OpenEdge® Development: Working with XML
Notices

© 2015 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, EasyI, Fathom, Icenium, Kendo UI, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, (Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, , Future Proof, High Performance Integration, Modulus, NativeScript, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at:

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

October 2015

Last updated with new content: Release 11.6.0
Table of Contents

Preface...........................................................................................................11
  Purpose.................................................................................................11
  Audience............................................................................................11
  Organization.......................................................................................12
  Using this manual..............................................................................12
    References to ABL compiler and run-time features........................13
    References to ABL data types..........................................................13
  Typographical conventions...............................................................13
  Examples of syntax descriptions.....................................................15
    Long syntax descriptions split across lines....................................16
    Complex syntax descriptions with both required and optional elements17
  Example procedures...........................................................................17
  OpenEdge messages..........................................................................18
    Obtaining more information about OpenEdge messages..............19

Chapter 1: Developing with XML in OpenEdge..........................................21
  About XML..........................................................................................21
    Role of XML.....................................................................................22
    Benefits of XML..............................................................................22
    Benefits of ABL XML development in OpenEdge..........................23
    Benefits of SOA XML development in OpenEdge..........................23
    Recommended resources for learning XML basics..........................24
  Developing XML-enabled ABL applications with DOM..................25
    DOM advantages.............................................................................25
    DOM limits......................................................................................25
    ABL support.....................................................................................25
    More information............................................................................26
  Developing XML-enabled ABL applications with SAX..................26
    SAX advantages.............................................................................26
    SAX limits.......................................................................................27
    ABL support.....................................................................................27
    More information............................................................................27
  Serializing ABL temp-tables and ProDataSets to and from XML........27
    Features..........................................................................................28
    Use cases.........................................................................................28
    ABL support.....................................................................................28
    More information............................................................................29
# Chapter 2: Reading and Writing XML with the Document Object Model (DOM)

- **XML terminology**
- **XML document structure**
- **XML prolog**
- **XML body**
- **DTDs and XML Schema documents**
- **XML elements**
- **XML attributes**
- **XML empty elements**
- **XML entities**
- **XML namespaces**

- The ABL DOM interface

- **Prerequisites**
- **Representing an XML document in ABL DOM**
- **Representing a DOM node in ABL**

- **Creating XML output from ABL**
  - The root node reference object
  - Creating and appending a node
  - Setting node attributes and values
  - Example of creating an output XML file
  - Writing an XML file to a MEMPTR, a stream, or a LONGCHAR

- **Reading XML input into ABL**
  - Loading an XML file
  - Loading an XML file from a MEMPTR
  - Accessing the child nodes
  - Using node attributes and values
  - Examples of reading an input XML file

- **Namespaces**
- **Internationalization**
- **Error handling**
- **Validation**

- **Method and attribute summary**

---

# Chapter 3: Reading XML Documents with the Simple API for XML (SAX)

- **Understanding ABL SAX**
- **SAX-reader object**
- **ABL SAX callbacks**
- **SAX-attributes object**

- Retrieving data from a SAX-attributes object
- Creating a SAX-attributes object
Chapter 4: Writing XML Documents with the Simple API for XML (SAX)....95

SAX-writer overview...........................................................................................................96
Creating a SAX-writer...........................................................................................................97
Configuring a SAX-writer.....................................................................................................98
Writing an XML document.....................................................................................................98
Examples...............................................................................................................................99
Creating an XML document from database data.................................................................99
Chapter 5: Reading and Writing XML Data from Temp-Tables and ProDataSets

Introduction.........................................................................................................................118
Use cases............................................................................................................................118
Other XML features...........................................................................................................119
Methods and attributes.........................................................................................................119
Attribute interactions..........................................................................................................121
  NAMESPACE-URI and NAMESPACE-PREFIX...........................................................................121
  XML-NODE-NAME and SERIALIZE-NAME..........................................................................123
  XML-NODE-TYPE and SERIALIZE-HIDDEN.......................................................................123
ABL-specific attributes in XML Schema...............................................................................123
Reading XML Schema into a temp-table, temp-table buffer, or ProDataSet...............................124
Creating a dynamic temp-table with XML Schema...............................................................127
Verifying a static temp-table against XML Schema..............................................................130
Creating a dynamic ProDataSet with XML Schema............................................................131
Reading XML into a temp-table, temp-table buffer, or ProDataSet..........................................131
Schema locations................................................................................................................136
Reading XML data into temp-tables....................................................................................136
Adding XML data to a populated temp-table.......................................................................138
Reading XML into a ProDataSet..........................................................................................140
Reading before-image data for ProDataSet temp-table buffers from XML..............................145
Writing XML Schema from a temp-table, temp-table buffer, or ProDataSet.............................146
Writing a temp-table definition to XML Schema....................................................................148
Writing a ProDataSet definition to XML Schema files........................................................149
Writing XML from a temp-table, temp-table buffers, or a ProDataSet......................................152
Writing temp-table data to XML..........................................................................................155
Writing XML from a temp-table buffer's current row.............................................................155
Writing a ProDataSet to XML..............................................................................................157
Minimizing XML document size..........................................................................................159
Omitting foreign key fields in nested child records................................................................159
Omitting fields with initial values........................................................................................160
Sample ProDataSet to XML round-trip..................................................................................161
Mapping ABL names to different XML element or attribute names.........................................163
Using XML Schema............................................................................................................164
Creating dynamic ABL objects from XML Schema.......................................................165
Verifying XML Schema against ABL definitions............................................................166
Inferring definitions from XML data..............................................................................166
   Inferring schema for a temp-table..........................................................................167
   Inferring schema for a ProDataSet..........................................................................167
Inferring data-relations from nested XML and XML Schema........................................168
Examples of additional XML support............................................................................169
   Simple example of a PARENT-ID-RELATION........................................................169
   XML schema with root element mapping to a temp-table........................................170
   XML schema with no explicit field definition for a parent table...............................172
   XML schema with parent table fields after nested child tables...............................174
   XML schema with parent table fields interspersed among child tables...............176
ABL representation of temp-tables and ProDataSets as XML Schema.........................177
Namespaces...........................................................................................................178
ABL-specific attributes on temp-table and ProDataSet definitions..............................178
Representing temp-tables in XML Schema................................................................179
Representing indexes in XML Schema........................................................................182
Representing data-relations in XML Schema.............................................................182

Appendix A: XML Schema and ABL Data Type Mappings.................................185
   Default mappings between ABL data types and XML Schema data types.............185
   Default mappings from other XML Schema data types to ABL data types............187

Appendix B: ABL to IANA Code Page Mappings..................................................189
   ABL to IANA code page mappings...........................................................................189

Appendix C: Command and Utility Reference.................................................193
   bproxsdto4gl utility.................................................................................................194

Index..................................................................................................................197
Preface

For details, see the following topics:

• Purpose
• Audience
• Organization
• Using this manual
• Typographical conventions
• Examples of syntax descriptions
• Example procedures
• OpenEdge messages

Purpose

This book provides a central programming guide for OpenEdge® developers who need their ABL (Advanced Business Language) procedures to work with XML documents.

Audience

This book is intended for ABL programmers working with XML documents in their ABL procedures. The book assumes that you have a basic working knowledge of XML. For advanced features, a basic working knowledge of XML Schema is also required.
Organization

Developing with XML in OpenEdge on page 21
Provides an overview of the role of XML and the ABL features available for developing ABL applications that work with XML documents.

Reading and Writing XML with the Document Object Model (DOM) on page 31
Describes ABL support for XML documents with DOM, including the receipt and processing of XML documents and the creation and transmission of XML documents.

Reading XML Documents with the Simple API for XML (SAX) on page 55
Describes ABL support for reading documents with SAX and includes provides a reference for SAX callback procedures.

Writing XML Documents with the Simple API for XML (SAX) on page 95
Describes ABL support for reading documents with SAX.

Reading and Writing XML Data from Temp-Tables and ProDataSets on page 117
Describes how temp-tables and ProDataSets can serialize their data to an XML document and serialize their definitions to XML Schema documents. This chapter also describes how you can read or read and load XML data and schema into a temp-table or ProDataSet.

XML Schema and ABL Data Type Mappings on page 185
Lists data type mappings needed to support ABL applications that interoperate with XML.

ABL to IANA Code Page Mappings on page 189
Lists code page mappings needed to support ABL applications that interoperate with XML.

Command and Utility Reference on page 193
Describes the bproxsdto4gl utility.

Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation Overview page on Progress Communities:
References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever integer appears, this is a reference to the INTEGER or INT64 data type.
- Wherever character appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.
- Wherever decimal appears, this is a reference to the DECIMAL data type.
- Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This manual uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>Convention</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a simultaneous key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a sequential key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed width</td>
<td>A fixed-width font is used in syntax, code examples, system output, and file names.</td>
</tr>
<tr>
<td>Fixed-width italics</td>
<td>Fixed-width italics indicate variables in syntax.</td>
</tr>
<tr>
<td>Fixed-width bold</td>
<td>Fixed-width bold italic indicates variables in syntax with special emphasis.</td>
</tr>
<tr>
<td>UPPERCASE fixed width</td>
<td>ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.</td>
</tr>
<tr>
<td>Period (.) or colon (;)</td>
<td>All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>{ }</td>
<td>Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding items.</td>
</tr>
</tbody>
</table>
Examples of syntax descriptions

In this example, **ACCUM** is a keyword, and *aggregate* and *expression* are variables:

**Syntax**

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, **STREAM stream**, **UNLESS-HIDDEN**, and **NO-ERROR** are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [, constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```

In this example, **EACH**, **FIRST**, and **LAST** are optional, but you can choose only one of them:

**Syntax**

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```
In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one expression or SKIP [ ( n ) ], and any number of additional expression or SKIP [ ( n ) ] is allowed:

**Syntax**

```
MESSAGE { expression | SKIP [ ( n ) ] } ... 
```

In this example, you must specify {include-file, then optionally any number of argument or &argument-name = "argument-value", and then terminate with }:

**Syntax**

```
{ include-file
  [ argument | &argument-name = "argument-value" ] ... }
```

**Long syntax descriptions split across lines**

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

**Syntax**

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```
Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:

Syntax

ASSIGN { [ FRAME frame ] } { field [ = expression ] } [ WHEN expression ] . . .

| { record [ EXCEPT field ] . . . ] }

Example procedures

This manual may provide example code that illustrates syntax and concepts. You can access many of the example files, and details for installing them, from the following locations:

- A self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center
- The OpenEdge Product Documentation Overview page on Progress Communities:

https://community.progress.com/technicalusers/w/opendedgegeneral/
1329.openedge-product-documentation-overview.aspx

Once installed, you can locate the example files for this manual in the following path under the OpenEdge Documentation and Samples installation directory:

<table>
<thead>
<tr>
<th>This directory . . .</th>
<th>Contains examples for the following documents . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>src\prodoc\dotnetobjects</td>
<td><strong>OpenEdge Development: GUI for .NET Programming</strong></td>
</tr>
<tr>
<td>src\prodoc\dynamics</td>
<td>The Progress Dynamics documentation</td>
</tr>
<tr>
<td>src\prodoc\getstartoop</td>
<td><strong>OpenEdge Development: Object-oriented Programming</strong></td>
</tr>
<tr>
<td>src\prodoc\handbook</td>
<td><strong>OpenEdge Getting Started: ABL Essentials</strong></td>
</tr>
<tr>
<td>src\prodoc\interfaces</td>
<td><strong>OpenEdge Development: Programming Interfaces</strong></td>
</tr>
<tr>
<td>src\prodoc\json</td>
<td><strong>OpenEdge Development: Working with JSON</strong></td>
</tr>
</tbody>
</table>
OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.

- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is **200**:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.
Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose Help > Recent Messages to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose Help > Messages and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the HELP key or F1.

On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press F3 to access the menu bar, then choose Help > Messages.
3. Type the message number and press ENTER. Details about that message number appear.
4. Press F4 to close the message, press F3 to access the Procedure Editor menu, and choose File > Exit.
Developing with XML in OpenEdge

This chapter introduces the OpenEdge® features that allow developers to use XML documents in their ABL (Advanced Business Language) applications. As background, the chapter also describes the importance of XML to current application development projects.

For details, see the following topics:

• About XML
• Developing XML-enabled ABL applications with DOM
• Developing XML-enabled ABL applications with SAX
• Serializing ABL temp-tables and ProDataSets to and from XML

About XML

The Extensible Markup Language (XML) is a data format used for exchanging structured data. The body of an XML document contains data and markup which encodes a description of the document's logical structure. XML is hardware and software independent and is the most widely used markup language today. It is also the preferred standard for data exchange between heterogeneous applications.

This section provides a quick overview of some essential XML information and concludes with some recommendations for getting more in-depth information about XML:

• Role of XML on page 22
• Benefits of XML on page 22
Role of XML

A great many of today’s application developers must address interoperability between applications not initially designed to communicate with each other. Small and large enterprises both tackle the problem of coaxing multiple applications to work together to keep up with ever-evolving business processes. Interoperability challenges can require tying together different:

- Hardware platforms
- Operating systems
- Data protocols
- Commercial software applications
- Local custom software applications

So how can a developer create business processes that leverage the functionality distributed on different corporate computing assets and make them interoperate? Assuming the networking infrastructure is in place to access the required assets, the last essential task needed to achieve interoperability is data exchange. Data exchange requires communicating the values that represent business knowledge and the schema that describes those values. Both are needed to drive the business processes built by linking distributed software applications.

Fortunately, the rise of the Internet has provided a universal, standards-driven platform to achieve interoperability between heterogeneous application assets within an enterprise or even among a larger circle of an enterprise and its business partners.

The development of the Internet is being guided by many groups that all promote interoperability. For example, the World Wide Web Consortium (W3C) develops interoperability technologies for the web. To that end, this consortium approved a data exchange standard called the Extensible Markup Language or XML. A markup language is a data description standard that includes both data and markup in a single document. (In XML, you can think of the markup and data as being analogous to field name and field value pairs presented in a logical structure.)

The W3C has also defined a companion standard to XML to provide rich schema definition for the elements of an XML document. This standard is known as XML Schema and it defines the means for defining the structure, content, and semantics of XML documents with much more detail. The language of XML Schema is known as the XML Schema Definition language (XSD).

The combination of XML and XML Schema provides an easy, standards-based vehicle for tackling a very wide array of data exchange challenges.

Since its introduction, XML has quickly grown to be the most widely-used markup language. XML makes it easy to describe data that must be used by more than one application. Its ease of use makes it a popular tool for data exchange in many scenarios, not just Internet applications. Any two applications that are XML-enabled can share data and schema.

Benefits of XML

Besides the fact that XML is a widely accepted standard, several other attributes of XML contribute to its rapid acceptance and growing popularity:
• Most importantly, XML is extensible. An XML document can grow to include and describe more data, yet each application that uses the document need only be concerned with the XML content that matters to it.

• XML is self-describing. The markup and hierarchical structure of the XML document can in some cases be interpreted by applications that do not know ahead of time what to expect from the document.

• XML is simple text. This fact makes it suitable and safe for transferring across platforms and operating systems that do not readily share more complex document types. As text, XML can also be readily displayed and edited in simple editors.

• XML is robust. It can describe any kind of data with the help of its associated schema-description languages.

• XML is easy to learn, easy to read, and easy to understand.

Benefits of ABL XML development in OpenEdge

Some of the most immediately noticeable ways OpenEdge supports your XML development work include:

• The standard installation of OpenEdge includes built in parsers for reading and writing XML.

• The Procedure Editor automatically applies hierarchical whitespace formatting and color to make reading and scanning XML documents easy.

• XML parser functionality is accessed using the familiar concepts of ABL objects, methods, and attributes.

• ABL XML-enablement features are simple enough to quickly implement basic use cases and rich enough to support the implementation of more complex use cases.

Benefits of SOA XML development in OpenEdge

The purpose of this document is to introduce the XML-enablement features available to OpenEdge ABL developers. But, it is also important to take a moment to relate XML and the ABL XML-enablement features to the latest application development methodology called Service Oriented Architecture (SOA).

What is SOA? Any traditional application is likely to include discrete functions that solve particular, well-defined business problems. If such a discrete business function were encapsulated and made available over a network, it could be reused by other applications. Conceptually, this type of business function is called a service and the applications that use it are called clients. A Service Oriented Architecture, then, is an application development, deployment, and management infrastructure where the major functions of the application are delivered as services.

A service is most useful when it is widely available to all current and potential clients, and therefore SOAs are built on widely accepted and widely available industry standards. XML is one of those key standards. Not only is XML widely used in SOAs, some of the new standards for SOA development are built upon XML.

Some important uses of XML in SOAs, include:

• Web services — A Web service is an application that can be accessed over the Internet using industry-standard protocols. To access this application, a client (Web service client) invokes operations that are described using Web Services Description Language (WSDL) and sent to the application using Simple Object Access Protocol (SOAP) over HTTP. WSDL and SOAP
are **defacto** industry Web service standards built on XML. Configuration information for Web services also uses XML.

- **Messaging** — Point-to-point and publish and subscribe messaging schemes can be part of an SOA in OpenEdge by using the Sonic ESB adapter. The Sonic messaging products provide this type of functionality by combining Java Servlet Engines with Java Messaging Services (JMS) for the exchange of information. The JMS messages are XML. In fact, XML is quiet a heavily used standard in the Java world in general.

- **Proxies** — The OpenEdge Open Client Proxy Generator (ProxyGen) utility generates the Web service (client interface) definition in the form of a *Web service mapping* (WSM) file that is used during Web service deployment. WSM files are XML documents.

All of these facts serve to emphasize that XML is now (and will continue to be) an important tool for programmers. If you plan to move towards SOA applications, taking the time to learn XML and XML Schema is essential. This is true even though OpenEdge provides many time-saving tools, like ProxyGen, that mask your need to know or work directly with particular kinds of XML standards, like WSDL files.

A second point to make is that OpenEdge provides the functionality to allow ABL applications to be exposed as Web services and for ABL applications to consume Web services from other available application domains. The XML-enablement features of ABL enrich the possibilities for participating in SOAs by giving you the tools to perform direct and indirect manipulation of XML documents within ABL.

### Recommended resources for learning XML basics

For XML learning, the basic definition documents and tutorials available from the W3C are a good first step. You can browse the available documents at this Web site:

```
http://www.w3.org
```

The following book is an excellent reference to XML and related technologies:


Progress Software's Education Services also offers two in-depth classes that are available over the Internet:

- **XML Essentials**
- **OpenEdge Development with XML**

Information about these classes can be found at this address:

```
http://www.progress.com/services/education/index.ssp
```
Developing XML-enabled ABL applications with DOM

The Document Object Model (DOM) is the original application programming interface (API) for XML documents, as defined by the W3C. It continues to be the W3C recommendation for XML parsing.

You use the DOM API to build a hierarchical tree structure that persists in memory and represents your entire XML document. With this API, you can call the parser to build such a tree for you in memory from an XML document. The DOM tree is then available for you to easily manipulate using standard tree-traversal logic. For example, an XML document may represent a list of address changes generated from a self-service Web site. Your application might need to load the XML document into a DOM tree, perform some logical validation on the data, and only then use the data to update customer address records in your database.

You could also use the API to programmatically build a DOM tree in memory and then write that tree out as an XML document. For example, you may need to generate a list of customer address changes for a business partner.

DOM advantages

The general advantages of DOM include:

- Data persists in memory
- You can go forwards and backwards in the tree (random access)
- You can make changes directly to the tree in memory

DOM limits

The general limits of DOM include:

- The entire document must be parsed before the tree is available to the application.
- You cannot load a subset of the document.
- Because the whole tree must exist in memory, it may not be suitable for handling very large XML documents.
- A single error in the XML file may prevent the document from being loaded. This limit makes DOM less attractive for use cases where XML is being retrieved in a stream rather than as a static document.

ABL support

ABL implements DOM with two ABL object types:

- **X-document** — Represents an entire XML document tree
- **X-noderef** — Represents a reference to a single node in the XML tree of the document

The objects are defined with the `CREATE X-DOCUMENT` and `CREATE X-NODEREF` statements. Methods and attributes on the objects provide the rest of the functionality.
If you provide a Document Type Definition document (DTD) or an XML Schema document, then ABL can perform the appropriate schema validation on reads of the DOM tree.

More information

For complete information, see Reading and Writing XML with the Document Object Model (DOM) on page 31.

Developing XML-enabled ABL applications with SAX

The Simple API for XML (SAX) is an application programming interface (API) for XML documents. It was developed by a group of programmers, and even though it is not a W3C recommendation, it is widely used industry standard.

SAX is a streaming model that processes one element at a time and provides mechanisms for you to respond to the current element before it is flushed and the parser moves on to the next element. Contrast this with the memory-resident model of DOM.

When an XML document is accessed by a SAX application, as the XML parser encounters an XML element, it parses that element and provides its information to the application immediately, through a callback procedure. The callback procedure allows you to provide custom programming on how to handle each type of information that the parser provides to your program. On the other hand, if you do not have any custom code, you do not need to provide callback procedures. The parser's default behavior is to simply read the whole document.

So, while the DOM API decomposes an XML document into a set of nodes in a hierarchical tree, the SAX API decomposes the document into a series of procedure calls. Your application must act on the information presented as it is provided to the application. SAX can only stream forward during the parse. At the conclusion of the parse, nothing is left in memory.

As an example, if you were parsing a list of address changes intended for your database, your application would need to validate and update each change as you retrieved it from the parse.

The SAX technology built into OpenEdge also allows you to write an XML document in a forward-streaming way.

SAX advantages

The general advantages of SAX include:

- The nature of a streaming model means that you need far less memory to process large XML documents.
- You do not have to process the entire document. Use callback procedures to identify and respond to only the XML elements you are interested in.
- You can halt the parse at any time.
- You can parse an XML document that is not well formed.
- SAX provides callback procedures that let you to provide more error handling.
SAX limits

The general limits of SAX include:

• You cannot back up in the parse.
• You must control the context. In other words, you must be able to grab the data you need as
  it goes by, while ignoring the data you don't need.
• There is no structure in memory to do in-place updates.
• The order in which you write the XML is important and you cannot modify the XML once it has
  been written.
• It is possible to create an XML document that is not well-formed.
• You do not know if an XML document is well-formed or conforms to a schema until you have
  parsed (and processed) the entire XML document.

ABL support

ABL implements SAX with three ABL object types:

• **SAX-reader** — Represents a parser object that reads an XML document
• **SAX-writer** — Represents an object used for writing an XML document as a stream of
  characters
• **SAX-attributes** — Contains any XML attribute values that may exist for the current XML
  element

The objects are defined with the `CREATE SAX-READER`, `CREATE SAX-ATTRIBUTES`, and `CREATE
SAX-WRITER` statements. Methods and attributes on the objects provide the rest of the functionality.

If you provide a Document Type Definition document (DTD) or an XML Schema document, then
ABL can perform the appropriate schema validation on SAX reads.

SAX support also includes definition of the callback methods expected by the SAX parser.

More information

For complete information, see **Reading XML Documents with the Simple API for XML (SAX)** on
page 55 and **Writing XML Documents with the Simple API for XML (SAX)** on page 95.

Serializing ABL temp-tables and ProDataSets to and from XML

ABL temp-tables and ProDataSets have the ability to serialize their data to an XML document and
serialize their definitions to XML Schema documents. Similarly, you can read XML data, load
schema, or both into a temp-table or ProDataSet.
You can consider the DOM and SAX features of ABL as direct manipulation of XML. Note that you must understand XML to use these features. By contrast, the temp-table and ProDataSet serialization features are convenience features that protect the developer from needing to know much about XML or accessing XML with ABL. On the other hand, the advanced abilities of these features can be used by knowledgeable XML developers for advanced use cases.

Features

The XML features of temp-tables and ProDataSets allow you to take advantage of their rich relational features while providing a standards-based method for sharing data and schema with application partners. These XML features include the following:

- Load XML Schema to create an empty temp-table or ProDataSet.
- Read XML data, XML Schema, or both to populate an empty temp-table or ProDataSet.
- Read XML data, XML Schema, or both into temp-tables and ProDataSets that already contain data and schema.
- Write XML data, XML Schema, or both from a temp-table or ProDataSet to XML documents.
- Perform round-trip XML write/reads. A round trip is an XML write of data during one session followed by an XML read in another session. For example, suppose your application needs to write data as XML to the file system when the user does not have a network connection. When the connection is established, your application can check for XML data files and read them back in.

The XML features are available as attributes and methods on:

- Temp-table objects
- Temp-table buffer objects (acts on the entire temp-table, not just the current buffer contents)
- ProDataSet objects

Use cases

The XML read and write features are robust and versatile. The examples described below demonstrate common problems that can be solved with the features:

- Provide interoperability between OpenEdge and another XML-enabled platform or application.
- Use XML data and XML Schema as a persistent storage mechanism between ABL sessions.
- Provide XML Schema from ABL for use in third-party tools.
- Simplify or replace existing ABL code that performs XML reads and writes on temp-tables and ProDataSets.

ABL support

The serialization features are provided by the following methods on temp-tables, temp-table buffers, and ProDataSets:

- **READ-XML( ) method** — Reads an XML document into an ABL object
• **READ-XMLSCHEMA( ) method** — Reads an XML Schema document into a dynamic ABL object to create its definition

• **WRITE-XML( ) method** — Writes the contents of the ABL object to an XML document

• **WRITE-XMLSCHEMA( ) method** — Writes the ABL definition of an object to an XML Schema document

• **SERIALIZE-ROW( ) method** — Serializes the buffer's current row to JSON or XML

You do not need to be familiar with XML and XML Schema to use these methods and the associated attributes.

**More information**

For complete information, see *Reading and Writing XML Data from Temp-Tables and ProDataSets* on page 117.
ABL supports two XML programming interfaces: the Document Object Model (DOM) and the Simple API for XML (SAX). This chapter provides a quick overview of key XML terminology and then covers the DOM interface.

In addition to basic XML knowledge, the minimum requirement for working with XML in ABL is a familiarity with using and manipulating ABL objects and a knowledge of network communication by way of either the Web or sockets.

For details, see the following topics:

- XML terminology
- The ABL DOM interface
- Creating XML output from ABL
- Reading XML input into ABL
- Namespaces
- Internationalization
- Error handling
- Validation
- Method and attribute summary
XML terminology

The following overview is a refresher of key XML terms and concepts that occur throughout this manual. It is provided to ensure the manual's use of familiar XML terminology matches your own, but it is not a substitute for an XML tutorial. If you are unfamiliar with XML, see Developing with XML in OpenEdge on page 21.

The Extensible Markup Language (XML) is a data format for structured document interchange on the Web and other networks. It is hardware-architecture neutral and application independent. XML documents are composed of storage units called elements that 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 storage layout and logical structure.

XML document structure

This section introduces the terms used to describe the parts of an XML document, starting at the top level and working down to the smallest parts.

XML documents are made up of two parts, called the prolog and the body.

XML prolog

The prolog contains optional information such as the XML version the document conforms to, information about the character encoding used to encode the contents of the document, and a reference to either a document type definition (DTD) or XML Schema document which describes the grammar and vocabulary of the document. The XML Schema document is the more modern way to describe XML grammar and vocabulary. XML Schemas and DTDs are usually stored in external documents and the prolog can reference both XML Schemas and DTDs.

