (train) in the rain.";
let regexp = /\(train\)/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

I mentioned escape characters in the previous example. Shown below is a table of commonly used escape characters.

\d	Matches any digit
\D	Matches any non-digit
\s	Matches any whitespace character
\S	Matches any non-whitespace character
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character

In addition to these escape characters, any characters that have special meaning to the regular expression interpreter, if that you want to match them, must be escaped. Shown below is a list of these special characters.

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

Note that we wanted to match the (and) parenthesis, so in the regular expression pattern we had to escape them. This example displays "(train)" in a message box.

How Many Matches?

CONTENTS

In this chapter you'll learn how to use regular expressions to count the number of matches. The example below will match the character a in the string "same sales sample".

```
<script>
var strTarget = "same sales sample";
var num = strTarget.match(/a/).length;
alert(num);
</script>
```

The code creates a text string "same sales sample" and then passes the regular expression /a/ to the JavaScript String Object method <i>match</i>. Even though we can see that there are 3 a's in the string, the code will return 1 because it will match only the first occurrence. To make it count all occurrences in the entire string we have to use the global identifier g, as shown below.

```
<script>
var strTarget = "same sales sample";
var num = strTarget.match(/a/g).length;
alert(num);
</script>
```

This time the code will return 3. We can search for more than single characters. The example below searches for occurrences if "ing" in the sentence "same sales sample".

```
<script>
var strTarget = "From beginning to ending it's
exciting.";
var num = strTarget.match(/ing/g).length;
alert(num);
```

```
</script>
```

The code will return 3. However the code shown below will return 2.

```
<script>
var strTarget = "From beginning to ENDING it's
exciting.";
var num = strTarget.match(/ing/g).length;
alert(num);
</script>
```

That's because by default the match is case sensitive. We solve that by adding the i switch to make the search case insensitive, as shown below.

```
<script>
var strTarget = "From beginning to ENDING it's
exciting.";
var num = strTarget.match(/ing/ig).length;
alert(num);
</script>
```

Now let's suppose we want to count the number of "oo" character sequences in the string "look smooth soon". We could use the code shown below.

```
<script>
var strTarget = "look smooth soon";
var num = strTarget.match(/o/g).length;
alert(num);
</script>
```

The code will return 6. That's because it counted successive o's. One way to fix that is to use the + specifier as shown below.

```
<script>
var strTarget = "look smooth soon";
var num = strTarget.match(/o+/g).length;
```

```
alert(num);
</script>
```

The code will return 3. The + specifier means the character can occur one or more times in each group. So successive characters will not be counted.

```
<script>
var strTarget = "same sales sample";
var num = strTarget.match(/\s/g).length;
alert(num);
</script>
```

One thing regular expression matches are frequently used for is countings spaces in text. This can be done with the code shown above. In the regular expression the back-slash is an escape character. It means the character following it is actually a code. The escape character \s means match any whitespace character. That includes tab (\t), new line (\n) and carriage return (\r).

The escape character \S means match any non-whitespace character.

The code shown below will return 3. That's because there is an extra space in "same sales sample" and the regular expression counts successive whitespace characters.

```
<script>
var strTarget = "same  sales sample";
var num = strTarget.match(/\s/g).length;
alert(num);
</script>
```

Again, we can solve this using the + specifier. The regular expression shown below returns 2.

```
<script>
var strTarget = "same  sales sample";
var num = strTarget.match(/\s+/g).length;
```

```
alert(num);
</script>
```

Why would we need a regular expression that match any whitespace character including tab (\t), new line (\n) and carriage return (\r)? In other words, how do we create a long text string that will fix in a small message box? This can be done as shown below.

```
<script>
var strTarget = "We hold these truths to be self-
evident,\n\
that all men are created equal, that they are endowed
by\n\
their creator with certain unalienable rights, that
among\n\
these are life, liberty and the pursuit of happiness.";

alert(strTarget);
</script>
```

Note the characters on the end of each string (except the last string). The \n escape character causes a new line, and the backslash \ at the end of each line tells JavaScript that the string will continue on the next line. This method is slightly more efficient then using the concatenation character (+) at the end of each line.

```
<script>
var strTarget = "We hold these truths to be self-
evident,\n\
that all men are created equal, that they are endowed
by\n\
their creator with certain unalienable rights, that
among\n\
these are life, liberty and the pursuit of happiness.";

var num = strTarget.match(/\s/g).length;
alert(num);
</script>
```

Now, the code shown above will count all spaces and new line characters, which can be used to get an accurate word count. It will return 34.

