Using XML with the Progress 4GL

Gus Bjorklund
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>4</td>
</tr>
<tr>
<td>WHAT IS XML?</td>
<td>4</td>
</tr>
<tr>
<td>4GL REPRESENTATION OF THE DOM</td>
<td>6</td>
</tr>
<tr>
<td>THE DOM STRUCTURE MODEL</td>
<td>7</td>
</tr>
<tr>
<td>THE NEW 4GL OBJECTS</td>
<td>8</td>
</tr>
<tr>
<td>4GL OBJECT TYPES</td>
<td>8</td>
</tr>
<tr>
<td>4GL DOCUMENT OBJECTS</td>
<td>9</td>
</tr>
<tr>
<td>4GL NODE REFERENCE OBJECTS</td>
<td>9</td>
</tr>
<tr>
<td>CREATING DOCUMENTS</td>
<td>9</td>
</tr>
<tr>
<td>CREATING NODE REFERENCES</td>
<td>9</td>
</tr>
<tr>
<td>CREATING NODES</td>
<td>9</td>
</tr>
<tr>
<td>CHARACTER SETS AND ENCODINGS</td>
<td>11</td>
</tr>
<tr>
<td>ERROR HANDLING</td>
<td>11</td>
</tr>
<tr>
<td>EXAMPLES</td>
<td>12</td>
</tr>
<tr>
<td>EXAMPLE 1: XML OUTPUT FROM 4GL</td>
<td>12</td>
</tr>
<tr>
<td>EXAMPLE 2: XML DOCUMENT PRODUCED BY EXAMPLE 1</td>
<td>13</td>
</tr>
<tr>
<td>EXAMPLE 3: XML INPUT FROM 4GL</td>
<td>15</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>17</td>
</tr>
<tr>
<td>XML</td>
<td>17</td>
</tr>
<tr>
<td>DOM</td>
<td>17</td>
</tr>
<tr>
<td>DOM PARSER</td>
<td>17</td>
</tr>
<tr>
<td>UNICODE</td>
<td>17</td>
</tr>
</tbody>
</table>
Introduction

This white paper describes a set of Progress 4GL capabilities, introduced in Version 9.1A, that enables Progress applications to use Extensible Markup Language (XML) as a means of data exchange.

XML is being rapidly and widely adopted throughout the computer industry and becoming the preferred method for encoding data exchanged among applications. One of the most important future uses is for business-to-business communication. XML is enormously useful for this purpose because it enables many different kinds of data exchange in a standards-based, simple, flexible, and therefore cheaper way than was possible with previous methods. Here’s what two respected industry analysts say about it:

"XML will revolutionize the exchange of business information similar to the way the phone, fax machine, and photocopier did when those devices were invented. Those [prior] innovations made a significant impact on how businesses viewed and exchanged information. XML is poised to impact the Internet area the same way."

Ron Rappaport, Zona Research, December 31, 1998

"Work by the World Wide Web Consortium (W3C) to flesh out XML will give sellers a standard way to define catalogs and processes. By 2000, this technology will proliferate from the desktop to the back office – creating a climate in which new buying models thrive and standards like OBI consolidate around XML."


What Is XML?

The Extensible Markup Language (XML™) is a data format for structured document interchange on the Web. It is hardware architecture neutral and application independent. XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996. The World-Wide Web Consortium’s official recommendation for XML and a variety of related materials can be found at the following URL: http://www.w3.org/XML/

XML is a subset of another markup language, called SGML, which was adopted as an international standard in 1986 [ISO 8879]. SGML is based on a still earlier markup language, called GML, which was developed by researchers at IBM in 1969.

XML describes a class of data objects called XML documents and partially describes the behavior of computer programs that process them. XML is an “application profile” or restricted form of SGML. By construction, XML documents are conforming SGML documents.

XML documents composed of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of a document's layout and logical structure. XML provides a mechanism to impose constraints on the layout and logical structure.

A software module called an XML processor is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another
module, called the *application*. The XML specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

**XML Documents**

XML documents are text composed of two parts: a prologue and a body. The optional prologue may contain the XML version the document conforms to, information about the character encoding used to encode the contents of the document, and a *document type definition* (DTD) which describes the grammar and vocabulary of the document. The body may contain elements, entity references, and other markup information.

