Microsoft Beefs Up VBScript with Regular Expressions

Vernon W. Hui
Microsoft Corporation

May 10, 1999

Contents

Wow! Talk about living in Internet time. Only nine months ago, the Microsoft® Scripting Technologies team released Version 4.0 of its scripting engines in Visual Studio® 6.0. Now, with the release of Internet Explorer 5.0, we roll out version 5.0 of the script engines—with lots of new bells and whistles! Some welcome improvements include HTML and ASP performance improvements, script encoding, and many Visual Basic® Scripting Edition (VBScript) and JScript® language features. Here's some insight into one of the most often requested features in VBScript—regular expressions!

Regular Expressions: What the Heck Are They?

So what are regular expressions? Regular expressions provide tools for developing complex pattern matching and textual search-and-replace algorithms. Ask any Perl, egrep, awk or sed developer, and they'll 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.

In this article, I'll describe all objects related to VBScript Regular Expressions, summarize common regular expression patterns, and provide some examples of using regular expressions in code.

VBScript RegExp Object

VBScript Version 5.0 provides regular expressions as an object to developers. In design, the VBScript **RegExp** object is similar to JScript's **RegExp** and **String** objects; in syntax it is consistent with Visual Basic. First, let me describe the object and its properties and methods. The VBScript **RegExp** object provides 3 properties and 3 methods to the user.

Properties	Methods

Pattern	Test (search-string)
IgnoreCase	Replace (search-string, replace-string)
Global	Execute (search-string)

- **Pattern** - A string that is used to define the regular expression. This must be set before use of the regular expression object. **Patterns** are described in more detail below.
- **IgnoreCase** - A Boolean property that indicates if the regular expression should be tested against all possible matches in a string. By default, IgnoreCase is set to False.
- **Global** - A Boolean property that indicates if the regular expression should be tested against all possible matches in a string. By default, **Global** is set to **False**.
- **Test** (string) - The **Test** method takes a string as its argument and returns True if the regular expression can successfully be matched against the string, otherwise False is returned.
- **Replace** (search-string, replace-string) - The **Replace** method takes 2 strings as its arguments. If it is able to successfully match the regular expression in the search-string, then it replaces that match with the replace-string, and the new string is returned. If no matches were found, then the original search-string is returned.
- **Execute** (search-string) - The **Execute** method works like **Replace**, except that it returns a **Matches** collection object, containing a **Match** object for each successful match. It doesn't modify the original string.

For more information, including some sample code, see the Microsoft Scripting Site.

VBScript Matches Collection Object

As I mentioned, the **Matches** collection object is only returned as a result of the **Execute** method. This collection object can contain zero or more **Match** objects, and the properties of this object are read-only.

Properties
Count
Item

- **Count** - A read-only value that contains the number of **Match** objects in the collection.
- **Item** - A read-only value that enables **Match** objects to be randomly accessed from the **Matches** collection object. The **Match** objects may also be incrementally accessed from the **Matches** collection object, using a For-Next loop.

For more information, including some sample code, see the Microsoft Scripting Site.

VBScript Match Object

Contained within each **Matches** collection object are zero or more **Match** objects. These represent each successful match when using regular expressions. The properties of these objects are also read-only, and contain information about each match.

Properties
FirstIndex
Length
Value

- **FirstIndex** - A read-only value that contains the position within the original string where the match occurred. This index uses a zero-based offset to record positions, meaning that the first position in a string is 0.
- **Length** - A read-only value that contains the total length of the matched string.
- **Value** - A read-only value that contains the matched value or text. It is also the default value when accessing the **Match** object.

For more information, including some sample code, see the Microsoft Scripting Site.

So What Does a Pattern Look Like?

Ok, this all sounds great and wonderful, but what do they look like? Regular expressions are almost another language by itself, but users familiar with Perl will feel right at home. VBScript derives its pattern set from Perl, and its core features are, therefore, similar to Perl. I'll try to describe some of the pattern sets used to define regular expressions. These sets can be broken down into several categories and areas:

Position Matching

Position matching involves the use of the ^ and $ to search for beginning or ending of strings. Setting the pattern property to "^VBScript" will only successfully match "VBScript is cool." But it will fail to match "I like VBScript."

Symbol	Function
^	Only match the beginning of a string. "^A" matches first "A" in "An A+ for Anita."
$	Only match the ending of a string. "t$" matches the last "t" in "A cat in the hat"
\b	Matches any word boundary "ly\b" matches "ly" in "possibly tomorrow."
\B	Matches any non-word boundary

Literals

Literals can be taken to mean alphanumeric characters, ACSII, octal characters, hexadecimal characters, UNICODE, or special escaped characters. Since some characters have special meanings, we must escape them. To match these special characters, we precede them with a "\" in a regular expression.