Matching a Set of Characters or Numbers

In this chapter you'll learn how to use regular expressions to match a set of of characters. The example below will match uppercase A-Z characters in the string "We hold these Truths to be self-evident, that All Men are Created Equal..."

```
<script>
let strTarget = "We hold these Truths to be self-evident,
that All Men are Created Equal...";
let regexp = /[A-Z]/g;

let result = strTarget.match(regexp);
alert(result);
</script>
```

To match a set of of characters, create a "character set" or "character class". Place the characters you want to match between square brackets. You can use a hyphen inside a character class to specify a range of characters, for example [A-Z].

In the example above, I store the regular expression /[A-Z]/g in a variable named regexp. I pass that variable to the JavaScript String object *match* method. The g specifier causes the regular expression to search for matches through the entire string.

The String object match method stores the returned matches in a variable named *result*. If there are no matches, the match method will return null. I use the *alert* method to display the content of the variable *result*.

This causes the String match method to return W,T,A,M,C,E.

• In previous examples, I used the keyword *var* to declare variables. Assuming that you will be programming for newer browser versions, the keyword *let* would be a better choice to declare variables. The difference between var and let is that var is function scoped and let is block scoped. This means that let provides much tighter scoping. (in programming a variables scope is the part of the code in which it will be visible).

Also you can actually use variables in JavaScript without declaring them first, which is bad programming practice, while variables declared with let are not accessible before they are declared.

In the example below, the regular expression /[aeiou]/g causes the String match method to return all lowercase vowels.

```
<script>
let strTarget = "We hold these Truths to be self-evident,
that All Men are Created Equal ...";
let regexp = /[aeiou]/g;

let result = strTarget.match(regexp);
alert(result);
</script>
```

This causes the String match method to return e,o,e,e,u,o,e,e,e,i,e,a,e,a,e,e,a,e,u,a.

In the example below, the regular expression /[A-Ma-m]/ will match all uppercase or lowercase letters from the first half of the alphabet.

```
<script>
let strTarget = "We hold these Truths to be self-evident,
that All Men are Created Equal ...";
let regexp = /[A-Ma-m]/g;

let result = strTarget.match(regexp);
alert(result);
</script>
```

This causes the String match method to return e,h,l,d,h,e,e,h,b,e,e,l,f,e,i,d,e,h,a,A,l,l,M,e,a,e,C,e,a,e,d,E,a,l.

In the example below, the caret character ^ at the beginning of the character group regular expression means match everything NOT in the list. In this case it will NOT match any characters in the first half of the alphabet. With the i switch for ignore case and it will also not match , (commas) and . (periods) and \s (blank space characters.

```
<script>
let strTarget = "We hold these Truths to be self-evident,
that All Men are Created Equal ...";
let regexp = /[^a-m,.\s]/ig;

let result = strTarget.match(regexp);
alert(result);
</script>
```

This causes the String match method to return W,o,t,s,T,r,u,t,s,t,o,s,-,v,n,t,t,t,n,r,r,t,q,u.

In the example below, to extract a number from the string "extract the number 82501 from this string", the regular expression [0-9] matches a single digit between 0 and 9. The + specifier means the character can occur one or more times.

```
<script>
let strTarget = "extract the number 82501 from this
string";
let regexp = /[0-9]+/;

let result = strTarget.match(regexp);
alert(result);
</script>
```

This causes the String match method to return 82501.

In the example below, signed numbers and unsigned numbers are extracted from the string "extract the number -825 and 501 from this string".

```
<script>
```

```
let strTarget = "extract the number -825 and 501 from
this string";
let regexp = /[-0-9]+/g;

let result = strTarget.match(regexp);
alert(result);
</script>
```

To extract signed numbers and unsigned numbers from a string the regular expression [-0-9] matches a single digit between 0 and 9 and between -0 and -9. The + specifier means the character can occur one or more times. The g specifier means that the regular expression should find matches in the entire string.

This causes the String match method to return -825,501.

In the example below, signed or unsigned floating point numbers are extracted from the string "extract the number -825.501 from this string".

```
<script>
let strTarget = "extract the number -825.501 from this
string";
let regexp = /[-.0-9]+/;

let result = strTarget.match(regexp);
alert(result);
</script>
```

To extract a signed or unsigned floating point numbers from from a string, the regular expression [-0-9] matches a single digit between 0 and 9 and between -0 and -9. The . (dot) matches a decimal point. The + specifier means the character can occur one or more times.

This causes the String match method to return -825.501.

