Getting Started with Regular Expressions

by Julian Robichaux, nsftools.com
originally published on lotususergroup.org, August 2011

String parsing. It's a pain, but all developers have to do it sometime. Maybe you're replacing a substring, or searching for a bit of text, or even just checking a file extension, but at some point you find yourself parsing an arbitrary hunk of text.

For simple searches, you can use InStr() in LotusScript or indexOf() in Java or JavaScript. For simple search-and-replace, you can use Replace() in LotusScript, which says it works with arrays but it works with scalar strings too. Java and JavaScript String objects have a replace() method, but sometimes you might need something with a bit more horsepower than just replacing simple strings. Sometimes you need a Regular Expression.

What's a Regular Expression?
A regular expression is a sort of language that uses pattern-matching in order to search for strings in text. While the simplest case of a regular expression can look exactly like a string — if you want to search for the word "red" in a sentence, your regular expression might simply be "red" — in more complex cases you will need to use things like wildcards, special characters, or groupings to match multiple variations of string patterns.

For example, say you want to find all instances of the words "write", "wrote", or "written" in a sentence. You could do that with three separate searches that all loop through the text, or you could have a single regular expression that handles all three cases and finds the matches in a single method call. In more advanced scenarios, there are some very interesting use-cases for regular expressions for parsing things like phone numbers and email addresses.

Regular expressions have been around for many years in various implementations. Unix and Linux command-line gurus and shell scripters are probably already familiar with them, as are Perl programmers. In modern implementations of Java (1.4+) and JavaScript (pretty much any browser still in common use), you can also use regular expressions natively.

Unfortunately, if you want to use them in LotusScript, you're pretty much stuck with using LS2J to do the processing in Java and pass back the results. There are probably some COM objects or similar tricks you could use as well, but it's not built-in. There is also a Like() function which kinda sorta does expression-matching, but only in a very basic sense.

How About Some Examples?
We'll run through a few rudimentary examples in this article, and then I'll point you to a few places you can go for more help. For our examples, we will parse the following text:

"It was the best of times, it was the worst of times. Time is sometimes like that."

We will also use the variable "str" to represent this string in the examples, so if you want to try it yourself you would always start with:

Java: String str = "It was the best…";
JavaScript: var str = "It was the best…";

Also please note that this article might end up formatted with curly quotes, so if you're copying and pasting the examples you might need to convert them to straight quotes for the strings to work properly in code.

Example 1: Count the occurrences of the string "time"
One way to count how many times the string "time" occurs is just to use indexOf() and loop through the text until you're at the end. But we're using regular expressions here, so instead we'll use the split() method.

The split() method simply "breaks" the original text every time it reaches an occurrence of the regular expression pattern you're looking for, so that you end up with an array of substrings containing the text that is found before and after your pattern match. For example, if you start with the sentence "A red ball, a red shoe, and a red hat" and you split it along the word "red", your resulting array will have 4 items:

"A "
" ball, a "
" shoe, and a "
" hat"

Notice that there are 4 items in the array, resulting from 3 occurrences of the word. Therefore, you can count the number of word occurrences using [array length - 1].

The Java Method
Back to our original example. In Java, looking for the string "time" in the test text, our code starts off looking something like this:

```
String[] arr = str.split("time", -1);
System.out.println("There are " + (arr.length-1) + " matches");
```

However, when you run this code, you'll see there are 3 matches, and yet there are 4 occurrences of the string "time" in the test text: time, time, Time, and sometimes. Which one did we miss?

In this case, we missed the third occurrence ("Time") because by default the regular expression match is **case-sensitive**. This is very important to remember, because it will come back to bite you if you don't take this into account. In order to make the search NOT case-sensitive, there are two ways we can handle it.

The easiest way to handle this specific case is to use the pattern "[Tt]ime" as our regular expression. This says that we can match either "Time" or "time". Running the code again, using this as our expression:

```
String[] arr = str.split("[Tt]ime", -1);
```

You will see that there are now 4 matches, as expected.

Another way to do it in Java is to pre-compile the expression "time" as a case-insensitive pattern. To do this, we can use the following code:

```
String[] arr = Pattern.compile("time", Pattern.CASE_INSENSITIVE).split( str );
System.out.println("There are " + (arr.length-1) + " matches");
```

This will achieve the same result, although this time we created a Pattern object where the entire expression was case-insensitive, not just the first letter.

The JavaScript Method

The JavaScript equivalent of what we started off doing above is this:

```
var arr = str.split( /time/ );
alert("There are " + (arr.length-1) + " matches.");
```

As with Java, we will end up with 3 matches in our result set instead of 4, for the same reason as before. You might also notice that the regular expression pattern is in a somewhat unusual syntax — it was entered as /time/ instead of "time". In JavaScript, that's just how you do it. Surrounding the pattern string with front-slashes tells the JavaScript compiler that it's a regular expression pattern instead of a String.

