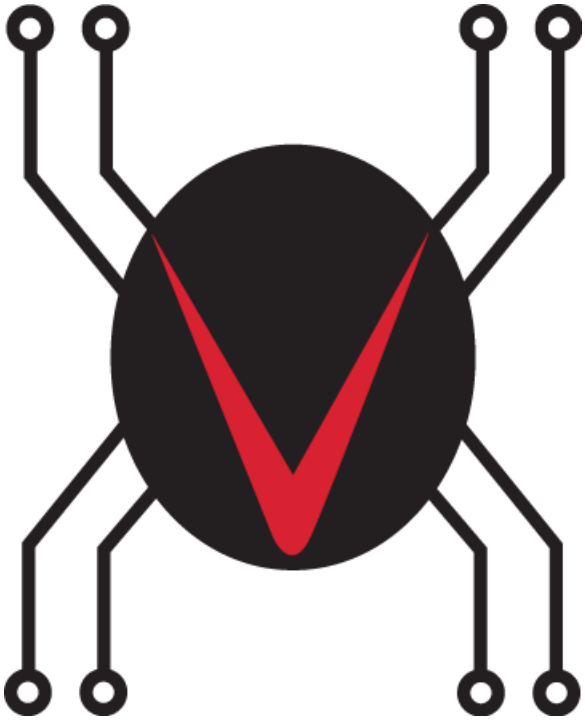
Pavuk Systems

Pavuk XML Processor



Pavuk XML Processor

Printed: January 2017 in Pearland, Texas, USA

Publisher

William G. Crowell
Pavuk Systems
2716 Shauntel Street
Pearland, Texas 77581 USA
+1 713.589.9711
sales@pavuk.com

Author

William G. (Bill) Crowell

Cover Graphic

Heather S. Geiser
JippityJuice Designs

Table of Contents

1 Introduction

The Pavuk XML processor enables the BASIC programmer to interact with XML documents using a unified interface.

The PXMLP requires that incoming XML documents be well-formed. If necessary, the programmer may use a variety of XML utilities to ensure that the XML is clean prior to processing.

A subset of Xpath syntax is used for querying the XML. This makes the PXMLP broadly compatible with other products.

2 Theory of Operation

PXMLP is implemented as an OpenQM BASIC Class module and utilizes the Object-Oriented Programming methodology as provided in the environment.

The steps of implementation are as follows:
1. Instantiate a QM BASIC object using the P.XML.CLS module.
2. Invoke the PARSE method to have the class module consume the XML document.
3. Perform Xpath methods to query the module.
4. Destroy the object to free resources.

An example program is provided.

3 Parsing

The PXMLP has an extremely efficient XML parser that is independent of the need for DTDs.

Internally, the parsed XML is stored inside of a QM Data Collections variable and subject to the QM constraints with respect to memory usage and size.

A quick review of XML:

We refer to any given tag pairing in XML as a "node". The <firstName></firstName> couplet is a node. Nodes may be nested. We refer to the content between the tags as the "value" of the node. Finally, we refer to the data contained in the opening tag as an "attribute". Referring to our starting example:

```
<records>
	<record id=”1234”>
		<firstName>Bill</firstName>
		<lastName>Crowell</lastName>
	</record>
	<record id="2345>
		<firstName>Dasha</firstName>
		<lastName>Crowell</lastName>
	</record>
</records>
```

The couplet, <records></records> is a node. Within this node are <record></record> nodes that contain the <firstName> and <lastName> nodes. The values are the names shown and there is an attribute named "id" in the null namespace.

By definition, other applications that handle an XML document may add value to the document by adding tags and attributes using their own namespace as a prefix , a colon and then an attribute name. In the example below, the application "foo" is adding information to the XML using its own namespace. This does not change the meaning of the data in the null namespace nor does it modify the content of the original application.

```
<records>
	<record id=”1234”>
		<firstName foo:bar=”William”>Bill</firstName>
		<lastName foo:baz=”Kroll”>Crowell</lastName>
		<foo:DOB>19631211</foo:DOB>
	</record>
	<record id=”2345”>
		<firstName>Dasha</firstName>
		<lastName foo:baz=”Kokulova”>Crowell</lastName>
	</record>
</records>
```

It is important that XML does not impose any method to ensure the uniqueness of any data node. The following example is perfectly acceptable as far as XML is concerned:

```
<records>
	<record>
		<firstName>Bill</firstName>
```

```
            <lastName>Crowell</lastName>
        </record>
        <record>
            <firstName>Dasha</firstName>
            <lastName>Crowell</lastName>
        </record>
</records>
```

This presents a challenge in accessing the nodes in a manner other than simply sequentially. In order to overcome challenges with parsing such as we've seen, Pavuk's XML processor adds information to the internal XML using the "pavuk" namespace. In addition, because multiple nodes are allowed at any given level without a unique identifier, PXMLP adds a sequence number to the internal structures.