In the example below, regular expression is used to extract the hexadecimal number 0xacc0fd from the string "extract the hexadecimal number 0xacc0fd from this string".

```
<script>
let strTarget = "extract the hexadecimal number 0xacc0fd
from this string";
let regexp = /0[xX][0-9a-fA-F]+/;

let result = strTarget.match(regexp);
alert(result);
</script>
```

The regular expression /0[xX][0-9a-fA-F]+/ matches a set of of characters starting with 0 followed by either a lower or uppercase x, followed by one or more characters in the ranges 0-9, or a-f, or A-F.

The hexadecimal number will need to be prefixed by 0x because hexadecimal digits are just regular alpha characters and so any regular expression can mistake regular text in the string for hexadecimal digits.

Pattern Replacing

CONTENTS

Another function useful in regular expressions is the String.replace() method. You pass a regular expression to the replace() method, and it returns a new string with the pattern matches replaced.

The example below matches the characters "sad" in the string "I am sad and today is a sad day." and replaces them with "happy".

```
<script>
let strTarget = "I am sad and today is a sad day.";
let regexp = /sad/g;
let res = strTarget.replace(regexp, "happy");
alert(res);
</script>
```

The g (global) control character causes all instances of "sad" to be replaced by "happy".

This example displays "I am happy and today is a happy day" in a message box.

Lets say you had some sloppily typed text where there where frequently two or more spaces between words, for example "Today is a good day." The example below matches all instances of two or more spaces and replaces them with a single space.

```
<script>
let strTarget = "Today  is a good  day.";
let regexp = /\s\s+/g;
let res = strTarget.replace(regexp, " ");
alert(res);
</script>
```

The regular expression consists of two \s escape characters which define spaces (white characters), the second one followed by a + control character which matches one or more of the previous character. The g control character causes a global search.

This example displays "Today is a good day." in a message box. Note it contains only single spaces between words.

Sometimes you have a space or spaces at the start of a string that you want to remove. Most programming languages have "String.trim" functions to accomplish that, but you can also do it with a regular expression. The example below trims the spaces from the front of the string " Today is a good day."

```
<script>
let strTarget = "   Today is a good day.";
let regexp = /^\s+/;
let res = strTarget.replace(regexp, "");
alert(res);
</script>
```

The regular expression begins with a ^ control character which means "match the beginning of the string". Then it has the \s escape character followed by a + control character which matches one or more spaces.

This example displays "Today is a good day." in a message box. Note it has no spaces at the beginning.

Similarly you may have a space or spaces at the end of a string that you want to remove. The example below trims the spaces from the end of the string "Today is a good day. "

```
<script>
let strTarget = "Today is a good day.   ";
let regexp = /\s+$/;
let res = strTarget.replace(regexp, "");
alert(res);
</script>
```

The regular expression ends with a $ control character which means "match the end of the string".

This example displays "Today is a good day." in a message box. Note it has no spaces at the beginning.

In this chapter you learned how to use the String.replace() method for regular expressions, and you learned how to remove extraneous white space from a string.

Intervals

Intervals tell us about the number of occurrences of a character in a string. Curly braces are used to define a precise count of how many occurrences you are checking for.

Expressions Intervals Syntax

{n}	Matches the preceding character exactly n times
{n,m}	Matches the preceding character a minimum of n times and a maximum of m times
{n,}	Matches the preceding character n times or more
{,n}	Matches the preceding character 0 times or more

The example shown below checks for occurrences of two lower-case s in the string "Stressed is desserts spelled backwards."

```
<script>
let strTarget = "Stressed is desserts spelled
backwards.";
let regexp = /s{2}/g;
let res = strTarget.match(regexp);
alert(res.length);
</script>
```

The expression s{2} defines to match lower case s where it appears exactly 2 times. The /g controls specifies to search the entire string for occurrences. The match .method searches the string for a match against a regular expression, and returns the matches, as an Array object. The array .length property is displayed by a message box.

This example displays the number 2 in the message box, indicating that it found 2 matches with ss in the string.

The example shown below matches occurrences of lower case s where they appear 1 time or 2 times in the string "Stressed is desserts ssspelled backwardss.", but not where they appear more than 2 times.

```
<script>
let strTarget = "Stressed is desserts ssspelled
backwardss.";
let regexp = /s{1,2}/g;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The expression s{1,2} defines to match lower case s where it appears 1 time or 2 times.

This example displays ss,s,ss,s,ss,s,ss in the message box.

The example shown below matches occurrences of lower case s where they appear 2 or more times in the string "Stressed is desserts ssspelled backwardss."