This simple example illustrates the prolog:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE maillist SYSTEM "maillist.dtd">
```

XML body

The body contains a single top-level element called the root element, which contains all other elements and other markup information.

This simple example illustrates the prolog and body, where addresses is the root element and encloses the entire body, or content, of the XML document:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE maillist SYSTEM "maillist.dtd">
<addresses>
  <address>...</address>
  <address>...</address>
  <address>...</address>
</addresses>
```
**DTDs and XML Schema documents**

DTDs and XML Schema documents are rules that define the elements that can exist in a particular document or group of documents, and the relationships among the various elements. A DTD or XML Schema can be part of the content of an XML document or can be separate from it and referred to by the XML documents. Best practice calls for the DTDs and XML Schema documents to be separate from the XML content for reuse and maintainability. Here is an example of a DTD:

```
<?xml encoding="US-ASCII"?>
<!-- DTD for a an XML document that stores Customer names and numbers-->
<!ELEMENT customer(name, cust-num)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT cust-num(#PCDATA)>
```

Here is an example of an XML Schema:

```
<?xml version="1.0"?>
<!-- XML Schema document for a an XML document that stores Customer names and numbers-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="">
  <xsd:element name="customer" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustNum" type="xsd:int"/>
        <xsd:element name="Name" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

**XML elements**

*Elements* represent the logical components of documents. They can contain data or other elements. For example, a customer element can contain a number of column (field) elements and each column element can contain one data value. An element is composed of a start tag, data, and an end tag. Here is an example of an element:

```
<table>
<thead>
<tr>
<th>Start tag</th>
<th>Data</th>
<th>End tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;customer-name&gt;</td>
<td>Clyde Peter Jones</td>
<td>&lt;/customer-name&gt;</td>
</tr>
</tbody>
</table>
```

Here is an example of elements that contain other elements:

```
<phone>
  <entry>
    <name>Chip</name>
    <extension>3</extension>
  </entry>
</phone>
```
In this example, name and extension are child elements of entry, which is a child element of phone. Similarly, phone is a parent element of entry, which is a parent element of name and extension.

The top-level element, the one that is the parent of all other elements is referred to as the root element. A child element that cannot have its own child elements, as defined by the DTD or XML Schema, is referred to as a leaf element. In the example above, name and extension are leaf elements.

To conclude, a typical discussion of XML elements that describes hierarchy will use these terms:

- Root element
- Parent element
- Child element
- Leaf element

In the previous example, the XML contains whitespace to make the logical, hierarchical structure of the XML apparent to human readers. XML is valid with or without this whitespace. The following version is also valid XML:

```xml
<phone>
  <entry>
    <name>Chip</name>
    <extension>3</extension>
  </entry>
</phone>
```

**XML attributes**

Elements can have additional information called attributes attached to them. Attributes describe properties of elements. Here is an example of an element with an attribute, `emp-num`:

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

An attribute is some data that further describes its element. When comparing an XML document to a database record, a programmer is free to interpret a particular database field as an attribute of the parent element (the record or database row) or as a child element of the parent. Thus, you could also see the previous example as:

```xml
<employee>
  <emp-num>1</emp-num>
  <name>Mary</name>
</employee>
```
XML empty elements

An empty element is an element that is complete by itself; it never contains other elements. Rather than being composed of a start tag, data, and an end tag, the empty element is a combined start and end tag. Typically, an empty element is used as a flag of some kind or perhaps as an element that contains its information in its attributes. In this example, \texttt{extension} has been converted to an empty tag:

\begin{verbatim}
<phone>
  <entry>
    <name>Chip</name>
    <extension number="3"/>
  </entry>
</phone>
\end{verbatim}

XML entities

An entity is simply a way to name a placeholder that the XML parser will replace with the correct value when it interprets the XML document. Probably most familiar are the predefined character entities that are used as placeholders for special characters that are illegal in XML or are frequently mishandled by platform-specific text handlers. For example \texttt{&lt;} and \texttt{&gt;} represent the "less than sign" character (<) and the "greater than sign" character (>).

More importantly, XML entities provide you with the ability to create placeholders for reoccurring text, or placeholders for values that will be resolved at parse time. Pointers to external resources such as DTDs or XML Schema documents often make use of entities.

If you are unfamiliar with the different entity types in XML, this is an area worth studying before beginning your XML development.

XML namespaces

Namespaces provide a mechanism for ensuring unique names for elements and attributes. Suppose you need to create a mailing list combining the customer data from two separate client companies. The ABC company uses the \texttt{name} element to hold the customer's company name while the XYZ company uses the \texttt{name} element to hold the customer's contact name. Combining these two data sets becomes a difficult task. Fortunately, each company's XML document has defined a unique namespace that acts as a prefix for all element names. So, your application can easily understand that \texttt{abccompany:name} and \texttt{xyzcompany:name} are two different things.

Like entities, the rules and options for defining XML namespaces are a concept you need to understand before beginning your XML development.

The ABL DOM interface

The Document Object Model (DOM) is the original application programming interface (API) for XML documents, as defined by the W3C. It continues to be the W3C recommendation.
A software module called an XML parser is used to read XML documents and provide access to their content and structure. The XML parser does its work on behalf of another software module—your application. ABL simplifies your interaction with DOM and its parser by providing the familiar ABL programming constructs of built-in objects, handles, attributes, and methods. This section describes the ABL objects that make up the DOM interface.

Note: The DOM API is designed to be compatible with a wide range of programming languages. In implementing DOM within the ABL, Progress Software Corporation adapted the specification to fit smoothly into the ABL environment. In some cases, the World Wide Web Consortium (W3C) naming conventions conflicted with familiar ABL naming conventions. In other cases, functionality in the DOM specification was already available in ABL. In all these cases, the established ABL naming and functionality was preserved.

Prerequisites

The ABL DOM interface is most quickly employed if:

- You understand XML
- You understand DOM programming
- You understand ABL programming including the use of built-in objects and their handles, methods, and attributes

If you have not yet learned XML, then DOM programming is not a productive starting point for you. Take the time to learn basic XML with any one of the many online or printed tutorials.

If you understand DOM, then you will find the ABL implementation to be a useful one that implements all the features of DOM that are needed by an ABL programmer. It is not a complete implementation of the entire DOM specification.

If you are not a DOM programmer, you can use this chapter to understand the ABL implementation and to develop pragmatic skills for reading and writing XML documents, as well as building XML documents or fragments in memory. The ABL implementation takes a somewhat difficult technology and makes it much easier to exploit within the ABL environment.

If you are an ABL programmer not yet experienced with built-in ABL objects, then you will find this an easy skill to acquire. If you do not understand something in this chapter, you can refer to the OpenEdge Development: ABL Reference for complete information about the object, handle, attribute, or method. Note that this ABL skill level does not require you to know or understand the support for true object-oriented programming also provided by the ABL. Think of ABL built-in or system objects as useful, consistent, easy-to-use encapsulations of system resources or technologies that increase the richness of the ABL programming environment.
Representing an XML document in ABL DOM

DOM builds a hierarchical tree structure that persists in memory and represents your entire XML document. With the ABL interface, you can call the parser to build such a tree for you in memory from an XML document. The DOM tree is then available for you to easily manipulate using standard tree-traversal logic. For example, an XML document may represent a list of address changes generated from a self-service Web site. Your application might need to traverse the tree and do some logical validation before using the XML data in the tree to update customer address records in your database.

You could also use the API to programmatically build a DOM tree in memory and then write that tree out as an XML document. For example, you may need to generate a list of customer address changes for a business partner.

Note: ABL has defined a set of extensions to ABL to allow the use of XML through the DOM interface. These extensions provide ABL 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.

In an ABL application, an X-document object can represent an XML document. Like all ABL objects, the programming flow for using an X-document object follows this pattern:

• Define a variable of type HANDLE
• Use the appropriate CREATE statement to create the object in memory and assign the pointer to that object to your HANDLE variable—the HANDLE variable is your interface to the object
• Use attributes and methods on the handle to initialize or configure the object
• Use attributes and methods on the handle to complete your programming tasks
• Use the DELETE OBJECT statement to destroy the object and remove it from memory

For example:

```
DEFINE VARIABLE hDocument AS HANDLE NO-UNDO.
CREATE X-DOCUMENT hDocument.
... 
DELETE OBJECT hDocument.
```

Since an X-document represents the tree of an XML document, it is a container for the branches and leaves of that tree. In XML, the branches and leaves are elements. In DOM, the branches and leaves are called nodes. There is a relationship between elements and nodes, but it is not a direct one-for-one correspondence. In DOM, a single XML element may be further decomposed into many nodes. The relationships between DOM nodes in a tree and the relationship of XML components can follow slightly different logic. For example, an attribute of an XML element is thought of as a part of the element. In DOM, that attribute becomes a separate node related to the node that represents its associated element.

So, how do you keep all this straight? Basically, when you begin a programming task, you need to ensure that your XML input documents are valid, well-formed XML and that the design of your output XML documents produces valid, well-formed XML. For the rest of your project, you will be immersed in ABL and DOM logic. Think in terms of DOM nodes and the rules of DOM relationships, and you will be successful.
Representing a DOM node in ABL

In essence, X-document represents the collection of nodes, their content of the nodes, and all the relationships between nodes.

The X-noderef object is an ABL object that is a reference to any node in a DOM tree (except a DOCUMENT node, which is the X-document). In ABL, you can create a handle to an X-noderef object, which in turn is a cursor pointing to a DOM node in a DOM tree (represented by an ABL X-document object).

The life cycle of an X-noderef object is similar to the X-document object's. For example:

```
DEFINE VARIABLE hNode AS HANDLE NO-UNDO.
CREATE X-NODEREF hNode.
... 
DELETE OBJECT hNode.
```

There are many types of DOM nodes as identified by the SUBTYPE attribute of both the X-document and X-noderef objects. 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 ABL X-noderef object can be one of the following DOM node types, as indicated by the SUBTYPE attribute:

- ATTRIBUTE
- CDATA-SECTION
- COMMENT
- DOCUMENT-FRAGMENT
- ELEMENT
- ENTITY-REFERENCE
- PROCESSING-INSTRUCTION
- TEXT

The default subtype is ELEMENT.

The following table shows X-noderef object SUBTYPE values, and how the NAME and NODE-VALUE attributes are different based on SUBTYPE.
Table 1: X-noderef SUBTYPE, NAME, and NODE-VALUE attributes

<table>
<thead>
<tr>
<th>SUBTYPE attribute</th>
<th>NAME attribute</th>
<th>NODE-VALUE attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE</td>
<td>The name of the attribute. Primarily used with namespaces</td>
<td>The value of the attribute</td>
</tr>
<tr>
<td>DOCUMENT-FRAGMENT</td>
<td>Ignored</td>
<td>Content of the fragment</td>
</tr>
<tr>
<td>ENTITY</td>
<td>XML tag name, including namespace information</td>
<td>Null</td>
</tr>
<tr>
<td>ENTITY-REFERENCE</td>
<td>The name of the entity referenced without the leading ampersand and trailing semicolon</td>
<td>Null</td>
</tr>
<tr>
<td>TEXT</td>
<td>Ignored</td>
<td>Content of the text node</td>
</tr>
<tr>
<td>CDATA-SECTION</td>
<td>Ignored</td>
<td>Content of the CDATA-SECTION</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Ignored</td>
<td>Content of the COMMENT</td>
</tr>
<tr>
<td>PROCESSING-INSTRUCTION</td>
<td>The target, which is the first XML token following the &lt;? markup</td>
<td>Content excluding the target</td>
</tr>
</tbody>
</table>

Creating XML output from ABL

This section describes how to create XML output from ABL. To create an output XML document:

1. Create an X-document object.
2. Create an X-noderef object to represent your root node.
3. Create an X-noderef object to represent nodes for your document.
4. Create the root node and append it to the document.
5. Create each specific node required.
6. Append each node to its parent.
7. Set node attributes. (Steps 5 through 7 should be repeated as necessary.)
8. Save the document as an XML file.
9. Delete the objects.
The root node reference object

The root node is the unique top-level node that is not a child of any other node. All other nodes are children or other descendents of the root node. A root node is necessary so that you have a top-level node to which you can append the child nodes.

Creating and appending a node

In order to create an actual XML node, you use the CREATE-NODE( ) method on the parent object. After the node is created, you must append it to its parent by using the APPEND-CHILD( ) method. The following code fragment is an example of creating and appending the root node:

```
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
CREATE X-NODEREF hRoot.
hDoc:CREATE-NODE(hRoot,"Root","ELEMENT").
hDoc:APPEND-CHILD(hRoot).
```

Setting node attributes and values

You can set the attributes of a node or the value of a node either before or after it is appended by using the SET-ATTRIBUTE( ) method or the NODE-VALUE attribute. The following code fragment depicts setting attributes of the "employee" ELEMENT node with the SET-ATTRIBUTE( ) method and setting the value of the "address" TEXT node with the NODE-VALUE attribute. Both objects are X-noderef objects. Note that in this case, the "employee" node is a child of the root node and the "address" node is a child of the "employee" node:

```
hDoc:CREATE-NODE(hEmp,"employee","ELEMENT").
hDoc:CREATE-NODE(hAddr,"","TEXT").
hEmp:SET-ATTRIBUTE("empID","10263").
hEmp:SET-ATTRIBUTE("empDept","Sales").
hRoot:APPEND-CHILD(hEmp).

hEmp:APPEND-CHILD(hAddr).
hAddr:NODE-VALUE = "121 State Street".
```

For more information on attributes and methods associated with the X-document object and the X-noderef object, see their entries in OpenEdge Development: ABL Reference.

Example of creating an output XML file

The following sample program creates an XML file consisting of all fields in all the customer records where the cust-num is less than "5". You must use the SAVE( ) method on the X-DOCUMENT object in order to create the actual XML file. For example:
/* i-outcus.p - Export the Customer table to an xml file*/
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
DEFINE VARIABLE hRow AS HANDLE NO-UNDO.
DEFINE VARIABLE hField AS HANDLE NO-UNDO.
DEFINE VARIABLE hText AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuf AS HANDLE NO-UNDO.
DEFINE VARIABLE hDBFld AS HANDLE NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hRow.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.

hBuf = BUFFER Customer:HANDLE.

/* Set up a root node */
hDoc:CREATE-NODE(hRoot,"Customers","ELEMENT").
hDoc:APPEND-CHILD(hRoot).

FOR EACH Customer WHERE Customer.CustNum < 5:
  hDoc:CREATE-NODE(hRow,"Customer","ELEMENT"). /* create a row node */
  hRoot:APPEND-CHILD(hRow). /* put the row in the tree */
  hRow:SET-ATTRIBUTE("CustNum", STRING(Customer.CustNum)).
  hRow:SET-ATTRIBUTE("Name", Customer.Name).

/* Add the other fields as tags in the xml */
REPEAT ix = 1 TO hBuf:NUM-FIELDS:
  hDBFld = hBuf:BUFFER-FIELD(ix).
  IF hDBFld:NAME = "CustNum" OR hDBFld:NAME = "Name" THEN NEXT.

  /* Create a tag with the field name */
  hDoc:CREATE-NODE(hField, hDBFld:NAME, "ELEMENT").

  /* Put the new field as next child of row */
  hRow:APPEND-CHILD(hField).

  /* Add a node to hold field value. The empty string ("") represents the
   value that will be set later. */
  hDoc:CREATE-NODE(hText, ",", "TEXT").

  /* Attach the text to the field */
  hField:APPEND-CHILD(hText).
  hText:NODE-VALUE = STRING(hDBFld:BUFFER-VALUE).
END.

/* Write the XML node tree to an xml file */
hDoc:SAVE("file","cust.xml").

DELETE OBJECT hDoc.
DELETE OBJECT hRoot.
DELETE OBJECT hRow.
DELETE OBJECT hField.
DELETE OBJECT hText.
A partial output of the above program appears below:

```xml
<?xml version='1.0' ?>
<Customers>
  <Customer Name="Lift Line Skiing" Cust-num="1">
    <Country>USA</Country>
    <Address>276 North Street</Address>
    <Address2></Address2>
    <City>Boston</City>
    <State>MA</State>
    <Postal-Code>02114</Postal-Code>
    <Contact>Gloria Shepley</Contact>
    <Phone>(617) 450-0087</Phone>
    <Sales-Rep>HXM</Sales-Rep>
    <Credit-Limit>66700</Credit-Limit>
    <Balance>42568</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
  </Customer>
  <Customer Name="Urpon Frisbee" Cust-num="2">
    <Country>Finland</Country>
    <Address>Rattipolku 3</Address>
    . . .
  </Customer>
</Customers>
```

**Note:** The previous example adds carriage returns and indentations for readability; the actual file contains one long string.

### Writing an XML file to a MEMPTR, a stream, or a LONGCHAR

You can also write an XML file to a MEMPTR or to an output stream or to a LONGCHAR as the following code fragments demonstrate.

The following fragment shows how to save an XML file to a MEMPTR:

```plaintext
DEFINE VARIABLE memfile AS MEMPTR NO-UNDO.
. . .
hDoc:SAVE("memptr",memfile). /* SAVE() will set the memptr size */
. . .
```

The following fragment shows how to save an XML file to a LONGCHAR:

```plaintext
DEFINE VARIABLE longstring AS LONGCHAR NO-UNDO.
. . .
hDoc:SAVE("LONGCHAR",longstring).
. . .
```
The following fragment shows how to save an XML file to an output stream:

```
DEFINE STREAM xmlstream.
... 
OUTPUT STREAM xmlstream TO custxml.xml.
hDoc:SAVE("stream","xmlstream").
OUTPUT CLOSE.
... 
```

### Reading XML input into ABL

This section describes how to read XML input into ABL.

To read in an XML file and process it:

1. Create an X-document object.
2. Create an X-noderef object to represent your root node.
3. Use the `LOAD()` method to read the input file.
4. Use the `GET-DOCUMENT-ELEMENT()` method to get the root node reference handle.
5. Create an X-noderef object to represent nodes for your document.
6. Using the `GET-CHILD()` method, read through the child nodes.
7. Using the `GET-ATTRIBUTE()` METHOD, the `NAME` attribute, the `NODE-VALUE` attribute and other attributes and methods, access the XML data.
8. Update the database or other fields as necessary.
9. Delete the objects.

### Loading an XML file

The `LOAD()` method reads the specified file into memory, parses it, optionally validates it, and makes its contents available to ABL. Once the XML file is in memory, you must get the handle to its root element by using the `GET-DOCUMENT-ELEMENT()` method. Once you have the root node handle, you can manipulate the remaining child nodes.

The following code fragment demonstrates loading an XML file called `myfile.xml`:

```
DEFINE VARIABLE hDocMine AS HANDLE NO-UNDO.
DEFINE VARIABLE hRootMine AS HANDLE NO-UNDO.

CREATE X-DOCUMENT hDocMine.
CREATE X-NODEREF hRootMine.

hDocMine:GET-DOCUMENT-ELEMENT(hRootMine).
... 
```
Loading an XML file from a LONGCHAR

An XML file can be read from a LONGCHAR as the following code fragment demonstrates:

```
DEFINE VARIABLE longstring AS LONGCHAR NO-UNDO.
... 
 hDoc:LOAD("LONGCHAR",longstring,FALSE).
... 
```

Loading an XML file from a MEMPTR

An XML file can be read from a MEMPTR as the following code fragment demonstrates:

```
DEFINE VARIABLE memfile AS MEMPTR NO-UNDO.
 FILE-INFO:FILE-NAME = "meminp.xml".
 SET-SIZE(memfile) = FILE-INFO:FILE-SIZE.

 INPUT FROM "meminp.xml" BINARY NO-CONVERT.
 IMPORT memfile.
 INPUT CLOSE.
 hDoc:LOAD("memptr",memfile,FALSE).
 ... 
```

Accessing the child nodes

Before you can work with a child node of an XML document, you must create a node reference object with which to access the node. Then you access the node by using the `GET-CHILD( )` method. The following code fragment shows obtaining the third child node of the parent node, hParent:

```
... 
 DEFINE VARIABLE hChildNode AS HANDLE NO-UNDO.
 CREATE X-NODEREF hChildNode.
 logvar = hParent:GET-CHILD(hChildNode,3).
 ... 
```
Using node attributes and values

You can get information about the child node by using various attributes and methods. For example, if you do not know how many nodes there are below the node referred to by the node reference, you can use the **NUM-CHILDREN** attribute. You can obtain or set the value of the node by using the **NODE-VALUE** attribute. For example:

```abl
DEFINE VARIABLE hChNode AS HANDLE NO-UNDO.
CREATE X-NODEREF hChNode.

REPEAT i = 1 TO hParent:NUM-CHILDREN:
    /* logvar is used for checking if the return value is zero. */
    logvar = hParent:GET-CHILD(hChNode, i).
    IF hChNode:_NODE-VALUE > 0 THEN
        hChNode:_NODE-VALUE = hChNode:_NODE-VALUE + i.
    . . .
```

You can obtain a list of an element's attribute names using the **ATTRIBUTE-NAMES** attribute, get the value of an attribute by using the **GET-ATTRIBUTE( )** method or set the value of an attribute by using the **SET-ATTRIBUTE( )** method. You can also **REMOVE-ATTRIBUTE( )**, as shown:

```abl
. . .
REPEAT i = 1 TO hNode1:NUM-CHILDREN:
    logvar = hNode1:GET-CHILD(hChNode, i).
    IF NOT logvar THEN LEAVE.
    entries = hNode1:ATTRIBUTE-NAMES.
    REPEAT j = 1 TO NUM-ENTRIES(entries):
        aname = ENTRY(j, entries).
        MESSAGE "attrname is " aname "value is " hNode1:GET-ATTRIBUTE(aname).
    END.
END.
. . .
```

In addition to creating nodes, you can **IMPORT-NODE( ), CLONE-NODE( ), and DELETE-NODE( )**. In addition to appending and getting a child, you can **REMOVE-CHILD( ), REPLACE-CHILD( ), and GET-PARENT( )**. The following example demonstrates the **CLONE-NODE( )** method:

**i-clone.p**

```abl
/* i-clone.p */
DEFINE VARIABLE hXref AS HANDLE NO-UNDO.
DEFINE VARIABLE hXref1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hText AS HANDLE NO-UNDO.
DEFINE VARIABLE hText1 AS HANDLE NO-UNDO.
DEFINE VARIABLE hClone AS HANDLE NO-UNDO.
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.

CREATE X-NODEREF hXref.
CREATE X-NODEREF hXref1.
CREATE X-NODEREF hText.
CREATE X-NODEREF hText1.
CREATE X-NODEREF hClone.
CREATE X-NODEREF hRoot.
```
CREATE X-DOCUMENT hDoc.

hDoc:CREATE-NODE(hRoot,"root","ELEMENT").
hDoc:INSERT-BEFORE(hRoot,?).
hDoc:CREATE-NODE(hXref,"customer","ELEMENT").
hDoc:CREATE-NODE(hXref1,"order","ELEMENT").
hDoc:CREATE-NODE(hText,?,"TEXT").
hDoc:CREATE-NODE(hText1,?,"TEXT").

/* Add the two element nodes to the root, each with a text*/
hXref:SET-ATTRIBUTE("id","54").
hXref:SET-ATTRIBUTE("name","Second Skin Scuba").
hRoot:APPEND-CHILD(hXref).
hXref:APPEND-CHILD(hText).
hXref1:SET-ATTRIBUTE("id","55").
hXref1:SET-ATTRIBUTE("name","Off the Wall").
hRoot:APPEND-CHILD(hXref1).
hXref1:APPEND-CHILD(hText1).

/* Save the file */

hDoc:SAVE("file","clone1.xml").

DELETE OBJECT hXref.
DELETE OBJECT hXref1.
DELETE OBJECT hText.
DELETE OBJECT hText1.
DELETE OBJECT hClone.
DELETE OBJECT hRoot.
DELETE OBJECT hDoc.

There are more methods and attributes that apply to the X-document object and the X-noderef objects. For more information on these attributes and methods, see their entries in the *OpenEdge Development: ABL Reference*.

### Examples of reading an input XML file

The *i-attnam.p* example shows reading in a file called "personal.xml", processing through all the child nodes and displaying information if the node name is "person".

#### i-attnam.p

```abl
/* i-attnam.p */
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
DEFINE VARIABLE lGood AS LOGICAL NO-UNDO.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.

hDoc:GET-DOCUMENT-ELEMENT(hRoot).

RUN getChildren(hRoot, 1).
DELETE OBJECT hDoc.
DELETE OBJECT hRoot.

PROCEDURE getChildren:
  DEFINE INPUT PARAMETER hParent AS HANDLE NO-UNDO.
```
DEFINE INPUT PARAMETER iLevel AS INTEGER NO-UNDO.

DEFINE VARIABLE ix AS INTEGER NO-UNDO.
DEFINE VARIABLE hNoderef AS HANDLE NO-UNDO.

CREATE X-NODEREF hNoderef.

REPEAT ix = 1 TO hParent:NUM-CHILDREN:
  lGood = hParent:GET-CHILD(hNoderef, ix).
  IF NOT lGood THEN LEAVE.
  IF hNoderef:SUBTYPE <> "element" THEN NEXT.
  IF hNoderef:NAME = "person" THEN
    MESSAGE "getattr id gives" hNoderef:GET-ATTRIBUTE("id")
    hNoderef:ATTRIBUTE-NAMES.
    RUN getChildren(hNoderef, (iLevel + 1)).
  END.
DELE   TE OBJECT hNoderef.
END PROCEDURE.

The i-incus.p example reads in the output file created by the previous program, i-outcus.p and creates temp-table entries.

i-incus.p

/* i-incus.p - Import the Customer table from an XML file */
DEFINE VARIABLE hDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hRoot AS HANDLE NO-UNDO.
DEFINE VARIABLE hTable AS HANDLE NO-UNDO.
DEFINE VARIABLE hField AS HANDLE NO-UNDO.
DEFINE VARIABLE hText AS HANDLE NO-UNDO.
DEFINE VARIABLE hBuf AS HANDLE NO-UNDO.
DEFINE VARIABLE hDBFld AS HANDLE NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
DEFINE VARIABLE jx AS INTEGER NO-UNDO.

/* So we can create new recs */
DEFINE TEMP-TABLE ttCustomer LIKE Customer.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hTable.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.
CREATE X-NODEREF hBuffer.
hBuf = BUFFER ttCustomer:HANDLE.

/* Read in the file created in i-outcus.p */
hDoc:LOAD("file", "cust.xml", FALSE).
hDoc:GET-DOCUMENT-ELEMENT(hRoot).

/* Read each Customer from the root */
REPEAT ix = 1 TO hRoot:NUM-CHILDREN:
  hRoot:GET-CHILD(hTable, ix).
  CREATE ttCustomer.

/* Get the fields given as attributes */
ttCustomer.CustNum = INTEGER(hTable:GET-ATTRIBUTE("CustNum").
ttCustomer.Name = hTable:GET-ATTRIBUTE("Name").

/* Get the remaining fields given as elements with text */
REPEAT jx = 1 TO hTable:NUM-CHILDREN:
  hTable:GET-CHILD(hField, jx).
  IF hField:NUM-CHILDREN < 1 THEN NEXT.
Namespaces

When you have nodes or attribute nodes that use namespaces, you need to supply the namespace information when you create the nodes. Therefore, there are different methods for creating nodes without namespaces and nodes with namespaces, as shown in the following table.

Table 2: X-document methods for creating nodes

<table>
<thead>
<tr>
<th>Use this method for no namespaces</th>
<th>Use this method for namespaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hX-document:CREATE-NODE()</code></td>
<td><code>hX-document:CREATE-NODE-NAMESPACE()</code></td>
</tr>
<tr>
<td><code>hX-noderef:SET-ATTRIBUTE()</code></td>
<td><code>hX-noderef:SET-ATTRIBUTE-NODE()</code></td>
</tr>
</tbody>
</table>

This is the syntax for the `CREATE-NODE-NAMESPACE( )` method:

**Syntax**

```plaintext
CREATE-NODE-NAMESPACE(x-noderef-handle, namespace-URI, qualifiedName, type)
```

*x-noderef-handle*

A valid X-noderef handle to use for the new namespace aware DOM node.

*namespace-URI*

Represents the URI reference, which identifies the namespace. The namespace name should have the characteristics of uniqueness and persistence. It does not need to be directly usable for retrieval of a schema, but ordinary URLs can also be used to achieve this effect.
qualifiedName

The qualified name is the name of an element or attribute optionally prefixed by a valid namespace prefix and a colon character. Unless you are using a default namespace, a prefix is required and should be set to the prefix specified when you declared the namespace using the `xmlns` attribute.

type

A character expression that represents the node SUBTYPE, which will be either `ELEMENT` or `ATTRIBUTE`.

The following example creates an `ELEMENT` node with a namespace and an `ATTRIBUTE` node with a namespace on an X-document object:

```
errStat = hDocument:CREATE-NODE-NAMESPACE
          (hNsNode,
           "http://www.progress.com/NS/MyNamespace",
           "MyNS:MyNSElt",
           "element").
```

This is the syntax for the `CREATE-NODE-NAMESPACE( )` method:

**Syntax**

```
SET-ATTRIBUTE-NODE (attr-node-handle)
```

`attr-node-handle`

A valid X-noderef handle created by `CREATE-NODE-NAMESPACE( )` or `CREATE-NODE( )`.

The following example creates an `ELEMENT` node with a namespace and an `ATTRIBUTE` node with a namespace on an X-document object:

```
ABL:

ASSIGN
  hX-noderef-withNS:NAMESPACE-URI = "http://www.progress.com/NS/MyNamespace"
  hX-noderef-withNS:NAMESPACE-PREFIX = "MyNS"
  hX-noderef-withNS:NODE-VALUE = "Some attribute value"
  errStat = hNsNode:SET-ATTRIBUTE-NODE(hx-noderef-wNS).

XML:

<MyNS:MyNamespaceElement MyNS:MyNamespaceAttribute = "Some attribute value">
```
**Note**: Do not mix calls to the `CREATE-NODE( )` method and the `CREATE-NODE-NAMESPACE( )` method in the same DOM tree.

### Internationalization

XML documents may be encoded using any of a wide a variety of character encoding. The DOM parser returns character data to ABL interpreter encoded, if possible, according to `-cpinternal`, the Internal Code Page parameter. This translation is performed by the DOM parser using its own translation functions. If the DOM parser cannot perform the translation according to -cpinternal, it translates to UTF8 which is then translated by the interpreter from UNICODE to the character encoding specified by `-cpinternal`. The encoding used in an XML document is specified by 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.

When the `LOAD` method is used to load an XML document, the `ENCODING` attribute of the X-document object will be set to the name of encoding found in the encoding declaration of the document. For output, you can set the X-document object'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 value 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 ABL is attempting to load a document, the document will be empty.

### Error handling

Any of the methods listed above may fail, but this does not normally raise the ABL `ERROR` condition. Instead, the method returns `FALSE` if that is appropriate. Also, the parsing 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, you should test for `ERROR-STATUS:NUM-MESSAGES` being greater than zero.

If the block containing the method calls includes a `CATCH` statement, any method that would normally add a message to `ERROR-STATUS:NUM-MESSAGES` after encountering a parsing error will raise the `ERROR` condition and the AVM will generate a `Progress.Lang.SysError` error object containing the message. For information on using `CATCH` statements, built-in method error handling, and structured error handling, see *OpenEdge Development: Error Handling*.

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

You can validate XML against a DTD or an XML Schema file. The DOM parser will always validate against an internal DTD or DTD reference. In addition, the SCHEMA-PATH attribute of the X-document allows you to point to another DTD for validation.

If you would like to validate against one or more XML Schema files, you can specify the location of these in the SCHEMA-LOCATION attribute or the NONAMESPACE-SCHEMA-LOCATION attribute. The following table describes the attributes and method that you configure to set up validation.

Table 3: X-document support for XML Schema validation

<table>
<thead>
<tr>
<th>X-document attribute or method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA-LOCATION attribute</td>
<td>Determines the list of namespace/location pairs of an XML Schema document to validate against.</td>
</tr>
<tr>
<td>NONAMESPACE-SCHEMA-LOCATION attribute</td>
<td>Determines the location the XML Schema document to validate against when elements do not contain a namespace.</td>
</tr>
<tr>
<td>ADD-Schema-LOCATION( ) method</td>
<td>An XML Schema document location is specified by providing a pair of values: a namespace and a physical location. This method allows you to specify that value pair and add it to the list of pairs stored in the SCHEMA-LOCATION attribute. The XML Schema documents listed in this attribute are used by an X-document to validate XML content.</td>
</tr>
</tbody>
</table>

For more information, see the reference entries for these methods and attributes in OpenEdge Development: ABL Reference.

Method and attribute summary

The following table summarizes the attributes and methods associated with the X-document object. For complete reference entries, see OpenEdge Development: ABL Reference.

Table 4: X-document attribute and method summary

<table>
<thead>
<tr>
<th>This attribute or method . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE-DOCUMENT-TYPE( ) method</td>
<td>Initialize document and set prolog information.</td>
</tr>
<tr>
<td>ENCODING attribute</td>
<td></td>
</tr>
<tr>
<td>This attribute or method . . .</td>
<td>Lets you . . .</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>SCHEMA-PATH</strong> attribute</td>
<td>Specify a search path for the Document Type Definition (DTD) or any other external entities. If you provide a DTD (or an XML Schema document), then ABL can perform the appropriate validation on reads and writes of the DOM tree.</td>
</tr>
<tr>
<td><strong>PUBLIC-ID</strong> attribute</td>
<td>Determine the logical (public) name or system name of the document's DTD.</td>
</tr>
<tr>
<td><strong>SYSTEM-ID</strong> attribute</td>
<td></td>
</tr>
<tr>
<td><strong>NAMESPACE-PREFIX</strong> attribute</td>
<td>Set namespace information at the document level.</td>
</tr>
<tr>
<td><strong>NAMESPACE-URI</strong> attribute</td>
<td></td>
</tr>
<tr>
<td><strong>SCHEMA-LOCATION</strong> attribute</td>
<td>The <strong>SCHEMA-LOCATION</strong> attribute lets you specify a whitespace separated list of namespace and XML Schema location pairs. The <strong>ADD-SCHEMA-LOCATION( )</strong> method allows you to add another pair to this list. X-document uses each XML Schema specified to validate XML elements with matching namespaces.</td>
</tr>
<tr>
<td><strong>ADD-SCHEMA-LOCATION( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>NONAMESPACE-SCHEMA-LOCATION</strong> attribute</td>
<td>Specify a single namespace and XML Schema location pair. X-document uses the XML Schema specified to validate XML elements that do not specify a namespace.</td>
</tr>
<tr>
<td><strong>SUPPRESS-NAMESPACE-PROCESSING</strong> attribute</td>
<td>Disable namespace processing.</td>
</tr>
<tr>
<td><strong>LOAD( )</strong> method</td>
<td>Create a complete DOM tree from an XML source document.</td>
</tr>
<tr>
<td><strong>NAME</strong> attribute</td>
<td>NAME is the name of the XML document. <strong>SUBTYPE</strong> is always DOCUMENT.</td>
</tr>
<tr>
<td><strong>SUBTYPE</strong> attribute</td>
<td></td>
</tr>
<tr>
<td><strong>APPEND-CHILD( )</strong> method</td>
<td>Programmatically build or modify a DOM tree.</td>
</tr>
<tr>
<td><strong>CREATE-NODE( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>CREATE-NODE-NAMESPACE( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>IMPORT-NODE( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>INSERT-BEFORE( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>REMOVE-CHILD( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>REPLACE-CHILD( )</strong> method</td>
<td></td>
</tr>
<tr>
<td><strong>GET-DOCUMENT-ELEMENT( )</strong> method</td>
<td>Traverse the DOM tree.</td>
</tr>
<tr>
<td><strong>NUM-CHILDREN</strong> attribute</td>
<td></td>
</tr>
<tr>
<td><strong>GET-CHILD( )</strong> method</td>
<td></td>
</tr>
</tbody>
</table>
### Method and attribute summary

<table>
<thead>
<tr>
<th>This attribute or method . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE( ) method</td>
<td>Write the DOM tree to an XML document.</td>
</tr>
<tr>
<td>TYPE attribute</td>
<td>Get the type of the object, which is always X-DOCUMENT.</td>
</tr>
<tr>
<td>HANDLE attribute</td>
<td>Get or set information concerning this particular X-document object.</td>
</tr>
<tr>
<td>INSTANTIATING-PROCEDURE attribute</td>
<td></td>
</tr>
<tr>
<td>UNIQUE-ID attribute</td>
<td></td>
</tr>
<tr>
<td>ENTITY-EXPANSION-LIMIT attribute</td>
<td>Set the maximum number of entity substitutions the XML parser will permit in a document.</td>
</tr>
<tr>
<td>STRICT-ENTITY-RESOLUTION attribute</td>
<td>Set whether or not the XML parser will attempt to resolve an external entity if the entity is located outside of the directories in the SCHEMA-PATH attribute.</td>
</tr>
</tbody>
</table>

The following table describes the attributes and methods that comprise the functionality of the X-noderef object. For complete reference entries, see *OpenEdge Development: ABL Reference*.

**Table 5: X-noderef attribute and method summary**

<table>
<thead>
<tr>
<th>This attribute or method . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE-NAMES attribute</td>
<td>Manage the relationships this node has with attribute nodes. Since a DOM attribute node is not a child of the element it is associated with, you must explicitly create the relationship from a node of subtype ELEMENT to the node of subtype ATTRIBUTE.</td>
</tr>
<tr>
<td>GET-ATTRIBUTE( ) method</td>
<td></td>
</tr>
<tr>
<td>GET-ATTRIBUTE-NODE( ) method</td>
<td></td>
</tr>
<tr>
<td>REMOVE-ATTRIBUTE( ) method</td>
<td></td>
</tr>
<tr>
<td>SET-ATTRIBUTE( ) method</td>
<td></td>
</tr>
<tr>
<td>SET-ATTRIBUTE-NODE( ) method</td>
<td></td>
</tr>
<tr>
<td>APPEND-CHILD( ) method</td>
<td>Manage the parent-child relationships this node has with other nodes.</td>
</tr>
<tr>
<td>CHILD-NUM attribute</td>
<td></td>
</tr>
<tr>
<td>GET-CHILD( ) method</td>
<td></td>
</tr>
<tr>
<td>GET-PARENT( ) method</td>
<td></td>
</tr>
<tr>
<td>INSERT-BEFORE( ) method</td>
<td></td>
</tr>
<tr>
<td>NUM-CHILDREN attribute</td>
<td></td>
</tr>
<tr>
<td>REPLACE-CHILD( ) method</td>
<td></td>
</tr>
<tr>
<td>REMOVE-CHILD( ) method</td>
<td></td>
</tr>
<tr>
<td>LOCAL-NAME attribute</td>
<td>Manage node namespaces.</td>
</tr>
<tr>
<td>NAMESPACE-PREFIX attribute</td>
<td></td>
</tr>
<tr>
<td>NAMESPACE-URI attribute</td>
<td></td>
</tr>
<tr>
<td>This attribute or method . . .</td>
<td>Lets you . . .</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>NODE-VALUE attribute</td>
<td>Set the content of the node and retrieve that content in various data formats.</td>
</tr>
<tr>
<td>LONGCHAR-TO-NODE-VALUE( ) method</td>
<td></td>
</tr>
<tr>
<td>NODE-VALUE-TO-LONGCHAR( ) method</td>
<td></td>
</tr>
<tr>
<td>MEMPTR-TO-NODE-VALUE( ) method</td>
<td></td>
</tr>
<tr>
<td>NODE-VALUE-TO-MEMPTR( ) method</td>
<td></td>
</tr>
<tr>
<td>OWNER-DOCUMENT attribute</td>
<td>Get the name of the X-document object the current X-noderef is associated with.</td>
</tr>
<tr>
<td>CLONE-NODE( ) method</td>
<td>Perform a copy of an existing node to the current node, or perform a deep copy of the subtree that this node parents.</td>
</tr>
<tr>
<td>NORMALIZE( ) method</td>
<td>Remove empty text nodes, merge adjacent text nodes, and clean up whitespace in ATTRIBUTE nodes such that the output of the DOM tree conforms to the XML specifications.</td>
</tr>
<tr>
<td>DELETE-NODE( ) method</td>
<td>Disconnects and removes this node and all of its descendent nodes from its associated X-document object.</td>
</tr>
<tr>
<td>NAME attribute</td>
<td>These attributes let you determine node type. The subtype is the DOM node type of the X-noderef object. The value of the NAME attribute and the NODE-VALUE attribute depend on the SUBTYPE attribute. See Table for more information.</td>
</tr>
<tr>
<td>SUBTYPE attribute</td>
<td></td>
</tr>
<tr>
<td>TYPE attribute</td>
<td>Get the type or the object, which is always X-NODEREF.</td>
</tr>
<tr>
<td>HANDLE attribute</td>
<td>Get or set information concerning this particular X-document object.</td>
</tr>
<tr>
<td>INSTANTIATING-PROCEDURE attribute</td>
<td></td>
</tr>
<tr>
<td>UNIQUE-ID attribute</td>
<td></td>
</tr>
</tbody>
</table>
This chapter assumes that the reader is an experienced ABL (Advanced Business Language) developer who understands XML, the Simple API for XML (SAX), and the basic OpenEdge development tools appropriate for developing ABL and WebSpeed applications. For more information on prerequisite topics, see the following documents:

- For XML, see the XML 1.0 specification, available at [http://www.w3.org](http://www.w3.org)
- For SAX, see the SAX 2.0 specification, available at [http://www.saxproject.org/apidoc](http://www.saxproject.org/apidoc)
- For WebSpeed, see OpenEdge Getting Started: WebSpeed Essentials

This chapter describes how to use the ABL’s interface to the Simple API for XML, commonly known as SAX. This chapter covers SAX basics and reading XML documents with SAX. For information on writing XML documents with the SAX-writer object, see Writing XML Documents with the Simple API for XML (SAX) on page 95.

For details, see the following topics:

- Understanding ABL SAX
- Developing ABL SAX applications
- ABL SAX and WebSpeed
- SAX API reference
Understanding ABL SAX

If you have already handled XML documents using the ABL support for DOM, then you know that an XML document in the ABL environment can be represented by an X-document object. An XML element can be represented by an X-noderef object. You interact with XML documents and XML elements using the attributes and methods of these ABL objects.

Similarly, the ABL SAX interface provides you with three ABL objects:

- SAX-reader
- SAX-attributes
- SAX-writer (see Writing XML Documents with the Simple API for XML (SAX) on page 95)

The ABL SAX-reader lets you do the following:

- Provide callback procedures for the SAX API to automatically invoke as it parses a document
- Enable or disable DTD and XML Schema validation and namespace processing
- Parse an XML document with one call or multiple calls
- Monitor the state of a parse
- Detect and handle errors

The ABL SAX-attributes object is automatically created and destroyed by the ABL Virtual machine (AVM) as needed at parse time to handle the attributes read for a single document element.

You can also create your own SAX-attributes objects to save the contents of automatically created SAX-attributes objects or, perhaps, to assemble a collection of XML attributes that you plan to apply to elements as you create a new XML document.

These topics are discussed in the following sections:

- SAX-reader object on page 56
- ABL SAX callbacks on page 58
- SAX-attributes object on page 60
- Validation on page 63
- Namespace processing on page 65
- Parsing with one call or with multiple calls on page 67
- Monitoring the state of the parse on page 67
- Error handling on page 67

SAX-reader object

The application creates a SAX-reader object and uses it to control the XML parser.

The application can parse an XML document in one call or in multiple calls. For more information on this topic, see Parsing with one call or with multiple calls on page 67.
When your application finishes parsing the XML document, you must delete the SAX-reader object.
The following table summarizes the attributes and methods associated with the SAX-reader object.
For complete reference entries, see *OpenEdge Development: ABL Reference*.

**Table 6: SAX-reader attribute and method summary**

<table>
<thead>
<tr>
<th>This attribute or method . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SET-INPUT-SOURCE( ) method</strong></td>
<td>Specify the XML input.</td>
</tr>
<tr>
<td><strong>SCHEMA-PATH attribute</strong></td>
<td>Specify a search path for the Document Type Definition (DTD) or any other external entities.</td>
</tr>
<tr>
<td><strong>SCHEMA-LOCATION attribute</strong></td>
<td>The <strong>SCHEMA-LOCATION</strong> attribute contains a whitespace separated list of namespace and XML Schema location pairs. The <strong>ADD-SCHEMA-LOCATION( ) method</strong> allows you to add another pair to this list. SAX-reader uses each XML Schema specified to validate XML elements with matching namespaces.</td>
</tr>
<tr>
<td><strong>NONAMESPACE-SCHEMA-LOCATION attribute</strong></td>
<td>Specify the location of an XML Schema that does not specify a target namespace. The SAX-reader object uses this XML Schema to validate XML elements that are not in a namespace. The value of this attribute is a single URI indicating the schema location, not pairs of namespace/location URIs for mapping namespaces to schema locations. Most schemas specify a target namespace. When one of those schemas is used, then the <strong>ADD-SCHEMA-LOCATION( ) method</strong> and <strong>SCHEMA-LOCATION attribute</strong> are used. If a schema does not specify a target namespace, but simply describes the structure of an XML document, the <strong>NONAMESPACE-SCHEMA-LOCATION</strong> is used.</td>
</tr>
<tr>
<td><strong>HANDLER attribute</strong></td>
<td>Specify the location of the SAX callback procedures.</td>
</tr>
<tr>
<td><strong>SUPPRESS-NAMESPACE-PROCESSING attribute</strong></td>
<td>Toggle parser options.</td>
</tr>
<tr>
<td><strong>VALIDATION-ENABLED attribute</strong></td>
<td>Toggle parser options.</td>
</tr>
<tr>
<td><strong>SAX-PARSE( ) method</strong></td>
<td>Start, continue, or stop parsing.</td>
</tr>
<tr>
<td><strong>SAX-PARSE-FIRST( ) method</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SAX-PARSE-NEXT( ) method</strong></td>
<td></td>
</tr>
<tr>
<td><strong>STOP-PARSING( ) method</strong></td>
<td></td>
</tr>
</tbody>
</table>
### This attribute or method . . . | Lets you . . .
---|---
LOCATOR-COLUMN-NUMBER attribute | Get the status of the parse.
LOCATOR-LINE-NUMBER attribute |
LOCATOR-PUBLIC-ID attribute |
LOCATOR-SYSTEM-ID attribute |
PARSE-STATUS attribute |
TYPE attribute | Get the type of the object, which is always SAX-READER.
ADM-DATA attribute | Get or set information concerning this particular SAX-reader object.
HANDLE attribute |
INSTANTIATINGPROCEDURE attribute |
PRIVATE-DATA attribute |
UNIQUE-ID attribute | Set the maximum number of entity substitutions the XML parser will permit in a document.
ENTITY-EXPANSION-LIMIT attribute |
strict-ENTITY-RESOLUTION attribute | Set whether or not the XML parser will attempt to resolve an external entity if the entity is located outside of the directories in the SCHEMA-PATH attribute.

## ABL SAX callbacks

When the XML parser encounters an XML token, ABL invokes the callback corresponding to that token—if that callback is provided by the developer. If not, the parser continues.

ABL SAX implements callbacks as internal procedures coded by the ABL developer using signatures specified by ABL. Normally, callbacks are placed in a procedure (.p) file that the application runs persistently. But callbacks can also be placed in the driver routine. A driver routine is one that calls the SAX-PARSE( ) or SAX-PARSE-FIRST( ) method. In either case, your application assigns the containing procedure's handle to the SAX-reader HANDLER attribute. (The HANDLER attribute defaults to a handle to the driver routine). Then, the application starts the parse.

**Note:** Although there are many callbacks in the interface specified by the SAX 2.0 standard, most applications use just a few. These are StartElement, Characters, and EndElement.

Within a callback, to get a handle to the SAX-reader object that invoked the callback, use the SELF system handle.

The following fragment uses SELF within a callback to call the SAX-reader STOP-PARSING( ) method:

```abl
SELF:STOP-PARSING( ).
```
The following fragment uses `SELF` within a callback to store data in the `SAX-reader PRIVATE-DATA` attribute:

```
SELF:PRIVATE-DATA = "xyz123".
```

For information on the SAX parser’s current location in the XML source, use the following attributes of `SAX-reader`:

- `LOCATOR-COLUMN-NUMBER`
- `LOCATOR-LINE-NUMBER`
- `LOCATOR-PUBLIC-ID`
- `LOCATOR-SYSTEM-ID`

**Note:** These attributes are valid only within a callback.

The following table summarizes the ABL SAX callbacks. For a complete description, see *SAX callback reference* on page 85.

**Table 7: SAX callback summary**

<table>
<thead>
<tr>
<th>This callback . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ResolveEntity</code> on page 89</td>
<td>Tell the parser where to find an external entity</td>
</tr>
<tr>
<td><code>StartDocument</code> on page 91</td>
<td>Process various XML tokens</td>
</tr>
<tr>
<td><code>ProcessingInstruction</code> on page 89</td>
<td></td>
</tr>
<tr>
<td><code>StartPrefixMapping</code> on page 92</td>
<td></td>
</tr>
<tr>
<td><code>EndPrefixMapping</code> on page 87</td>
<td></td>
</tr>
<tr>
<td><code>StartElement</code> on page 91</td>
<td></td>
</tr>
<tr>
<td><code>Characters</code> on page 85</td>
<td></td>
</tr>
<tr>
<td><code>IgnorableWhitespace</code> on page 88</td>
<td></td>
</tr>
<tr>
<td><code>EndElement</code> on page 86</td>
<td></td>
</tr>
<tr>
<td><code>EndDocument</code> on page 86</td>
<td></td>
</tr>
<tr>
<td><code>NotationDecl</code> on page 88</td>
<td>Process notations and unparsed entities</td>
</tr>
<tr>
<td><code>UnparsedEntityDecl</code> on page 92</td>
<td></td>
</tr>
<tr>
<td><code>Warning</code> on page 93</td>
<td>Handle errors</td>
</tr>
<tr>
<td><code>Error</code> on page 87</td>
<td></td>
</tr>
<tr>
<td><code>FatalError</code> on page 87</td>
<td></td>
</tr>
</tbody>
</table>
SAX-attributes object

When the parser encounters the start tag of a new XML element, it knows to call the \texttt{startElement} callback procedure. At this time, the parser creates a SAX-attributes object and passes it to the \texttt{startElement} callback as an input parameter. Thus, when you define your \texttt{startElement} callback, you define the name of the parameter that holds the handle to this automatically created object and this is how your callback can access the attributes of the XML element.

For example, suppose the parser encounters an XML start tag that has one or more attributes, such as:

\begin{verbatim}
<Customer name="John Smith" id="2543">
\end{verbatim}

The parser creates and populates the \texttt{StartElement} callback \texttt{attributes} parameter with information on the XML element's attributes. In the example, \texttt{name} and \texttt{id} are attributes. The \texttt{attributes} parameter is a handle to a SAX-attributes object.

\textbf{Note:} The SAX-attributes object is similar to the \texttt{Attributes} interface of the Java Sax2 API.

For most basic use cases, you will be working with this automatically created SAX-attributes object. For information on programmatically creating your own SAX-attributes object, see Creating a SAX-attributes object on page 62.

The following table summarizes the attributes and methods of the SAX-attributes object. For complete reference entries, see \textit{OpenEdge Development: ABL Reference}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
This attribute or method & Lets you . . .
\hline
\texttt{NUM-ITEMS attribute} & See how many attributes the XML element has
\hline
\texttt{GET-INDEX-BY-NAMESPACE-NAME ( ) method} & Get where on the attribute list a particular attribute resides
\hline
\texttt{GET-INDEX-BY-QNAME ( ) method} & Get the name of a particular attribute
\hline
\texttt{GET-LOCALNAME-BY-INDEX ( ) method} & Get the type of a particular attribute
\hline
\texttt{GET-QNAME-BY-INDEX ( ) method} & Get the value of a particular attribute
\hline
\texttt{GET-URI-BY-INDEX ( ) method} & Get the value of a particular attribute
\hline
\end{tabular}
\caption{SAX-attributes attribute and method summary}
\end{table}
A SAX-attributes object represents the complete collection of all the attributes for a given element. No matter how many attributes an element has, the `StartElement` callback gets passed only one SAX-attributes handle.

**Note:** The order of the elements in the SAX-attributes object list might not be the same as the order in which they appear in the document—which is consistent with the SAX2 Java API specification.

### Retrieving data from a SAX-attributes object

To get information about each attribute from a SAX-attributes object, use the `GET-xxx` methods. These methods let your application get the SAX-attributes data in two different ways:

- You can traverse the list, getting each attribute's data in turn by using the `GET-xxx-BY-INDEX` methods
- You can use an attribute's name to get its data

**Note:** This approach has variations, depending on whether you are using namespace processing or not. For more information on this point, see Namespace processing on page 65.

There are six facts you can get about each attribute, although three are variations on the attribute name. Each `GET-xxx` method retrieves one fact. The following table describes the six facts and the function of the `GET-xxx` methods.

<table>
<thead>
<tr>
<th>This attribute or method . . .</th>
<th>Lets you . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>INSERT-ATTRIBUTE( )</code> method</td>
<td>Manipulate the attributes in the list</td>
</tr>
<tr>
<td><code>UPDATE-ATTRIBUTE( )</code> method</td>
<td></td>
</tr>
<tr>
<td><code>REMOVE-ATTRIBUTE( )</code> method</td>
<td></td>
</tr>
<tr>
<td><code>TYPE</code> attribute</td>
<td>Get the type of the object (which is always <code>SAX-ATTRIBUTES</code>)</td>
</tr>
<tr>
<td><code>ADM-DATA</code> attribute</td>
<td>Get or set information concerning this particular SAX-attributes object</td>
</tr>
<tr>
<td><code>HANDLE</code> attribute</td>
<td></td>
</tr>
<tr>
<td><code>INSTANTIATING-PROCEDURE</code> attribute</td>
<td></td>
</tr>
<tr>
<td><code>PRIVATE-DATA</code> attribute</td>
<td></td>
</tr>
<tr>
<td><code>UNIQUE-ID</code> attribute</td>
<td></td>
</tr>
<tr>
<td><code>COPY-SAX-ATTRIBUTES( )</code> method</td>
<td>Copy the contents of the specified SAX-attributes object to this SAX-attributes object</td>
</tr>
</tbody>
</table>
Table 9: The GET-xxx methods

<table>
<thead>
<tr>
<th>To get this information on an attribute . . .</th>
<th>Use these methods . . .</th>
</tr>
</thead>
</table>
| Its position on the list                    | GET-INDEX-BY-NAMESPACE-NAME( )  
|                                             | GET-INDEX-BY-QNAME( )      |
| Its namespace URI, if namespace information is available | GET-URI-BY-INDEX( )        |
| Its local name, if namespace processing is available | GET-LOCALNAME-BY-INDEX( )  |
| Its Qualified name (qName)                  | GET-QNAME-BY-INDEX( )     |
| Its XML attribute type (declared in the DTD or XML Schema) | GET-TYPE-BY-INDEX( )     
|                                             | GET-TYPE-BY-NAMESPACE-NAME( )  
|                                             | GET-TYPE-BY-QNAME( )      |
| Its value                                   | GET-VALUE-BY-INDEX( )     
|                                             | GET-VALUE-BY-NAMESPACE-NAME( )  
|                                             | GET-VALUE-BY-QNAME( )      |

If the parser is processing namespaces—that is, if SAX-reader object SUPPRESS-NAMESPACE-PROCESSING attribute is set to NO (the default)—each attribute that has a namespace prefix will have non-empty URI, localName, and qName data. An attribute that has the noNamespace prefix will have an empty URI, but its localName and qName will have values.

**Note:** An unprefixed attribute name does not use the default namespace, if one exists. The attribute is not associated with any namespace. Contrast this case with the case for unprefixed elements. Unprefixed elements do use the default namespace, if one exists.

If the parser is not processing namespaces, each attribute will only have a qName. In all cases, the qName will be exactly what appears in the XML document.

**Creating a SAX-attributes object**

Use the CREATE SAX-ATTRIBUTES statement to create an instance of a SAX-attributes object and assign its handle to the handle variable specified.

**Syntax**

```
CREATE SAX-ATTRIBUTES handle [ IN WIDGET-POOL pool-name ] [ NO-ERROR ]
```
handle

Variable of type HANDLE which stores the handle of the new SAX-attributes object.

IN WIDGET-POOL pool-name

Specifies the widget pool where the AVM creates the new object. If you do not specify a widget pool, the AVM creates the object in the current default widget pool.

NO-ERROR

Specifies that the AVM should suppress errors occurring during the creation of the SAX-attributes handle. After the CREATE SAX-ATTRIBUTES statement completes, you can check the ERROR-STATUS system handle for information about errors that might have occurred.

In ABL, a SAX-attributes object can be automatically created and managed by the SAX parser, or programatically created and managed by you. Automatic SAX-attributes objects are created, populated, and destroyed during the scope of theStartElement callback procedure. TheStartElement callback is called by the SAX-reader object each time the SAX parser encounters a new XML element during a read operation. The SAX parser populates the SAX-attributes object, but you may interact with the object during the scope of theStartElement callback procedure. For example, you may add elements, update elements, and remove elements from the SAX-attributes object.

At the conclusion of the callback, the SAX-attributes object is deleted and your access to the element list is lost. To save the XML attributes list, you would need to create another SAX-attributes object that is not tied to the scope of theStartElement callback procedure. Then, from within theStartElement callback, you would use theCOPY-SAX-ATTRIBUTES( ) method on the new object and provide the handle to the automatically created SAX-attributes object, which is the attributes input parameter of theStartElement callback. Finally, you would likely pass the XML element and its attributes to another procedure outside of the callback's scope for further processing.

The SAX-writer also supports use of the SAX-attributes object. Here, you could pass it a copy of an attribute list from an XML element, or you might define a SAX-attributes object that contains a standard set of attribute values to apply to particular XML elements output by the SAX-writer object.

For an example of using a user-created SAX-attributes object, see Writing XML Documents with the Simple API for XML (SAX) on page 95.

Validation

The SAX-reader object always checks that the XML document is well formed. In addition, the SAX-reader object can validate the XML document against a DTD or XML Schema or both.

Validation of an XML document read into a SAX-reader object can be accomplished by providing a DTD, one or more XML Schemas, or both. The following table provides a summary of how to perform the validation tasks required to support your desired validation scheme.

Table 10: Validation schemes

<table>
<thead>
<tr>
<th>Validation task</th>
<th>Required action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable DTD validation</td>
<td>Set the VALIDATION-ENABLED attribute of the SAX-reader object to FALSE. All DTDs will be ignored.</td>
</tr>
</tbody>
</table>
### Validation task | Required action
--- | ---
Enable DTD validation | Set the `VALIDATION-ENABLED` attribute of the SAX-reader object to `TRUE`. If the parser does not find a DTD, the parse fails. Validation errors are caught in a callback event procedure named `error`. If this callback procedure is not implemented, then validation errors cannot be caught. If the DTD is not included within the parsed XML document, provide the DTD location by setting the `SCHEMA-PATH` attribute of the SAX-reader object. If this attribute is set, DTDs found or referenced from within the XML document are ignored in favor of the specified DTD.

Disable XML Schema validation | Set the `SCHEMA-LOCATION` and `NONAMESPACE-SCHEMA-LOCATION` attributes to the empty string (""").

**Note:** XML documents may refer to the schemas to which they must conform using an `xsi:schemaLocation` attribute and/or an `xsd:noNamespaceSchemaLocation` attribute. If the XML document does include these references, setting these SAX-reader attributes to the empty string ("") does not disable validation against the referenced schemas.

Enable XML Schema validation | Specify the location of one or more XML Schema documents with the `SCHEMA-LOCATION` attributes and/or the `NONAMESPACE-SCHEMA-LOCATION` attribute of the SAX-reader object.

Enable both DTD and XML Schema validation | Set the `VALIDATION-ENABLED` attribute to `TRUE` and provide valid `SCHEMA-PATH`. Specify the location of one or more XML Schema documents with the `SCHEMA-LOCATION` attribute and/or the `NONAMESPACE-SCHEMA-LOCATION` attributes of the SAX-reader object.

---

A DTD might be completely included in the XML document. Alternatively, part or all of the DTD might reside in one or more external files named in the XML document. In the latter case, to override the locations given in the XML document, use one or both of the following techniques:

- Implement the `ResolveEntity` callback procedure
- Set the SAX-reader object's `SCHEMA-PATH` attribute

**Note:** Even when `VALIDATION-ENABLED` is `FALSE`, the parser still reads any specified DTD (internal or external) to get information on entities.

For more information, see the reference entries for the `VALIDATION-ENABLED` and `SCHEMA-PATH` attributes in *OpenEdge Development: ABL Reference*. 
Similarly, to use XML Schema for validation, set the following attributes or use the method to declare your namespace and location pairs for the XML Schema documents:

- **SCHEMA-LOCATION** attribute
- **ADD-SCHEMA-LOCATION( )** method
- **NONAMESPACE-SCHEMA-LOCATION** attribute

For more information, see these reference entries in *OpenEdge Development: ABL Reference*.

**Note:** The parser will validate against both a DTD and XML Schema if both types of validation are properly specified.

### Namespace processing

Namespace processing is enabled by default. To disable it, set the **SUPPRESS-NAMESPACE-PROCESSING** attribute to **TRUE**. XML namespace processing is potentially complex. This section provides a summary of the expected behavior from an ABL SAX application.

### Namespace declarations

An XML document that uses namespaces has one or more namespace declarations, which appear as attributes of elements. A namespace declaration might appear as an attribute of any element. Frequently, the document element (the one that encloses all the other elements) has a global namespace declaration or declarations.

A namespace declaration associates a prefix with a URI. Once associated, the prefix might appear with element names, attributes names, or both, to distinguish the names from identical names that might mean something else. For example, an XML element whose name is `memory` might mean completely different things and have different valid attributes, depending on whether it appears in a computer inventory or in a psychological report. You can distinguish the uses by having a `computer-inventory` namespace and a `psychological-report` namespace.

In the SAX2 interface, XML namespaces affect the values of the parameters of the **StartElement** and **EndElement** callback procedures, as well as the attribute data in the **attributes** parameter of the **StartElement** callback. There can be slight variations in the way that the SAX-reader object handles namespace processing, depending on the configuration of the SAX-reader object.

### Effects of namespace processing

In ABL SAX, namespace processing affects the behavior of:

- **The StartElement and EndElement callbacks**
- **Attribute data**

The following table describes the effect of namespace processing on the **StartElement** and **EndElement** callbacks.
### Table 11: Effect of namespace processing on StartElement and EndElement

<table>
<thead>
<tr>
<th>Where namespace processing is enabled</th>
<th>Where namespace processing is suppressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>If an element’s name has a namespace prefix:</td>
<td>Whether or not an element's name has a namespace prefix:</td>
</tr>
<tr>
<td>• namespaceURI is the URI associated with the prefix</td>
<td>• namespaceURI is empty</td>
</tr>
<tr>
<td>• localName is the name as given in the XML document, without the prefix</td>
<td>• localName is empty</td>
</tr>
<tr>
<td>• qName is the name as given in the XML document</td>
<td>• qName is the name as given in the XML document (including the prefix, if one exists)</td>
</tr>
<tr>
<td>If an element’s name has no prefix, then:</td>
<td></td>
</tr>
<tr>
<td>• namespace URI is empty</td>
<td></td>
</tr>
<tr>
<td>• localName is the name as given in the XML document</td>
<td></td>
</tr>
<tr>
<td>• qName is the name as given in the XML document</td>
<td></td>
</tr>
<tr>
<td>If there is a default namespace in effect and an element name has no prefix, then:</td>
<td></td>
</tr>
<tr>
<td>• namespaceURI is the URI of the default namespace</td>
<td></td>
</tr>
<tr>
<td>• localName is the name as given in the XML document</td>
<td></td>
</tr>
<tr>
<td>• qName is the name as given in the XML document (there is no prefix, since the default namespace does not specify a prefix)</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the effect of namespace processing on attribute data.

### Table 12: Effect of namespace processing on attribute data

<table>
<thead>
<tr>
<th>Where namespace processing is enabled</th>
<th>Where namespace processing is suppressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>The behavior is identical to element name behavior, except that a default namespace does not apply to attribute names, so an attribute with no prefix will never have a value for its namespaceURI.</td>
<td>Whether or not an attribute’s name has a namespace prefix:</td>
</tr>
<tr>
<td>• namespaceURI is empty</td>
<td></td>
</tr>
<tr>
<td>• localName is empty</td>
<td></td>
</tr>
<tr>
<td>• qName will be the name as given in the XML document (including any prefix)</td>
<td></td>
</tr>
</tbody>
</table>

For more information on namespace processing, see the reference entry for the SUPPRESS-NAMESPACE-PROCESSING attribute in *OpenEdge Development: ABL Reference*. 
Parsing with one call or with multiple calls

An ABL SAX application can parse an XML document using one of the following techniques:

- Single call
- Progressive scan

To use the *single-call* technique, the application calls the `SAX-PARSE()` method once. The parser parses the entire XML document (unless errors occur), calling all appropriate callbacks, and returns control to the line in the code following `SAX-PARSE()`.

To use the *progressive-scan* technique, the application calls the `SAX-PARSE-FIRST()` method once to initiate parsing, then calls the `SAX-PARSE-NEXT()` method repeatedly to parse each XML token in the document. As each XML token is detected, the parser invokes the corresponding callback. After each call to `SAX-PARSE-FIRST()` or `SAX-PARSE-NEXT()`, control returns to the line in the code following the `SAX-PARSE-FIRST()` or `SAX-PARSE-NEXT()`.

Consider using progressive scan:

- If your business logic is complex and processes individual XML elements extensively.
  
  To do significant processing with a single call, your callback code might have to call directly into your business logic. This might be awkward, especially when adding SAX to existing code.
  
  To do significant processing with progressive scan, your business logic can simply call `SAX-PARSE-FIRST()` or `SAX-PARSE-NEXT()` to extract the next piece of data. The callbacks could then store incoming data, one piece at a time, for the business logic to process after the return from `SAX-PARSE-FIRST()` or `SAX-PARSE-NEXT()`.

- To parse two XML sources concurrently.
  
  After calling the `SAX-PARSE()`, `SAX-PARSE-FIRST()`, or `SAX-PARSE-NEXT()` methods, the application checks the value of the `PARSE-STATUS` attribute, as explained in Monitoring the state of the parse on page 67.

Monitoring the state of the parse

An ABL SAX application keeps track of the status of the parse by monitoring the value of the SAX-reader object's `PARSE-STATUS` attribute, which can have the following values:

- `SAX-UNINITIALIZED`
- `SAX-RUNNING`
- `SAX-COMPLETE`
- `SAX-PARSER-ERROR`

For more information, see the `PARSE-STATUS` attribute reference entry in OpenEdge Development: ABL Reference.

Error handling

When an ABL SAX application parses an XML document, it could encounter one of three distinct parse-related error situations. They are:
• The `SAX-PARSE( )`, `SAX-PARSE-FIRST( )`, or `SAX-PARSE-NEXT( )` detects an error and cannot complete its current parse operation.

This might be caused by one of the following:

• The specified XML source does not exist.

• The handle to the procedure containing the callbacks is invalid.

• `SAX-reader` was not in the appropriate state. For example, this can happen if `SAX-PARSE-NEXT( )` was called before `SAX-PARSE-FIRST( )`.

If this error condition occurs, parsing stops and the AVM raises an error.

If `NO-ERROR` was specified, the AVM sets `ERROR-STATUS:ERROR` to `YES`, sets `ERROR-STATUS:NUM-MESSAGES` to the number of errors encountered, and returns OpenEdge error messages in response to a statement with the following syntax:

```
ERROR-STATUS:GET-MESSAGE(err-msg-num).
```

Where `err-msg-num` indicates a number between 1 and `ERROR-STATUS:NUM-MESSAGES` inclusive.

You can also handle the error by way of a `CATCH` block if the method call does not contain the `NO-ERROR` option.

If the error is not trapped by a `NO-ERROR` option or a `CATCH` block, or handled by an explicit `ON ERROR` phrase or `UNDO` statement, then the AVM executes the default error handling behavior for the block type containing the method call. For more information, see *OpenEdge Development: Error Handling*.

• A callback raises an error by using the `RETURN` statement `ERROR` option.

If this error situation occurs, parsing stops, the AVM raises an error, and the `RETURN-VALUE` is set to whatever string the `RETURN` statement included.

If `NO-ERROR` was specified, the AVM sets `ERROR-STATUS:ERROR` to `TRUE` and sets `ERROR-STATUS:NUM-MESSAGES` to zero.

You can also handle the error by way of a `CATCH` block if the method call does not contain the `NO-ERROR` option.

If `NO-ERROR` was not specified, the AVM looks in the driver routine for the closest block that has the error property and behaves as if the block has an explicit `ON ERROR` phrase. For more information, see *OpenEdge Development: Error Handling*.

**Note:** If a callback procedure calls a second procedure, the second procedure calls a third procedure, and so on, and the final procedure in the chain executes `RETURN ERROR`, each preceding procedure in the chain must also execute `RETURN ERROR`, otherwise the error condition is never communicated to the driver routine. This is standard AVM behavior.

• While a callback is executing, the AVM raises an error that the callback does not handle; for example, a `FIND CUSTOMER` that fails to find the specified customer.

If this error occurs:

• The error is displayed (which is standard AVM behavior).
• In the callback, the AVM finds the closest block that has the error property (which might be the PROCEDURE block) and behaves according to the block’s explicit or implicit ON ERROR phrase. This might cause the AVM to break out of the callback.

• The PARSE-STATUS attribute and the ERROR-STATUS handle are unaffected.

• The parse continues.

Developing ABL SAX applications

The following sections describe how to develop an ABL SAX application:

• Basic tasks of OpenEdge SAX application on page 69

• Example code: retrieving names and phone numbers on page 70

• Example code: reading customer data and writing a TEMP-TABLE on page 76

• ABL SAX and WebSpeed on page 80

• Example code: reading XML data using WebSpeed on page 80

• SAX and the AppBuilder on page 81

Basic tasks of OpenEdge SAX application

A typical ABL SAX application performs the following tasks:

1. Creates a SAX-reader object
2. Runs a persistent procedure that contains the SAX callbacks
3. Configures the SAX-reader object by:
   a. Setting its HANDLER attribute to the handle of the routine that contains the SAX callbacks
   b. Turning namespace processing and validation on or off as needed
   c. Specifying the input source using the SET-INPUT-SOURCE( ) method
4. Starts the parser by calling the SAX-PARSE( ) or SAX-PARSE-FIRST( ) method
5. Handles XML data passed to the callbacks as the parser proceeds
6. Monitors the state of the parse by checking error codes and the parse status after each call to the SAX-PARSE( ), SAX-PARSE-FIRST( ), and SAX-PARSE-NEXT( ) methods
7. Releases resources, including deleting the SAX-reader object

Most of these tasks can be performed using the attributes and methods of the SAX-reader object, which are summarized in the following table.

Table 13: Tasks handled by SAX-reader attributes and methods

<table>
<thead>
<tr>
<th>To perform this task . . .</th>
<th>Use this attribute or method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify the XML input</td>
<td>SET-INPUT-SOURCE( ) method</td>
</tr>
</tbody>
</table>
To perform this task . . . | Use this attribute or method . . .
---|---
Specify a search path for the DTD | SCHEMA-PATH attribute
Specify the location of XML Schema documents with namespace and location pairs | SCHEMA-LOCATION attribute  
| ADD-SCHEMA-LOCATION( ) method
Specify the location of the XML Schema used to validate XML elements that do not specify a namespace | NONAMESPACE-SCHEMA-LOCATION attribute
Tell the parser where the callbacks reside | HANDLER attribute
Toggle parser options | SUPPRESS-NAMESPACE-PROCESSING attribute  
| VALIDATION-ENABLED attribute
Start, continue, or stop parsing | SAX-PARSE( ) method  
| SAX-PARSE-FIRST( ) method  
| SAX-PARSE-NEXT( ) method  
| STOP-PARSING( ) method
Get the status of the parse | LOCATOR-COLUMN-NUMBER attribute  
| LOCATOR-LINE-NUMBER attribute  
| LOCATOR-PUBLIC-ID attribute  
| LOCATOR-SYSTEM-ID attribute  
| PARSE-STATUS attribute
Get or set information concerning this particular SAX-reader object | PRIVATE-DATA attribute

Example code: retrieving names and phone numbers

This example retrieves names and phone numbers. It is presented in two versions in the following sections:
- Without namespace processing on page 70
- With namespace processing on page 73

Without namespace processing

In this example of a SAX driver procedure, i-sax1d.p, the driver’s logic closely parallels the tasks in Basic tasks of OpenEdge SAX application on page 69.
DEFINE VARIABLE hHandler AS HANDLE NO-UNDO.
DEFINE VARIABLE hParser AS HANDLE NO-UNDO.

/* Create the SAX-reader object */
CREATE SAX-READER hParser.

/* Run the persistent procedure that contains the callbacks */
RUN "i-sax1h.p" PERSISTENT SET hHandler.

/* Give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse. This XML file does not use namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "i-sax1.xml").
hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, our callbacks have been called as many times as necessary and we're done processing the XML document (or there was an error) */
IF ERROR-STATUS:ERROR THEN DO:
    IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
        /* Unable to begin the parse */
        MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
    ELSE
        /* Error detected in a callback */
        MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
    MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.

DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.

This is the associated sample XML file, i-sax1.xml. Each entry contains a name and phone number:

```
<?xml version='1.0' ?>
<Phonelist>
    <Entry ContactName="Jane Jones"> 555 555-5555 </Entry>
    <Entry ContactName="John Smith"> 555 555-1111 </Entry>
</Phonelist>
```

i-sax1h.p is the sample handler procedure with SAX callbacks, i-sax1h.p.

/* This small example uses a very simple approach to keeping track of where it is in the processing of the document. It uses currentPerson and currentNum, which are variables global to this procedure that enable the application to tie together the several different pieces of information that it gets for each element in the XML document. */

/* Name attribute for the current entry. App gets it during the StartElement callback */
DEFINE VARIABLE currentPerson AS CHARACTER NO-UNDO.
/* Phone number from the current entry. App gets it during the Characters
   callback because it is the character data for the element. */
DEFINE VARIABLE currentNum AS CHARACTER NO-UNDO.

/* This procedure is called when the parser finds the start tag for an element.
   For this particular XML doc, the app simply looks for "Entry" elements and
digs out the "ContactName" attribute during the StartElement call, saving
it in currentPerson */
PROCEDURE StartElement:
   DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER qname AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER hAttributes AS HANDLE NO-UNDO.
   IF qName = "Entry" THEN
      currentPerson = hAttributes:GET-VALUE-BY-QNAME("ContactName").
   END PROCEDURE.

/* This callback gets passed the character data for an element. Note that SAX
does not guarantee that all the characters for an element get passed in
one call -- that's why the app has to maintain the currentNum global
variable and append to it when handling Characters, and also why it has to
wait for EndElement before displaying the message box. (Note also that
some
apps may need to use a MEMPTR to accumulate the character data, which may
exceed the 32K ABL CHARACTER variable limit) */
PROCEDURE Characters:
   DEFINE INPUT PARAMETER charData AS MEMPTR NO-UNDO.
   DEFINE INPUT PARAMETER numChars AS INTEGER NO-UNDO.
   /* Assume that any call to Characters is for an Entry's text value, because
      we know what the document looks like. If this weren't the case, we'd
      have
to keep track of the localName passed to the most recent call to
StartElement. */
   currentNum = currentNum + GET-STRING(charData, 1, GET-SIZE(charData)).
END PROCEDURE.

/* This callback is called when the parser finds the end tag for an Element.
   Note that this app only cares about the end of an Entry element. */
PROCEDURE EndElement:
   DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER qname AS CHARACTER NO-UNDO.
   IF qName = "Entry" THEN DO:
      MESSAGE "Name: " currentPerson SKIP
      "Phone Number: " currentNum VIEW-AS ALERT-BOX.
      ASSIGN
         currentNum = ""
         currentPerson = "".
      END.
   END PROCEDURE.

/* Knowing the structure of the XML doc, the app could have done this in the
EndElement call for the Phonelist element and could then have omitted
EndDocument altogether. */
PROCEDURE EndDocument:
   MESSAGE "All Done" VIEW-AS ALERT-BOX.
END PROCEDURE.

When the sample is run, it produces the following trace:
Trace of SAX driver without namespace processing

Callback function: StartDocument
Callback function: StartElement
   namespaceURI: 
   localName: Phonelist
   qName: Phonelist
   SAX-ATTRIBUTE has 0 items:
Callback function: StartElement
   namespaceURI: 
   localName: Entry
   qName: Entry
   SAX-ATTRIBUTE has 1 items:
      Attribute 1 :
         namespaceURI: 
         localName: ContactName
         qName: ContactName
         type: CDATA
         value: Jane Jones
Callback function: Characters
   charData: 555 555-5555
Callback function: EndElement
   namespaceURI: 
   localName: Entry
   qName: Entry
Callback function: StartElement
   namespaceURI: 
   localName: Entry
   qName: Entry
   SAX-ATTRIBUTE has 1 items:
      Attribute 1 :
         namespaceURI: 
         localName: ContactName
         qName: ContactName
         type: CDATA
         value: John Smith
Callback function: Characters
   charData: 555 555-1111
Callback function: EndElement
   namespaceURI: 
   localName: Entry
   qName: Entry
Callback function: EndElement
   namespaceURI: 
   localName: Phonelist
   qName: Phonelist
Callback function: EndDocument

With namespace processing

This section shows another version of the driver example where the XML document uses namespaces. Consequently, the StartElement andEndElement callbacks in the handler procedure use the namespaceURI and localName parameters rather than the qName parameter.

**Note:** The original example could have used localName by itself, but did not.

i-sax1dn.p is the SAX driver procedure with namespace processing.
i-sax1dn.p

```openedge
DEFINE VARIABLE hHandler AS HANDLE NO-UNDO.
DEFINE VARIABLE hParser AS HANDLE NO-UNDO.

/* Create the SAX-READER object */
CREATE SAX-READER hParser.

/* Run the persistent procedure that contains the callbacks */
RUN "i-sax1h-ns.p" PERSISTENT SET hHandler.

/* Give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse. This XML file uses namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "i-sax1-ns.xml").
hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, our callbacks have been called as many times as necessary and we're done processing the XML document (or there was an error. */
IF ERROR-STATUS:ERROR THEN DO:
  IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
    /* Unable to begin the parse */
    MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
  ELSE
    /* Error detected in a callback */
    MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
  MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.

DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.
```

i-sax1n.xml is the associated XML file with namespaces.

i-sax1n.xml

```xml
<?xml version='1.0' ?>
<Phonelist xmlns="http:/www.wmhwmh.biz/ns/Default"
    xmlns:pl="http://www.wmhwmh.biz/ns/phonelist">
    <pl:Entry pl:ContactName="Jane Jones"> 555 555-5555 </pl:Entry>
    <pl:Entry pl:ContactName="John Smith"> 555 555-1111 </pl:Entry>
</Phonelist>
```

i-sax1hn.p is the handler procedure with namespace processing.

i-sax1hn.p

```openedge
/* Name attribute for the current entry. App gets it during the StartElement callback */
DEFINE VARIABLE currentPerson AS CHARACTER NO-UNDO.

/* Phone number from the current entry. App gets it during the Characters callback because it is the character data for the element. */
DEFINE VARIABLE currentNum AS CHARACTER NO-UNDO.

/* This procedure is called when the parser finds the start tag for an element. */
```
For this particular XML doc, the app simply looks for "Entry" elements and digs out the "ContactName" attribute during theStartElement call, saving it in currentPerson. The code assumes that Namespace processing is enabled and checks to make sure that name parameters are part of the correct namespace. */

PROCEDURE StartElement:
DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER attributes AS HANDLE NO-UNDO.

IF namespaceURI = "http://www.wmhwmh.biz/ns/phonelist" THEN DO:
   IF localName = "Entry" THEN
      currentPerson = attributes:GET-VALUE-BY-NAMESPACE-NAME
                       ("http://www.wmhwmh.biz/ns/phonelist", "ContactName").
END.
END PROCEDURE.

/* This callback gets passed the character data for an element. SAX does not guarantee that all the characters for an element get passed in one call -- that's why the app has to maintain the currentNum global variable and append to it when handling Characters, and also why it has to wait for EndElement before displaying the message box. (Some apps may need to use a MEMPTR to accumulate the character data, which may exceed the 32K ABL CHARACTER variable limit) */

PROCEDURE Characters:
DEFINE INPUT PARAMETER charData AS MEMPTR NO-UNDO.
DEFINE INPUT PARAMETER numChars AS INTEGER NO-UNDO.

/* Can assume that any call to Characters is for an Entry's text value, because we know what the document looks like. If this weren't the case, we'd have to keep track of the localName passed to the most recent call to StartElement) */
currentNum = currentNum + GET-STRING(charData, 1, GET-SIZE(charData)).
END PROCEDURE.

/* This callback is called when the parser finds the end tag for an Element. This app only cares about the end of an Entry element. */

PROCEDURE EndElement:
DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.

IF namespaceURI = "http://www.wmhwmh.biz/ns/phonelist" THEN DO:
   IF localName = "Entry" THEN DO:
      MESSAGE "Name: " currentPerson SKIP
            "Phone Number: " currentNum VIEW-AS ALERT-BOX.
      ASSIGN
         currentNum = 
         currentPerson = "".
      END.
   END.
END.

PROCEDURE EndDocument:
MESSAGE "All Done" VIEW-AS ALERT-BOX.
END PROCEDURE.

When this driver with namespace processing is run, it produces the following trace:
Trace of SAX driver with namespace processing

Callback function: StartDocument
Callback function: StartElement
  namespaceURI: http://www.wmhwmh.biz/ns/Default
  localName: Phonelist
  qName: Phonelist
  SAX-ATTRIBUTE has 0 items:
Callback function: StartElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
  SAX-ATTRIBUTE has 1 items:
    Attribute 1 :
      namespaceURI: http://www.wmhwmh.biz/ns/phonelist
      localName: ContactName
      qName: pl:ContactName
      type: CDATA
      value: Jane Jones
Callback function: Characters
  charData: 555 555-5555
Callback function: EndElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
Callback function: StartElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
  SAX-ATTRIBUTE has 1 items:
    Attribute 1 :
      namespaceURI: http://www.wmhwmh.biz/ns/phonelist
      localName: ContactName
      qName: pl:ContactName
      type: CDATA
      value: John Smith
Callback function: Characters
  charData: 555 555-1111
Callback function: EndElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
Callback function: EndElement
  namespaceURI: http:/www.wmhwmh.biz/ns/Default
  localName: Phonelist
  qName: Phonelist
Callback function: EndDocument

Example code: reading customer data and writing a TEMP-TABLE

This example is a SAX version of the DOM example described in Reading and Writing XML with the Document Object Model (DOM) on page 31. The example reads an XML file containing the Customer table from the Sports database and writes the data to a temp-table. The example uses qname, assumes there is no namespace prefix, and, for clarity, omits code for transaction scoping and validation. The SAX driver procedure, i-sax2d.p, is shown here:
i-sax2d.p

DEFINE VARIABLE hParser AS HANDLE NO-UNDO.
DEFINE VARIABLE hHandler AS HANDLE NO-UNDO.

/* Create the SAX-READER object */
CREATE SAX-READER hParser.

/* Run the persistent procedure that contains the callbacks */
RUN "i-sax2h.p" PERSISTENT SET hHandler.

/* Give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse. This XML file does not use namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "i-sax2.xml").
hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, the callbacks have been called as many times as necessary and we’re done processing the XML document (or there was an error) */
IF ERROR-STATUS:ERROR THEN DO:
   IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
      /* unable to begin the parse */
      MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
   ELSE
      /* error detected in a callback */
      MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
   END.
ELSE
   MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.

DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.

i-sax2.xml is the associated XML document.

i-sax2.xml

<?xml version='1.0' ?>
<Customers>
   <Customer Name="Lift Line Skiing" Cust-num="1">
      <Country>USA</Country>
      <Address>276 North Street</Address>
      <Address2></Address2>
      <City>Boston</City>
      <State>MA</State>
      <Postal-Code>02114</Postal-Code>
      <Contact>Gloria Shepley</Contact>
      <Phone>(617) 450-0087</Phone>
      <Sales-Rep>HXM</Sales-Rep>
      <Credit-Limit>66700</Credit-Limit>
      <Balance>42568</Balance>
      <Terms>Net30</Terms>
      <Discount>35</Discount>
      <Comments>This customer is on credit hold.</Comments>
   </Customer>
   <Customer Name="Hoops " Cust-num="3">
      <Country>USA</Country>
      <Address>Suite 415</Address>
      <Address2>40 Grove St.</Address2>
      <City>Atlanta</City>
      <State>GA</State>
   </Customer>
</Customers>
i-sax2h.p is the handler procedure.

```
DEFINE VARIABLE hBuf AS HANDLE NO-UNDO.
DEFINE VARIABLE hDBFld AS HANDLE NO-UNDO.
/* Variable in which to accumulate all the text data for one element coming in through potentially multiple calls (per element) to the Characters procedure */
DEFINE VARIABLE currentFieldValue AS CHARACTER NO-UNDO.
/* Simple-minded state machine - the code makes minimal use of it, but it could easily be used to validate the structure of the document in this example. */
DEFINE VARIABLE iProcessingState AS INTEGER NO-UNDO.
/* So we can create new records*/
DEFINE TEMP-TABLE ttCustomer LIKE Customer.

&SCOPED-DEFINE READY-TO-START 1
&SCOPED-DEFINE GETTING-RECORDS 2
&SCOPED-DEFINE GETTING-FIELDS 3
&SCOPED-DEFINE DONE 4

hBuf = BUFFER ttCustomer:HANDLE.
PROCEDURE StartDocument:
iProcessingState = {&READY-TO-START}.
END PROCEDURE.

PROCEDURE StartElement:
DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER attributes AS HANDLE NO-UNDO.

IF qName = "Customers" THEN
  iProcessingState = {&GETTING-RECORDS}.
ELSE IF qName = "Customer" THEN DO:
  /* Starting a new customer, so create the record */
  CREATE ttCustomer.
END IF
ASSIGN
/* Get the fields that are in the XML doc as attributes */
ttCustomer.CustNum =
       INTEGER(attributes:GET-VALUE-BY-QNAME("CustNum"))
ttCustomer.Name   = attributes:GET-VALUE-BY-QNAME( "Name" )
```

**Note:** There is no DTD or XML Schema and no use of namespace prefixes. The lack of a DTD or XML schema means that the handlers need to validate the document, but this example omits that validation for the sake of clarity.
iProcessingState = {&GETTING-FIELDS}.
END.
ELSE IF iProcessingState = {&GETTING-FIELDS} THEN DO:
    /* Get a handle to the field whose name corresponds to the element name */
    hDBFld = hBuf:BUFFER-FIELD(qName).
    /* Re-init the variable in which we accumulate the field value */
    currentFieldValue = "".
END.
END PROCEDURE.

PROCEDURE Characters:
    DEFINE INPUT PARAMETER charData AS MEMPTR NO-UNDO.
    DEFINE INPUT PARAMETER numChars AS INTEGER NO-UNDO.
    /* Get the text value of the field (hDBFld was set to the correct field
     in StartElement */
    currentFieldValue = currentFieldValue +
                      GET-STRING(charData, 1, GET-SIZE(charData)).
END PROCEDURE.

PROCEDURE EndElement:
    DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
    IF localName = "Customers" THEN
        iProcessingState = {&DONE}.
    ELSE IF localName = "Customer" THEN
        iProcessingState = {&GETTING-RECORDS}.
    ELSE IF iProcessingState = {&GETTING-FIELDS} THEN
        hDBFld:BUFFER-VALUE = currentFieldValue.
    END PROCEDURE.

PROCEDURE EndDocument:
    /* Show that data made it by displaying temp-table */
    FOR EACH ttCustomer:
        DISPLAY ttCustomer.Name.
    END.
    RUN Cleanup.
END PROCEDURE.

PROCEDURE FatalError:
    DEFINE INPUT PARAMETER errorMessage AS CHARACTER NO-UNDO.
    /* Not necessary to do anything with PRIVATE-DATA, this is just an example
    of what you could do */
    SELF:PRIVATE-DATA = "FATAL".
    RUN Cleanup.
    /* RETURN ERROR in an error handler implicitly calls SELF:STOP-PARSING( ),
    sets SELF:PARSE-STATUS to SAX-PARSER-ERROR, and raises the ABL ERROR
    condition. */
    RETURN ERROR errorMessage
          + "(Line " + STRING(SELF:LOCATOR-LINE-NUMBER)
          + ", Col " + STRING(SELF:LOCATOR-COLUMN-NUMBER) + ")".
END PROCEDURE.

/* This is not a SAX callback; it is just a local utility */
PROCEDURE Cleanup:
    /* In case we have parsed previous documents */
    hBuf:EMPTY-TEMP-TABLE( ).
END.
Note: Alternately, you could use the `RETURN ERROR` error-object-expression syntax and handle the resulting error object with a `CATCH` block in the caller. For more information on this type of structured error handling, see *OpenEdge Development: Error Handling*.

ABL SAX and WebSpeed

This section describes how to use ABL SAX with WebSpeed applications. To use SAX with WebSpeed applications:

1. Check that the `WEB-CONTEXT` object’s `IS-XML` attribute is `TRUE`. This indicates that the WebSpeed transaction server recognizes that an XML document was posted to it.

   Note: The `WEB-CONTEXT` object’s `VALIDATE-XML` attribute applies only to DOM, not to SAX.

2. After you create the SAX-reader object, run the `SET-INPUT-SOURCE()` method as follows:

   ```abl
   ```

   At this point, proceed with the WebSpeed application as if it were any other ABL SAX application.

Example code: reading XML data using WebSpeed

This example reads XML data using WebSpeed using the `i-sax3s.p` server procedure. The example can use the callbacks in `i-sax2h.p` (the example handler procedure from the previous example code).

`i-sax3s.p`

```abl
/* This particular procedure is intended to be run on a server with an available web server and functioning WebSpeed broker/messenger. */
/* This is needed to support webspeed applications */
{src/web/method/cgidefs.i}

DEFINE VARIABLE hHandler AS HANDLE NO-UNDO.
DEFINE VARIABLE hParser AS HANDLE NO-UNDO.

CREATE SAX-READER hParser.

/* Run the persistent procedure that contains the callbacks */
RUN "i-sax2h.p" PERSISTENT SET hHandler.

/* Give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Check to see if there is an XML document available on the webstream and if true, give it to the sax parser. */
IF (WEB-CONTEXT:IS-XML) THEN
```
hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, our callbacks have been called as many times as necessary and we're done processing the XML document (or there was an error) */

IF ERROR-STATUS:ERROR THEN DO:
  IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
    /* Unable to begin the parse */
    MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
  ELSE
    /* Error raised in a callback */
    MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
  MESSAGE "Document parsed successfully".

DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.

SAX and the AppBuilder

To use the AppBuilder to develop ABL SAX applications, perform the following tasks:

• Create a SAX handler object
• Supply an override for each callback your application requires
• Handle the context (optional)

To create a SAX Handler object which corresponds to a procedure (.p) file to contain the SAX callbacks:

1. From the AppBuilder main menu, select File > New. The New dialog box appears.
2. Select the Procedures toggle box. The SAX Handler Template appears:

3. Select Sax Handler and click OK. The new Sax Handler object appears, along with the Section Editor for it:
You use the Section Editor to create the callbacks required by your SAX application.

4. Change the **Section** combo box to **Procedure**. The **New Procedure** dialog box appears.

5. From the **Name** drop-down list, select the name of the desired callback. Then select the **Override** button and click **OK**.

![New Procedure dialog box](image)

**Note:** ABL implements SAX callbacks as super procedures (which you can override) of the SAX Handler object.

The **Section Editor** displays the selected callback procedure, as shown:

![Section Editor](image)

6. Modify the callback as desired, then save it.
Note: If you misspell the name of a callback, it is not invoked at run time. Rather, the corresponding internal procedure in the super procedure is invoked.

Storing and recalling context information

The SAX specification does not say how to store and recall context information; that is, information on how XML elements are related to each other. For example, the SAX specification says that when the SAX parser encounters a new element, a startElement event should be triggered and the StartElement callback should be invoked. However, the SAX specification does not say how to determine the new element’s parent.

ABL SAX provides a solution. Three of the AppBuilder templates for SAX callbacks refer to a temp-table. The temp-table and its records can be used as a stack to record context information related to that callback. When this feature is turned on:

- The AVM creates a new temp-table record each time the SAX parser encounters the beginning of a new element
- The AVM deletes the temp-table record each time the SAX parser encounters the end of the element

The information recorded in each temp-table record includes the parameters passed to the StartElement callback and the element’s path (position) in the element hierarchy. For example, in the following XML example, the path of the customer element is /customer and the path of the order element is /customer/orders/order:

```
<customer custnum="1" name="Lift Line Skiing">
  <orders>
    <order ordernum="1">
      </order>
    </orders>
</customer>
```

To activate context management (which is inactive by default), call the setContextMode( ) function, as demonstrated in the following code fragment:

```
RUN myHandler.p PERSISTENT SET hHandler.
DYNAMIC-FUNCTION("setContextMode" IN hHandler, TRUE).
hParser:HANDLER = hHandler.
```

ABL SAX provides context management for the following SAX callbacks:

- StartDocument
- StartElement
- EndElement
Context management example

Here is a fragment that demonstrates the OpenEdge context management system. The fragment retrieves the handle to the context management table, then finds the element added most recently, as shown:

```abl
PROCEDURE getTopElement:
  DEFINE OUTPUT PARAMETER cElementname AS CHARACTER NO-UNDO.
  DEFINE VARIABLE fld AS HANDLE NO-UNDO.
  DEFINE VARIABLE bh AS HANDLE NO-UNDO.
  mhStack = DYNAMIC-FUNCTION("getStackHandle").
  IF VALID-HANDLE(mhStack) THEN DO:
    bh = mhStack:DEFAULT-BUFFER-HANDLE.
    bh:FIND-LAST( ) NO-ERROR.
    fld = bh:BUFFER-FIELD("cQName") NO-ERROR.
    cElementname = fld:BUFFER-VALUE NO-ERROR.
  END.
END PROCEDURE.
```

SAX API reference

This reference contains the following sections:

- **SAX error message reference** on page 84
- **SAX callback reference** on page 85

For definitions of the ABL elements related to the SAX-reader and SAX-attributes objects, see *OpenEdge Development: ABL Reference*.

SAX error message reference

The following table explains the error messages that ABL SAX provides.

<table>
<thead>
<tr>
<th>Error message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Couldn't initialize proxml (or libproxml)</td>
<td>proxml.dll (or libproxml.so) was missing or incomplete, or XML could not be initialized</td>
</tr>
<tr>
<td>Parser not running for SAX-PARSE-NEXT</td>
<td>Could not read the next part of the XML document: the parser is not running</td>
</tr>
<tr>
<td>Document not found</td>
<td>file-name was not found</td>
</tr>
<tr>
<td>Handler procedure not found</td>
<td>Could not process XML document: invalid procedure handle for the handler</td>
</tr>
</tbody>
</table>
SAX callback reference

This section contains a reference entry for each callback ABL SAX supports. Each entry specifies the signature and defines each parameter. The callbacks, in alphabetical order, are:

- **Characters** on page 85
- **EndDocument** on page 86
- **EndElement** on page 86
- **EndPrefixMapping** on page 87
- **Error** on page 87
- **FatalError** on page 87
- **IgnorableWhitespace** on page 88
- **NotationDecl** on page 88
- **ProcessingInstruction** on page 89
- **ResolveEntity** on page 89
- **StartDocument** on page 91
- **StartElement** on page 91
- **StartPrefixMapping** on page 92
- **UnparsedEntityDecl** on page 92
- **Warning** on page 93

These callbacks closely match those defined in the SAX2 documentation at [www.saxproject.org](http://www.saxproject.org). For more information on these callbacks, see this Web site.

**Characters**

Invoked when the XML parser detects character data.

**Syntax**