*Elements* represent the logical components of documents. They can contain either data or other elements. For example, a “phone list element” could contain a number of phone list entry elements and each entry element the data values for a single entry. Here is an example of a simple element:

```
<name>Clyde</name>
```

Elements can have additional information called *attributes* attached to them. Attributes are used to describe properties of elements. Here is an example of an element with an attribute:

```
<name emp-num="1">Mary</name>
```

Here is an example of some elements that contain other elements and data:

```
<phonelist>
  <entry>
    <name>Chip</name>
    <extension>3</extension>
  </entry>
  <entry>
    <name>Gus</name>
    <extension>5</extension>
  </entry>
</phonelist>
```

**Document Type Definitions**

There are obviously an infinite variety of possible kinds of documents, such as a repair manual for a vehicle, a dictionary, a telephone directory, an order for equipment, an invoice, and so forth. Each kind of document can have a unique structure and organization that can be used over and over. The descriptions of classes of documents are called *Document Type Definitions* or DTDs. DTDs are rules that define the elements that can exist in a particular document or group of documents, and what the relationships among the various elements are. A DTD can be part of the content of an XML document or can be separate from it and referred to by the document.

Here is an example of a document that includes a DTD in its prologue:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customer [
<!ELEMENT customer (name, cust-num)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT cust-num (#PCDATA)>
]>
<customer>
  <name>Lift Line Skiing</name><cust-num>1</cust-num>
```
The DOM

In an application, a raw XML document is cumbersome to work with because it just is one long string of characters. To allow an application to use data encoded in XML easily, something more is needed. The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of XML documents and the way a document is accessed and manipulated. The W3C’s official recommendation for the DOM and a variety of related materials may be found at the following URL: http://www.w3.org/DOM/.

In the DOM specification, the term “document” is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage these data.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.

4GL Representation of the DOM

For Version 9.1A, we have defined an initial set of extensions to the Progress 4GL to allow the use of XML through the DOM interface. A DOM parser is built-in to the 4GL interpreter. These extensions are designed to provide 4GL applications with the basic input, output, and low-level data manipulation capabilities required to use data contained in XML documents. They are not intended to provide access to the entire DOM interface, nor are they intended to include all the high-level constructs that one might expect to have for ease-of-use.

The DOM is an object model for representing XML documents in a platform-neutral and application-independent form as a hierarchy or “tree” of objects. The model and the API have been designed to provide language bindings for object-oriented languages such as C++, Java, and ECMAScript. The Progress 4GL is not like these other object-oriented languages. It is a hybrid, mainly procedural, with some object capabilities. Due to the 4GL’s nature, the 4GL binding is intentionally different from the other language bindings mentioned.

The DOM API is designed to be compatible with a wide range of programming languages, and the naming convention chosen by the W3C does not match that already used in the Progress 4GL. It would be awkward to introduce an entirely new set of names for existing 4GL language constructs. In some cases, we have elected to use the familiar names already used in the 4GL rather than the names used by the DOM specification.

Similarly, where there are existing 4GL features that provide the same capability as the DOM interfaces, we have chosen to use the 4GL approach rather to introduce duplicate language features merely to match the DOM more closely. This approach allows 4GL programmers to use concepts they are already familiar with and also reduces the number of new language constructs.

The DOM Level 1 Specification is separated into two parts: Core and HTML. The Core DOM Level 1 section provides a low-level set of fundamental interfaces that can represent any structured document, as well as defining extended interfaces for representing an XML document. The HTML Level 1 section

\[1\text{ However, many existing HTML documents do not conform to the HTML specification. They would have to be reformulated to conform before they could be used by an XML application.}\]
will provide additional, higher-level interfaces that are used with the fundamental interfaces defined in the Core Level 1 section to provide a more convenient view of an HTML document. The language extensions provided with Progress Version 9.1A cover the Core DOM Level 1 only.

**The DOM Structure Model**

Before we discuss the 4GL enhancements, it is helpful to look at the DOM structure model and its object types.

From the DOM recommendation:

“The DOM presents documents as a hierarchy of "Node" objects that also implement other, more specialized interfaces. Some types of nodes may have child nodes of various types, and others are leaf nodes that cannot have anything below them in the document structure. The node types, and which node types they may have as children, are as follows:

- **Document** -- Element (maximum of one), ProcessingInstruction, Comment, DocumentType
- **DocumentFragment** -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- **DocumentType** -- Notation, Entity
- **EntityReference** -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- **Element** -- Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
- **Attribute** -- Text, EntityReference
- **ProcessingInstruction** -- no other nodes
- **Comment** -- no other nodes
- **Text** -- no other nodes
- **CDATASection** -- no other nodes
- **Entity** -- no other nodes
- **Notation** -- no other nodes

The DOM also specifies a "NodeList" interface to handle ordered lists of nodes, such as the children of a node, or the elements returned by the `Element:getElementsByTagName` method, and also a NamedNodeMap interface to handle unordered sets of Nodes referenced by their name attribute, such as the Attributes of an Element. NodeLists and NamedNodeMaps in the DOM are "live", that is, changes to the underlying document structure are reflected in all relevant NodeLists and NamedNodeMaps. For example, if a DOM user gets a NodeList object containing the children of an Element, then subsequently adds more children to that Element (or removes children, or modifies them), those changes are automatically reflected in the NodeList without further action on your part. Likewise changes to a Node in the tree are reflected in all references to that Node in NodeLists and NamedNodeMaps.”

The text below illustrates a short example document taken from the DOM specification:

```
<TABLE>
<TBODY>
<TR>
<TD>Shady Grove</TD>
```
The New 4GL Objects

4GL Object Types
The DOM Core APIs present two different sets of interfaces, one presenting an "object-oriented" approach with a hierarchy of inheritance, and a "simplified" flattened view that allows all manipulation to be done via the Node interface. Because the 4GL does not have inheritance, we have chosen to use the "simplified" view and to implement the Node interface. Though the simplified interface provides access to all DOM tree manipulation functionality, it is also useful to use some of the other specialized interfaces. The DOM objects that have these specialized interfaces are implemented as SUBTYPEs of the Progress object. This approach does not limit us to using only the DOM Node interface because the Progress object is extended with the methods and attributes that are only valid for objects that are of the appropriate SUBTYPEs.

The Document interface also inherits from the Node interface. We have extended it to provide specialized methods for operations like LOAD() and SAVE(), Therefore it is implemented as a separate object instead of as a SUBTYPE of a generic node object.
This gives us two new object types in the 4GL for XML document manipulation: the X-DOCUMENT, which represents an entire XML document tree, and the X-NODEREF, which represents a reference to a single node in the XML document tree.

### 4GL Document Objects

The X-DOCUMENT is a 4GL object that represents a DOM Document object. X-DOCUMENT objects are used to manipulate the XML document and its tree representation. They can be assigned to variables of the 4GL data-type HANDLE, be created with the CREATE statement, and be identified through the TYPE attribute on the handle.

### 4GL Node Reference Objects

The X-NODEREF object is a 4GL object that is a reference to any arbitrary node in an XML tree, except a Document node. X-NODEREF objects are used to manipulate the DOM nodes. They can be assigned to variables of the 4GL data-type HANDLE, be created with the CREATE statement, and be identified through the TYPE attribute on the handle.

Note that an X-NODEREF object is not an actual node in the XML tree but is more like a “cursor” which is used to navigate the tree and manipulate the nodes in it.

### Creating Documents

The creation and persistent saving of a document is not part of the DOM Core API, and is left to the application (i.e. the Progress 4GL interpreter) that calls the DOM API. You create an XML document using the 4GL CREATE statement and use the methods of the Progress object to save a new XML document or to load an existing XML document.

```progress
CREATE X-DOCUMENT <handle>.
```

This statement will create a Progress handle for an object of the type “X-DOCUMENT” that “wraps” an XML document. You may start adding nodes to it right away or use the LOAD() method to populate it from an existing XML document.

### Creating Node References

The X-NODEREF is also a new Progress object that you will need to create:

```progress
CREATE X-NODEREF <handle>.
```

This creates a Progress handle, which is ready to be used as a parameter or return-value for methods that will associate the handle with a new or an existing XML–node. This object is not a node in its own right, but merely used as a reference to some node in the tree. It provides a path to access and manipulate the underlying XML document nodes. Operations on the X-NODEREF object’s attributes and methods in turn manipulate the contents of the XML document.

### Creating Nodes

The Progress 4GL uses a common CREATE-NODE method on an X-DOCUMENT object to create an XML nodes of the various SUBTYPEs.

The complete list of DOM interfaces that inherit from Node are as follows:
DocumentType

Represents the Document Type Declaration or Schema declaration of the XML document.

DocumentFragment

Represents a lightweight object used to store sections of an XML document temporarily.

Element

Represents an element node. This interface represents the data, or more precisely the tags of the XML document. It’s important to notice that the text of the element is stored in a Text or CDATASection node, which is the child of the element.

Attribute

Represents an attribute of a document or an element. Typically the allowable values for the attribute are defined in a document type definition. Note that the attributes are NOT considered as child nodes of the element they describe.

Entity

Represents an entity, either parsed or unparsed, in the XML document.

EntityReference

Represents a reference to an entity within the XML document.

Notation

Represents a notation declared within the DTD.

CharacterData

Provides methods and properties that are inherited by Text, Comment, and CDATASection.

Text

Represents a Text node that is a child of an element node.

CDATASection

CDATA sections are used to escape blocks of text that would otherwise be regarded as markup. The primary purpose is for including XML fragments, without needing to escape all the delimiters.

Comment

Represents the content of a comment.

ProcessingInstruction

The “Processing Instruction” is a way to keep processor-specific information in the text of the document.

Implementation

Provides access to methods and properties that are application specific and independent of any specific implementation of the DOM.

The Progress 4GL provides the following SUBTYPEs for the X-NODEREF object:

- CDATA-SECTION
- COMMENT
- ELEMENT
- ENTITY-REFERENCE
- PROCESSING-INSTRUCTION
- TEXT-NODE

The default SUBTYPE will be ELEMENT.

The “simplified” DOM Node interface’s nodeName and nodeValue gives access to these interfaces as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>nodeName</th>
<th>nodeValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentType</td>
<td>Document type name</td>
<td>Null</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>#document-fragment</td>
<td>Null</td>
</tr>
<tr>
<td>Element</td>
<td>Tag name</td>
<td>Null</td>
</tr>
<tr>
<td>Attribute</td>
<td>Name of attribute</td>
<td>Value of attribute</td>
</tr>
<tr>
<td>Entity</td>
<td>Entity name</td>
<td>Null</td>
</tr>
<tr>
<td>EntityReference</td>
<td>Name of entity referenced</td>
<td>Null</td>
</tr>
<tr>
<td>Notation</td>
<td>Notation name</td>
<td>Null</td>
</tr>
<tr>
<td>Text</td>
<td>#text</td>
<td>Content of the text node</td>
</tr>
<tr>
<td>CDATASection</td>
<td>#cdata-section</td>
<td>Content of the CDATA section</td>
</tr>
<tr>
<td>Comment</td>
<td>#comment</td>
<td>Content of the comment</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>Target</td>
<td>Content excluding the target</td>
</tr>
</tbody>
</table>

2 Names of elements, attributes, etc. in XML documents are constrained by the rules laid forth in the XML specification. The gist of these rules is that: Names must begin with a letter, digit, comma, hyphen, or underscore. Names may not start with the string “xml” or anything that would match it if case were insignificant.
The Progress 4GL NAME attribute is used to obtain the nodeName, while the new NODE-VALUE character attribute is used to obtain or set the node’s nodeValue.

Character Sets and Encodings

XML documents can contain data encoded in any one of a wide a variety of character sets. The encoding used in an XML document is determined from an optional encoding declaration at its very beginning. If the encoding declaration is present, it specifies the encoding used in the remainder of the document. If the declaration is not present, the document’s encoding is assumed to be UTF-8 or UTF-16.

The DOM parser always returns UNICODE UTF-16 encoded character data to the Progress 4GL interpreter. The translation from a document’s encoding to UTF-16 is performed by the DOM parser’s built-in translation library. Character data are translated from UTF-16 to the character encoding specified by the value of –cpinternal by the 4GL interpreter.

When the LOAD() method is used to load an XML document, theENCODING attribute of the X-DOCUMENT will be set to the name of encoding found in the encoding declaration of the document.

For output, you can set the X-DOCUMENT’s ENCODING attribute to the name of the desired encoding.

When the SAVE() method is used to write an output XML document from a memory-resident DOM tree, the generated XML text is encoded by the DOM parser according to the setting of the ENCODING attribute.

When you SAVE a document to a stream, the specified encoding is used, and the value of cpstream is ignored.

According to the XML recommendation, “it is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process”. If this error occurs while Progress is attempting to load a document, the document will be empty.

Error Handling