The sequence number is an integer that begins at 1 at each level of nesting of nodes. This sequence number further enables the BASIC programmer to select nodes out of the XML using counters and loops.

It is easiest to examine the internal structure of the data collection using the path syntax.

```
records:00001
records:00001/record:00001/id = 1234
records:00001/record:00001/firstName:00001/foo:bar = William
records:00001/record:00001/firstName:00001/pavuk:value = Bill
records:00001/record:00001/LastName:00002/foo:baz = Kroll
records:00001/record:00001/lastName:00002/pavuk:value = Crowell
records:00001/record:00001/foo:DOB:00003/pavuk:value = 19631211

records:00001/record:00002/id = 2345
records:00001/record:00002/firstName:00001/pavuk:value = Dasha
records:00001/record:00002/lastName:00002/foo:baz = Kokulova
```

The attribute "pavuk:value" is utilized to declare the name for what is contained inside the node's value. When using the Xpath syntax for queries, either the attribute "pavuk:value" may be used or Xpath's standard operator "text()" may be used as they are equivalent.

It is important to note that the counter at any given level is incremented as data is read in through the parsing mechanism. The use of the counter to determine the field position is strongly discouraged as it may be different between nodes. In our example, we have "foo:DOB:00003", but had this been encountered first, it would be stored as "foo:DOB:00001".

The choice of a right-justified, zero-filled counter was made to simplify the work of the BASIC programmer.

4 Object Oriented Programming Overview

Most BASIC programmers have not been exposed to Object-Oriented Programming (OOP). This overview is to give a quick introduction to the concepts as they are embodied in OpenQM.

An "object" is a container of both executable BASIC code and data. The concept behind the object is that the internal code is a "black box" to the calling program. That the programmer who utilizes an object need not understand the internal workings of the object, but concentrate on his own task. Likewise, the person writing the object's internal code need not be concerned with what code uses the object.

BASIC has a number of different variables. Variables may be:
1. Regular variables - strings and numbers
2. Dynamic Arrays
3. Dimensioned arrays
4. File Descriptors
5. Select Lists
6. Data Collections
7. NULL
8. Boolean (True/False)

QM introduces a 9th type of variable, the Object. An object variable contains the executable code and internal storage. Like any other BASIC variable, it may be copied into a new variable. The process of managing the object variable is handled transparently by the QM runtime.

Unlike a called SUBROUTINE, an Object maintains its local storage between requests. Because of the local storage and of Data Collections, there is no need to create a temporary file for processing XML.

In OOP, the features of an object are referred to as "properties" and "methods". Properties are simply variables that are contained inside of the object that the programmer has made available to the code outside of the object. Methods are simply named subroutines that live inside of the object and which may be called by programs outside of the object. Objects may have many internal variables, functions and subroutines that are not visible to the outside program as they are "private".

In QM, objects are called "class modules". Though not required, the standard suffix is .CLS. Class modules are written in BASIC and compiled like any other program and are loaded into the catalog for use. P.XML.CLS is the name of the Pavuk XML Processor. The executable is loaded into the catalog.

Here is an example of creating (instantiating) an object variable:

```
MYOBJ = OBJECT('P.XML.CLS')
```

MYOBJ is calling the QM CLASS() function and passing it the name of the class module. QM reads the module and assigns an "instance" of it to MYOBJ. At this point, MYOBJ may be used.

To talk to the object, we use the direction operator "->" which is familiar in other languages.

```
MYOBJ->PARSE(data)
```

Calls the internal subroutine "PARSE" and passes in some data. This is referred to as a Method.

```
PRINT MYOBJ->MS.PARSE
```

Displays the number of milliseconds required to parse the data. This is referred to as a Property.

```
MYOBJ->XML.KEY = 'SAVENAME'
```

```
MYOBJ2 = MYOBJ
```

Unlike normal variable assignments in BASIC, MYOBJ2 is NOT a copy of MYOBJ. It is a reference or alias. At present, there is no method to directly copy one object to another.

```
MYOBJ  = ''
```

Setting the object variable to the Null variable causes the object and all of its data to be destroyed and removed from memory.