```plaintext
PROCEDURE Characters:
    DEFINE INPUT PARAMETER charData AS { MEMPTR | LONGCHAR }.
    DEFINE INPUT PARAMETER numChars AS INTEGER.
```

- **charData**

  A MEMPTR or LONGCHAR that contains a chunk of character data.

- **numChars**

  The number of characters contained in the MEMPTR.

**Note:** If a character requires more than one byte to encode, the value of numChars might not match the value returned by MEMPTR:GETSIZE().
The parser calls this method to report each chunk of character data. It might report contiguous character data in one chunk, or split it into several chunks. If validation is enabled, whitespace is reported by the `IgnorableWhitespace` callback.

Although this callback is intended to be called by the parser, the application can call `Characters` directly. Whoever calls `Characters` must free the `charData MEMPTR`. When ABL calls `Characters`, the AVM is responsible for freeing the `charData MEMPTR` (although if the application frees it, no harm results). If the application calls `Characters`, the application is responsible for freeing the `charData MEMPTR`.

To copy the `charData MEMPTR` such that the memory used by the copy is completely separate from the memory used by the original, use ABL assignment, which performs a deep copy. The following fragment demonstrates this:

```abla
memptrA = memptrB
```

For more information on ABL assignment, see *OpenEdge Development: ABL Reference*.

### EndDocument

Invoked when the XML parser detects the end of an XML document.

**Syntax**

```abla
PROCEDURE EndDocument:
```

### EndElement

Invoked when the XML parser detects the end of an element.

**Syntax**

```abla
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName AS CHARACTER.
```

**namespaceURI**

A `CHARACTER` string indicating the namespace URI of the element. If namespace processing is not enabled, or the element is not part of a namespace, the string is of length zero.

**localName**

A `CHARACTER` string indicating the nonprefixed element name. If namespace processing is not enabled, the string is of length zero.
**qName**

A CHARACTER string indicating the actual name of the element in the XML source. If the name has a prefix, qName includes it, whether or not namespace processing is enabled.

This callback corresponds to a preceding `StartElement` after all element content is reported.

**EndPrefixMapping**

Invoked when the XML parser detects that a prefix associated with namespace mapping has gone out of scope.

**Syntax**

```plaintext
PROCEDURE EndPrefixMapping:  
   DEFINE INPUT PARAMETER prefix AS CHARACTER.
```

**prefix**

A character string representing the prefix for a namespace declaration.

This callback is invoked only when namespace processing is enabled. It provides information not required by normal namespace processing. However, in some situations, this callback might be useful and even required.

**Error**

Invoked to report an error encountered by the parser while parsing the XML document.

**Syntax**

```plaintext
PROCEDURE Error:  
   DEFINE INPUT PARAMETER errMessage AS CHARACTER.
```

**errMessage**

A character string indicating the error message.

After this callback is invoked, the parser continues where it left off.

**FatalError**

Invoked to report a fatal error.

**Syntax**

```plaintext
PROCEDURE FatalError:  
   DEFINE INPUT PARAMETER errMessage AS CHARACTER.
```
**errMessage**

A character string indicating the error message.

The application must assume that after a fatal error is reported, the document is unusable and future parsing events might not be reported. However, the parser might try to continue to parse the document. To stop the parser after reporting a fatal error, execute `RETURN ERROR`.

**Note:** If you stop the parser by executing `STOP-PARSING()`, parsing stops, but no error condition is raised, no error message is reported, the SAX-reader object’s `PARSE-STATUS` attribute is set to `SAX-COMPLETE` rather than to `SAX-PARSER-ERROR`, and the driver might not know that an error occurred. For this reason, Progress Software Corporation recommends that to stop the parser after reporting a fatal error, execute `RETURN ERROR`.

---

**IgnorableWhitespace**

Invoked when the XML parser detects ignorable whitespace.

**Syntax**

```plaintext
PROCEDURE IgnorableWhitespace:
  DEFINE INPUT PARAMETER charData AS CHARACTER.
  DEFINE INPUT PARAMETER numChars AS INTEGER.
```

- **charData**
  - A `CHARACTER` string representing a contiguous block of ignorable whitespace in an XML document.

- **numChars**
  - An `INTEGER` expression indicating the size, in characters, of the character string.

If validation is enabled, the XML parser reports ignorable whitespace through this callback. If validation is not enabled, the XML parser reports whitespace through the `Characters` callback.

The data type of `charData` is `CHARACTER`, not `MEMPTR`, because it is unlikely that an XML document has over 32K of contiguous ignorable whitespace.

---

**NotationDecl**

Invoked when the XML parser detects a notation declaration.

**Syntax**

```plaintext
PROCEDURE NotationDecl:
  DEFINE INPUT PARAMETER name AS CHARACTER.
  DEFINE INPUT PARAMETER publicID AS CHARACTER.
  DEFINE INPUT PARAMETER systemID AS CHARACTER.
```
name

A character string representing the name of the notation.

publicID

Optional. A character string indicating the public identifier of the entity. If none is supplied, the string is of length zero.

systemID

Optional. A character string indicating the system identifier of the entity. If none is supplied, the string is of length zero. systemID must be one of the following:

- Absolute file path
- Relative file path
- Absolute URI

ProcessingInstruction

Invoked when the XML parser detects a processing instruction.

Syntax

```plaintext
PROCEDURE ProcessingInstruction:
    DEFINE INPUT PARAMETER target AS CHARACTER.
    DEFINE INPUT PARAMETER data AS CHARACTER.
```

target

A character string indicating the target of the processing instruction.

data

A character string indicating the data associated with the processing instruction. If the processing instruction has no data, the length of the string is zero.

Note: A processing instructions can appear before or after a root element.

ResolveEntity

Invoked to let the application specify the location of an external entity (such as a DTD or XML Schema).

When the parser finds an external entity reference, it calls ResolveEntity, passing it the system identifier and public identifier (if any) contained in the XML. This gives the application a chance to override the location specified in the XML.
Syntax

PROCEDURE ResolveEntity:
DEFINE INPUT PARAMETER publicID AS CHARACTER.
DEFINE INPUT PARAMETER systemID AS CHARACTER.
DEFINE OUTPUT PARAMETER filePath AS CHARACTER.
DEFINE OUTPUT PARAMETER memPointer AS { MEMPTR | LONGCHAR }.

publicID

Optional. A character string indicating the public identifier of the entity. If none is supplied, the string is of length zero.

systemID

A character string indicating the system identifier of the entity. The character string will not be of length zero, as this parameter is required. systemID will be one of the following:

• Absolute file path
• Relative file path
• Absolute URL

filePath

Optional. A character string indicating the actual location of the entity being resolved. This tells the parser where to actually get the entity, in preference to the location specified by the system identifier.

filePath will be one of the following:

• Absolute file path
• Relative file path
• HTTP URI

If you do not supply filePath, set it to the Unknown value (?).

memPointer

Optional. A MEMPTR or LONGCHAR containing the entity being resolved. Use memPointer to return XML representing an entity that is not stored as a stand-alone file.

If you do not supply memPointer, set it to the Unknown value (?).

Caution: Supplying both filePath and memPointer is an error.

If the application does not implement this callback, or if the callback sets both filePath and memPointer to the Unknown value (?), the entity is resolved according to the following rules (which are also the rules that the ABL DOM interface uses):
1. If the location given in the XML source is a relative path and the `SAX-reader:SCHEMA-PATH` attribute has been set, try appending the relative path to each entry in `SCHEMA-PATH` and retrieving the file there.

2. If the location is a relative file path and the `SAX-reader:SCHEMA-PATH` attribute has the `Unknown` value (?), try retrieving the file relative to the working directory.

3. If the location given in the XML source is an absolute path to a local file or if it is an HTTP URI, try retrieving the file at the specified location.

4. If the file cannot be found, the parser calls the `FatalError` callback (if there is one) and stops processing the XML.

**StartDocument**

Invoked when the XML parser detects the start of an XML document.

**Syntax**

```
PROCEDURE StartDocument:
```

_StartDocument_ does not provide any data.

**StartElement**

Invoked when the XML parser detects the beginning of an element.

**Syntax**

```
PROCEDURE StartElement:
    DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
    DEFINE INPUT PARAMETER localName AS CHARACTER.
    DEFINE INPUT PARAMETER qName AS CHARACTER.
    DEFINE INPUT PARAMETER attributes AS HANDLE.
```

*namespaceURI*

A character string indicating the namespace URI of the element. If namespace processing is not enabled or the element is not part of a namespace, the string is of length zero.

*localName*

A character string indicating the non-prefixed element name. If namespace processing is not enabled, the string is of length zero.

*qName*

A character string indicating the actual name of the element in the XML source. If the name has a prefix, *qName* includes it, whether or not namespace processing is enabled.
attributes

A handle to a SAX-attributes object, which provides access to all attributes specified for
the element. If the element has no attributes, attributes is still a valid handle, and
the NUM-ITEMS attribute is zero.

For every invocation of StartElement, there is a corresponding invocation of EndElement.
The contents of the element are reported in sequential order before the corresponding EndElement
is invoked.

When StartElement returns, the SAX-attributes object, which was created by the AVM is deleted
by the AVM.

Note: If the application deletes it first, however, no harm is done.

StartPrefixMapping

Invoked when the XML parser detects that a prefix associated with namespace mapping is coming
into scope.

Note: This callback is invoked only when namespace processing is enabled.

Syntax

PROCEDURE StartPrefixMapping:
   DEFINE INPUT PARAMETER prefix AS CHARACTER.
   DEFINE INPUT PARAMETER uri AS CHARACTER.

prefix

A character string representing the prefix for a namespace declaration.

uri

A character string representing the URI that identifies the namespace being declared.

This callback does not normally need to be implemented, since the information it provides is not
required for normal namespace processing. But, it might be useful (and even required) in some
situations.

UnparsedEntityDecl

Invoked when the XML parser detects an entity that it does not parse (where "unparsed entity"
has the definition given in the XML 1.0 specification).
Syntax

PROCEDURE UnparsedEntityDecl:
  DEFINE INPUT PARAMETER name AS CHARACTER.
  DEFINE INPUT PARAMETER publicID AS CHARACTER.
  DEFINE INPUT PARAMETER systemID AS CHARACTER.
  DEFINE INPUT PARAMETER notationName AS CHARACTER.

name

A character string indicating the name of the entity.

publicID

Optional. A character string indicating the public identifier of the entity. If publicID is not supplied, the character string is of length zero.

systemID

Optional. A character string representing the system identifier of the entity. If systemID is not supplied, the character string is of length zero. systemID must be one of the following:

• Absolute file path
• Relative file path
• Absolute URI

notationName

A character string indicating the name of the notation associated with the entity.

Warning

Invoked to report a warning.

Syntax

PROCEDURE Warning:
  DEFINE INPUT PARAMETER errMsg AS CHARACTER.

eMsg

A character string indicating the error message.

A warning is a condition that is less severe than an error or a fatal error, as determined by the XML parser. After this callback is invoked, the parser continues where it left off.
This chapter assumes that you are familiar with Reading XML Documents with the Simple API for XML (SAX) on page 55.

A SAX-writer is an ABL object that streams (writes) an XML document to a specified target using the built-in support for the SAX API. ABL methods and attributes allow you to easily set up and control the XML write. This chapter describes the SAX-writer.

For details, see the following topics:

- SAX-writer overview
- Creating a SAX-writer
- Configuring a SAX-writer
- Writing an XML document
- Examples
- Handling namespaces
- Handling errors
SAX-writer overview

The SAX-writer is an ABL object created with the `CREATE SAX-WRITER` statement and accessed through a SAX-writer object handle. A collection of attributes and methods on the handle allow you to configure options and stream the XML content element by element to a variety of output destinations.

Writing XML with a SAX-writer object is an alternative to outputting the DOM document tree of an X-document object. The DOM document tree needs to be completely built and stored in memory before you can output it. In contrast, the SAX-writer object only needs enough memory to handle the largest single element in your XML output. The streaming nature of the SAX-writer object makes it a better choice for writing large XML documents.

The following table summarizes the attributes and methods of the SAX-writer object.

Table 15: SAX-writer attributes and methods

<table>
<thead>
<tr>
<th>Attribute or Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCODING attribute</td>
<td>Returns the name of the character encoding used to encode the contents of an XML document (for example, UTF-8, UTF-16, ASCII, and so on). The values must match the standard IANA encoding values.</td>
</tr>
<tr>
<td>FORMATTED attribute</td>
<td>Determines whether the XML output has extra formatting to make it a human-readable document.</td>
</tr>
<tr>
<td>FRAGMENT attribute</td>
<td>Specifies if the output of a SAX-writer object is a complete document or a fragment.</td>
</tr>
<tr>
<td>STANDALONE attribute</td>
<td>Determines the value of the standalone attribute in the XML declaration.</td>
</tr>
<tr>
<td>STRICT attribute</td>
<td>Determines if the SAX-writer object should ensure that the XML document is well formed XML.</td>
</tr>
<tr>
<td>VERSION attribute</td>
<td>Determines the value of the version string in the XML declaration.</td>
</tr>
<tr>
<td>WRITE-STATUS attribute</td>
<td>The current state of an XML write in a SAX-writer object. Certain method calls are only valid with certain status values.</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE( ) method</td>
<td>Adds a namespace declaration to an XML element.</td>
</tr>
<tr>
<td>END-DOCUMENT( ) method</td>
<td>Closes the XML document.</td>
</tr>
<tr>
<td>END-ELEMENT( ) method</td>
<td>Ends an XML element based upon the specified element name.</td>
</tr>
<tr>
<td>INSERT-ATTRIBUTE( ) method</td>
<td>Adds a single attribute to a start tag in an XML element.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>RESET() method</td>
<td>Closes the open stream and resets the SAX-writer object to its default values.</td>
</tr>
<tr>
<td>SET-OUTPUT-DESTINATION() method</td>
<td>Defines the target of the XML document that the SAX-writer object creates.</td>
</tr>
<tr>
<td>START-DOCUMENT() method</td>
<td>Creates the XML document with the prolog information.</td>
</tr>
<tr>
<td>START-ELEMENT() method</td>
<td>Starts an XML element based upon the specified name.</td>
</tr>
<tr>
<td>WRITE-CDATA() method</td>
<td>Adds a CDATA block to an XML element.</td>
</tr>
<tr>
<td>WRITE-CHARACTERS() method</td>
<td>Adds character data to an XML element.</td>
</tr>
<tr>
<td>WRITE-COMMENT() method</td>
<td>Adds a comment to the XML document.</td>
</tr>
<tr>
<td>WRITE-DATA-ELEMENT() method</td>
<td>Adds a complete XML element.</td>
</tr>
<tr>
<td>WRITE-EMPTY-ELEMENT() method</td>
<td>Creates an empty XML element.</td>
</tr>
<tr>
<td>WRITE-EXTERNAL-DTD() method</td>
<td>Adds an external Document Type Definition (DTD) reference to an XML document.</td>
</tr>
<tr>
<td>WRITE-ENTITY-REF() method</td>
<td>Adds an entity reference to the XML stream.</td>
</tr>
<tr>
<td>WRITE-FRAGMENT() method</td>
<td>Adds character data to the XML element.</td>
</tr>
<tr>
<td>WRITE-PROCESSING-INSTRUCTION() method</td>
<td>Creates a processing instruction node in an XML document.</td>
</tr>
</tbody>
</table>

## Creating a SAX-writer

Use the `CREATE SAX-WRITER` statement to create a SAX-writer and assign its handle to a handle variable.

```
CREATE SAX-WRITER handle [ IN WIDGET-POOL pool-name ] [ NO-ERROR ]
```

Here is an example:

```
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
CREATE SAX-WRITER hSAXWriter.
```
Configuring a SAX-writer

Next, you configure the values that control how the write occurs. Below are some steps you might want to consider:

- If you are reusing a SAX-writer object for multiple writes, call the RESET method to clear the object and reset the default values.
- Use the SET-OUTPUT-DESTINATION method to declare the output destination and its type (file, longchar, memptr, stream).
- Set the VERSION, ENCODING, and STANDALONE attributes to ensure that the XML documents prolog is correctly configured.
- Set the FRAGMENT attribute if you are writing XML content that is not a complete XML document.
- Set the FORMATTED attribute if you want the XML output to have extra whitespace for easy readability.
- Set the STRICT attribute to FALSE if you do not want the write to fail if the parser detects invalid XML.

Here is an example:

```plaintext
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE lok AS LOGICAL NO-UNDO.

CREATE SAX-WRITER hSAXWriter.
hSAXWriter:FORMATTED = TRUE.
lok = hSAXWriter:SET-OUTPUT-DESTINATION("file", "sw-example.xml").
```

Writing an XML document

The program flow of an XML write follows this general pattern in a forward-only stream:

- The write begins with the START-DOCUMENT method and the document prolog is written.
- To build an element, use the START-ELEMENT method to create the start tag for the desired element type. If you have created a SAX-attributes either to programatically build a list of attributes for the element or to capture a set of attributes during a SAX read operation, you can pass it as an optional parameter to the START-ELEMENT method. (The WRITE-EMPTY-ELEMENT and WRITE-DATA-ELEMENT methods also can take a SAX-attributes object as an optional parameter.)
- Use the following methods to build the content of the tag:
  - INSERT-ATTRIBUTE( ) method
  - DECLARE-NAMESPACE( ) method
  - WRITE-CDATA( ) method
  - WRITE-CHARACTERS( ) method
• WRITE-ENTITY-REF( ) method
• WRITE-FRAGMENT( ) method

• Use the END-ELEMENT method to provide the closing tag for the named element.
• Use the other methods to provide other types of XML content:
  • WRITE-EMPTY-ELEMENT( ) method
  • WRITE-DATA-ELEMENT( ) method
  • WRITE-EXTERNAL-DTD( ) method
  • WRITE-PROCESSING-INSTRUCTION( ) method
  • WRITE-COMMENT( ) method

• Use the END-DOCUMENT method to close the XML document.

Examples

This section shows examples of common use cases.

Creating an XML document from database data

The sw-example.p example outputs part of the Customer table of the Sports database.

sw-example.p

/* Write out the Customer table of the Sports2000 sample database using the methods of the SAX-WRITER Object */
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE lOK AS LOGICAL NO-UNDO.
CREATE SAX-WRITER hSAXWriter.
hSAXWriter:FORMATTED = TRUE.
lOK = hSAXWriter:SET-OUTPUT-DESTINATION("file", "sw-example.xml").
lOK = hSAXWriter:START-DOCUMENT( ).
lOK = hSAXWriter:START-ELEMENT("customers").
FOR EACH Customer NO-LOCK WHERE Customer.CustNum < 5:
  ASSIGN
  lOK = hSAXWriter:START-ELEMENT("customer").
lOK = hSAXWriter:INSERT-ATTRIBUTE("CustNum", STRING(Customer.CustNum)).
lOK = hSAXWriter:INSERT-ATTRIBUTE("Name", Customer.Name).
lOK = hSAXWriter:WRITE-DATA-ELEMENT("Phone", Customer.Phone).
lOK = hSAXWriter:WRITE-DATA-ELEMENT("CreditLimit", ")
Partial output from the `sw-example.p` program is shown below.

**sw-example.xml (partial output)**

```xml
<?xml version="1.0"?>
<customers>
  <customer Cust-Num="1" Name="Lift Line Skiing">
    <Address>276 North Street</Address>
    <City>Boston</City>
    <State>MA</State>
    <Postal-Code>02114</Postal-Code>
    <Country>USA</Country>
    <Phone>(617) 450-0087</Phone>
    <Contact>Gloria Shepley</Contact>
    <Credit-Limit>66700</Credit-Limit>
    <Balance>42568</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
  </customer>
  <customer Cust-Num="2" Name="Urpon Frisbee">
    <Address>Rattipolku 3</Address>
    <City>Valkeala</City>
    <State>Uusimaa</State>
    <Postal-Code>45360</Postal-Code>
    <Country>Finland</Country>
    <Phone>(60) 532 5471</Phone>
    <Contact>Urpo Leppakoski</Contact>
    <Credit-Limit>27600</Credit-Limit>
    <Balance>17166</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>Ship all products 2nd Day Air.</Comments>
  </customer>
...  
</customers>
```
Here is an example of a simple SAX-writer application that creates envelope information:

```plaintext
/* Program to write an envelope address */
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
CREATE SAX-WRITER hSAXWriter.
hSAXWriter:FORMATED = TRUE. /* Format output so it is easy to read */
hSAXWriter:SET-OUTPUT-DESTINATION("file", "mailing.xml").

hSAXWriter:START-DOCUMENT( ).
/* The ENCODING attribute defaults to UTF-8 */
/* The FRAGMENT attribute defaults to FALSE */
/* The STRICT attribute defaults to TRUE */

hSAXWriter:START-ELEMENT("psc:mailingaddress").
hSAXWriter:DECLARE-NAMESPACE("www.progress.com", "psc").
RUN xmlData(INPUT "name", INPUT "John Smith").

hSAXWriter:START-ELEMENT("psc:address").
hSAXWriter:INSERT-ATTRIBUTE("type", "personal"). /* Node has an attribute */
RUN xmlData(INPUT "psc:street", INPUT "411 Whatsup St.").
RUN xmlData(INPUT "psc:city", INPUT "Somerville").
RUN xmlData(INPUT "psc:state", INPUT "MA").
RUN xmlData(INPUT "psc:zipcode", INPUT "02143").
hSAXWriter:END-ELEMENT("psc:address").

hSAXWriter:START-ELEMENT("psc:address").
hSAXWriter:INSERT-ATTRIBUTE("type", "business").
RUN xmlData(INPUT "psc:name", INPUT "Progress Software").
RUN xmlData(INPUT "psc:street", INPUT "14 Oak Park").
RUN xmlData(INPUT "psc:city", INPUT "Bedford").
RUN xmlData(INPUT "psc:state", INPUT "MA").
RUN xmlData(INPUT "psc:zip", INPUT "01730").
hSAXWriter:END-ELEMENT("psc:address").

hSAXWriter:WRITE-EMPTY-ELEMENT("psc:default").
hSAXWriter:INSERT-ATTRIBUTE("type", "personal").

hSAXWriter:END-ELEMENT("psc:mailingaddress").
hSAXWriter:END-DOCUMENT( ). /* Document written to working directory. */

PROCEDURE xmlData:
DEFINE INPUT PARAMETER xmlNode AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER charData AS CHARACTER NO-UNDO.

    hSAXWriter:START-ELEMENT(xmlNode).
    hSAXWriter:WRITE-CHARACTERS(charData).
    hSAXWriter:END-ELEMENT(xmlNode).
END PROCEDURE.
```

The code from the previous example produces a document like the following:

```xml
<?xml version="1.0">
<psc:mailingaddress xmlns:psc="www.progress.com">
    <psc:name>John Smith</psc:name>
    <psc:address type="personal">
        <psc:street>411 Whatsup St.</psc:street>
        <psc:city>Somerville</psc:city>
        <psc:state>MA</psc:state>
        <psc:zipcode>02143</psc:zipcode>
    </psc:address>
</psc:mailingaddress>
```
Instead of a procedure, you could also use the method for creating a leaf node. For example:

```xml
<psc:name>John Smith</psc:name>
<psc:address type="business">
    <psc:name>Progress Software</psc:name>
    <psc:street>14 Oak Park Drive</psc:street>
    <psc:city>Bedford</psc:city>
    <psc:state>MA</psc:state>
    <psc:zipcode>01730</psc:zipcode>
</psc:address>
<psc:default type="personal"/>
</psc:mailingaddress>
```

Concurrently reading and writing XML documents

One common use case for the ABL SAX objects is to read a source XML document element by element, transform the elements, and output the updated elements to a new XML document. During this process, you might want to:

- Decide if the element should be included or excluded in the output
- Alter the attributes of the element
In this example, a simple XML document that represents address data is read in and transformed to create a new XML address list. You can read the comments embedded in the code to see examples of the kinds of changes that are possible. The example is provided as a single procedure which serves as both the SAX driver and the SAX handler (includes internal callback procedures). The program transforms the source XML document in a single parse. This technique forces custom processing logic down into the callback procedures. This can quickly lead to added complexity as you mingle your transformation logic with the logic of the parser life cycle. In reality, you will likely use a progressive scan parse for anything more than simple adjustments to source XML. In this scenario, you would use the SAX-reader to feed your driver procedure the next piece of XML data, use the logic in your driver procedure to process the data, and output the desired XML data with the SAX writer object.

The following is a snippet of the source XML document `sampledata2.xml`.

```
sampledata2.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<QAddress>
 <QHeader Revision="1.0" Mime="text/xml">
  <Queue OperatorID="299" Name="Agent Addresses" Region="New England"/>
  <Comments Text="This is a sample list of addresses for mailing labels or envelopes."/>
 </QHeader>
 <QBody>
  <Address Row="1" Catalog="No">
   <Contact>Joe Smith</Contact>
   <Name>Pedal Power Cycles</Name>
   <Urbanization>P.O. Box 1719</Urbanization>
   <Street>304 Hancock Street</Street>
   <City>Bangor</City>
   <State>ME</State>
   <Country>US</Country>
   <Zip>04402</Zip>
  </Address>
  ... 
 </QBody>
</QAddress>
```

This is the sample code:

```
sax-readwrite.p
/* This sample uses the SAX-reader, SAX-attributes and SAX-writer objects. SAX-reader reads the XML document. The SAX parser passes the attributes of each new element to the StartElement callback procedure in a SAX-attributes object. The StartElement transforms the XML document by manipulating attributes and passes the altered SAX-attributes object to the SAX-writer object. The SAX-writer object writes the data out to a new XML document. */
DEFINE VARIABLE hSAXReader AS HANDLE NO-UNDO.
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
```

Examples
/* Keep the current element name in a global scope. */
DEFINE VARIABLE CurrentTag AS CHARACTER NO-UNDO.

/* For a domestic mailing, do not use <Country> elements in new XML. */
DEFINE VARIABLE NoCountry AS LOGICAL NO-UNDO INITIAL TRUE.

/* For customers with PO Boxes, do not use <Street> elements in new XML. */
DEFINE VARIABLE NoStreet AS LOGICAL NO-UNDO.

/* New envelope line to request address correction. */
DEFINE VARIABLE EnvelopeSlug AS CHARACTER NO-UNDO
   INITIAL "Attention Post Master: Address Correction Requested.".

CREATE SAX-WRITER hSAXWriter.
   hSAXWriter:FORMATTED = TRUE.
   hSAXWriter:STANDALONE = TRUE.
   hSAXWriter:ENCODING = "UTF-8".
   hSAXWriter:SET-OUTPUT-DESTINATION("FILE", "sax-readwrite.xml").

hSAXWriter:START-DOCUMENT( ).
CREATE SAX-READER hSAXReader.
   hSAXReader:SET-INPUT-SOURCE("FILE", "sampledata.xml").
   hSAXReader:SAX-PARSE( ).
   hSAXWriter:END-DOCUMENT( ).
DELETE OBJECT hSAXWriter.
DELETE OBJECT hSAXReader.

******************************************************************
/* Callback procedures for SAX parser (SAX-reader object) */
******************************************************************

PROCEDURE StartElement:
   DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
   DEFINE INPUT PARAMETER hSAXAttributes AS HANDLE NO-UNDO.
   ASSIGN CurrentTag = localName.
   IF localName = "Qheader" THEN
      hSAXAttributes:INSERT-ATTRIBUTE("MailDate", STRING(TODAY)).
   ELSE IF localName = "Address" THEN DO:
      hSAXAttributes:REMOVE-ATTRIBUTE("Row").
      hSAXAttributes:UPDATE-ATTRIBUTE("Catalog", "Yes").
   END.
   /* This address will use a PO Box instead of a Street Address */
   IF localName = "Urbanization" THEN ASSIGN NoStreet = TRUE.
   /* Only call SAX-writer for elements wanted in new XML. */
   CASE localName:
      WHEN "Country" THEN IF NOT NoCountry THEN
         hSAXWriter:START-ELEMENT(localName, namespaceURI, hSAXAttributes).
      WHEN "Street" THEN IF NOT NoStreet THEN
         hSAXWriter:START-ELEMENT(localName, namespaceURI, hSAXAttributes).
      OTHERWISE hSAXWriter:START-ELEMENT(localName, namespaceURI, hSAXAttributes).
   END CASE.
   END PROCEDURE.

PROCEDURE Characters:
   DEFINE INPUT PARAMETER charData AS MEMPTR NO-UNDO.
   DEFINE INPUT PARAMETER numChars AS INTEGER NO-UNDO.
   DEFINE VARIABLE cData AS CHARACTER NO-UNDO.
   ASSIGN cData = GET-STRING(charData, 1, GET-SIZE(charData)).
   /* Only use 5 digit zip codes. */
IF CurrentTag = "Zip" THEN
  cData = SUBSTRING(cData, 1, 5).
/* Only write content with SAX-writer if a start tag is waiting for content.
"5" is a status of "SAX-WRITE-ELEMENT", which indicates that an end tag
was last written and this current data is associated with a element not
wanted in the output mailing list. */
IF hSAXWriter:WRITE-STATUS NE 5 THEN
  hSAXWriter:WRITE-CHARACTERS(cData).
END PROCEDURE.

PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
  /* Only call SAX-writer for elements wanted in output mailing list. */
  CASE localName:
    WHEN "Country" THEN IF NOT NoCountry THEN
      hSAXWriter:END-ELEMENT(localName, namespaceURI).
    WHEN "Street" THEN IF NOT NoStreet THEN
      hSAXWriter:END-ELEMENT(localName, namespaceURI).
    OTHERWISE hSAXWriter:END-ELEMENT(localName, namespaceURI).
  END CASE.
  /* Add another envelope line after Zip Code. */
  IF localName = "Zip" THEN DO:
    hSAXWriter:START-ELEMENT("EnvelopeSlug", namespaceURI).
    hSAXWriter:WRITE-CHARACTERS(EnvelopeSlug).
    hSAXWriter:END-ELEMENT("EnvelopeSlug", namespaceURI).
  END.
  /* Reset check for PO Box versus Street address. */
  IF localName = "Address" THEN
    ASSIGN NoStreet = FALSE.
  END PROCEDURE.

The following partial sax-readwrite.xml output file shows key changes in bold:

sax-readwrite.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<QAddress>
  <QHeader Revision="1.0" Mime="text/xml" MailDate="10/25/06">
    <Queue OperatorID="299" Name="Agent Addresses" Region="New England">
    </Queue>
    <Comments Text="Sample list of addresses...">
    </Comments>
  </QHeader>
  <QBody>
    <Address Catalog="Yes">
      <Contact>Joe Smith</Contact>
      <Name>Pedal Power Cycles</Name>
      <Urbanization>P.O. Box 1719</Urbanization>
      <City>Bangor</City>
      <State>ME</State>
      <Zip>04402</Zip>
      <EnvelopeSlug>Attention Post Master: Address Correction Requested.</EnvelopeSlug>
    </Address>
    .
    .
    .
  </QBody>
</QAddress>
```