Symbol	Function
Alphanumeric	Matches alphabetical and numerical characters literally.
\n	Matches a new line
\f	Matches a form feed
\r	Matches carriage return
\t	Matches horizontal tab
\v	Matches vertical tab
\?	Matches ?
*	Matches *
\+	Matches +
\.	Matches .
|	Matches |
\{	Matches {
\}	Matches }
\\	Matches \
\[	Matches [
\]	Matches]
\(	Matches (
\)	Matches)
\xxx	Matches the ASCII character expressed by the octal number xxx. "\50" matches "(" or chr (40).

\xdd	Matches the ASCII character expressed by the hex number dd. "\x28" matches "(" or chr (40).
\uxxxx	Matches the ASCII character expressed by the UNICODE xxxx. "\u00A3" matches "£".

Character Classes

Character classes enable customized grouping by putting expressions within [] braces. A negated character class may be created by placing ^ as the first character inside the []. Also, a dash can be used to relate a scope of characters. For example, the regular expression "[^a-zA-Z0-9]" matches everything except alphanumeric characters. In addition, some common character sets are bundled as an escape plus a letter.

Symbol	Function
[xyz]	Match any one character enclosed in the character set. "[a-e]" matches "b" in "basketball".
[^xyz]	Match any one character not enclosed in the character set. "[^a-e]" matches "s" in "basketball".
.	Match any character except \n.
\w	Match any word character. Equivalent to [a-zA-Z_0-9].
\W	Match any non-word character. Equivalent to [^a-zA-Z_0-9].
\d	Match any digit. Equivalent to [0-9].
\D	Match any non-digit. Equivalent to [^0-9].
\s	Match any space character. Equivalent to [\t\r\n\v\f].
\S	Match any non-space character. Equivalent to [^ \t\r\n\v\f].

Repetition

Repetition allows multiple searches on the clause within the regular expression. By using repetition matching, we can specify the number of times an element may be repeated in a regular expression.

Symbol	Function
{x}	Match exactly x occurrences of a regular expression. "\d{5}" matches 5 digits.
{x,}	Match x or more occurrences of a regular expression. "\s{2,}" matches at least 2 space characters.
{x,y}	Matches x to y number of occurrences of a regular expression. "\d{2,3}" matches at least 2 but no more than 3 digits.
?	Match zero or one occurrences. Equivalent to {0,1}. "a\s?b" matches "ab" or "a b".
*	Match zero or more occurrences. Equivalent to {0,}.
+	Match one or more occurrences. Equivalent to {1,}.

Alternation & Grouping

Alternation and grouping is used to develop more complex regular expressions. Using alternation and grouping techniques can create intricate clauses within a regular expression, and offer more flexibility and control.

Symbol	Function
()	Grouping a clause to create a clause. May be nested. "(ab)?(c)" matches "abc" or "c".
|	Alternation combines clauses into one regular expression and then matches any of the individua "(ab)|(cd)|(ef)" matches "ab" or "cd" or "ef".

BackReferences

Backreferences enable the programmer to refer back to a portion of the regular expression. This is done by use of parenthesis and the backslash (\) character followed by a single digit. The first parenthesis clause is referred by \1, the second by \2, etc.

Symbol	Function
()\n	Matches a clause as numbered by the left parenthesis

	"(\w+)\s+\1" matches any word that occurs twice in a row, such as "hubba hubba."

Gimme an Example Already!

This example covers some of the things I've discussed in this article. It's a simple app that makes use of regular expressions to test if a valid input has been entered. It will repeatedly prompt the user for input until a valid input has been entered. First, I'll explain the initial pattern in detail.

- "^\s*((\$\s?)|(£\s?))?((\d+(\.(\d\d)?)?)|(\.\d\d))\s*(UK|GBP|GB|USA|US|USD)?)\s*$"
- "^\s*..." and "...\s*$" - means that there can be any number of leading and end space characters, and the input must be on a line by itself
- "((\$\s?)|(£\s?))?" - means an optional $ or £ sign followed by an optional space
- "((\d+(\.(\d\d)?)?)|(\.\d\d))" - searches for at least one digit, followed by an optional decimal and two digits (which are optional) or a decimal and two digits. This means that input such as 6., 23.33, .88 are all allowed, but 5.5 is not.
- "\s*(UK|GBP|GB|USA|US|USD)?" - means that any number of space characters are valid followed by optional and acceptable arguments to the string.

In this example, regular expressions are used to determine if the user entered US dollars or British pounds. I search for the strings £, UK, GBP, or GB. If the regular expression is true, then the user has entered an amount in British pounds. Otherwise, I assume USD currency.