```
<script>
let strTarget = "Stressed is desserts ssspelled
backwardss.";
let regexp = /s{2,}/g;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The expression s{2,} defines to match lower case s where it appears 2 times or more.

This example displays ss,ss,sss,ss in the message box.

The example shown below matches occurrences of lower case s where it appears 2 or more times in the string "treed desert pelled backward."

```
<script>
```

```
let strTarget = "treed deert pelled backward.";
let regexp = /s{2}/g;
let res = strTarget.match(regexp);

if(res === null) alert(0);
else alert(res.length);
</script>
```

Because the string does not contain an occurrence where they lower case s appears 2 or more times, the .match method will return null. The code if(res === null) tests for the null value, and if found displays 0 in the message box.

Note: the === comparison operator tests both the value and the type of the value, so this test will pass ONLY for null, not pass for "", undefined, false, 0, or NaN.

You might want to test the example by changing the word "deert" to "dessert". Then the message box will display 1. This testing for the null value is useful for many regular expression code statements.

Boundaries

CONTENTS

A Boundary controls where a regular expression matches a string, on a word or non-word boundary, or at the beginning or end of a string, or at the beginning or end of a line.

Word Boundary Control Character

The \b control character starts matching at:

• a boundary between a non-alpha character and an alpha character,
• or a boundary between an alpha character and a non-alpha character,
• or a beginning or end of a string (called zero-length character).

These are referred to as *word boundaries*.

The example below replaces "people" in the word "peopleare" because it finds the zero-length character at the beginning of the string and it finds the non-alpha character (space) after "peopleare".

```
<script>
let strTarget = "peopleare governed by thepeoplefor
thepeople2";
let res = strTarget.replace(/\bpeople/g, "trees");
alert(res);
</script>
```

This example displays "treesare governed by thepeoplefor thepeople2" in a message box.

The example below replaces "people" at the end of the string because it finds the boarder between the word "people" and the zero-length character end of the string.

```
<script>
let strTarget = "thepeopleare governed by thepeoplefor
people";
let res = strTarget.replace(/\bpeople/g, "trees");
alert(res);
</script>
```

This example displays "thepeopleare governed by thepeoplefor trees" in a message box.

The example below replaces the word "people" in the middle of the string because it finds the non-alpha character (space) before "people".

```
<script>
let strTarget = "thepeopleare governed by people
thepeople2";
let res = strTarget.replace(/\bpeople/g, "trees");
alert(res);
</script>
```

This example displays "thepeopleare governed by trees thepeople2" in a message box.

The NOT Word Boundary Control Character

The \B control character is the opposite of \b. \B matches at every position where \b does not.

The example below replaces the word "people" in "thepeoplefor" and in "thepeople2".

```
<script>
let strTarget = "peopleare governed by thepeoplefor
thepeople2";
var res = strTarget.replace(/\Bpeople/g, "trees");
alert(res);
</script>
```

This example does NOT replace "people" in "peopleare" because unlike \b, \B ignores the boundary between a zero-length character and a alpha character at the beginning of a string.

It does replace "people" in the word "thepeoplefor" because unlike \b, \B does start matching at a boundary between two alpha characters.

Similarly, it does replace "people" in "thepeople2" at the end of the string because unlike \b, \B does start matching at a boundary between two alpha characters.

This example displays "peopleare governed by thetreesfor thetrees2" in a message box.

Multiline Mode

The m modifier sets multiline mode. The example below matches "the" at the beginning of each line.

```
<script>
let strTarget = "the people are governed by\nthe people
for\nthe people";
let regexp = /^the/gm;
let res = strTarget.match(regexp);
alert(res);
</script>
```

Strings are delimited by a \n (newline) character. When in multiline mode It will understand the beginning (^) and end ($) characters to mean the beginning or end of each line of a string.

As in any regular expression, without the "g" (global) control character it will stop after the first match. Without the "i" (case-insensitive) control character it will be a case-sensitive match.

This example displays "the,the,the" in the message box.

In this chapter you learned how to controls where a regular expression matches a string, on a word or non-word boundary, at the beginning or end of a string. You also learned about multiline mode.

Alternation

CONTENTS

Alternation allows allows a match with one or more patterns in a regular expression. The alternate patterns are separated by a pipe character referred to the alternation operator. The syntax for alternation is shown below.

