

## Regular Expressions: What are They?

Regular expressions are a tiny, highly specialized programming language. They used to only be familiar to Unix users. Text editors like vi allowed regular expressions-formatted searches. Finally, Microsoft decided to give the same power to us and implemented it in Interdev. Most likely people haven't noticed it when they are using Find in Interdev.

When Microsoft started creating scripting languages for the Windows platform, only JScript contained regular expressions, leaving VBScript alone in the dark. That changed with version 5 of the VBScript engine. To ensure that Visual Basic (VB) developers can use regular expressions, the VBScript regular expressions engine has been implemented as a COM object. This makes them much more powerful, since they can be called from various sources outside of VBScript, such as Visual Basic or C.

Regular expressions provide tools for developing complex pattern-matching and textual search-and-replace algorithms. Any Perl, egrep, awk, or sed developer will tell you that regular expressions are one of the most powerful utilities available for manipulating text and data. By creating patterns to match specific strings, a developer has total control over searching, extracting, or replacing data. In short, to master regular expressions is to master your data.

A regular expression is a series of characters that define a pattern. The pattern is then compared to a target string to see whether there are any matches to the pattern in the target string.

### Patterns

Regular expressions are almost another language by itself. A pattern defines the criteria to search for within a string. Regular expressions can be as simple as plain text, or use a unique language consisting of special characters and modifiers to build these patterns.

### Alphabetic Data:

In a regular expression, the period (.) represents exactly one occurrence of any character other than a new line. Thus, the regular expression `m.n` will be matched not only by `man`, but also by `remind` and `mint`, since each of them contain an `m` and an `n` separated by exactly one letter.

```
m.n --- matches man, men, met, mint, remind
```

To narrow the range of acceptable characters in a regular expression, a character class can be used. This is simply a list of one or more characters surrounded by square brackets `[ ]`, and it is matched by only the characters within the brackets.

```
m[ai]n --- matches man, mint, remind
```

The above expression only matches any string which has a or i in between m and n. Inside the square brackets you could specify the range of characters.

```
m[a-z]n --- matches any alphabetic character between m and n
```

```
m[a-z0-9]n --- matches any alpha-numeric character between m and n
```

If you want to further reduce the criteria, such as m should be the first character, then use the following:

```
^m[ai]n --- matches man, mint
```

A caret (^) at the beginning of the pattern string enforces that the target string should start with the pattern string. But if you put the caret inside the square brackets `[^]`, then it is altogether a different meaning. Caret inside the square bracket means "don't match the characters inside the brackets."

```
m[^ai]n --- matches men, mend, diamond
```

Putting a caret at the beginning of the above expression reduces the search criteria.

```
^m[^ai]n --- matches men, met
```

As caret enforces the pattern to match from the starting of the string, `$` enforces that pattern should match at the end of the string.

```
m[ai]n$ --- matches man, min, cumin
```

So putting caret (^) at the beginning and dollar (\$) at the end forces the string to match the pattern both at the beginning and the end.

```
^m[ai]n$ --- matches man, min
```

```
^m[^ain]$ --- matches men
```

Finally, let's see two more special characters, \* and +. \* represents 0 or more characters.

```
m[O]*n --- matches mn, mon, moon, mooon
```

+ is almost the same as \*. The plus sign assures that there should be one or more characters that match.

```
m[O]+n --- matches mon, moon, mooon
```

+ and \* are used to match to any number of characters. If you want to match only for a certain number of characters, then specify the number of characters inside {}.

```
m[O]{3}n --- matches moon
```

You can even give a range of characters.

```
m[O]{2,4}n --- matches moon, mooon, mooon
```

Here there can be 2 to 4 o's between m and n.

### Numeric Data:

So far we have seen the string patterns. Let's see how to match numeric data. If you want to match the numbers, \d should be used. \d matches one numeric character

```
\d --- matches 1, 23423, abc1, abc1def
```

You can use \*, +, ^, \$, [], {} along with this and they will have the same meaning as with the strings.

So:

```
\d+ --- matches 1, 123, abc1def  
\d* --- matches 1, 123, abc1def, abc  
^\d+$ --- matches 1, 123  
^\d{3}$ --- matches 123, 234  
^\d{2,4}$ --- matches 12, 123, 1234
```

So far we have seen what the patterns look like. There are many more types of patterns. Discussing them is beyond the scope of this article. For further reference about the special characters, visit <http://msdn.microsoft.com/scripting/default.htm?/scripting/jscript/doc/jsgrpregxpsyntax.htm>.