To run this code, you can save it as CurrencyEx.vbs and run it using Windows Script Host, copy it to VB (need to add references to Microsoft VBScript Regular Expressions) or embed the code in an HTML file.

```
Sub CurrencyEx
Dim inputstr, re, amt
Set re = new regexp  'Create the RegExp object

'Ask the user for the appropriate information
inputstr = inputbox("I will help you convert USA and CAN currency. Please enter the
amount to convert:")
'Check to see if the input string is a valid one.
re.Pattern =
"^\s*((\$\s?)|(£\s?))?((\d+(\.(\d\d)?)?)|(\.\d\d))\s*(UK|GBP|GB|USA|US|USD)?)\s*$"
re.IgnoreCase = true
do while re.
Test(inputstr) <> true
'Prompt for another input if inputstr is not valid
inputstr = inputbox("I will help you convert USA and GBP currency. Please enter the
amount to(USD or GBP):")

loop

'Determine if we are going from GBP->US or USA->GBP
re.Pattern = "£|UK|GBP|GB"
if re.Test(inputstr) then
'The user wants to go from GBP->USD

re.Pattern = "[a-z$£ ]"
re.Global = True
```

```
amt = re.Replace(inputstr, "")
amt = amt * 1.6368
amt = cdbl(cint(amt * 100) / 100)
amt = "$" & amt
else
'The user wants to go from USD->GBP

re.Pattern = "[a-z$£ ]"
re.Global = True
amt = re.Replace(inputstr, "")
amt = amt * 0.609
amt = cdbl(cint(amt * 100) / 100)
amt = "£" & amt
end if

msgbox ("Your amount of: " & vbTab & inputstr & vbCrLf & "is equal to: " & vbTab &
amt)
End sub
```

More Power to You!

To ensure that Visual Basic developers can use regular expressions, the VBScript regular expression 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.As an example, I have written a small Visual Basic application that sniffs through your contact list in Outlook® 97, Outlook 98 or Outlook 2000, and returns the names of contacts that live in those particular cities.

This program is simple. First, it prompts the user to enter the names of the cities to search for, separated by commas. Second, it prompts for the name of the new contact folder to create within Outlook. After each contact match, the contact is copied to the newly created contact folder.

By adding references to the Microsoft VBScript Regular Expressions object library we can do some fancy early binding. An early binding object has some advantages. It is faster and coding programs becomes simpler. This is because you can add the reference to your object and then cut and paste code from VBScript directly to VB since "new RegExp" will work right away.

I've also referenced the Outlook 9.0 object library in the same manner as regular expressions, and for the same reasons. Of course, you can still make COM calls using**CreateObject()**, but this proves to be simpler. Afterward creating these objects, it is just simple code that accesses the contact folders and tries to match the name of cities. I have a small helper-function, **compareCollectionObjects(x,y)** that takes 2 collection objects and compares them to see if they are equal.

To try this program, simply copy the code to VB(need to add references) and call the function **FindCityContacts()**.

```
Sub FindCityContacts()

    Dim strTemp
    Dim index
    Dim citySearch
    Dim myNameSpace, myContacts, newCityContacts, newCityContactsName
    Dim contact
```

```
    Dim newContact

    'Set the early binding objects
    Dim re as New RegExp
    Dim myApp as New Outlook.Application

    re.Global = True
    re.IgnoreCase = True

    citySearch = InputBox("Please enter the cities of your search, separated by
commas.")
    newCityContactsName = InputBox("Please enter the new contact folder name")

    'Set some of the objects and create the new Contacts folder
    Set myNameSpace = myApp.GetNamespace("MAPI")
    'olFolderContacts = 10
    Set myContacts = myNameSpace.GetDefaultFolder(10)
    Set newCityContacts = myContacts.Folders.Add(newCityContactsName)

    'Set cities, using regular expressions to contain the city names
    re.Pattern = "[^,]+"
    Set cities = re.Execute(citySearch)
    For Each city In cities

       'Set citytokens to be the individual tokens in the city name
       'Then we compare them to the address tokens in each contact
        re.Pattern = "[^ ]+"
        Set citytokens = re.Execute(city)

        For i = 1 to myContacts.Items.Count
            re.Pattern = "[^ ]+"
            Set contact = myContacts.Items.Item(i)

            Set HomeAddressCityTokens = re.Execute(contact.HomeAddressCity)
            If compareCollectionObjects(HomeAddressCityTokens, citytokens) = 1 Then

                Set newContact = contact.Copy
                newContact.Move newCityContacts
            End If

            Set OtherAddressCityTokens = re.Execute(contact.OtherAddressCity)
            If compareCollectionObjects(OtherAddressCityTokens, citytokens) = 1 Then
                Set newContact = contact.Copy
                newContact.Move newCityContacts
            End If

            Set BusinessAddressCityTokens = re.Execute(contact.BusinessAddressCity)
            If compareCollectionObjects(BusinessAddressCityTokens, citytokens) = 1
Then
                Set newContact = contact.Copy
                newContact.Move newCityContacts
            End If
        Next
    Next
```

```
MsgBox "done"

End Sub

'This function is provided as a helper-function
' to compare two collection objects.
Function compareCollectionObjects(x, y)

    Dim index
    Dim flag
    flag = 1

    If x.Count <> y.Count Then
        flag = 0
    Else
        index = x.Count

        For i = 0 To (index - 1)
            If StrComp(x.Item(i), y.Item(i), 1) Then
                flag = 0
            End If
        Next
    End If

    compareCollectionObjects = flag

End Function
```