/pattern1|pattern2|pattern3/

In the example below, the name of one or more programming languages in the sentence is matched and displayed in a message box.

```
<script>
let strTarget = "The JavaScript programming language is
my favorite.";
let regexp = /VBScript|Python|JavaScript/gi;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The regular expression defines an alternation containing the words VBScript, Python, and JavaScript. Note that the words are separated by a pipe character.

This example displays "JavaScript" in a message box. Try changing the sentence by replacing JavaScript with your favorite programming language.

In the example below, the names of the kinds of pets in the sentence is matched and displayed in a message box.

```
<script>
let strTarget =  "What kind of pets do you have, cats,
dogs, fish or birds.";
let regexp = /cats|dogs|fish|birds/g;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The regular expression defines an alternation containing the words cats, dogs, fish, and birds. Given the g (global) control character, the alternation regular expression will return, in an array, all the matches it finds to every alternate pattern.

This example displays "cats,dogs,fish,birds" in a message box.

In this chapter you learned how to create an alternation regular expression that will match alternate patterns in a string.

Subexpressions

CONTENTS

In case you are not familiar with how to execute code like that in this article, you open a new text file (use a basic ASCII text editor like Windows Notepad, not a word processor. Word processors add formatting characters to the text). Paste the code into the text file. Save the text file with a name that has the file extension .htm. (say test.htm) Double-click on the file to open it in your web browser.

A regular expression subexpression is created by enclosing a separate match in parenthesis. Each subexpression is stored temporarily in memory. You can then access reference each subexpression using a backreference. The example below show how to use subexpressions and backreferences to format names last name first, first name last.

```
<script>
let strTarget = "Robert Sled";
let regexp = /(\w+)\s(\w+)/;

let newstr = strTarget.replace(regexp, "$2, $1");
alert(newstr);
</script>
```

The subexpressions are numbered $1 to $9. $1 will reference the first match in parenthesis, $2 will reference the second match in parenthesis and so on. The $x value persist until another regular expression is encountered.

The example uses the replace() method to switch the words in the string. It references the value in $2 (last name) first, and then the value in $1 (first), placing a comma between the two names. The result of the replace() method is stored in the variable newstr, which is displayed in a message box using the alert() method.

In case you need an explanation of the regular expression, the \w control character matches any character, the + operator indicates one or more occurrences of the previous control character. the \s control character matches one white space character.

The output of this example will display "Sled, Robert". The example below uses the same technique to re-arrange the words in the sentence "dog jumps over cat".

```
<script>
let strTarget = "dog jumps over cat";
let regexp = /(\w+)(\s.*\s)(\w+)/;

let newstr = strTarget.replace(regexp, "$3 $2 $1");
alert(newstr);
</script>
```

The regular expression is also similar, except it contains three subexpressions referenced by the ids $1 $2 $3. Subexpression $2 uses the .* control characters to match any number of characters. The replace() method references the subexpression values in reverse, resulting in the display of the message "cat jumps over dog".

The example below uses a subexpression to extract the temperature numbers from the sentence "Temperature is 26 degrees C today", uses a formula to convert that value from Celsius to Fahrenheit, and displays the same message with the Fahrenheit temperature.

```
<script>
let strTarget = "Temperature is 26 degrees C today";
let regexp = /(.*)\s([0-9]+)\s(degrees)\s(\w)\s(\w+)/;

let tempC = strTarget.replace(regexp, "$2");
let tempF = (tempC*9/5) + 32;

let newstr = strTarget.replace(regexp, "$1 " + tempF + "
$3" + ' F ' + "$5");
alert(newstr);
</script>
```

The temperature numbers are saved in the subexpression with the id $2. The replace() method references the subexpression's value and stores it in a variable named tempC. The $n subexpressions are static, read-only properties, so they can't be modified directly.

The value in tempC is used in the Celsius to Fahrenheit conversion formula. The resulting Fahrenheit temperature value is saved in a variable named tempF. The replace() method is then used a second time to reconstruct the sentence, replacing the temperature value and the units identifier character with "F", resulting in the display of the message "Temperature is 78.8 degrees F today".

ECMAScript 2018 introduced named groups into JavaScript regexes. The example below uses named groups to change a date in day, month, year format to a date in month, day, year format.

```
<script>
let regexp = /(?<day>\d{2})-(?<month>\d{2})-(?
<year>\d{4})/;
let match = regexp.exec("28-06-2020");
alert(match.groups.month + "-" + match.groups.day + "-" +
match.groups.year);
</script>
```

This example uses the regular expression exec() method to test for a match in a string. You set up the regular expression, and then you pass the string as an argument to the exec() method. If it finds a match, it returns the matched text, otherwise it returns null.