5 Example Program

The following example program process an Excel worksheet document.

```
0001    PROGRAM XML.TEST
0002     INPUT.FILE = '/tmp/xl/workbook/sheet1.xml'
0003
0004     OSREAD D.XML FROM INPUT.FILE ELSE D.XML = ''
0005
0006     XOBJ->OBJECT('P.XML.CLS')
0007     XOBJ->PARSE(D.XML)
0008
0009     MYPATH = '/worksheet/sheetData/row/c[@t='s']/v/text()'
0010
0011     XOBJ->XPATH(MYPATH)
0012
0013     MYVALUES =     XOBJ->XPATH.VALUES
0014     MYNODES  =     XOBJ->XPATH.NODES
0015     MYCOUNT  =     XOBJ->XPATH.COUNT
0016
0017     FOR J = 1 TO MYCOUNT
0018              PRINT J:' ':MYVALUES<J>
0019     NEXT J
0020
0021    XOBJ->XML.FILE = 'MYXMLFILE'
0022    XOBJ->XML.KEY  = 'SHEET1'
0023
0024    XOBJ->SAVE()
0025
0026    XOBJ = ''
0027    END
```

Referring to the example by line number:

0002 The variable INPUT.FILE is set to the path to the XML file we wish to read.

0004 OSREAD is used to populate the variable D.XML using the file path specified.

0006 A new variable named "XOBJ" is created inside of QM. It is created as an object and it is "instantiated" by calling the QM/BASIC function OBJECT(). OBJECT takes a single argument which is the name of the module previously cataloged. Inside the module, P.XML.CLS, subroutines are triggered to configure the internal variables and arrays. This is transparent to the XML.TEST program itself.

0007 The D.XML variable holding the XML is passed into the XOBJ's PARSE() function. At this point, the XOBJ performs the parsing of the XML and places it into a QM Collection variable for subsequent operations. For the BASIC programmer, this is analogous to "CALL PARSE(D.XML)"; however, it is much more powerful because there may be many objects available inside of one BASIC program - each has independent local data.

0009 The MYPATH variable is set up as a string containing an XPath query. The string follows the path of an Excel worksheet to select rows and colums with a "t" attribute of 's' and returning the node

value.

0011 The MYPATH variable is sent to the XPATH() method. XOBJ processes the query and the return values/properties are set.

0013-15 These lines of code illustrate the ability to make a copy of properties of the XOBJ. They are included so that the BASIC programmer can grow accustomed to the concepts. They are not needed for any other purpose.

0017-19 These lines show a simple loop that will display the values of the data returned by the XPath query. As you can see, it is a normal dynamic array.

0021 This line is where we are setting the property variable XML.FILE to the name of a file where we wish to store our data collection from the parser.

0022 This line is assigning the key field to the file.

0024 This line tells the XOBJ to save the data collection variable.

0026 Setting XOBJ to null causes the object to be destroyed. All resources are released back to the system.

6 XPath Queries

The Pavuk XML Processor performs queries based upon the XPath Syntax. A full discussion on XPath may be found here Wikipedia Article.

Essentially, the way that nodes are returned from XML is using a query that is constructed similar to a Unix file path.

Data contained in XML:

<comment xml:lang='en'>Portable Document Format</comment>

Example program to return the 'Portable Document Format':

```
0001    XP = \/mime-type/comment[@xml:lang='en']/text()\
0002    MYOBJ->XPATH(XP)
0003
0004    PRINT 'NODES SELECTED ':MYOBJ->XPATH.COUNT
0005    PRINT 'TIME ':MYOBJ->MS.XPATH
0006
0007    FOR I = 1 TO MYOBJ->XPATH.COUNT
0008            PRINT MYOBJ->XPATH.VALUES<I>
0009    NEXT I
```

In this example program by line number:

0001 The XP variable is an XPath string that selects <comment> nodes with the attribute "xml:lang" equal to 'en' and returns the node value.

0002 The XPATH() method is called with MYOBJ to select the specified nodes.

0004 MYOBJ->XPATH.COUNT is a property returning the number of nodes selected by the previous XPATH method.
0005 MYOBJ->MS.XPATH is a property containing the time required to perform the query.

0007-9 This loop prints the values returned by the query. Because the 'text()' directive was used, the value contained inside the node is returned.

7 Object Reference

Property	Description
DATA.COL	Data Collection variable containing parsed XML
XPATH.COL	Data Collection variable containing Xpath locator strings
XPATH.ROOT	Path to the root node of the XML {to be implemented}
XPATH.NODES	Dynamic array list of nodes matched with the Xpath query method
XPATH.VALUES	Dynamic array list of values matched with the Xpath query method
XPATH.COUNT	Integer counter of selected nodes
XML.FILE	QM file to save DATA.COL - defaults to $XML
XML.KEY	Record key for saving
MS.XPATH	Execution time for the XPATH method (in milliseconds)
MS.PARSE	Execution time for the PARSE() method (in milliseconds)

Method	Description
PARSE(*arg*)	Parses XML data contained in *arg* into DATA.COL
XPATH(*arg*)	Executes the query contained in *arg*.
SAVE()	Saves the DATA.COL record to the QM collection file

8 Pavuk Namespace

The PXMLP parses the source XML into a QM Data Collection. The DC maintains the structure of the XML and facilitates the access to the information inside of it.