Any of the methods of the X-DOCUMENT and X-NODEREF objects may encounter an error condition and fail, but this will not normally cause the Progress error status to be raised. Instead, the method will generally return FALSE. Also, the parser may encounter errors that do not cause the operation as a whole to fail. So instead of testing for ERROR-STATUS:ERROR after running a method with NO-ERROR, one should test for ERROR-STATUS:NUM-MESSAGES being greater than 0.

Note that the DOM parser may detect errors in an input XML document even if validation is not specified for the LOAD() method call. Validation checks the document for conformance to a DTD, but there could be structure or other errors, such a missing end tag, mismatched, or improperly nested tags. The parser will report these errors even if validation against a DTD is not performed.

---

3 According to the XML recommendation, “entities encoded in UTF-16 must begin with the Byte Order Mark described by ISO/IEC 10646 Annex E and Unicode Appendix B (the ZERO WIDTH NO-BREAK SPACE character, #xFEFF). This is an encoding signature, not part of either the markup or the character data of the XML document. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents”.

Examples

In this section, we provide several simple examples of 4GL programs using XML, and the XML documents they use. These examples show how to:

- Load an XML document
- Save an XML document
- Generate an XML document that contains data obtained from the database
- Obtain data from an XML document and store it into the database.

Example 1: XML Output from 4GL

In this example we create a new document with a root element ‘customers’, put in all the rows from the Customer table in the Sports2000 sample database, and then write it into a file called “cust.xml”.

/* Declarations */
DEFINE VARIABLE hDoc AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE hRow AS HANDLE.
DEFINE VARIABLE hField AS HANDLE.
DEFINE VARIABLE hText AS HANDLE.
DEFINE VARIABLE hBuf AS HANDLE.
DEFINE VARIABLE hDBFlId AS HANDLE.
DEFINE VARIABLE i AS INTEGER.

/* Create the objects we need. */
CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hRow.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.

/* Get a buffer for the Customer table. */

hBuf = BUFFER Customer:HANDLE.

/* Set up a root node. */

hDoc:CREATE-NODE (hRoot, "Customers", "ELEMENT").
hDoc:APPEND-CHILD (hRoot).

FOR EACH Customer WHERE cust-num < 5:

/* Create a customer row node. */

hDoc:CREATE-NODE (hRow, "Customer", "ELEMENT").
hRow:APPEND-CHILD (hRow).

/* Put the row in the tree. Cust-num and Name are attributes of this element. The remaining fields are elements. */
hRow:SET-ATTRIBUTE ("Cust-num", STRING (cust-num)).
hRow:SET-ATTRIBUTE ("Name", NAME).

/* Add the other fields as elements. */
REPEAT i = 1 TO hBuf:NUM-FIELDS:
    hDBFld = hBuf:BUFFER-FIELD (i).
    /* We already did Cust-num and Name above so skip them. */
    IF hDBFld:NAME = "Cust-num" OR hDBFld:NAME = "NAME"
        THEN NEXT.
    /* Create an element with the field name as the tag.
       Note that the field name is the same as the element
       name. The rules for allowed names in XML are less
       stringent than the rules for Progress column names. */
    hDoc:CREATE-NODE (hField, hDBFld:NAME, "ELEMENT").
    hRow:APPEND-CHILD (hField).
    /* Make new field next row child. */
    hDoc:CREATE-NODE (hText, "", "TEXT").
    /* Node to hold value. */
    hField:APPEND-CHILD (hText).
    /* Attach text to field */
    hText:NODE-VALUE = STRING (hDBFld:BUFFER-VALUE).
END.
END.

/* Write the XML node tree to an xml file. */

hDoc:SAVE ("file", "cust.xml").

/* Delete the objects. Note that deleting the document
object deletes the DOM structure under it also. */

DELETE OBJECT hDoc.
DELETE OBJECT hRoot.
DELETE OBJECT hRow.
DELETE OBJECT hField.
DELETE OBJECT hText.

Example 2: XML Document Produced by Example 1

The following text illustrates the xml document produced by the preceding example. The original did
not include line breaks in the document body. We have added those to improve readability:
<?xml version='1.0' encoding='utf-8' ?>
<Customers>
  <Customer Name="Lift Line Skiing" Cust-num="1">
    <Country>USA</Country>
    <Address>276 North Street</Address>
  </Customer>
  <Customer Name="Urpon Frisbee" Cust-num="2">
    <Country>Finland</Country>
    <Address>Rattipolku 3</Address>
  </Customer>
  <Customer Name="Hoops Croquet Co." Cust-num="3">
    <Country>USA</Country>
    <Address>Suite 415</Address>
  </Customer>
  <Customer Name="Go Fishing Ltd" Cust-num="4">
    <Country>United Kingdom</Country>
    <Address>Unit 2</Address>
  </Customer>
</Customers>
<Contact>Alan Frogbrook</Contact>
<Phone>081 883 6827</Phone>
<Sales-Rep>SLS</Sales-Rep>
<Credit-Limit>15000</Credit-Limit>
<Balance>689</Balance>
<Terms>Net30</Terms>
<Discount>10</Discount>
<Comments/>
</Customer>
</Customers>

Now that you've had a chance to read through the example and the output it produces, (you did read it didn't you?), we'd like to make a few observations about it:

- As previously mentioned, the line breaks aren’t really there. We added them later to make the example more readable. The original document contained only two lines, the prologue and the body.
- There is no DTD.
- We chose to make the “Name” and “Cust-num” columns of the Customer table into attributes of the “Customer” element. The remaining columns are elements contained by the “Customer” element. Normally you probably wouldn’t make some table columns into attributes and some into elements, but we wanted to show you that you can do it either way.
- Columns that have unknown values show up as empty elements – there is no text between the opening and closing tag.
- The customer records expressed as XML are verbose. That is the nature of XML.

**Example 3: XML Input from 4GL**

This example reads a document from a file called “Cust.xml” and loads the data in it into the Customer table in theSports2000 sample database:

```4GL
/* Declarations */
DEFINE VARIABLE hDoc AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE hTable AS HANDLE.
DEFINE VARIABLE hField AS HANDLE.
DEFINE VARIABLE hText AS HANDLE.
DEFINE VARIABLE hBuf AS HANDLE.
DEFINE VARIABLE hDBF1d AS HANDLE.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.

/* Create the objects we will need. */
CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hTable.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.

hBuf = BUFFER Customer:HANDLE.
```
/* Read in the file created by the output example. Note that the entire file is read and parsed at this point. */


/* Get the root of the structure */

hDoc:GET-DOCUMENT-ELEMENT (hRoot).

/* Read each Customer from the root. */

REPEAT i = 1 TO hRoot:NUM-CHILDREN:
    hRoot:GET-CHILD (hTable,i).

    /* Create a customer record in the database. */
    CREATE Customer.

    /* Cust-num and Name are attributes, so we have to get those values from there. */
    cust-num = integer (hTable:GET-ATTRIBUTE ("Cust-num")).
    NAME = hTable:GET-ATTRIBUTE ("Name").

    /* The remaining fields are elements with text values. */
    REPEAT j = 1 TO hTable:NUM-CHILDREN:
        hTable:GET-CHILD (hField,j).

        /* skip any null values */
        IF hField:NUM-CHILDREN < 1 THEN NEXT.

        hDBFld = hBuf:BUFFER-FIELD (hField:NAME).

        /* Get text value of field and put it in the customer buffer. Note that the field name is assumed to be the element name. But the rules for allowed names in XML are less stringent than the rules for Progress column names. */

        hField:GET-CHILD (hText,1).
        hDBFld:BUFFER-VALUE = hTEXT:NODE-VALUE.

    END.

END.

/* Delete the objects we created. Note that when we delete hDoc, the structure under it will be deleted as well. */

DELETE OBJECT hDoc.
DELETE OBJECT hRoot.
DELETE OBJECT hTable.
DELETE OBJECT hField.
DELETE OBJECT hText.
References

XML


DOM

DOM Parser

UNICODE