In order to make the search not case-sensitive, all we have to do is add an "i" at the end of the pattern, like so:

```
var arr = str.split( /time/i );
```

That makes the entire search case-insensitive, and will therefore catch the uppercase "T" in "Time".

Example 2: Count the occurrences of the words “time” or “times”
In the previous example, we found all occurrences of the substring “time” in our test text. However, what if we wanted to only find the word “time”? Or, in fact, we want both the word “time” and “times”, but not inside of another word (like “sometimes”).

There are two considerations here. First, we want both “time” and “times”. This can be handled by a pattern like: time(s?) . That translates into “The string ‘time’ followed by zero or one ‘s’”.

There are also other wildcard options like (s*) for “zero or more ‘s’” and (s+) for “one or more ‘s’”, along with options for specific ranges of numbers of characters (used in situations where you want to match a string with between 3 and 5 numbers in a row or something).

The second consideration is that we only want the <u>word</u> “time(s)”, not occurrences inside other words. To do this, we can use the special “word boundary” character: \b . Putting this before and after an expression in a pattern indicates that any time there is a character that would act as a word break (a space, a period, a comma, etc.), that’s a match.

So our pattern now becomes: \btime(s?)\b

As you might guess, regular expression patterns get more and more cryptic as you keep adding new conditions. Our code from before becomes:

<u>Java</u>
```
        arr = Pattern.compile("\\btime(s?)\\b", Pattern.CASE_INSENSITIVE).split( str );
        System.out.println("There are " + (arr.length-1) + " matches");
```

<u>JavaScript</u>
```
        var arr = str.split( /\btimes?\b/i );
        alert("There are " + (arr.length-1) + " matches.");
```

You’ll notice a few language differences in the syntax above. First, in Java the parenthesis around the (s?) are optional — it works with or without the parenthesis around the “s?”, although I like to use them because it makes it easier to read the pattern string. In JavaScript, the parenthesis will cause the split() method to break the text at the word “time” as well as after the letter “s”. So you should NOT use parenthesis on split() patterns in JavaScript, at least not on the version of Firefox I was testing on.

The other difference is that in Java the word boundary character is “\\b” while in JavaScript it’s just “\b”. This is because Java needs to escape the “\” character in the string used to indicate the pattern, while in JavaScript no escaping is necessary.

The greater lesson here is that you can't necessarily find a regular expression pattern on the Internet, paste it into your code, and expect it to work. Different languages and different regular expression engines have very subtle and non-obvious differences between them, so you really have to do a lot of testing.

Example 3: Replace the word "time" with "thyme"
In our final example, we will replace the word "time" (along word boundaries) with the word "thyme" — maybe because this regular expression is being used to edit a cookbook, I don't know. You can make up your own use-case here.

The Java Method
The Java String object has a replaceAll() method that works well for this task, although we still have the case-sensitive issue to deal with. So we can do either:

```
String s2 = str.replaceAll("\\b[Tt]ime(s?)\\b", "Thyme");
```

or:

```
String s2 = Pattern.compile("\\btimes?\\b", Pattern.CASE_INSENSITIVE).
        matcher(str).replaceAll("Thyme");
```

to return our resulting string:

"It was the best of Thyme, it was the worst of Thyme. Thyme is sometimes like that."

The JavaScript Method
In JavaScript, there is simply a replace() method of the String object that can handle this situation for us. Interestingly, if you pass a String as a parameter for replace(), the method will perform a simple string replace. If you pass a regular expression (surrounded by front-slashes instead of quotes), it will perform a regular expression replace.

The biggest thing to remember here is that you have to add a "g" modifier at the end of the pattern in order to replace ALL the values in the text. Otherwise, it will only replace the first value it finds. The code will look something like this:

```
alert( str.replace( /\btime(s?)\b/gi, "Thyme" ));
```

Notice that the syntax is /pattern/gi , such that "gi" indicates that it's a global search as well as a case-insensitive search. Notice also that in this situation it was perfectly legal to use parenthesis around the "s?" in the pattern, even though it caused problems before in the split() method.

A Few Resources for Learning More
Regular expressions are generally a confusing thing to learn when you're getting started, and we have just barely scratched the surface here with this article and examples. There is a LOT more you can do. A few tutorials and reference pages you might want to look at are:

Oracle Java Regular Expression Tutorial
http://download.oracle.com/javase/tutorial/essential/regex/

Java API Documents for the Pattern class
http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html

JavaScript Regular Expressions (Mozilla)
https://developer.mozilla.org/en/JavaScript/Guide/Regular_Expressions

JavaScript Regular Expressions (Microsoft MSDN)
http://msdn.microsoft.com/en-us/library/6wzad2b2(v=VS.94).aspx