OpenEdge Development: Working with XML 105
Note that the SAX-writer object `Start-Element( )` method takes a SAX-attributes object as an optional parameter. In this variation of the last example, all the transformation logic is stripped out and a SAX-attributes object is created and populated with attributes at the procedure’s top (global) scope. The calls to the SAX-writer object in the callback procedure ignore the SAX-attributes object created by the parser for SAX-reader and only pass the global SAX-attributes object to the SAX-writer if the current element is `address`. The end result is that all attribute data in the source XML is lost and only the attribute data created by the procedure is output to the new XML document. Note the changes shown in bold:

**sax-readwrite2.p**

```plaintext
/* This sample shows a programatically created SAX-attributes object used to override the attributes of the input XML document. */
DEFINE VARIABLE hSAXWriter AS HANDLE NO-UNDO.
DEFINE VARIABLE hSAXReader AS HANDLE NO-UNDO.
DEFINE VARIABLE hMySAXAttributes AS HANDLE NO-UNDO.
CREATE SAX-WRITER hSAXWriter.
hSAXWriter:FORMATTED = TRUE.
hSAXWriter:STANDALONE = TRUE.
hSAXWriter:ENCODING = "UTF-8".
hSAXWriter:SET-OUTPUT-DESTINATION("FILE", "sax-readwrite2.xml").

CREATE SAX-ATTRIBUTES hMySAXattributes.
hMySAXAttributes:INSERT-ATTRIBUTE("MailDate", STRING(TODAY)).
hMySAXAttributes:INSERT-ATTRIBUTE("Catalog", "Yes").
CREATE SAX-READER hSAXReader.
hSAXReader:SET-INPUT-SOURCE("FILE", "sampledata.xml").
hSAXReader:SAX-PARSE( ).
hSAXWriter:END-DOCUMENT( ).
DELETE OBJECT hSAXWriter.
DELETE OBJECT hSAXReader.
DELETE OBJECT hMySAXAttributes.

/* Callback procedures for SAX parser (SAX-reader object) */
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER localName AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER qName AS CHARACTER NO-UNDO.
  DEFINE INPUT PARAMETER hSAXAttributes AS HANDLE NO-UNDO.
  IF localName = "Address" THEN
    hSAXWriter:START-ELEMENT(localName, namespaceURI, hMySAXAttributes).
  ELSE
    hSAXWriter:START-ELEMENT(localName, namespaceURI).
  END.
...
```

The following partial output XML document is the result:
Handling namespaces

There are three kinds of namespaces:

- A local namespace, which is one that is declared in that tag
- An inherited namespace, which is one specified in an ancestor tag; for example, a namespace declared by the root node can be used in the entire document
- The default namespace, which is an unspecified namespace (no prefix provided)

There are two ways for a namespace to be created when writing an XML document. They can be created implicitly when starting an element or they can be declared explicitly after a tag has been started. Implicit namespaces are created with the methods `START-ELEMENT`, `WRITE-EMPTY-ELEMENT`, and `WRITE-DATA-ELEMENT`. Explicit namespaces are created with the methods `DECLARE-NAMESPACE` or `INSERT-ATTRIBUTE`. The following table describes these methods.
<table>
<thead>
<tr>
<th>Method call example and resulting tag</th>
<th>Use case explanation</th>
</tr>
</thead>
</table>
| START-ELEMENT("name", "")<name>      | **Case**: You supply an element *name*, but no namespace prefix and no namespace URI.  
**Result**: The element is written without a prefix and the default namespace is used. |
| START-ELEMENT("prefix:name", "")<prefix:name> | **Case**: You supply an element *name* with a namespace prefix, but no namespace URI. The supplied prefix has been previously associated with a namespace URI.  
**Result**: The element is written with the supplied *prefix* and the previously associated namespace URI is used. |
| START-ELEMENT("prefix:name", ")error | **Case**: You supply an element *name* with a namespace prefix, but no namespace URI. The supplied prefix has *not* been previously associated with a namespace URI. The **STRICT** attribute is set to TRUE.  
**Result**: The call generates an error. Only the default namespace can be set to an empty string (""). |
| START-ELEMENT("prefix:name", "")<prefix:name xmlns:prefix=""| **Case**: You supply an element *name* with a namespace prefix, but no namespace URI. The supplied prefix has *not* been previously associated with a namespace URI. The **STRICT** attribute is set to FALSE.  
**Result**: The element is written with the supplied *prefix* and an empty namespace is specified. |
| START-ELEMENT("name", "namespaceUri")<prefix:name> | **Case**: You supply an element *name* without a namespace prefix. You supply a *namespaceUri* that has been previously associated with a namespace prefix.  
**Result**: The element is written using the previously associated prefix and the declared namespace URI is used. |
| START-ELEMENT("name", "namespaceUri")<name xmlns="namespaceUri"> | **Case**: You supply an element *name* without a namespace prefix. You supply a *namespaceUri* that has *not* been previously associated with a namespace prefix.  
**Result**: The element is written without a prefix and the *namespaceUri* is set to the namespace URI associated with the default namespace. |
| START-ELEMENT("prefix:name", "namespaceUri")<prefix:name> | **Case**: You supply an element *name* with a namespace prefix and a *namespaceUri* that has been previously associated with the supplied prefix.  
**Result**: The element is written with the supplied prefix and name and the previously associated namespace URI is used. Since the namespace matching the supplied |
<table>
<thead>
<tr>
<th>Method call example and resulting tag</th>
<th>Use case explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>START-ELEMENT</strong>(&quot;prefix:name&quot;, &quot;namespaceUri&quot;)&lt;prefix:name xmlns:prefix=&quot;namespaceUri&quot;&gt;</td>
<td>prefix and URI pair has already been declared, it will not be redeclared.</td>
</tr>
<tr>
<td><strong>Case:</strong> You supply an element <em>name</em> with a namespace <em>prefix</em> and a <em>namespaceUri</em>. Either one or both of the namespace prefix and the namespace URI has already been used in a previous declaration, but the pair have not been declared together. <strong>Result:</strong> The element is written with the supplied <em>prefix</em> and <em>name</em> and <em>namespaceUri</em> declared. This amounts to the declaration of a new namespace.</td>
<td></td>
</tr>
<tr>
<td><strong>START-ELEMENT</strong>(&quot;prefix:name&quot;, &quot;namespaceUri&quot;)error</td>
<td><strong>Case:</strong> You supply an element <em>name</em> with a namespace <em>prefix</em> and a <em>namespaceUri</em>. The namespace prefix matches the prefix used in the element, but the URIs do not match. The STRICT attribute is set to <strong>TRUE</strong>. <strong>Result:</strong> The method call generates an error message. Within an element, namespaces are like attributes and must be unique. <strong>Exception:</strong> In the case where the element's URI is the empty string, the tag will be written with the supplied <em>namespaceUri</em>.</td>
</tr>
<tr>
<td><strong>Case:</strong> You supply an element <em>name</em> with a namespace <em>prefix</em> and a <em>namespaceUri</em>. The namespace prefix matches the prefix used in the element, but the URIs do not match. The STRICT attribute is set to <strong>FALSE</strong>. <strong>Result:</strong> The tag is written, but it is not valid XML.</td>
<td></td>
</tr>
<tr>
<td><strong>DECLARE-NAMESPACE</strong>(&quot;namespaceUri&quot;, &quot;prefix&quot;)&lt;qname xmlns:prefix=&quot;namespaceUri&quot;&gt;</td>
<td><strong>Case:</strong> You supply a new prefix and new <em>namespaceUri</em> that do not overlap with those declared in the element. <strong>Result:</strong> Creates the expected namespace without error.</td>
</tr>
<tr>
<td><strong>Declare-NAMESPACE</strong>(&quot;namespaceUri&quot;, &quot;prefix&quot;)&lt;prefix:name xmlns:prefix=&quot;namespaceUri&quot;&gt;</td>
<td><strong>Case:</strong> The provided <em>namespaceUri</em> matches the namespace URI declared in the element and <em>prefix</em> matches the namespace prefix declared in the element. <strong>Result:</strong> Re-declaring an implicitly created namespace (the namespace declared in the element) does no harm and does not generate an error message.</td>
</tr>
<tr>
<td><strong>DECLARE-NAMESPACE</strong>(&quot;namespaceUri&quot;, &quot;prefix&quot;)&lt;prefix:name xmlns:prefix=&quot;namespaceUri&quot;&gt;</td>
<td><strong>Case:</strong> <em>namespaceUri</em> is provided and the namespace URI declared in the element is the empty string (&quot;&quot;), and the provided <em>prefix</em> matches the namespace prefix declared in the element. <strong>Result:</strong> The namespace is created without error. In other words, if you are going to explicitly create your...</td>
</tr>
<tr>
<td>Method call example and resulting tag</td>
<td>Use case explanation</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;namespaceUri&quot;, &quot;prefix&quot;)&lt;br&gt;error</td>
<td><strong>Case:</strong> <code>namespaceUri</code> is provided but it does not match the namespace URI declared in the element and the supplied <code>prefix</code> does match the namespace prefix declared in the element. The <code>STRICT</code> attribute is set to <code>TRUE</code>. <strong>Result:</strong> The method call generates an error. Within an element, namespaces are like attributes and must be unique.</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;namespaceUri&quot;, &quot;&quot;).&lt;qname xmlns=&quot;namespaceUri&quot;&gt;</td>
<td><strong>Case:</strong> You provide only a <code>namespaceUri</code> and an empty string (&quot;&quot;) as the namespace prefix. <strong>Result:</strong> You create the default namespace.</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;namespaceUri&quot;, &quot;&quot;).error</td>
<td><strong>Case:</strong> The element has declared a default namespace with a different namespace URI than the one you provide. <strong>Result:</strong> The method call generates an error. Within an element, namespaces are like attributes and must be unique.</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;namespaceUri&quot;, &quot;&quot;).&lt;name xmlns=&quot;namespaceUri&quot; xmlns=&quot;namespaceUri&quot;&gt;</td>
<td><strong>Case:</strong> The element has declared a default namespace with a different namespace URI than the one you provide. The <code>STRICT</code> attribute is set to <code>FALSE</code>. <strong>Result:</strong> The tag is written, but it is not valid XML.</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;&quot;, &quot;prefix&quot;)&lt;br&gt;error</td>
<td><strong>Case:</strong> You provide an empty string (&quot;&quot;) as the namespace URI and supply the <code>prefix</code>. The <code>STRICT</code> attribute is set to <code>TRUE</code>. <strong>Result:</strong> The method call generates an error. Only the default namespace can be declared as the empty string (&quot;&quot;).</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE(&quot;&quot;, &quot;prefix&quot;)&lt;br&gt;&lt;qname xmlns:prefix=&quot;&quot;&quot;&gt;</td>
<td><strong>Case:</strong> You provide an empty string (&quot;&quot;) as the namespace URI and supply a <code>prefix</code> that matches the namespace prefix declared in the element. The <code>STRICT</code> attribute is set to <code>FALSE</code>.</td>
</tr>
</tbody>
</table>
### Handling errors

The `WRITE-STATUS` attribute is your view into the error status of a SAX-writer object. The value of the attribute is interpreted differently depending upon the value of the `STRICT` attribute. When `STRICT` is set to `TRUE`, any method call that would result in invalid XML sets the `WRITE-STATUS` attribute to `SAX-WRITE-ERROR`. When `STRICT` is `FALSE`, only errors that prevent the write from starting or raise exceptions set the `WRITE-STATUS` attribute to `SAX-WRITE-ERROR`. Essentially, `STRICT = TRUE` is more sensitive and useful for development and troubleshooting mode.

`STRICT = FALSE` will only stop a document from being written if the error prevents the write from starting or continuing and may be a better choice for production mode.

If the SAX-writer does have a status of `SAX-WRITE-ERROR`, only the methods which reinitialize the object can be called: `SET-OUTPUT-DESTINATION` and `RESET`. The object attributes will still be readable and writable.

<table>
<thead>
<tr>
<th>Method call example and resulting tag</th>
<th>Use case explanation</th>
</tr>
</thead>
</table>
| DECLARE-NAMESPACE("", "prefix"). <prefix:name xmlns:prefix="namespaeUri" xmlns:prefix=""> | **Result:** This can be useful when not running in `STRICT` mode to reset the element namespace prefix to the empty string (""").  
**Case:** You provide an empty string ("") as the namespace URI and supply a `prefix`. The `STRICT` attribute is set to `FALSE`.  
**Result:** The tag is written, but it is not valid XML. |
| DECLARE-NAMESPACE("", ""). <gname xmlns=""> | **Case:** You provide an empty string ("") as the namespace URI and as the namespace prefix.  
**Result:** You create the default namespace as empty. |
| DECLARE-NAMESPACE("", ""). error | **Case:** The element has declared a specific default namespace and you provide `namespaceUri`, which is the empty string (""). The `STRICT` attribute is set to `TRUE`.  
**Result:** The method call generates an error. Within an element, namespaces are like attributes and must be unique. |
| DECLARE-NAMESPACE("", "). <name xmlns="namespaceUri" xmlns=""> | **Case:** The element has declared a specific default namespace and you provide `namespaceUri`, which is the empty string (""). The `STRICT` attribute is set to `FALSE`.  
**Result:** The tag is written, but it is not valid XML. |
How errors are handled when STRICT is FALSE

If STRICT is set to FALSE, then the SAX-writer will ignore the structure and content of the document, and not generate any messages. The method will succeed, the WRITE-STATUS attribute will be properly set by the method, and the document stream remains open.

How errors are handled when STRICT is TRUE

If STRICT is set to TRUE, then the writer will validate the structure of the XML document as it writes it out. With this attribute, the SAX-writer attempts to keep the developer from creating invalid XML documents. It does this by ensuring that the methods are called in correct order.

If SAX-writer finds an incorrect method call:

- The method fails and returns FALSE
- It generates an error message
- The SAX-writer changes the WRITE-STATUS to SAX-WRITE-ERROR
- The SAX-writer closes the document stream

The following table describes the conditions that generate an error message when STRICT is set to TRUE.

**Table 17: Common error messages**

<table>
<thead>
<tr>
<th>Methods effected</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>If a particular method is called when the WRITE-STATUS attribute status value is an illegal value for the method, the method fails and generates an error message. See Errors raised by invalid method calls during SAX-writer states on page 113 for more information.</td>
</tr>
<tr>
<td>END-DOCUMENT( )</td>
<td>If the root tag is not closed and END-DOCUMENT is called, the method fails and generates the following message: END-DOCUMENT attempted to close the document before the root tag was closed. You must close the root tag with END-ELEMENT before closing the document.</td>
</tr>
<tr>
<td>START-ELEMENT( )</td>
<td>If FRAGMENT is FALSE, then there may be only one root node, that is, the document-level node. If a call to one of these methods would result in a second document-level node then the following error message is generated: &lt;method&gt; attempted to create a second document level node in the document.</td>
</tr>
<tr>
<td>WRITE-EMPTY-ELEMENT( )</td>
<td></td>
</tr>
<tr>
<td>WRITE-DATA-ELEMENT( )</td>
<td></td>
</tr>
</tbody>
</table>
## Methods effected

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>You must limit each XML document to one document-level node.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>END-ELEMENT( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each START-ELEMENT method call, you must have a corresponding and matching END-ELEMENT method call. Start and end all nodes to properly form parent and child relationships. All nodes must be started and ended (excluding empty nodes). If END-ELEMENT is called with the incorrect name and prefix, then it fails and generates the message:</td>
</tr>
<tr>
<td>END-ELEMENT attempted to close the tag &lt;qname&gt; when the current tag to close was &lt;qname&gt;.</td>
</tr>
<tr>
<td>Do not interweave different start and end tags.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WRITE-CHARACTERS( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only call WRITE-CHARACTERS from within the document. That is, call it from within the root node. If it is called at the document level, it will fail and generate the following error:</td>
</tr>
<tr>
<td>WRITE-CHARACTERS attempted to write character data at the document level.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INSERT-ATTRIBUTE( ) DECLARE-NAMESPACE( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute names must be unique. If a call to either of these methods results in a repeated instance of a name, then the methods fail and the following error message will be generated:</td>
</tr>
<tr>
<td>&lt;method&gt; attempted to create a second instance of the name &lt;name&gt;.</td>
</tr>
<tr>
<td>Also, these methods can only be called when the WRITE-STATUS is SAX-WRITE-TAG or they fail and generate an error message as described in Errors raised by invalid method calls during SAX-writer states on page 113. For example:</td>
</tr>
</tbody>
</table>

```xml
hSAXWriter:STRICT = TRUE.
hSAXWriter:START-DOCUMENT( ).
hSAXWriter:START-ELEMENT("root","").
hSAXWriter:START-ELEMENT("name","").
hSAXWriter:WRITE-CHARACTER("Fred Smith", 16).
hSAXWriter:END-ELEMENT("root","").
/*method fails */
hSAXWriter:END-DOCUMENT( ).
```

This code fragment would generate an error because root was closed before name.

## Errors raised by invalid method calls during SAX-writer states

Certain methods may only be called when the writer is in certain states. For example, most of the writing methods, like START-ELEMENT, can only be called while WRITE-STATUS is neither SAX-WRITE-IDLE nor SAX-WRITE-COMPLETE.

The following table describes the invalid method calls WRITE-STATUS vs. methods and attributes.
Table 18: Valid method calls by WRITE-STATUS value

<table>
<thead>
<tr>
<th>Method</th>
<th>Invalid for these WRITE-STATUS values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET-OUTPUT-DESTINATION</td>
<td>SAX-WRITE-BEGIN, SAX-WRITE-TAG,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>START-DOCUMENT</td>
<td>SAX-WRITE-BEGIN, SAX-WRITE-TAG,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>END-DOCUMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-CHARACTERS</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-EMPTY-ELEMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-DATA-ELEMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-FRAGMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-PROCESSING-INSTRUCTION</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-COMMENT</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-BEGIN,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>INSERT-ATTRIBUTE</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-BEGIN,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>DECLARE-NAMESPACE</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-BEGIN,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-EXTERNAL-DTD</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-BEGIN,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-CONTENT, SAX-WRITE-ELEMENT,</td>
</tr>
<tr>
<td></td>
<td>SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
</tbody>
</table>
### Example of changing values in the WRITER-STATUS attribute

Here is a code snippet (just the writer methods) that demonstrates the way the `WRITER-STATUS` changes:

```
/* hSAXWriter:WRITE-STATUS */
/* SAX-WRITE-IDLE */
/* SAX-WRITE-IDLE */
hSAXWriter:START-DOCUMENT( ). /* SAX-WRITE-BEGIN */
hSAXWriter:WRITE-COMMENT("comment"). /* SAX-WRITE-BEGIN */
hSAXWriter:START-ELEMENT("root", ""). /* SAX-WRITE-TAG */
hSAXWriter:START-ELEMENT("person", ""). /* SAX-WRITE-TAG */
hSAXWriter:START-ELEMENT("name", ""). /* SAX-WRITE-TAG */
hSAXWriter:WRITE-CHARACTERS("John"). /* SAX-WRITE-CONTENT */
hSAXWriter:END-ELEMENT("name", ""). /* SAX-WRITE-ELEMENT */
hSAXWriter:WRITE-EMPTY-ELEMENT("status", ""). /* SAX-WRITE-TAG */
hSAXWriter:INSERT-ATTRIBUTE("office", "1073"). /* SAX-WRITE-TAG */
hSAXWriter:END-ELEMENT("person", ""). /* SAX-WRITE-ELEMENT */
hSAXWriter:END-ELEMENT("root", ""). /* SAX-WRITE-ELEMENT */
hSAXWriter:END-DOCUMENT( ). /* SAX-WRITE-COMPLETE */
```

If `STRICT` is set to `FALSE` then calling a method while in the wrong state does not affect the XML document or the status of the writer. The method does not fail, the writer writes out the data, and the `WRITER-STATUS` is set accordingly. `WRITER-STATUS` is only set to `SAX-WRITE-ERROR` if there is a problem with the destination or `proxm1` library at `START-DOCUMENT`.

If `STRICT` is set to `TRUE` then each of these invalid cases returns `FALSE` and an error message is generated:

```
<method_name> invalid while WRITE-STATUS is <state_name>.
```

In this case, the stream is closed and the `WRITER-STATUS` is change to `SAX-WRITE-ERROR`. 

---

**Handling errors**

<table>
<thead>
<tr>
<th>Method</th>
<th>Invalid for these WRITE-STATUS values</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE-ENTITY-REF</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>WRITE-CDATA</td>
<td>SAX-WRITE-IDLE, SAX-WRITE-COMPLETE, SAX-WRITE-ERROR</td>
</tr>
<tr>
<td>RESET</td>
<td>None</td>
</tr>
</tbody>
</table>
If there is a problem with the output destination or the proxml library, the WRITE-STATUS is set to SAX-WRITE-ERROR regardless of the setting of STRICT. One of the following error messages is generated:

Unable to write to output destination.

Or...

The proxml.dll or libproxml.so was missing or incomplete or XML could not be initialized.

Handling codepage conversions

The SAX-writer object outputs the XML document in the codepage specified by the ENCODING attribute (UTF-8 by default). However, the character input to the document might use several codepages, as described:

• A CHARACTER variable uses the codepage set by the ABL parameter -cpinternal
• A LONGCHAR variable uses the codepage individually set for it

The SAX-writer handles these issues automatically by converting all character data to the UTF-16 codepage during processing before converting the XML output to the correct specified codepage.
Reading and Writing XML Data from Temp-Tables and ProDataSets

The AVM can serialize the data from ABL temp-tables and ProDataSets to an XML document and serialize their definitions to XML Schema documents. Similarly, you can read or read and load XML data and schema into a temp-table or ProDataSet. (Note that "AVM" refers to the ABL Virtual Machine (AVM), which is the ABL runtime engine.) To locate the code samples used in this chapter, see Example procedures on page 17. The samples use the Sports2000 database.

For details, see the following topics:

- Introduction
- Methods and attributes
- Reading XML Schema into a temp-table, temp-table buffer, or ProDataSet
- Reading XML into a temp-table, temp-table buffer, or ProDataSet
- Writing XML Schema from a temp-table, temp-table buffer, or ProDataSet
- Writing XML from a temp-table, temp-table buffers, or a ProDataSet
- Sample ProDataSet to XML round-trip
- Using XML Schema
Introduction

The XML features of temp-tables and ProDataSets allow you to take advantage of their rich relational features while providing a standards-based method for sharing data and schema with remote clients. These XML features include the following:

• Read XML data, XML Schema, or both to populate an empty temp-table or ProDataSet.

• Read XML data, XML Schema, or both into temp-tables and ProDataSets that already contain data and schema. For ProDataSets, the XML document can also contain before-image information.

• Write XML data, XML Schema, or both from a temp-table or ProDataSet to XML documents. XML Schema is written using the XML Schema Definition language (XSD). If ProDataSet temp-table buffers have before-image information, that information can also be written.

• Perform round-trip XML write/reads. ABL temp-table and ProDataSets are feature-rich and their definitions cannot be completely represented by basic XSD. ABL adds ABL-specific attributes to the XML Schema it writes so that temp-tables and ProDataSets can be fully restored when the XML Schemas are read back into the temp-table or ProDataSet objects.

The XML features are available as attributes and methods on:

• Temp-table objects

• Temp-table buffer objects

• ProDataSet objects

Note: The AVM can also serialize data from these ABL data objects to and from JSON (JavaScript Object Notation). For more information on this feature, see OpenEdge Development: Working with JSON.

During an XML read, only the XML elements and attributes relevant to ABL temp-tables and ProDataSets are used. If the source XML contains extra information and you subsequently write the XML document from ABL, it might differ from the original XML in the following ways:

• Original formatting (white space, line indents, line breaks) is ignored

• Elements or attributes that are not relevant for ABL are ignored

Note: If preserving the original XML data is important to your application, you may want to use the ABL DOM or SAX interfaces for more control over your read and write operations.

Use cases

The XML read and write features are robust and versatile. The following examples demonstrate common problems that can be solved with the features:

• Provide interoperability between OpenEdge and another XML-enabled platform or application. For example, Crystal Reports can accept XML data and XML Schema when building and displaying reports.

• Use XML data and XML Schema as a persistent storage mechanism between ABL sessions.
• Provide XML Schema from ABL for use in third-party tools. For example, Microsoft Visual Studio .NET has a utility that creates an ADO.NET dataset from an .xsd file. By writing a ProDataSet definition to an .xsd file, you can then use the Microsoft utility to quickly model a ProDataSet as an ADO.NET DataSet class.

• You might have existing ABL code that performs XML reads and writes with the DOM or SAX interface that you can simplify and replace with the temp-table and ProDataSet object XML read and write features.

**Other XML features**

You do not need to know XML to successfully use the XML features described in this chapter. However, you might be interested in ABL features that support working directly with XML. For example:

• For information on reading and writing XML documents with ABL using the Document Object Model (DOM) interface, see Reading and Writing XML with the Document Object Model (DOM) on page 31

• For information on reading and writing XML documents with ABL using the Simple API for XML (SAX) interface, see Reading XML Documents with the Simple API for XML (SAX) on page 55 and Writing XML Documents with the Simple API for XML (SAX) on page 95

**Methods and attributes**

To use the XML read and write features described in this chapter, you use the handle to a temp-table, temp-table buffer, or ProDataSet to access methods and attributes of the object. The following table describes the available methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-XMLSCHEMA( )</td>
<td>Creates a dynamic temp-table or ProDataSet from the specified XML Schema. Or, it can verify an existing ABL definition against the XML Schema.</td>
</tr>
<tr>
<td>READ-XML( )</td>
<td>Loads data into a static or dynamic temp-table or ProDataSet from the specified XML. Optionally, it can create the temp-table or ProDataSet definition or verify the definition against the specified XML Schema.</td>
</tr>
<tr>
<td>WRITE-XMLSCHEMA( )</td>
<td>Writes the schema of the specified static or dynamic temp-table or ProDataSet as XSD. Options allow you to control the formatting, encoding, and level of definition detail.</td>
</tr>
<tr>
<td>WRITE-XML( )</td>
<td>Writes the data and, optionally, the definition of the specified static or dynamic temp-table or ProDataSet as XML. Options allow you to control the formatting, encoding, and level of definition detail.</td>
</tr>
</tbody>
</table>

The XML related attributes are described in the following table.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data type</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREIGN-KEY-HIDDEN¹</td>
<td>LOGICAL</td>
<td>Data-relation</td>
<td>Specifies whether the WRITE-XML( ) method should hide foreign key fields in the child records of a nested data-relation in a ProDataset.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAMESPACE-URI²</td>
<td>CHARACTER</td>
<td>temp-table temp-table</td>
<td>Specifies the namespace, a Uniform Resource Indicator (URI) used to uniquely identify attribute and element names and prevent collisions. Interacts with the NAMESPACE-PREFIX attribute.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td>bufferProDataSet</td>
<td></td>
</tr>
<tr>
<td>NAMESPACE-PREFIX²</td>
<td>CHARACTER</td>
<td>temp-table temp-table</td>
<td>Specifies the prefix which identifies elements belonging to the namespace specified by the NAMESPACE-URI attribute.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td>bufferProDataSet</td>
<td></td>
</tr>
<tr>
<td>NESTED</td>
<td>LOGICAL</td>
<td>Data-relation</td>
<td>Specifies whether the AVM embeds child rows within a parent row in the XML. This affects both the data and schema.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SERIALIZE-HIDDEN³</td>
<td>LOGICAL</td>
<td>Buffer-fieldProDataSet</td>
<td>Indicates whether a temp-table field and its value or ProDataSet name is written when the object is serialized, for example into JSON or XML. Interacts with the XML-NODE-TYPE attribute.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SERIALIZE-NAME⁴</td>
<td>CHARACTER</td>
<td>ProDataSettemp-tabletemp-table</td>
<td>Optionally specifies the name of a ProDataSet, a temp-table, a temp-table buffer, or a temp-table buffer-field object as it should appear when serialized, for example into JSON or XML. Interacts with the XML-NODE-NAME attribute.</td>
</tr>
<tr>
<td>(Readable and writeable)</td>
<td></td>
<td>buffertemp-table buffer field</td>
<td></td>
</tr>
</tbody>
</table>

¹ See Minimizing XML document size on page 159 for more information.
² For more information, see NAMESPACE-URI and NAMESPACE-PREFIX on page 121.
³ For more information, see XML-NODE-TYPE and SERIALIZE-HIDDEN on page 123.
### Attribute interactions

Several attributes used to serialize ABL data to and from XML interact. The following sections discuss how these pairs of attributes interact.