## RegExp Object

Microsoft provided us with the RegExp object. In order to use RegExp object in VB, select the References menu item from the VB Project menu, then check the "Microsoft VBScript Regular Expressions" entry. No reference or CreateObject is required if you want to use it in VBScript. Once the reference is set, the RegExp object is instantiated with the following code:

```
Dim regEx As RegExp  
Set regEx = New RegExp
```

The RegExp object has a property called IgnoreCase. When set to false, it makes the comparison case sensitive.

```
regEx.IgnoreCase = False ' Set case sensitivity.
```

Once the Case option is set, we have to set the pattern and then invoke the Test method of the object with the target string to compare. The Test method returns a Boolean. If the pattern matches the target string, then the Test method returns true, otherwise false.

```
regEx.Pattern = "^\\d{2,4}$" ' Set pattern.  
Validate = regEx.Test("123") ' Execute the search test.
```

In the above code, the pattern is set to be a number that is 2 to 4 characters and the target string is "123," a number of 3 characters in length. Since the pattern matches the target string, the Test method returns true.

There is also an Execute method in the RegExp object that returns Match object and Match collection, which we won't discuss here.

## Patterns to Match User Inputs

### Numeric Data Types:

Below are some sample cases that use regular expressions and its patterns in VB to validate user inputs.

This case uses only positive integers. The pattern would be

```
Positive Integer --- ^\d+$
```

The caret in the beginning and dollar at the end insure that the target string should be a number (\d) and the + enforces there should be at least one numeric character. If you want to test only for negative numbers, then the pattern is

```
Negative Integer --- ^-\d+$
```

Observe the "-" after the caret (^) that forces the user to enter "-" before the numeric character. Finally, if you want to test an integer number that can be positive or negative, then the pattern is

```
Integer --- ^-{0,1}\d+$
```

0, 1 inside the curly brackets {} tells the pattern to have 0 or 1 "-" before the numeric character.

Extend the same logic to match the numbers. (A number can have a decimal; and if it has decimal, then it can have only one decimal and there should be at least one numeric character after the decimal.)

The pattern to match a positive number is

```
Positive Number --- ^\d*\.{0,1}\d+$
```

\d\* means there can be 0 or more numeric characters before a decimal. \ represents a period (or decimal). As a period has special meaning in regular expressions, \ (backslash) overrides its special meaning so that a period can be matched. The curly brackets next to the period forces it to have only one decimal or no decimal. And the \d+ at the end means there should be at least one numeric character after the decimal.

The pattern for checking for a negative number and number (can be positive or negative) is

```
Negative Number --- ^-\d*\.{0,1}\d+$
```

```
Positive Number or Negative Number --- ^-{0,1}\d*\.{0,1}\d+$
```

A zip code can have 5 digits (99999) or can have a 5-4 (99999-9999) pattern. The two possible cases can't be represented by one pattern. Based on whether there is "-" in the zip code, use the pattern which suits. This is illustrated by the following code. (Here strValue is the target string and strPattern is the pattern.)

```
Zip Code
```

```
'it can be a 5 digits(99999) Or 5-4(99999-9999) pattern
```

```
If InStr(strValue, "-") = 0 Then
```

```
    strPattern = "^{\d}{5}$"
```

```
Else
```

```
    strPattern = "^{\d}{5}-{\d}{4}$"
```

```
End If
```

Validating a Social Security Number is almost similar to the zip code. A social security number can also be represented by two forms (999-99-9999 Or 999999999). The code to match this pattern is

```
Social Security Number
```

```
'it should of the pattern 999-99-9999 Or 999999999
```

```
If InStr(strValue, "-") = 0 Then
```

```
    strPattern = "^{\d}{9}$"
```

```
Else
```

```
    strPattern = "^{\d}{3}-{\d}{2}-{\d}{4}$"
```

```
End If
```

### Alphabetic Data Types:

Below are patterns for validating alphabetic data types. If you have a first-name field in your form, then that field should be validated for alphabets only. (In one of my applications, customer service received an email saying that a first name was not accepted by the form because it contained some numbers.) The pattern to match only alphabets follows:

```
Alphabets --- ^[a-zA-Z]+$
```

Instead of a first name, if you have a name field that takes first name and last name, then you should also allow space for the space between the two names.. That pattern string is

```
Alphabets with space --- ^[a-zA-Z ]+$
```

The pattern for an alphanumeric string (alphabets and numbers) is

```
AlphaNumeric --- ^[a-zA-Z0-9]+$
```

The following is a tough pattern. The typical email pattern looks like name@domain.extension. The name must start with an alphabetic character and can contain a number or a special character, such as period (.), underscore (\_), and dash (-), but it should not end with special characters. So the pattern to match the name in the email address is "[a-zA-Z][\w\.-]\*[a-zA-Z0-9]"

Domain name follows the same rules.

The extension cannot have special characters such as "-" and "\_". Putting all of these together will give the pattern for an email address.

Email --

```
^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z][\w\.-]*[a-zA-Z0-9]\.[a-zA-Z][a-zA-Z0-9]*[a-zA-Z]$
```

Nagaraj.NET