



Recursion reminds me of...recursion

Shedding a Little Light on XML

To understand recursion you must first understand recursion. Okay, so maybe it's the oldest recursion joke in the book, but fortunately I don't have to make my livelihood as a comedian. This month's column is dedicated to questions on recursion, in both the XML DOM and XSLT.

re-cur-sion \ri-'kər-zhən\ *n* [mathematics] 1 : SEE RECURSION...

I don't often get questions that specifically ask, "How do I use recursion to...?" But I do get questions in which recursion plays a key role in the solution. The questions in this month's column are prime examples. If I were to guess why people avoid recursion, I'd say it could be because it appears to be the harder road to take. But IMHO it isn't. And at times it may be the only road.

After you see a few questions and demos, including a game of XSLT tic-tac-toe (you vs XSLT recursion), I hope to change the way you view the subject. Many of the answers in this month's column can be truly appreciated only after you download and examine the source code. Here's to shedding some light on recursion...

Q: *Why does XSLT require recursion for what seem to be simple tasks? After all, many other languages can solve the same problems without using recursion.*

A: The prevalent use of recursion is a characteristic of functional languages – languages that don't have side effects like global variables. If you have a language with global variables, you don't need recursion as you can iterate through anything with a changing pointer. If you *don't* have global variables, you have to reintroduce a stack frame with a new set of localized variables with the iterated values and sets (be they sets of nodes or sets of characters).

Functionally (no pun intended), XSLT works the same magic as other functional languages. Essentially, this is because the "call-template" construct opens up a new stack frame with a new set of locally

scoped variables that can be bound with values different from the previous stack frame's locally scoped variables. This allows working with a particular item of a set and then recursively calling the same code with either a subset of the string or a different index into the set.

A forefather of XSLT, DSSSL (Document Style Semantics and Specification Language), is also a functional language and many algorithms need to be implemented recursively. XSLT is a templating language but is, in computing theory, Turing-complete, and the "call-template" construct is no different from a function call (except in its verbose syntax, which can make it awkward to write and review). If you understand the use of recursion, it's only the syntax that gets in the way, making it more difficult than the syntax of imperative languages.

Q: *How do I iterate through a string of characters in XSLT?*

A: You don't (well, not exactly). For this problem the desired result is to place an HTML line break (
) before each uppercase character that's processed. The purpose of this algorithm is to iterate through a string of characters and insert an HTML line break ("
") before each uppercase character. To tackle this type of problem step by step, the first part of the solution is to create a template that will recursively process the string character by character. In fact, the bulk of the solution is contained within this template, called "ProcessString". Listing 1 shows ProcessString.xml, the inner workings of our string processing template. Once it's coded, all that remains is to call the template, passing to it the string that needs

parsing. The following breakdown explains how to pass the string to the template and, more important, what happens within the template.

Solution

Initially, the template is called, passing each string to be parsed. In this implementation a simple "xsl:for-each" is used to feed strings contained within the "SingleString" element to the template. (Full details on ShowProcessString.xml, the recursive XSLT string processing template, are given in Listing 2.)

```
<xsl:template match="AllStrings">
  <xsl:for-each select="SingleString">

    <xsl:value-of select="'Original
      String: '"/>
    <xsl:value-of select="."/ /><br/>
    <xsl:value-of select="'Converted
      String: '"/>

    <!-- Call the template that will
      loop through each character of a
      string -->
    <xsl:call-template name="Iterate">
      <xsl:with-param name="strInput"
        select="."/>
    </xsl:call-template>
  <br/><br/>
</xsl:for-each>
</xsl:template>
```

Once the "SingleString" element is passed to the template, the remaining processing is performed within the "ProcessString" template. Listing 3, ShowProcessString.xml, provides the XML document containing the strings that will be processed. The template will process the entire string, character by character, placing a "
" before the character if it tests true as uppercase. Within each iteration the template calls itself with a new string that is one character shorter until all the characters are exhausted.

AUTHOR BIO

David Silverlight is chief XML evangelist for Infoteria Corporation (www.infoteria.com), an XML software development company based in Tokyo and Beverly, Massachusetts. He also maintains www.XMLPitstop.com, a resource for XML examples and everything else XML.

Q: What is recursion useful for?

A: As part of the source code for this article, a number of other examples demonstrating recursion are included:

- “Splitting XML Elements into Smaller XML Elements” (from Infoteria’s StylesheetCentral)
- “Transformation of Linefeed Characters to HTML” (from Mike J. Brown at Fourthought, Inc.)
- “XSLT Template to Perform Substring Replacements” (also from Mike J. Brown)
- “Trimming the Leading Spaces from a String” (from Infoteria’s StylesheetCentral)
- “Trimming the Trailing Spaces from a String” (from Infoteria’s StylesheetCentral)
- “Returning the Min() or Max() Value from a Nodeset” (from Infoteria’s StylesheetCentral)

The approach used in the foregoing resources is similar to that for the previous question.