**NAMESPACE-URI and NAMESPACE-PREFIX**

**NAMESPACE-URI** and **NAMESPACE-PREFIX** interact in a **WRITE-XML** method call in the following ways:

---

4 For more information, see XML-NODE-NAME and SERIALIZE-NAME on page 123.
If `NAMESPACE-URI` is specified and `NAMESPACE-PREFIX` is not, the WRITE-XML method writes the XML document using a default namespace (xmlns="namespaceUri"). For example:

```sql
DEFINE TEMP-TABLE ttCustomer NO-UNDO
   NAMESPACE-URI "http://myCompanyServer.com/myNamespace"
   FIELD CustNum LIKE Customer.CustNum
   FIELD Name LIKE Customer.Name FORMAT "x(30)".
```

This example produces XML like the following:

```xml
<?xml version="1.0"?>
<ttCustomer
   xmlns="http://myCompanyServer.com/myNamespace"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ttCustRow
      xmlns="http://myCompanyServer.com/myNamespace">
      <CustNum>3</CustNum>
      <Name>Hoops</Name>
   </ttCustRow>
</ttCustomer>
```

If both `NAMESPACE-URI` and `NAMESPACE-PREFIX` are specified, all elements in the XML document will start with the `NAMESPACE-PREFIX`. For example:

```sql
DEFINE TEMP-TABLE ttCustomer NO-UNDO
   NAMESPACE-URI "http://myCompanyServer.com/myNamespace"
   NAMESPACE-PREFIX "myPrefix"
   FIELD CustNum LIKE Customer.CustNum
   FIELD Name LIKE Customer.Name FORMAT "x(30)".
```

This example produces XML like the following:

```xml
<?xml version="1.0"?>
<myPrefix:ttCustomer
   xmlns:myPrefix="http://myCompanyServer.com/myNamespace"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <myPrefix:ttCustRow>
      <myPrefix:CustNum>3</myPrefix:CustNum>
      <myPrefix:Name>Hoops</myPrefix:Name>
   </myPrefix:ttCustRow>
</myPrefix:ttCustomer>
```

If `NAMESPACE-PREFIX` is specified and `NAMESPACE-URI` is not, the WRITE-XML method will ignore the prefix and write the document with no namespace information. The WRITE-XML method behaves as if neither `NAMESPACE-URI` nor `NAMESPACE-PREFIX` is specified. For example:

```sql
DEFINE TEMP-TABLE ttCustomer NO-UNDO
   NAMESPACE-PREFIX "myPrefix"
   FIELD CustNum LIKE Customer.CustNum
   FIELD Name LIKE Customer.Name FORMAT "x(30)".
```

This example produces XML like the following:

```xml
<?xml version="1.0"?>
<ttCustomer
   xmlns="http://myCompanyServer.com/myNamespace"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ttCustRow>
      <CustNum>3</CustNum>
      <Name>Hoops</Name>
   </ttCustRow>
</ttCustomer>
```
This example produces XML like the following:

```xml
<?xml version="1.0"?>
<ttCustomer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ttCustRow>
    <CustNum>3</CustNum>
    <Name>Hoops</Name>
  </ttCustRow>
</ttCustomer>
```

**XML-NODE-NAME and SERIALIZE-NAME**

The `XML-NODE-NAME` attribute's purpose overlaps with the `SERIALIZE-NAME` attribute. Because of this overlap, the attributes interact as follows:

- The `READ-XML()` and `WRITE-XML()` methods always use the `XML-NODE-NAME` attribute value. If you set a value for the `XML-NODE-NAME` attribute, it keeps that value regardless of how you set the `SERIALIZE-NAME` attribute.
- If you do not set the `XML-NODE-NAME` attribute and set the `SERIALIZE-NAME` attribute, the AVM sets `XML-NODE-NAME` equal to `SERIALIZE-NAME`.
- If you do not set either attribute, the AVM sets both to the ABL object name.

**XML-NODE-TYPE and SERIALIZE-HIDDEN**

The `XML-NODE-TYPE` attribute's purpose overlaps with the `SERIALIZE-HIDDEN` attribute. Because of this overlap, the attributes interact as follows:

- The `WRITE-XML()` method always uses the `XML-NODE-TYPE` attribute value. If you set a value for the `XML-NODE-TYPE` attribute, it keeps that value regardless of how you set the `SERIALIZE-HIDDEN` attribute.
- If you do not set the `XML-NODE-TYPE` attribute and set the `SERIALIZE-HIDDEN` attribute to `TRUE`, the AVM sets `XML-NODE-TYPE` to "HIDDEN".
- If you do not set the `XML-NODE-TYPE` attribute and set the `SERIALIZE-HIDDEN` attribute to `FALSE`, the AVM sets `XML-NODE-TYPE` to "ELEMENT".
- If you do not set either attribute, the AVM sets `XML-NODE-TYPE` to "ELEMENT" and sets `SERIALIZE-HIDDEN` to `FALSE`.

**ABL-specific attributes in XML Schema**

ABL provides a set of XML Schema attributes to specify ABL properties that are extensions to basic XML Schema. These properties enable you to completely describe temp-table and ProDataSet definitions with XML Schema. A non-OpenEdge application that works directly with an OpenEdge-generated XML Schema will ignore these extensions. See Using XML Schema on page 164 for more information.

These attributes allow temp-table and ProDataSet definitions to be fully restored when XML Schemas are read back into ABL. You can control whether some of the ABL XML Schema extensions are present in the XSD with options on the `WRITE-XML()` and `WRITE-XMLSCHEMA()` methods.
Reading XML Schema into a temp-table, temp-table buffer, or ProDataSet

Reading XML Schema using READ-XMLSCHEMA( ) has two distinct meanings, depending on your use case:

- If you are creating a dynamic object, then reading XML Schema is the process of creating the definition of the dynamic object from the specified XML Schema.
- If the object already has a definition, then reading XML Schema is the process of comparing the existing definition to the specified XML Schema. This comparison is known as verification. Since static objects cannot exist without a definition, the purpose of reading XML Schema into a static object is always verification. If a dynamic object already has its definition, then the purpose of reading XML Schema into it is also verification.

Verification allows you to test the compatibility of the object with the specified XML Schema document. If the object and XML Schema are not an exact or close match, the READ-XMLSCHEMA( ) method call fails and returns FALSE. The verification-mode parameter (described later) controls how close the match must be.

**Note:** The XML document must be an XML Schema written in the XML Schema Definition (XSD) language in the 2001 XML Schema namespace (http://www.w3.org/2001/XMLSchema). Non-OpenEdge applications might use XSD in many ways. The READ-XMLSCHEMA( ) method attempts to parse any XSD elements that form an obvious relational structure into ABL temp-tables and ProDataSets. However, the method might not be able to handle every XSD element generated by a non-OpenEdge source. The closer a particular XSD element conforms to what the WRITE-XMLSCHEMA( ) method creates, the more likely that the READ-XMLSCHEMA( ) method will succeed.

Here is the syntax for READ-XMLSCHEMA( ). The method returns TRUE or FALSE to indicate whether the operation was successful.

**Syntax**

```plaintext
READ-XMLSCHEMA ( source-type, { file | memptr | handle | longchar },
override-default-mapping [, field-type-mapping [, verify-schema-mode] ] )
```

**source-type**

A CHARACTER expression that specifies the source XML document type. Valid values are: "FILE", "MEMPTR", "HANDLE", and "LONGCHAR".

**file**

A CHARACTER expression that specifies the name of an XML Schema file. You can specify an absolute pathname, a relative pathname (based on the current working directory), or a URL pathname. Valid URL protocols include FILE and HTTP (the HTTPS protocol is not supported). The AVM verifies that the file exists and is accessible.
memptr

A MEMPTR variable that contains the XML Schema document text. The size of the MEMPTR variable must match the size of the XML document text.

handle

A WEB-CONTEXT system handle, X-document object handle, or X-noderef object handle.

longchar

A LONGCHAR variable that contains the XML Schema document text.

override-default-mapping

A LOGICAL expression where TRUE directs the AVM to override the default mapping between XML Schema string and binary data types and ABL data types when creating ABL temp-table definition from an XML Schema. The default value is FALSE.

The XML Schema string data type maps to the ABL CHARACTER data type by default, and the XML Schema base64Binary and hexBinary data types map to the ABL RAW data type by default. If you specify TRUE, the READ-XMLSCHEMA( ) method creates a temp-table schema with CLOB and BLOB fields instead of CHARACTER and RAW fields.

If you specify the Unknown value (?), the method uses the default value of FALSE.

Note: If the temp-table or ProDataSet has an ABL definition, this option is ignored.

field-type-mapping

An optional CHARACTER expression that evaluates to a comma-separated list of field name, data type pairs using the following syntax:

\[
\text{field-name}_1, \text{data-type}_1[, \text{field-name}_n, \text{data-type}_n ]...\]

This option allows you to specify the ABL data type for a specific field from the XML Schema. Generally, this option is only used to map fields from non-ABL generated schema. When reading and writing ABL-generated XML Schema, there is little need to override field mappings because of the ABL extensions to standard XML Schema. (The ABL extensions use an identifying prefix prodata.)

field-name

The name of the specified field. For a ProDataSet object, you must qualify the field name with the buffer name from the XML Schema. That is, buffer-name.field-name.

data-type

The target ABL data type for the specified field. The data type must be a valid ABL data type, and it must be compatible with the XML Schema type based on the ABL XML data type mapping rules. For example, any XML Schema type can be mapped to ABL CHAR or CLOB, but an XML Schema dateTime can be mapped only to ABL DATE, DATETIME or DATETIME-TZ.
If you specify the unknown value (?), the method uses the default data type.

**Note:** If the temp-table or ProDataSet has an ABL definition, this option is ignored.

For more information about the ABL XML data type mapping rules, see XML Schema and ABL Data Type Mappings on page 185.

**verify-schema-mode**

An optional CHARACTER expression that specifies the mode in which the READ-XMLSCHEMA( ) method verifies any XML Schema against an existing ABL definition. The expression must evaluate to "LOOSE" or "STRICT". The default value is "LOOSE".

**Note:** For a dynamic temp-table or ProDataSet temp-table buffer that does not have ABL definition (that is, the object is in the CLEAR state), this option is ignored.

The following table lists the READ-XMLSCHEMA( ) method schema verification modes.

**Table 21: READ-XMLSCHEMA( ) method verification mode**

<table>
<thead>
<tr>
<th>For this mode ...</th>
<th>And this object ...</th>
<th>The READ-XMLSCHEMA( ) method ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRICT</td>
<td>temp-table</td>
<td>Matches temp-table columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. The XML Schema must define a field for every temp-table field and cannot define any additional fields.</td>
</tr>
<tr>
<td></td>
<td>ProDataSet</td>
<td>Matches temp-tables and columns by name. There must be a temp-table defined in the XML Schema for each table of the ProDataSet. There can be no tables defined in the XML Schema that are not in the ProDataSet definition. There must be field defined in the XML Schema for each field in the temp-table, with the same data type and extent, and there can be no fields defined in the XML Schema that are not in the temp-table definition. There must also be a data relationship defined in the XML Schema for every data-relation in the ProDataSet, and there can be no data relationships defined in the XML Schema that are not defined in the ProDataSet. The field mapping between the parent and child buffers must match.</td>
</tr>
</tbody>
</table>
### Reading XML Schema into a temp-table, temp-table buffer, or ProDataSet

<table>
<thead>
<tr>
<th>For this mode . . .</th>
<th>And this object . . .</th>
<th>The READ-XMLSCHEMA( ) method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOSE</td>
<td>temp-table</td>
<td>Matches temp-table columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. The XML Schema might be a subset or superset of the temp-table definition. Any columns that are not present in both the XML Schema and the temp-table are ignored.</td>
</tr>
<tr>
<td>ProDataSet</td>
<td>ProDataSet</td>
<td>Matches temp-tables and columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. Data relationships are matched by parent buffer and child buffer names. For each match between a data relationship in the XML Schema that a data-relation in the ProDataSet, the field mapping must match for the parent and child buffers. The XML Schema may be a subset or superset of the ProDataSet definition. Any temp-tables, columns, or data-relations that are in the ProDataSet, but not in the XML Schema, are ignored. For a dynamic ProDataSet object, the method adds temp-tables and data-relations to the object when the temp-tables and data-relations are defined in the XML Schema, but are not members of the ProDataSet. Fields are not added to existing temp-tables. For a static ProDataSet object, any temp-tables or data-relations that are in the XML Schema, but not in the ProDataSet, are ignored.</td>
</tr>
</tbody>
</table>

If you specify the **Unknown value (?)**, the method uses the default value of LOOSE.

If the XML Schema verification fails, the method generates an error message indicating the XML Schema element that caused the failure and returns FALSE.

---

### Creating a dynamic temp-table with XML Schema

A dynamic temp-table needs its definition supplied before it can be put into a **PREPARED** state for reading data. Using `READ-XMLSCHEMA( )` on a temp-table with no definition creates the definition from the XML Schema. The method then places the temp-table in the **PREPARED** state. The code example that follows does several things:

- Creates a dynamic temp-table
- Reads in XML Schema
- Displays some information about the definition of the temp-table
The read and write XML methods have many parameters. To make the code samples in this chapter more readable, each parameter is represented by a descriptively named variable. All these variable definitions are stored in an include file, as shown:

```/* pi-tfx-parameterVarDefs.i */
/* Variables representing parameter values in the READ-XML( ),
READ-XMLSCHEMA( ), WRITE-XML( ), and WRITE-XMLSCHEMA( ) methods. */

DEFINE VARIABLE cSourceType AS CHARACTER NO-UNDO.
DEFINE VARIABLE cTargetType AS CHARACTER NO-UNDO.
DEFINE VARIABLE cFile AS CHARACTER NO-UNDO.
DEFINE VARIABLE cReadMode AS CHARACTER NO-UNDO.
DEFINE VARIABLE cSchemaLocation AS CHARACTER NO-UNDO.
DEFINE VARIABLE lOverrideDefaultMapping AS LOGICAL NO-UNDO.
DEFINE VARIABLE cFieldTypeMapping AS CHARACTER NO-UNDO.
DEFINE VARIABLE cVerifySchemaMode AS CHARACTER NO-UNDO.
DEFINE VARIABLE cEncoding AS CHARACTER NO-UNDO.
DEFINE VARIABLE lFormatted AS LOGICAL NO-UNDO.
DEFINE VARIABLE lWriteSchema AS LOGICAL NO-UNDO.
DEFINE VARIABLE lMinSchema AS LOGICAL NO-UNDO.
DEFINE VARIABLE lWriteBeforeImage AS LOGICAL NO-UNDO.
```

Here is the code sample:

```/* pi-tfx-read-1.p */
/* Provides XML Schema for a new dynamic temp-table. */

(pi-tfx-parameterVarDefs.i)

DEFINE VARIABLE httCust AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE TEMP-TABLE httCust.

ASSIGN
  cSourceType = "FILE"
  cFile = "ttCustomer.xsd"
  lOverrideDefaultMapping = ?
  cFieldTypeMapping = ?
  cVerifySchemaMode = ?.

DISPLAY "Is dynamic temp-table PREPARED? " httCust:PREPARED SKIP.
DISPLAY "Reading XML Schema..." SKIP.

lReturn = httCust:READ-XMLSCHEMA(cSourceType, cFile,
  lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).
IF lReturn THEN DO:
  DISPLAY "Is dynamic temp-table now PREPARED? " httCust:PREPARED SKIP.
  DISPLAY "How many columns in dynamic temp-table? "
  httCust:DEFAULT-BUFFER-HANDLE:NUM-FIELDS SKIP.
END.
```

The code displays the following:

```
Is dynamic temp-table PREPARED?   no
Reading XML Schema...
Is dynamic temp-table now PREPARED?  yes
How many columns in dynamic temp-table?  1
```
In this next version, a non-ABL generated XML Schema file will be used to create a new dynamic temp-table. The schema is very similar to the Feedback table of the Sports2000 database. The purpose of this example is to demonstrate mapping an XML Schema field to something more useful for your ABL application.

Here is the ttFeedback.xsd file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xsd:element name="ttFeedback">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ttFeedbackRow" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Contact" type="xsd:string"/>
              <xsd:element name="Company" type="xsd:string"/>
              <xsd:element name="EmailAddress" type="xsd:string"/>
              <xsd:element name="Phone" type="xsd:string"/>
              <xsd:element name="Fax" type="xsd:string"/>
              <xsd:element name="Comments" type="xsd:string"/>
              <xsd:element name="Department" type="xsd:string"/>
              <xsd:element name="Rating" type="xsd:integer"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Elements with the XML Schema type "string" are mapped to CHARACTER fields by default.

Here is the code that maps the Comments field to a CLOB:

```abl
/* pi-tfx-read-1b.p */
/* Provides XML Schema for a new dynamic temp-table. */
{pi-tfx-parameterVarDefs.i}

DEFINE VARIABLE httFeedback AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE TEMP-TABLE httFeedback.

ASSIGN
cSourceType = "FILE"
cFile = "ttFeedback.xsd"
lOverrideDefaultMapping = ?
cFieldTypeMapping = "Comments, CLOB"
cVerifySchemaMode = ?.

DISPLAY "Is dynamic temp-table PREPARED? " httFeedback:PREPARED SKIP.
DISPLAY "Reading XML Schema..." SKIP.

lReturn = httFeedback:READ-XMLSCHEMA(cSourceType, cFile, lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).
IF lReturn THEN DO:
  DISPLAY "Is dynamic temp-table now PREPARED? " httFeedback:PREPARED SKIP.
  DISPLAY "What is the data type of the Comments field? "
  httFeedback:DEFAULT-BUFFER-HANDLE:BUFFER-FIELD("Comments"):DATA-TYPE.
END.
```
Here is the code sample output:

```prolog
/* pi-tfx-read-2.p */
/* Verifies XML Schema against a static temp-table. */
{pi-tfx-parameterVarDefs.i)
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
DEFINE TEMP-TABLE ttCustomer NO-UNDO
FIELD CustNum LIKE Customer.CustNum
FIELD Name LIKE Customer.Name FORMAT "X(30)".
ASSIGN
cSourceType = "FILE"
cFile = "ttCustomer.xsd"
lOverrideDefaultMapping = ?
cFieldTypeMapping = ?
cVerifySchemaMode = "STRICT".
DISPLAY "Reading XML Schema..." SKIP.
lReturn = TEMP-TABLE ttCustomer:READ-XMLSCHEMA(cSourceType, cFile,
lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).
IF lReturn THEN
   DISPLAY "Success: Temp-table definition verified against XML Schema.".
END.
```

As written, this code sample fails with the following error messages:

Because the verification mode is "STRICT", the presence of more fields in the XML Schema than are actually defined for the temp-table causes the method to fail. The definition and the XML Schema are not an exact match. Change the verification mode to "LOOSE", and the code completes successfully because the temp-table definition is a partial match for the XML Schema.
Creating a dynamic ProDataSet with XML Schema

In this code sample, an XML Schema file that defines four temp-table buffers is read into a new dynamic ProDataSet:

```abl
/* pi-tfx-read-3.p */
/* Provides a new dynamic ProDataSet with XML Schema. */
{pi-tfx-parameterVarDefs.i}

DEFINE VARIABLE hDSET AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE DATASET hDSET.

ASSIGN
  cSourceType = "FILE"
  cFile = "Dept400.xsd"
  lOverrideDefaultMapping = ?
  cFieldTypeMapping = ?
  cVerifySchemaMode = ?.

DISPLAY "Reading XML Schema..." SKIP.

lReturn = hDSET:READ-XMLSCHEMA(cSourceType, cFile,
  lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).
      IF lReturn THEN
        DISPLAY "How many temp-table buffers in the dynamic ProDataSet?"
        hDSET:NUM-BUFFERS "."
      END.
```

The code displays the following information:

The `READ-XML()` method is used for loading data into static or dynamic temp-table, temp-table buffer, and ProDataSet objects from an XML document. Optionally, you can specify an XML Schema to create an ABL definition for a dynamic object or to verify the XML Schema against a static or dynamic object definition.

Here is the syntax for `READ-XML()`. The method returns `TRUE` or `FALSE` to indicate if the operation was successful.
Syntax

```
READ-XML ( source-type, {file | memptr | handle | longchar}, read-mode,
schema-location, override-default-mapping[, field-type-mapping[, verify-schema-mode]] )
```

**source-type**

A CHARACTER expression that specifies the source XML document type. Valid values are: "FILE", "MEMPTR", "HANDLE", and "LONGCHAR".

**file**

A CHARACTER expression that specifies the name of a file. You can specify an absolute pathname, a relative pathname (based on the current working directory), or a URL pathname. Valid URL protocols include FILE and HTTP (the HTTPS protocol is not supported). The AVM verifies that the file exists and is accessible.

**memptr**

A MEMPTR variable that contains the XML document text. The size of the MEMPTR variable must match the size of the XML document text.

**handle**

A WEB-CONTEXT system handle, X-document object handle, or X-noderef object handle.

**longchar**

A LONGCHAR variable that contains the XML document text.

**read-mode**

A CHARACTER expression that specifies the mode in which the READ-XML( ) method reads data from the XML document into a temp-table or ProDataSet member buffer. The expression must evaluate to "APPEND", "EMPTY", "MERGE", or "REPLACE". The default value is "MERGE".

The following table lists the READ-XML( ) method modes for reading data.

**Table 22: READ-XML( ) method read modes**

<table>
<thead>
<tr>
<th>When the mode is . . .</th>
<th>The READ-XML( ) method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>Reads data from the XML document into the ProDataSet or temp-table object by adding new records to the existing records, without performing any record comparisons. If a record from the XML document exists in the object (that is, it results in a duplicate unique key conflict), the method generates an error message and returns FALSE.</td>
</tr>
</tbody>
</table>
When the mode is . . . | The READ-XML() method . . .
---|---
EMPTY | Empties the contents of the ProDataSet or temp-table object before reading in data from the XML document.
MERGE | Reads data from the XML document into the ProDataSet or temp-table object by merging new records with existing records in the table. If a record from the XML document exists in the object (that is, it results in a duplicate unique key conflict), the method does not replace the existing record. If the record from the XML document does not exist in the object, the method creates a new record.
REPLACE | Reads data from the XML document into the ProDataSet or temp-table object by merging new records with existing records in the table. If the record from the XML document exists in the object (that is, it results in a duplicate unique key conflict), the method replaces the existing record with the new record. If the record from the XML document does not exist in the object, the method creates a new record.

schema-location

A CHARACTER expression that specifies the name of an external XML Schema file to use in creating or verifying the object’s definition when reading in the XML document. You can specify an absolute pathname, a relative pathname (based on the current working directory), or a URL pathname. Valid URL protocols include FILE and HTTP (the HTTPS protocol is not supported). The AVM verifies that the file exists and is accessible. When specified, the AVM ignores any XML Schema defined or referenced in the source XML document.

If you specify the empty string (""") or the Unknown value (?), the AVM creates or verifies the object’s definition using any XML Schema defined or referenced in the XML document.

override-default-mapping

A LOGICAL expression where TRUE directs the AVM to override the default mapping between XML Schema string and binary data types and ABL data types when creating a temp-table definition from an XML Schema. The default value is FALSE.

The XML Schema string data type maps to the ABL CHARACTER data type by default, and the XML Schema base64Binary and hexBinary data types map to the ABL RAW data type by default. If you specify TRUE, the READ-XML() method creates a temp-table schema with CLOB and BLOB fields instead of CHARACTER and RAW fields.

If you specify the Unknown value (?), the method uses the default value of FALSE.

Note: If the temp-table or ProDataSet has an ABL definition, this option is ignored.
**field-type-mapping**

An optional CHARACTER expression that evaluates to a comma-separated list of field name, data type pairs using the following syntax:

```
field-name-1, data-type-1[, field-name-n, data-type-n] . . .
```

This option allows you to specify the ABL data type for a specific field from the XML Schema. Generally, this option is only used to map fields from non-ABL generated schema. When reading and writing ABL-generated XML Schema, there is little need to override field mappings because of the ABL extensions to standard XML Schema.

**field-name**

The name of the specified field. For a ProDataSet object, you must qualify the field name with the buffer name from the XML Schema: that is, `buffer-name.field-name`.

**data-type**

The target ABL data type of the specified field. The data type must be a valid ABL data type, and it must be compatible with the XML Schema type based on the ABL XML data type mapping rules. For example, any XML Schema type can be mapped to ABL CHARACTER or CLOB, but an XML Schema dateTime can be mapped only to an ABL DATE, DATETIME or DATETIME-TZ.

**Note:** If the temp-table or ProDataSet has an ABL definition, this option is ignored.

If you specify the Unknown value (?), the method uses the default data type mapping. For more information about the ABL XML data type mapping rules, see XML Schema and ABL Data Type Mappings on page 185.

**verify-schema-mode**

An optional CHARACTER expression that specifies the mode in which the READ-XML( ) method verifies any XML Schema against existing ABL definitions. The expression must evaluate to "IGNORE", "LOOSE", or "STRICT". The default value is "LOOSE".

**Note:** For a dynamic temp-table or ProDataSet member buffer that does not have an ABL definition (that is, the object is in the CLEAR state), this option is ignored.

The following table lists the READ-XML( ) method modes for schema verification.

**Table 23: READ-XML( ) method schema verification modes**

<table>
<thead>
<tr>
<th>For this mode . . .</th>
<th>And this object . . .</th>
<th>The READ-XML( ) method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE</td>
<td>Any</td>
<td>Ignores any XML Schema specified in <code>schema-location</code>, or defined in the XML document.</td>
</tr>
</tbody>
</table>
### Reading XML into a temp-table, temp-table buffer, or ProDataSet

<table>
<thead>
<tr>
<th>For this mode . . .</th>
<th>And this object . . .</th>
<th>The READ-XML( ) method . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOSE</td>
<td>temp-table</td>
<td>Matches temp-table columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. The XML Schema may be a subset or superset of the temp-table schema. Any columns that are in the XML Schema but not in the temp-table are ignored. Any columns that are in the temp-table, but not in the XML Schema, are ignored.</td>
</tr>
<tr>
<td>ProDataSet</td>
<td></td>
<td>Matches temp-tables and columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. Data relationships are matched by parent buffer and child buffer names. For every data relationship in the XML Schema that matches a data-relation in the ProDataSet, the field mapping between the parent and child buffers must match. The XML Schema may be a subset or superset of the ProDataSet definition. Any temp-tables, columns, or data-relations that are in the ProDataSet, but not in the XML Schema, are ignored. For a dynamic ProDataSet object, the method adds temp-tables and data-relations to the object when the temp-tables and data-relations are defined in the XML Schema, but are not members of the ProDataSet. Fields are not added to existing temp-tables. For a static ProDataSet object, any temp-tables or data-relations that are in the XML Schema, but not in the ProDataSet, are ignored.</td>
</tr>
<tr>
<td>STRICT</td>
<td>temp-table</td>
<td>Matches temp-table columns by name. The data type and extent of the column in the XML Schema must match those for the matching column in the temp-table. Other field attributes in the XML Schema are ignored. The XML Schema must define a field for every temp-table field, and there can be no extra fields in the XML Schema that are not in the temp-table schema.</td>
</tr>
<tr>
<td>ProDataSet</td>
<td></td>
<td>Matches temp-tables and columns by name. There must be a temp-table defined in the XML Schema for each table of the ProDataSet. There can be no tables defined in the XML Schema that are not in the ProDataSet definition. There must be field defined in the XML Schema for each field in the temp-table, with the same data type and extent, and there can be no fields defined in the XML Schema that are not in the temp-table definition. There must also be a data relationship defined in the XML Schema for every data-relation in the ProDataSet, and there can be no data relationships defined in the XML Schema that are not defined in the ProDataSet. The field mapping between the parent and child buffers must match.</td>
</tr>
</tbody>
</table>

If you specify the *Unknown value* (?), the method uses the default value of LOOSE.
If the XML Schema verification fails, the method generates an error message indicating the XML Schema element that caused the failure and returns FALSE.

Schema locations

The READ-XML( ) method can find and use XML Schema from the following sources, in the order defined below:

1. The XML Schema specified with the schema-location option of the READ-XML( ) method. If this option is used, XML Schema references embedded in XML data are ignored.

2. A <schema> child element of the root element of the XML data document. Provided the method does not specify a schema-location, then any one or combination of this and the remaining list item are used.

3. An xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute on an instance data in the XML data document.

4. If no schema is present, the READ-XML( ) method will attempt to infer schema.

Reading XML data into temp-tables

XML information read into a temp-table or temp-table buffer includes:

- Schema for defining temp-table fields and indexes
- Data for the temp-table rows

This code example takes a temp-table with field definitions from the Customer table and reads an XML data file with three Customer records in it. The following data file shows the XML representation of the first record in the file:

```xml
<?xml version="1.0"?>
<ttCustomer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ttCustRow>
    <CustNum>1</CustNum>
    <Country>USA</Country>
    <Name>Lift Tours</Name>
    <Address>276 North Drive</Address>
    <City>Burlington</City>
    <State>MA</State>
    <PostalCode>01730</PostalCode>
    <Contact>Gloria Shepley</Contact>
    <Phone>(617) 450-0086</Phone>
    <SalesRep>HXM</SalesRep>
    <CreditLimit>66700.0</CreditLimit>
    <Balance>903.64</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
    </ttCustRow>
  ...
</ttCustomer>
```
An include file sets up the static temp-table definition:

```c
/* pi-tfx-ttSetup-4.i */
/* Definition of a static temp-table. */

DEFINE TEMP-TABLE ttCustomer NO-UNDO
FIELD CustNum LIKE Customer.CustNum
FIELD Country LIKE Customer.Country
FIELD Name LIKE Customer.Name
FIELD Address LIKE Customer.Address
FIELD Address2 LIKE Customer.Address2
FIELD City LIKE Customer.City
FIELD State LIKE Customer.State
FIELD PostalCode LIKE Customer.PostalCode
FIELD Contact LIKE Customer.Contact
FIELD Phone LIKE Customer.Phone
FIELD SalesRep LIKE Customer.SalesRep
FIELD CreditLimit LIKE Customer.CreditLimit
FIELD Balance LIKE Customer.Balance
FIELD Terms LIKE Customer.Terms
FIELD Discount LIKE Customer.Discount
FIELD Comments LIKE Customer.Comments
FIELD Fax LIKE Customer.Fax
FIELD EmailAddress LIKE Customer.EmailAddress

INDEX CountryPost Country PostalCode
INDEX Comments IS WORD-INDEX Comments
INDEX CustNum IS UNIQUE PRIMARY CustNum
INDEX NAME Name
INDEX SalesRep SalesRep.
```

Here is the code sample:

```c
/* pi-tfx-read-4.p */
/* Populates an empty static temp-table with records from an XML file. */
{pi-tfx-parameterVarDefs.i}
{pi-tfx-ttSetup-4.i}

DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

ASSIGN
   cSourceType = "FILE"
   cFile = "ttCustomer.xml"
   cReadMode = "EMPTY"
   cSchemaLocation = ?
   lOverrideDefaultMapping = ?
   cFieldTypeMapping = ?
   cVerifySchemaMode = ?.

lReturn = TEMP-TABLE ttCustomer:READ-XML(cSourceType, cFile, cReadMode, cSchemaLocation, lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).

IF lReturn THEN
   FOR EACH ttCustomer NO-LOCK:
      DISPLAY CustNum Name FORMAT "X(30)".
   END.
```

The code specifies the XML source file and displays a list of customer numbers and names after the read completes. Note the required TEMP-TABLE keyword in the method call.
Adding XML data to a populated temp-table

If you need to merge data in an ABL temp-table with XML data, you can control the way in which the AVM handles duplicate unique key conflicts with the `read-mode` parameter. In this example, a new temp-table is set up with definition from the `Customer` table and two new records. This set-up code is stored in an include file. Note that the first record uses a customer number that conflicts with one in the XML document. The second record has no conflict with the XML document, as shown:

```abl
/* pi-tfx-ttSetup-5.i */
/* Creates a new static temp-table and creates two new records for it. */
DEFINE TEMP-TABLE ttCustomer NO-UNDO
/* Field and index definitions. */
CREATE ttCustomer.
ASSIGN
  ttCustomer.Name = "Extreme XML Sports"
  ttCustomer.CustNum = 1
  ttCustomer.Balance = 111.11
  ttCustomer.Country = "USA"
  ttCustomer.Comments = "Enthusiastic new customer!".
CREATE ttCustomer.
ASSIGN
  ttCustomer.Name = "Low-Impact XML Sports"
  ttCustomer.CustNum = 4
  ttCustomer.Balance = 444.44
  ttCustomer.Country = "USA"
  ttCustomer.Comments = "Mellow new customer.".
```

The following is an abbreviated copy of the `ttCustomer.xml` file:

```xml
<?xml version="1.0"?>
<ttCustomer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ttCustRow>
    <CustNum>1</CustNum>
    <Country>USA</Country>
    <Name>Lift Tours</Name>
    ...
  </ttCustRow>
  <ttCustRow>
    <CustNum>2</CustNum>
    <Country>Finland</Country>
    <Name>Urpon Frisbee</Name>
    ...
  </ttCustRow>
  <ttCustRow>
    <CustNum>3</CustNum>
    <Country>USA</Country>
    <Name>Hoops</Name>
    ...
  </ttCustRow>
</ttCustomer>
```

To illustrate the different results obtained with each `read-mode` option, there is a unique index defined for the `CustNum` field. There is a record in both the include file and the XML file that uses the same `CustNum` value (1). This is known as a duplicate unique key conflict. The different read modes respond to such a conflict in different ways.
The code sample below performs the same `READ-XML()` method call you saw in the last example. Replace the highlighted variable value in the code with each of the following four read mode tokens and compare the results:

- EMPTY
- APPEND
- MERGE
- REPLACE

```plaintext
* pi-tfx-read-5.p */
/* Merges records in a temp-table with records from an XML file. */
{pi-tfx-parameterVarDefs.i}
{pi-tfx-ttSetup-5.i}

DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

ASSIGN
  cSourceType       = "FILE"
  cFile            = "ttCustomer.xml"
  cReadMode        = "EMPTY"
  cSchemaLocation  = ?
  lOverrideDefaultMapping = ?
  cFieldTypeMapping = ?
  cVerifySchemaMode = ?.

lReturn = TEMP-TABLE ttCustomer:READ-XML(cSourceType, cFile, cReadMode,
                                         cSchemaLocation, lOverrideDefaultMapping, cFieldTypeMapping,
                                         cVerifySchemaMode).
IF lReturn THEN
  FOR EACH ttCustomer:
    DISPLAY ttCustomer.CustNum ttCustomer.Name FORMAT "X(30)".
  END.
```

Compare your results to those shown and explained in the following table.