In this chapter you learned how to use subexpressions to rearrange matches in a regular expression and how to replace the value of a subexpression match in the results.

Groups

A group allows you to define a separate part of an expression by enclosing it within parentheses. A group acts as an entity.

An *entity* is a character/number/symbol or multiple of such, that work as a unit to provide information, such as a date, time, email address, or serial number.

There are several kinds of groups: capturing group, non-capturing group, conditional group, and alternative group.

Capturing Group

A *capturing group* is stored by the regular expression processor, and it is given a number, starting with 1, that can be used to reference the group using a backreference.

Shown below is an example of a group that uses the meta character \d to match any single digit. It might be used to extract the number of people from the string.

```
<script>
let strTarget = "We will need 3 people to do the job";
let regexp = /(\d\speople)/;
let res = strTarget.match(regexp);
alert(res[1]);
</script>
```

This example displays 3 in a a message box.

The example below uses capturing groups to extract the the number of people and the number of days, but in the results it reveres to display the number of days first.

It uses the pattern [A-Za-z]+ to match the characters and spaces before the first number. It uses the pattern [a-z]+ to match the characters between the numbers. It uses the pattern [a-z.]+ to match the characters and dot after the second number.

The references $1 and $2 match the two capturing groups in the string and the result is displayed using backreferences.

```
<script>
var strTarget = "We will need 3 people for 6 days to do
the job.";
var regexp = /[ A-Za-z]+(\d)[ a-z]+(\d)[ a-z.]+/;
var res = strTarget.replace(regexp, '$2 days $1 people');
alert(res);
</script>
```

This example displays "6 days 3 people" in a a message box.

Non-capturing Group

A *non-capturing group* is not stored and not given a number. You can still match expressions in a non-capturing group, but you will be matching the whole expression. A non-capturing group is recognized by ?: immediately after the groups opening parentheses.

In the example below we use the array element indexes 1 and 2 returned by the .match method. Since the second number in the string is a non-capturing group, it will not have an element in the returned array. Actually the entire string can be returned in array 0.

```
<script>
const strTarget = "We will need 3 people for 6 days and
1000 dollars to do the job.";
const regex = /[ A-Za-z]+(\d*)[ a-z]+(?:\d*)[ a-z]+
(\d{4})[ a-z.]+/;
const res = strTarget.match(regex);
alert(res[1] + " people" + res[2] + " dollars");
</script>
```

This example displays "3 people1000 dollars" in a a message box.

Named Capturing Group

A *named capturing group* identifies a group with a name. A named capturing group is recognized by ? immediately after the groups opening parentheses. The ? is immediately followed by the group's name within less-than and greater-than characters.

```
(?<name>expression)
```

The name can contain letters and numbers, but must start with a letter. The name can be descriptive, making the patterns easier to understand and more maintainable. The named capturing group can be referenced by its name in a backreference.

```
<script>
let strTarget = "We will need 3 people for 6 days and
1000 dollars to do the job.";
let regex = /[ A-Za-z]+(?<people>\d*)[ a-z]+(?<days>\d*)[
a-z]+(?<cost>\d*)/;
let groups = strTarget.match(regex).groups;
alert(groups.people + " people " + groups.days+ " days "
+ groups.cost + " dollars");
</script>
```

This example displays "3 people 6 days 1000 dollars" in a message box.

Alternation Group

A *alternative group* includes alternate patterns separated by pipe symbols.

```
(pattern 1|pattern 2|pattern 3)
```

The example below contains three groups. Each group contains an alternation that selects either a number or that number as a word.

```
<script>
let strTarget = "3 six 12";
let regexp = /([0-9]+|[A-za-z]+)\s([0-9]+|[A-za-
z]+)\s([0-9]+|[A-za-z]+)/;
let res = strTarget.match(regexp);
alert(res[1] + " " + res[2] + " " + res[3]);
</script>
```

This example displays "3 six 12" in a message box.

Conditional Capturing Group

A *conditional capturing group* includes three patterns; the first is a non-capturing group that defines the condition, that is followed by two capturing groups separated by a pipe (|) symbol.

```
(?(condition)true|false)
```

If the conditional statement is *true*, the first capturing group will be saved and given a reference number. If the conditional statement is *false*, the second capturing group will be saved and given a reference number.