Q: How can I load a tree view in Visual Basic with XML data?

A: Finally, a chance to use recursion in VB. Displaying the contents of an XML document as a tree control (with VB, XML DOM, and recursion) can be accomplished without a tremendous amount of effort. Due to the hierarchical nature of XML, it’s a prime candidate for display as a tree view. It’s also a prime candidate for recursive processing. Here’s how the three technologies combine into a single solution.

Overview

There are really two parts to loading the tree control from your XML. The algorithm is rather trivial once you see it.

Breakdown

The first part of the algorithm involves setting up the tree view. In this function the

DOM is initialized, the TreeView control is reset, and the root node is extracted. In the Visual Basic application the ShowScreen-TreeView function performs the environment setup and makes the initial call to the recursive function – PopulateTreeWithChildren – passing it the root node and the TreeView control (see Listing 4).

The second part involves calling PopulateTreeWithChildren, passing the initial root node and a treeview control as parameters. This function does all of the recursive work in processing the XML DOM object and will continue to call itself recursively while objNode.childNodes.length is greater than 1.

During each iteration, contents of the element will be added as a leaf in the tree if it’s the innermost element; otherwise it will be added as a branch of the tree (see Listing 5).

That’s about it. Once again, the recursive function in this VB application will call itself until all nodes are added to the tree. The same concepts described in earlier questions apply.

Some XML/XSL tree views worth checking out:

- Fancy XML tree view
- Crane Softwrights Showtree
- XML TreeView in SVG
- Tree view menu implemented using XML/XSL

Q: What’s the difference between tail-end recursion and embedded recursion?

A: For starters, think of recursion in XSLT as a template that calls itself as a template. Questions then arise: At what point does it call itself? At the beginning of the template? Somewhere in the middle? At the end? The point at which it recursively calls itself could have serious implications on the consumption of resources. Tail end, as has been demonstrated in all of the examples thus far, includes the recursive call at the very end (the “tail” end); embedded recursion makes the call to itself anywhere within.

When determining what constitutes the “very end,” the rule of thumb is that it shouldn’t be the last call written in the

code block, but in the flow of logic the call is the last operation of the block. This means that a single block of code may have many recursive calls, all of which can be characterized as tail-end calls because each is at the end of different flows of logic within the module, not at the end of the syntax of the module.

A drawback to recursive algorithms can be the amount of stack consumed and the indeterminate amount of system resources that may be required by the processor. An XSLT processor that implements tail-end recursion will survive recursive-based algorithms far longer than an XSLT processor, which does not, because tail recursion recognizes that a stack frame is no longer required during a recursive call and disposes of the old stack frame before using the new one, thus reducing the demand for stack space.

Implementation is also an important consideration when coding recursive algorithms. Saxon supports tail-end recursion while XT does not. Other processors will vary. If the coder of the routine has *any* logic after the recursive call, the implementation is obliged to keep the stack frame around and can quickly run out of stack space...hence the burden on the programmer to code carefully.

Elements of Design Tic-Tac-Toe Using XSLT Recursion

Let’s have some fun with XSLT. This month, as a demonstration of recursion and XSLT, I’ve written a game of Tic-Tac-Toe. Here’s the catch. You’ll be using your brain to figure out your moves; the computer will be using pure XSLT recursion. Think of it as a variation on Kasparov vs Deep Blue. Are you up for it? The source code is included in this article.

Are you wondering how I did it? I started with a very common recursion algorithm (<http://erwnerve.tripod.com/tic-tactoe.htm>) and created my first proof of concept in VB6. I followed it up with a secondary version using VBScript and HTML. After the smoke cleared, and with many cups of coffee behind me, as well as a few tears...voilà: XSLT Tic-Tac-Toe. Enjoy.