During the parsing phase of XML processing, the PXMLP adds structural information about the XML inside of the data collection. If the program saves the parsed data collection to a file, it may be examined with the CED editor. The structural information is defined as being in the "pavuk" namespace. It is expected that this namespace will be unique.

Tag	Description
pavuk:node	A single byte flag of the type of XML node (see below)
pavuk:value	A name to refer to the content of the node and is equivalent to the text() Xpath
pavuk:#:@	Inherited attributes from nodes above the selected node where 'pavuk' is the namespace, # is the level above the node as an integer value from 1-n, where @ is the attribute name. For example: pavuk:1:xmlns "http://schemas.openxmlformats.org..."

Node Type	Description	Example
X	XML Header Node	<?xml version="1.0"...
S	Starting of a nest of nodes	<sheetData>
E	End of a nest of nodes	</sheetData>
D	Data node	<v>440</v>
F	Self-terminated (void) node	<dimenson ref="A1:Z55" />
C	CDATA node	<![CDATA[this is cdata content]]>
M	Comment node	<!-- THIS IS A COMMENT -->

Example XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<worksheet xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" xmlns:r="http://
schemas.openxmlformats.org/officeDocument/2006/relationships">
  <dimension ref="A1:Z55" />
  <sheetViews>
    <sheetView tabSelected="1" workbookViewId="0" />
  </sheetViews>
  <sheetFormatPr defaultRowHeight="15" />
  <sheetData>
    <row r="1" spans="1:26">
      <c r="A1" t="s">
        <v>440</v>
      </c>
    </row>
```

9 CED Editor Screenshots

Parsed XML may be saved as a QM Data Collection. The default collection file name is $XML. The default record name is the internal date _ internal time (17690_72447).

Text in red shows Pavuk structural elements. Text in green shows elements inherited from previous levels and the level number where they occurred.

Example:

:CED $XML 17690_72247

9.1 Level 0

>worksheet:00001 (Collection, 10 elements)

$XML 17690_72447
(top level)

9.2 Level 1

>dimension:00001 (Collection, 4 elements)
drawing:00007 (Collection, 4 elements)
hyperlinks:00005 (Collection, 4 elements)
pageMargins:00006 (Collection, 9 elements)
pavuk:node (String): "S"
sheetData:00004 (Collection, 57 elements)
sheetFormatPr:00003 (Collection, 4 elements)
sheetViews:00002 (Collection, 4 elements)
xmlns (String): "http://schemas.openxmlformats.org/sp
xmlns:r (String): "http://schemas.openxmlformats.org

$XML 17690_72447
worksheet:00001

9.3 Level 2

>pavuk:1:xmlns (String): "http://sch
pavuk:1:xmlns:r (String): "http://s
pavuk:node (String): "S"
row:00001 (Collection, 6 elements)

row:00002 (Collection, 7 elements)
row:00003 (Collection, 31 elements)
row:00004 (Collection, 33 elements)
row:00005 (Collection, 33 elements)
row:00006 (Collection, 33 elements)
row:00007 (Collection, 33 elements)
row:00008 (Collection, 33 elements)
row:00009 (Collection, 33 elements)
row:00010 (Collection, 33 elements)
row:00011 (Collection, 33 elements)
row:00012 (Collection, 33 elements)
row:00013 (Collection, 33 elements)
row:00014 (Collection, 33 elements)
row:00015 (Collection, 33 elements)
row:00016 (Collection, 33 elements)
row:00017 (Collection, 31 elements)
row:00018 (Collection, 33 elements)
row:00019 (Collection, 33 elements)
$XML 17690_72447
worksheet:00001/sheetData:00004

9.4 Level 3

>c:00001 (Collection, 8 elements)
pavuk:1:xmlns (String): "http://schemas.o
pavuk:1:xmlns:r (String): "http://schemas
pavuk:node (String): "S"
r (String): "1"
spans (String): "1:26"

$XML 17690_72447
worksheet:00001/sheetData:00004/row:00001

9.5 Level 4

>pavuk:1:xmlns (String): "http://schemas.openxmlfor
pavuk:1:xmlns:r (String): "http://schemas.openxmlf
pavuk:3:r (String): "1"
pavuk:3:spans (String): "1:26"
pavuk:node (String): "S"
r (String): "A1"
t (String): "s"
v:00001 (Collection, 8 elements)

$XML 17690_72447
worksheet:00001/sheetData:00004/row:00001/c:00001

9.6 Level 5

>pavuk:1:xmlns (String): "http://schemas.openxmlformats.o
pavuk:1:xmlns:r (String): "http://schemas.openxmlformats
pavuk:3:r (String): "1"
pavuk:3:spans (String): "1:26"
pavuk:4:r (String): "A1"
pavuk:4:t (String): "s"
pavuk:node (String): "D"
pavuk:value (String): "440"

$XML 17690_72447
worksheet:00001/sheetData:00004/row:00001/c:00001/v:00001

Back Cover