The JavaScript regular expression engine doesn't have support for conditionals, but you can mimic it with a look-ahead where an explicit pattern is provided as the condition.

```
(?(conditional-pattern)yes-pattern|no-pattern)
```

use the regexp.test method, which will return true if the test pattern matches the condition pattern, and false if it does not.

```
<script>
let strTarget = "#ffffff";
let regexp = /^#?([0-9A-F]{3}|[0-9A-F]{6})$/i;
let res = regexp.test(strTarget);
alert(res);
```

```
</script>
```

The above example tests the input string to determine if it is a hexadecimal coded number. If it is, the result returns *true*, if it is not, the result returns *false*.

Backreferences

If you need to match a pattern only when it comes after another pattern (called a lookbehind) or only when it comes before another pattern (called a lookahead) you need these operations, cumulatively called *lookarounds*.

Lookbehind Positive

In the example below, we want to match the price in the string "I purchased 24 item number 6602 for $12 each."

```
<script>
let strTarget = "I purchased 24 item number 6602 for $12
each.";
let regexp = /(?<=\$)\d+/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The syntax for *lookbehind positive* is shown below.

(?<=B)A

It means match expression A, but only where it is preceded by expression B.

In this example expression A is one or more numerical digits, but they are matched only if preceded by expression B, which is a $ sign. Note that the dollar sign is escaped, \$, to make sure it's not confused with the $ end of string control character.

This example displays the number 12 in a message box.

Lookbehind Negative

In the example below, we want to match the quantity in the string "I purchased 24 item number 6602 for $12 each."

```
<script>
let strTarget = "At $9 each, I purchased 24 item number
6602.";
let regexp = /(?<!\$)\d+/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The syntax for *lookbehind negative* is shown below.

(?<!B)A

It means match expression A, only where it is NOT preceded by expression B.

In this example expression A is one or more numerical digits, but they are matched only if NOT preceded by expression B. Expression B is a $ sign followed by a numerical digit. So it will find the first numerical match which is NOT preceded by a $ sign.

This example displays the number 24 in a message box.

Lookahead Positive

In the example below, we want to match the item number in the string "I purchased 24 item number 6602 for $12 each."

```
<script>
let strTarget = "I purchased 24 item number 6602 for $12
each.";
let regexp = /\d+(?=\sfor)/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The syntax for *lookahead positive* is shown below.

A(?=B)

It means match expression A only where expression B follows.

In this example expression A is one or more numerical digits, but they are matched only if followed by expression B. Expression B is a space followed by the word "for". So it will NOT match the quantity 24, it will match the number 6602.

Note that expression A must be followed immediately by expression B, if this wasn't true, the quantity number would be the match because it is also followed by expression B, in fact it would be the first match.

This example displays the number 6602 in a message box.

Lookahead Negative

In the example below, we want to match the quantity in the string "I purchased 24 item number 6602 for $12 each."

```
<script>
let strTarget =  "I purchased 24 item number 6602 for $12
each.";
let regexp = /\d+(?!\sfor)/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

The syntax for *lookahead negative* is shown below.

A(?!B)

It means match expression A only where expression B does NOT follow.

In this example expression A is one or more numerical digits, but they are matched only if NOT followed by expression B. Expression B is a space followed by the word "for". The first match for one or more numerical digits is the quantity, 24, and since it is not followed by a space and the word "for", it is the qualified match.

This example displays the number 24 in a message box.

In this chapter you learned how to match a pattern only when it comes after another pattern (or after not that pattern) or only when it comes before another pattern (or before not that pattern). Lookaround syntax can be difficult to master, but it's very powerful.

Applying Regular Expressions

Validate eMail Address

One use for which regular expressions should be an obvious candidate is email address validation. However, if you search the web you'll find hundreds of solutions, some of them enormously long, and all that fail at some point. The consensus on the web is that there is no 100% regular expression solution for email address validation.

However, I offer you the following solution provided by Rae Lee on stackoverflow.com.

```
<script>
let strTarget = "email@domain.com";
let regexp = /^[\w!#$%&'*+\-/=?\^_`{|}~]+(\.[\w!#$%&'*+\-
/=?\^_`{|}~]+)
*@((([\-\w]+\.)+[a-zA-Z]{2,4})|(([0-9]{1,3}\.){3}[0-9]
{1,3}))$/;
alert(regexp.test(strTarget));
</script>
```

• To use this example you'll need to put the regular expression on a single line. I broke it into two lines to fit in this book.

I've set this up with the regexp.test() method, which returns *true* if there is a match, and *false* if the match fails. The following things cause the example to return false: no dot, double dot, front space, end space, and double ampersand. An email address with none of these flaws causes the regexp.test() method to return true.