DSILVERLIGHT @ INFOTERIA.COM

LISTING 1 ProcessString.xsl

```
<?xml version='1.0' encoding='utf-8' ?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>
<xsl:template name="ProcessString">
<!--This template will recursively process a string of
characters and place a <br/> before each uppercase character-->

<xsl:param name="strInput" select="" />

<xsl:variable name="strCurrChar" select=
"substring($strInput, 1, 1)"/>
<xsl:variable name="strRemainingChars" select=
```

```
"substring($strInput, 2)"/>

<xsl:choose>
<!--the test below will be false when strCurrChar is
empty. -->
<xsl:when test="$strCurrChar">

<!-- Test to see if the character is upper case (in a
very cool fashion),
if so, output a <br/> -->
<xsl:if test="not(translate($strCurrChar, 'ABCDE-
FGHIJKLMNQRSTUWXYZ', ''))">
<br/>
```

```

</xsl:if>

<!--BTW, the test above is known as "Ken's trick" (submitted by Ken G. Holman) for a test of an uppercase character. It is impressively clever, but a bit subtle. Here's how it works:
a) Translate() won't touch characters that aren't in the second argument.
b) If the first argument contains only uppercase characters, they are all deleted and nothing else is touched.
c) If the resulting string is empty (meaning every character was uppercase and was deleted), the empty string tests as FALSE and the not() changes it to TRUE.
d) So...the entire "test=" is TRUE if the first argument contains only uppercase characters.
-->

<!-- Output the current character -->
<xsl:value-of select="$strCurrChar"/>

<!-- At this point, the template will recursively call itself with the remaining characters in the string -->
<xsl:call-template name="ProcessString">
  <xsl:with-param name="strInput"
select="$strRemainingChars"/>
</xsl:call-template>
</xsl:when>

</xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

LISTING 2 ShowProcessString.xsl

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html"/>
  <xsl:include href="Iterate.xsl"/>

  <!-- Template for root rule -->
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <!-- Template for AllStrings rule -->
  <xsl:template match="AllStrings">
    <xsl:for-each select="SingleString">

      <xsl:value-of select="'Original String: '"/>
      <xsl:value-of select="."/ ><br/>
      <xsl:value-of select="'Converted String: '"/>

      <!-- Call the template that will loop through each character of a string -->
      <xsl:call-template name="ProcessString">
        <xsl:with-param name="strInput" select="."/>
      </xsl:call-template>
      <br/><br/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>

```

LISTING 3 ShowProcessString.xml

```

<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="ShowProcessString.xsl"?>
<AllStrings>
  <SingleString>OneTwoThreeFour</SingleString>

  <SingleString>MondayTuesdayWednesdayThursdayFridaySaturdaySunday</SingleString>

  <SingleString>JanFebMarAprMayJunJulAugSeptOctNovDec</SingleString>
</AllStrings>

```

LISTING 4 ShowScreenTreeView function

```

Function ShowScreenTreeView() As Boolean

  Dim objXMLelement As IXMLDOMElement
  Dim tvwRoot As Node
  Dim objXMLRoot As IXMLDOMElement
  Dim objDOM As New DOMDocument
  Dim bSuccess As Boolean
  Dim strXML As String

  On Error GoTo Procedure_Err
  strgLastFunction = "ShowScreenTreeView"

  'Assume failure
  bSuccess = False

```

```

'A) Initialize and load the DOM object from the selected filename
  LoadDOMObjectFromFile objDOM, txtInputFile.Text

'B) Remove any existing elements from the TreeView
  ClearTreeView tvwData

'C) Get the root element of the XML - bypassing the comments, PI's etc
  Set objXMLRoot = objDOM.documentElement

'D) Add a child to the root node of the TreeView
  Set tvwRoot = tvwData.Nodes.Add()
  tvwRoot.Text = "<" & objXMLRoot.baseName & ">"
  tvwRoot.Expanded = True

'E) Call the PopulateTreeWithChildren function which will recursively drill
  ' down, starting at the root node, until all child nodes are added
  ' to the tree control
  PopulateTreeWithChildren objXMLRoot, tvwData

  'If you made it to here, you have no errors
  bSuccess = True

Procedure_exit:
  ShowScreenTreeView = bSuccess
  Exit Function

Procedure_Err:
  HandleError Err
  bSuccess = False
  Resume Procedure_exit
  Resume

End Function

```

LISTING 5 PopulateTreeWithChildren() function

```

Dim objNode As IXMLDOMNode
Dim tvwChildElement As Node
Dim bSuccess As Boolean
Dim nLevel As Long

On Error GoTo Procedure_Err
strgLastFunction = "PopulateTreeWithChildren"

'Assume failure
bSuccess = False

nLevel = tvwData.Nodes.Count
For Each objNode In objDOMNode.childNodes

  'If the current node has child nodes, add the current
  'name and text to the tree.
  If objNode.childNodes.length > 1 Then
    If tvwData.Nodes.Count > 1 Then
      'Add a child to the tree.
      Set tvwChildElement =
tvwData.Nodes.Add(nLevel, tvwChild)
      tvwChildElement.Text = "<" & objNode.nodeName
& ">"
      tvwChildElement.Expanded = True
    End If

    'Recursively call PopulateTreeWithChildren with
    the current node
    PopulateTreeWithChildren objNode, tvwData
  Else
    'No more children. Add the current node without
    a recursive call
    Set tvwChildElement = tvwData.Nodes.Add(nLevel,
tvwChild)
    tvwChildElement.Text = objNode.xml
  End If
  Next

  'If you made it to here, you have success
  bSuccess = True

Procedure_exit:
  PopulateTreeWithChildren = bSuccess
  Exit Function

Procedure_Err:
  HandleError Err
  bSuccess = False
  Resume Procedure_exit
  Resume

```