Extract a URL From a String

Another handy use for regular expressions is to extract web addresses from text. Again if you search the web you'll find hundreds of solutions, some of them enormously long, not many of them actually work.

I found the following solution provided by naT erraT on stackoverflow.com.

```
<script>
let strTarget = "Follow me
https://twitter.com/StephenBucaro on twitter.";
let regexp = /(?:(?:(?:ftp|http)[s]*:\/\/|www\.)[^\.]+\.
[^ \n]+)/;
let res = strTarget.match(regexp);
alert(res);
</script>
```

In addition to the string in the example code, I tested the following strings: "The site map is found at http://bucarotechelp.com/search/000300.asp this URL"; "My information on amazon https://www.amazon.com/s?k=Stephen+Bucaro is here.";

Compress IPv6 Address

As you may know, IPv4 addresses are 32-bit numeric addresses written in decimal as four numbers separated by dots. Very easy to understand. Since we have run out of IPv4 addresses, The Internet Engineering Task Force came up with IPv6 to replace it. IPv6 addresses are 128-bit written in hexadecimal and separated by colons. It requires 39 characters to write an IPv6 address. So they have come up with a 2-step process to compress them.

Shown below is an example IPv6 address.

2001:0db8:ac10:0000:0000:0000:0000:ffff

Step 1 is omit the leading zeros in each segment.

```
<script>
var IPv6Str = "2001:0db8:ac10:0000:0000:0000:0000:ffff";
var newStr = IPv6Str.replace(/(^|:)0+(?!(?::|$))/g,
"$1");
```

```
alert(newStr);
</script>
```

This example displays the number 2001:db8:ac10:0:0:0:0:ffff in a message box. Note that all leading zeros have been omitted. Many times, this is as far as you want to go, because step 2 compresses consecutive fields of zeros, leaving instead a field of 4 colons. This makes the address shorter, but incomprehensible by human or computer.

```
<script>
var IPv6Str = "2001:db8:ac10:0:0:0:0:ffff";
var newStr = IPv6Str.replace(/(:(?:0:){2,})(?!\S*
(?:\1)0:)/, "::");
alert(newStr);
</script>
```

This example displays 2001:db8:ac10::ffff in a message box. Note that the consecutive fields of zeros (:0:0:0:0:) have been replaced with ::. Step 2 can only be used once on an address because if there were more fields of 4 colons, there is no information about how many fields of zeros are contained in each one.

In this chapter you learned some common applications of regular expressions: email address validation, extracting a URL from a String, and Compressing an IPv6 Compress IPv6 network address.

Appendix

CONTENTS

Regex Object Methods

In JavaScript, regular expressions are *Regex* objects. To create a new RegExp object is to simply use the syntax: regexp = /pattern/. Methods of the Regex object are shown below.

Method	Returns	Description
exec()	array	Executes a search for a match. It returns an array of information or null on a mismatch
test()	true/false	Tests for a match. It returns true or false
match()	array	Returns an array containing the matches or null if no match is found

String Object Methods

Along with the Regex methods, String objects are also used in regular expressions. Methods of the String object are shown below.

Method	Returns	Description
matchAll()	iterator	Returns an iterator containing all the matches
search()	index	Searches for a match in a string. It returns the index of the match, or -1 if no match found
replace()	replacement	Searches for a match in a string, and replaces the matched substring with a replacement substring
split()	array	Breaks a string into an array of substrings

Metacharacters

Metacharacters are characters with a special meaning in a regular expression.

Metacharacter	Description
.	Match a single character, except newline
\w	Match a word character
\W	Match a non-word character
\d	Match a digit
\D	Match a non-digit character
\s	Match a whitespace character
\S	Match a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match, not at the beginning/end of a word
\0	Match a NUL character
\n	Match a new line character
\f	Match a form feed character
\r	Match a carriage return character
\t	Match a tab character
\v	Match a vertical tab character
\xxx	Match a character specified by an octal number
\xdd	Match a character specified by a hexadecimal number
\udddd	Match a Unicode character specified by a hexadecimal number

Quantifiers

Quantifiers are characters that specifys how often that a preceding element is allowed to occur.

Quantifier	Description
?	Match zero or one occurrences of the preceding element
*	Match zero or more occurrences of the preceding element
+	Match one or more occurrences of the preceding element
{n}	Match the preceding item exactly n times
{min,}	Match the preceding item min or more times
{min,max}	Match the preceding item at least min times, but not more than max times

Contents