**Table 24: Read mode examples**

<table>
<thead>
<tr>
<th>Read mode</th>
<th>List result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td><img src="image" alt="Procedure Editor - Run" /></td>
<td>The method empties the existing data from the temp-table. It then reads the XML document and adds records to the temp-table. Since the temp-table is empty, there is no duplicate key conflict. The result is a temp-table with records exclusively from the XML document.</td>
</tr>
</tbody>
</table>
The method reads the XML document and adds records to the temp-table. Due to the duplicate key conflict between the record in the temp-table and the record in the XML with a CustNum of 1, the method stops loading records and stops. Records added before a conflict is detected are retained. The AVM displays the error message shown.

The method reads the XML document and adds records to the temp-table. Due to the duplicate key conflict between the record in the temp-table and the record in the XML with a CustNum of 1, the record from the XML is ignored. The result is all the existing records from the temp-table, plus all of the records from the XML except for the record with a CustNum of 1.

The method reads the XML document and adds records to the temp-table. Due to the duplicate key conflict between the record in the temp-table and the record in the XML with a CustNum of 1, the record from the XML replaces the record in the temp-table. The result is all the existing records from the temp-table except for the record with a CustNum of 1, plus all of the records from the XML.

Note: To see a version of the last example that uses a dynamic temp-table, look at the following sample files in the Doc_samples\xml directory: pi-tfx-ttSetup-6.i and pi-tfx-read-6.p

Reading XML into a ProDataSet

XML information read into a ProDataSet includes:

- Schema for defining a ProDataSets's temp-table buffers, temp-table indexes, and temp-table data-relations
- Data for the temp-table buffers
- Before-image information for the temp-table buffers

In this first code example, two static temp-tables are defined in an include file:
DEFINE TEMP-TABLE ttCustomer NO-UNDO BEFORE-TABLE ttCustBefore
FIELD CustNum LIKE Customer.CustNum
FIELD Name LIKE Customer.Name
FIELD Country LIKE Customer.Country
FIELD Comments LIKE Customer.Comments FORMAT "x(40)"
INDEX CustNum IS PRIMARY UNIQUE CustNum
INDEX Name Name
INDEX Comments IS WORD-INDEX Comments.

/* Definition for temp-table ttOrder */
DEFINE TEMP-TABLE ttOrder NO-UNDO
FIELD OrderNum LIKE Order.OrderNum
FIELD CustNum LIKE Order.CustNum
FIELD OrderDate LIKE Order.OrderDate
INDEX OrderNum IS PRIMARY UNIQUE OrderNum
INDEX CustOrder IS UNIQUE CustNum OrderNum
INDEX OrderDate OrderDate.

Here is a snippet of the XML file:

```xml
<?xml version="1.0"?>
<dsCustomerOrders xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ttCustomer>
    <CustNum>1</CustNum>
    <Country>USA</Country>
    <Name>Lift Tours</Name>
    <Address>276 North Drive</Address>
    <City>Burlington</City>
    <State>MA</State>
    <PostalCode>01730</PostalCode>
    <Contact>Gloria Shepley</Contact>
    <Phone>(617) 450-0086</Phone>
    <SalesRep>HXM</SalesRep>
    <CreditLimit>66700.0</CreditLimit>
    <Balance>903.64</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
  </ttCustomer>
  <ttOrder>
    <OrderNum>6</OrderNum>
    <CustNum>1</CustNum>
    <OrderDate>1998-02-11</OrderDate>
    <ShipDate>1998-02-16</ShipDate>
    <Carrier>Standard Mail</Carrier>
    <Instructions/>
    <PO/>
    <Terms>Net30</Terms>
    <SalesRep>HXM</SalesRep>
    <BillToID>0</BillToID>
    <ShipToID>0</ShipToID>
    <OrderStatus>Shipped</OrderStatus>
    <WarehouseNum>0</WarehouseNum>
    <Creditcard>American Express</Creditcard>
  </ttOrder>
  ...
</dsCustomerOrders>
```

Note: The child records are nested within the parent records.
The following code sample defines the static ProDataSet, reads the previous XML file, and reports all the orders by customer number:

```plaintext
/* pi-tfx-read-7.p */
/* Populates a static ProDataSet with records from an XML file. */

{pi-tfx-parameterVarDefs.i}
{pi-tfx-ttSetup-7.i}
DEFINE VARIABLE hPDS AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

DEFINE DATASET dsCustomerOrders FOR ttCustomer, ttOrder
    DATA-RELATION custOrd FOR ttCustomer, ttOrder
        RELATION-FIELDS(CustNum, CustNum) NESTED.

hPDS = DATASET dsCustomerOrders:HANDLE.
ASSIGN
cSourceType = "FILE"
cFile = "dsCustomerOrders.xml"
cReadMode = "EMPTY"
cSchemaLocation = ?
lOverrideDefaultMapping = NO.

lReturn = hPDS:READ-XML (cSourceType, cFile, cReadMode,
    cSchemaLocation, lOverrideDefaultMapping).
IF lReturn THEN DO:
    FOR EACH ttCustomer BY ttCustomer.CustNum:
        FOR EACH ttOrder WHERE ttOrder.CustNum = ttCustomer.CustNum:
            DISPLAY ttCustomer.CustNum ttCustomer.Name FORMAT "x(30)"
                ttOrder.OrderNum.
        END.
    END.
END.
```

The code displays a report of all the orders for each customer. For example:
Reading XML into a temp-table, temp-table buffer, or ProDataSet

Press space bar to continue.
This next code sample provides a dynamic ProDataSet handle with XML Schema and data:

```plaintext
/* pi-tfx-read-8.p */
/* Populates an empty dynamic ProDataSet with XML Schema and records from an XML document. */

{pi-tfx-parameterVarDefs.i}

DEFINE VARIABLE hPDS AS HANDLE NO-UNDO.
DEFINE VARIABLE hQuery AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE DATASET hPDS.

ASSIGN
cSourceType = "FILE"
cFile = "Dept400-2.xml"
cReadMode = "EMPTY"
cSchemaLocation = "Dept400-2.xsd"
lOverrideDefaultMapping = ?
cFieldTypeMapping = ?
cVerifySchemaMode = ?.

lReturn = hPDS:READ-XML(cSourceType, cFile, cReadMode, cSchemaLocation,
lOverrideDefaultMapping, cFieldTypeMapping, cVerifySchemaMode).
IF lReturn THEN
  DISPLAY SKIP "How many temp-table buffers in the ProDataSet?"
  hPDS:NUM-BUFFERS SKIP
  "How many data-relations in the ProDataSet?"
  hPDS:NUM-RELATIONS SKIP(2).

CREATE QUERY hQuery.

hQuery:SET-BUFFERS(hPDS:GET-BUFFER-HANDLE("ttEmp")).
hQuery:QUERY-PREPARE("FOR EACH ttEmp").
hQuery:QUERY-OPEN( ).
hQuery:GET-FIRST( ).

DISPLAY "Displaying Employee Roster..." SKIP(2)
"Employee No. Last Name First Name".

REPEAT WHILE NOT hQuery:QUERY-OFF-END:
  DISPLAY
    hPDS:GET-BUFFER-HANDLE("ttEmp"):BUFFER-FIELD("EmpNum"):BUFFER-VALUE
    hPDS:GET-BUFFER-HANDLE("ttEmp"):BUFFER-FIELD("LastName"):BUFFER-VALUE
    hPDS:GET-BUFFER-HANDLE("ttEmp"):BUFFER-FIELD("FirstName"):BUFFER-VALUE.
  hQuery:GET-NEXT( ).
END. /* REPEAT */
```

The following is the output for this code:
In this sample, a handle was defined and used to create a new dynamic ProDataSet. The ProDataSet is in the CLEAR state at this point, meaning it has no ABL definition. When the code calls the READ-XML method, the method sees that XML Schema has been provided. Since the dynamic ProDataSet is in the clear state, it knows the schema is being provided to create ABL definitions for the object from the XML Schema. If the ProDataSet had not been in the CLEAR state, the method would have to verify the XML Schema against whatever ABL definition existed for the ProDataSet.

The method can proceed to read in the data from the XML document provided in the method call. The code displays the number of temp-tables and data-relations in the dynamic object as well as a summary of the employee records now in the ttEmp temp-table.

**Reading before-image data for ProDataSet temp-table buffers from XML**

The read and write XML methods support before-image tables in ProDataSets. If the ProDataSet has defined before-image tables, then they will be handled appropriately:

- If a ProDataSet has before-tables and the READ-XML method finds before-image data in the XML data, then that data is read in. Before-image data can be in one of two formats:
  - ABL before-image format
  - Microsoft DiffGram format

- If a static ProDataSet member buffer does not define a before-image table and the XML data contains before-image data for that buffer, then READ-XML fails.
• If the \texttt{READ-XML( )} method reads data into a dynamic ProDataSet and detects before-image data for one or more of the temp-table buffers, then the method automatically creates a before-image table for each buffer with before-image data.

## Writing XML Schema from a temp-table, temp-table buffer, or ProDataSet

The \texttt{WRITE-XMLSCHEMA( )} method writes an XML representation (an XML Schema document) of the definition for a temp-table, temp-table buffer, or ProDataSet object. The XML Schema is written using the XML Schema Definition (XSD) language.

When writing XML Schema for a temp-table or temp-table buffer object, the AVM writes temp-table and index definitions. When writing XML Schema for a ProDataSet object, the AVM writes all temp-table definitions, index definitions, and data-relations.

Here is the syntax for \texttt{WRITE-XMLSCHEMA( )}. The method returns \texttt{TRUE} or \texttt{FALSE} to indicate whether the operation was successful.

### Syntax

\begin{verbatim}
WRITE-XMLSCHEMA ( mode, \{ file | stream | stream-handle | memptr | handle | longchar \} [, formatted [, encoding [, min-xmlschema [, omit-initial-values] ]]] )
\end{verbatim}

### target-type

A \texttt{CHARACTER} expression that specifies the target XML Schema document type. Valid values are: "FILE", "STREAM", "MEMPTR", "HANDLE", and "LONGCHAR".

### file

A \texttt{CHARACTER} expression that specifies the name of a file to which the AVM writes the XML Schema document text. You can specify an absolute pathname or a relative pathname (based on the current working directory). If a file with the specified name already exists, the AVM verifies that the file is writable and overwrites the file.

### stream

A \texttt{CHARACTER} expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the XML Schema document text to the default unnamed output stream.

### memptr

A \texttt{MEMPTR} variable to contain the XML Schema document text. The method allocates the required amount of memory for the XML document text and sets the size of the variable. When you are finished using the \texttt{MEMPTR}, you must free the associated memory by executing \texttt{SET-SIZE(memptr) = 0} on the \texttt{MEMPTR}.
handle

An X-document object handle or X-noderef object handle. If the specified handle contains
XML text, the AVM deletes the existing text first.

longchar

A LONGCHAR variable to contain the XML Schema document text.

The AVM saves the XML Schema document text to the LONGCHAR variable in the code
page that corresponds to the character encoding you specify in the encoding option. If you do not specify a character encoding for the XML Schema document text, the AVM
saves the LONGCHAR variable in UTF-8.

If the LONGCHAR variable's code page is fixed (that is, set using the FIX-CODEPAGE
function) and the fixed code page is not equivalent to the character encoding you specify
in the encoding option, the WRITE-XMLSCHEMA( ) method returns an error and the
XML Schema document is not saved to the LONGCHAR.

formatted

An optional LOGICAL expression where TRUE directs the AVM to format the XML Schema
document text in a hierarchical manner using extra white space, carriage returns, and
line feeds. The default value is FALSE.

If you specify the Unknown value (?), the method uses the default value of FALSE.

encoding

An optional CHARACTER expression that specifies the name of the character encoding
the AVM uses to write the XML Schema document text. The default encoding is UTF-8.

The encoding name must be an Internet Assigned Numbers Authority (IANA) name
supported by the ABL XML parser. For a list of supported IANA encodings and their
respective ABL code pages, see ABL to IANA Code Page Mappings on page 189.

Note: The AVM records this character encoding in the encoding declaration in the XML
document's prologue. If you specify the empty string (""), or the Unknown value (?),
The AVM uses the default encoding of UTF-8. In this case, The AVM does not record
the character encoding in the XML document's encoding declaration.

If target-type is HANDLE, the X-document's ENCODING attribute is also set to UTF-8.

min-xmlschema

An optional LOGICAL expression where TRUE directs the AVM to write the minimum
amount of XML Schema for the object, and FALSE directs the AVM to write the complete
XML Schema including ABL-specific XML Schema extensions. The default value is
FALSE. If you specify the Unknown value (?), the method uses the default value of
FALSE.

When TRUE, most ABL-specific XML Schema information is omitted from the XML
Schema. For a complete list of ABL XML Schema extensions, see Using XML Schema
on page 164. If the ABL data type of the temp-table field is not the default ABL data type
for the XML Schema type, the AVM writes the prodata:dataType XML Schema
attribute for the field. If the initial value of the temp-table field is TODAY, NOW, or UNKNOWN
(and UNKNOWN is not the default initial value for the field's data type), the AVM writes
the prodata:initial XML Schema attribute for the field.

When TRUE, the XML Schema will contain any ABL-specific XML Schema attributes
needed to define the data relations for a ProDataSet.

omit-initial-values

An optional LOGICAL expression where TRUE directs the AVM to exclude temp-table
fields containing their initial values from the XML document, and FALSE directs the AVM
to include all temp-table field data in the XML. The default value is FALSE. If you specify
the Unknown value (?), the method uses the default value of FALSE.

Note: For an array field to be omitted, each element of the array must contain the initial
value.

See Examples of additional XML support on page 169 for more information.

Writing a temp-table definition to XML Schema

The following code example creates a dynamic temp-table and writes its definition to an XML
Schema file:

```abl
/* pi-tfx-write-l.p */
/* Writes an XSD file from a dynamic temp table. */

{pi-tfx-parameterVarDefs.i}

DEFINE VARIABLE httCust AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE TEMP-TABLE httCust.
httCust:CREATE-LIKE("Customer").
httCust:TEMP-TABLE-PREPARE("ttCustomer").

ASSIGN
cTargetType = "FILE"
cFile = "ttCust2.xsd"
lFormatted = YES
cEncoding = ?
lMinSchema = NO.

lReturn = httCust:WRITE-XMLSCHEMA(cTargetType, cFile, lFormatted,
cEncoding, lMinSchema).
```

Note that the min-xmlschema method option (lMinSchema = NO) is set to NO, which means
that the AVM will use all the XML Schema extension attributes necessary to fully restore objects
in the ABL environment. Here is an example portion of the .xsd file generated by this code:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns=""
xmlns:prodata="urn:schemas-progress-com:xml-prodata:0001">
  <xsd:element name="ttCustomer" prodata:proTempTable="true">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ttCustRow" minOccurs="0" maxOccurs="unbounded">
```

OpenEdge Development: Working with XML
Writing XML Schema from a temp-table, temp-table buffer, or ProDataSet

Re-run this code with the `min-xmlschema` method option set to YES (`lMinSchema = YES`), which means that the AVM will use only use a small subset of its XML Schema extension attributes. The XML Schema will still identify temp-tables, ProDataSets, and data relations, and, if necessary, it will provide `datatype` and `initial` attributes. For example, here is a portion of the `.xsd` file generated by this code with `min-xmlschema` set to YES:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns=""
xmlns:prodata="urn:schemas-progress-com:xml-prodata:0001">
  <xsd:element name="ttCustomer" prodata:proTempTable="true">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ttCustRow" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CustNum" type="xsd:int" nillable="true"/>
              <xsd:element name="Country" type="xsd:string" nillable="true" default="USA" prodata:format="x(20)" prodata:help="Please enter a country."/>
              <xsd:element name="Name" type="xsd:string" nillable="true" prodata:format="x(30)" prodata:help="Please enter a name."/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Writing a ProDataSet definition to XML Schema files

Temp-table members of a ProDataSet can have a `NAMESPACE-URI` attribute that differs from the `NAMESPACE-URI` of the ProDataSet. In this case, the `WRITE-XMLSCHEMA( )` method uses a separate XML Schema document for the temp-table definition, since the XML Schema `TargetNamespace` attribute is different.

In other words, suppose your ProDataSet has three temp-table buffers. The ProDataSet defines a default namespace. The first temp-table buffer does not define a default namespace. Therefore, it uses the default namespace defined by its parent ProDataSet. The second temp-table buffer defines the same default namespace as the ProDataSet. So far, only a single XML Schema file is needed because only one namespace has been used. Suppose the third temp-table buffer defines a different default namespace. The method now needs to create a second XML Schema file for this temp-table buffer.

This code sample demonstrates the ability of the `WRITE-XMLSCHEMA( )` method to write different XML Schema files for different namespaces. For example:

- The method writes one XML Schema file for the namespace associated with the ProDataSet definition.
- The method writes one XML Schema file for each temp-table with a different namespace.
The following is an abbreviated version of the include file that sets up the ProDataSet. Note the variety of namespaces used:

```plaintext
/* pi-tfx-writeSetup-2.i */
/* Define a ProDataSet and temp-table buffers that use multiple namespaces. */
DEFINE TEMP-TABLE ttDeptartment NO-UNDO
    NAMESPACE-URI "urn:Dept-Temp-Table" NAMESPACE-PREFIX "pfxDept"
/* Field definitions. */
DEFINE TEMP-TABLE ttEmployee NO-UNDO
    NAMESPACE-URI "urn:Emp-Temp-Table" NAMESPACE-PREFIX "pfxEmp"
/* Field definitions. */
DEFINE TEMP-TABLE ttFamily NO-UNDO
    NAMESPACE-URI "urn:Fam-Temp-Table" NAMESPACE-PREFIX "pfxFam"
/* Field definitions. */
DEFINE DATASET dsEmployees
    NAMESPACE-URI "urn:PRODATASET" NAMESPACE-PREFIX "pfxDS"
/* ProDataSet definition. */
/* Populate ProDataSet. */
```

The following is the main code (note that only a single target file name is specified):

```plaintext
/* pi-tfx-write-2.p */
/* Write ProDataSet definition to XML Schema files. */
{pi-tfx-parameterVarDefs.i}
{pi-tfx-writeSetup-2.i}
ASSIGN
    cTargetType = "FILE"
    cFile = "prodataset.xsd"
    lFormatted = YES
    cEncoding = ?
    lMinSchema = NO.

lReturn = DATASET dsEmployees:WRITE-XMLSCHEMA(cTargetType, cFile, lFormatted, cEncoding, lMinSchema).
```

This is the standard `Open` dialog box set up to show `.xsd` files in the working directory:
The method call created four .xsd files and used the specified file name as the main ProDataSet-level XML Schema file. It also used the name as a root for naming other .xsd files it needed to create for temp-table buffers with different namespaces.

When reading this XML Schema, an application would find the child XML Schemas from the import directives in the in the main XML Schema. For example, here is a snippet of prodataset.xsd showing the imports:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="urn:PRODATASET" xmlns:pfxDS="urn:PRODATASET"
    targetNamespace="urn:PRODATASET" elementFormDefault="qualified"
    xmlns:prodata="urn:schemas-progress-com:xml-prodata:0001"
    xmlns:pfxDept="urn:Dept-Temp-Table" xmlns:pfxEmp="urn:Emp-Temp-Table"
    xmlns:pfxFam="urn:Fam-Temp-Table">
    <xsd:import namespace="urn:Dept-Temp-Table"
        schemaLocation="prodataset_ttDept.xsd"/>
    <xsd:import namespace="urn:Emp-Temp-Table"
        schemaLocation="prodataset_ttEmp.xsd"/>
    <xsd:import namespace="urn:Fam-Temp-Table"
        schemaLocation="prodataset_ttFam.xsd"/>
    ...
</xsd:schema>
```

**Note:** This feature works exclusively when writing XML Schema documents to operating system files (target-type "FILE"), otherwise it returns an error.
Writing XML from a temp-table, temp-table buffers, or a ProDataSet

The WRITE-XML() method writes an XML document containing the data of a temp-table or ProDataSet object. The method can be used on a temp-table buffer, but here it writes the entire contents of the associated temp-table, not just the contents of the buffer. You can optionally write the definition of the object along with the data. If you include the object’s definition, it is written using the XML Schema Definition (XSD) language.

When writing data from a ProDataSet object, the AVM writes the current version of data in each row of each table in the ProDataSet object. However, you can also include any before-image data, so that both the current and original versions of the data for each table row are written.

When writing XML Schema for a ProDataSet object, the AVM writes all table definitions as well as relation and index definitions. When writing XML Schema for a temp-table or temp-table buffer object, the AVM writes table and index definitions.

Here is the syntax for WRITE-XML(). The method returns TRUE or FALSE to indicate if the operation was successful.

Syntax

```
WRITE-XML ( mode, { file | stream | stream-handle | memptr | handle | longchar }
              [, formatted [, encoding [, schema-location [, write-xmlschema
              [, min-xmlschema [, write-before-image
              [, omit-initial-values]]]]]] ])
```

target-type

A CHARACTER expression that specifies the target XML document type. Valid values are: "FILE", "STREAM", "MEMPTR", "HANDLE", and "LONGCHAR".

file

A CHARACTER expression that specifies the name of a file to which the AVM writes the XML document text. You can specify an absolute pathname or a relative pathname (based on the current working directory). If a file with the specified name already exists, the AVM verifies that the file is writable and overwrites the file.

stream

A CHARACTER expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the XML document text to the default unnamed output stream.

memptr

A MEMPTR variable to contain the XML document text. The method allocates the required amount of memory for the XML document text and sets the size of the variable. When
you are finished using the MEMPTR, you must free the associated memory by executing
SET-SIZE(memptr) = 0 on the MEMPTR.

handle

An X-document object handle or X-noderef object handle. If the specified handle contains
XML text, the AVM deletes the existing text first.

longchar

A LONGCHAR variable to contain the XML document text.

The AVM saves the XML document text to the LONGCHAR variable in the code page that
corresponds to the character encoding you specify in the encoding option. If you do
not specify a character encoding for the XML document text, the AVM saves the
LONGCHAR variable in UTF-8.

If the LONGCHAR variable’s code page is fixed (that is, set using the FIX-CODEPAGE
function) and the fixed code page is not equivalent to the character encoding you specify
in the encoding option, the WRITE-XML( ) method returns an error and the XML
document is not saved to the LONGCHAR.

formatted

An optional LOGICAL expression where TRUE directs the AVM to format the XML
document text in a hierarchical manner using extra white space, carriage returns, and
line feeds. The default value is FALSE.

If you specify the Unknown value (?), the method uses the default value of FALSE.

encoding

An optional CHARACTER expression that specifies the name of the character encoding
the AVM uses to write the XML document text. The default encoding is UTF-8.

The encoding name must be an Internet Assigned Numbers Authority (IANA) name
supported by the ABL XML parser. For a list of supported IANA encodings and their
their corresponding ABL code pages, see ABL to IANA Code Page Mappings on page 189.

Note: The AVM records this character encoding in the encoding declaration in the XML
document’s prologue. If you specify the empty string (""") or the Unknown value (?),
the AVM uses the default encoding of UTF-8. In this case, the AVM does not record
the character encoding in the XML document’s encoding declaration.

If target-type is HANDLE, the X-document’s ENCODING attribute is also set to UTF-8.

schema-location

An optional CHARACTER expression that specifies the name of an external XML Schema
file. The method uses this value to set the xsi:schemaLocation or
xsi:noNamespaceSchemaLocation attribute in the XML document.

Note: You must provide the location of an actual XML Schema document. Consider
using the WRITE-XMLSCHEMA( ) method to generate the XML Schema document.
write-xmlschema

An optional LOGICAL expression where TRUE directs the AVM to write the ProDataSet or temp-table object's definition as in-line XML Schema along with the data, and FALSE directs the AVM to write only the data. The default value is FALSE.

**Note:** If you specify TRUE and the NAMESPACE-URI attribute value for a temp-table buffer within a ProDataSet object is different than that of the ProDataSet object, the method creates a separate XML Schema file for the temp-table definition. The namespace URI for the temp-table is imported into the ProDataSet schema, with a schemaLocation pointing to a separate XML Schema file containing the temp-table definition. Multiple namespaces are supported only when target-type is "FILE". If the ProDataSet object contains multiple namespaces and target-type is not "FILE", the method generates an error and returns FALSE.

min-xmlschema

An optional LOGICAL expression where TRUE directs the AVM to write the minimum amount of schema when it writes the XML Schema representation of the object, and FALSE directs the AVM to write the complete schema including the AVM-specific XML Schema extensions. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

When TRUE, most ABL-specific XML Schema information is omitted from the XML Schema. For a complete list of ABL XML Schema extensions, see Using XML Schema on page 164. If the ABL data type of the temp-table field is not the default ABL data type for the XML Schema type, the AVM writes the prodata:dataType XML Schema attribute for the field. If the initial value of the temp-table field is TODAY, NOW, or UNKNOWN (and UNKNOWN is not the default initial value for the field's data type), the AVM writes the prodata:initial XML Schema attribute for the field.

When TRUE, the XML Schema will contain any ABL-specific XML Schema attributes needed to define the data relations for a ProDataSet.

write-before-image

An optional LOGICAL expression where TRUE directs the AVM to write any before-image table data and error information in addition to the ProDataSet object data, and FALSE directs the AVM to write only the ProDataSet object data. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

omit-initial-values

An optional LOGICAL expression where TRUE directs the AVM to exclude temp-table fields containing their initial values from the XML document, and FALSE directs the AVM to include all temp-table field data in the XML. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

**Note:** For an array field to be omitted, each element of the array must contain the initial value.

See Examples of additional XML support on page 169 for more information.
Writing temp-table data to XML

Having taken a tour through several scenarios for reading XML data, examples of writing XML data will be mostly familiar. If you wish to experiment with these code samples, the best way to examine your results is to open the XML data files and see how your changes affected the XML.

The following code example creates a static temp-table, copies a subset of Customer records to it, then serializes the data to an XML file:

```
/* pi-tfx-write-3.p */
/* Populates a temp-table with a subset of Customer records and writes it to an XML file. */

{pi-tfx-parameterVarDefs.i}
{pi-tfx-writeSetup-3.i}
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

FOR EACH Customer NO-LOCK WHERE Customer.CustNum < 4:
  CREATE ttCustomer.
  BUFFER-COPY Customer TO ttCustomer.
END.

ASSIGN
  cTargetType = "FILE"
  cFile = "ttCustomer.xml"
  lFormatted = YES
  cEncoding = ?
  cSchemaLocation = ?
  lWriteSchema = NO
  lMinSchema = NO.

lReturn = TEMP-TABLE ttCustomer:WRITE-XML(cTargetType, cFile, lFormatted, cEncoding, cSchemaLocation, lWriteSchema, lMinSchema).
```

Writing XML from a temp-table buffer's current row

The SERIALIZE-ROW( ) method serializes a buffer's current row to JSON or XML.

The syntax for SERIALIZE-ROW( ) is shown below. The method returns TRUE or FALSE to indicate whether the operation was successful.

Syntax

```
SERIALIZE-ROW ( target-format, target-type,
  { file | stream | stream-handle | memptr | longchar }
  [, formatted [, encoding [, omit-initial-values
  [, omit-outer-object]]] ] ] )
```

**target-format**

A CHARACTER expression that specifies the target for the JSON string. Valid values are "JSON" and "XML".
**target-type**

A CHARACTER expression that specifies the target XML Schema document type. Valid values are: "FILE", "STREAM", "MEMPTR", "HANDLE", and "LONGCHAR".

**file**

A CHARACTER expression that specifies the name of a file to which the AVM writes the JSON string. You can specify an absolute pathname or a pathname relative to the current working directory. If a file with the specified name already exists, the AVM verifies that the file is writeable and overwrites the file.

**stream**

A CHARACTER expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the JSON string to the default unnamed output stream. For WebSpeed, write the JSON string to the WebSpeed-defined output stream (WEBSTREAM).

**stream-handle**

A HANDLE variable that specifies a stream object handle.

**memptr**

A MEMPTR variable to contain the JSON string in memory. If you do not specify the encoding parameter, the AVM encodes the text written to the MEMPTR as UTF-8. This method allocates the required amount of memory for the JSON string and sets the size of the variable. When you are finished using the MEMPTR, you must free the associated memory, by setting the MEMTER to zero bytes with the SET-SIZE statement.

**longchar**

A LONGCHAR variable to contain the JSON string in memory.

The AVM saves the JSON string to the LONGCHAR variable in the code page that corresponds to the character encoding you specify in the encoding option. If you do not specify a character encoding for the JSON string, the AVM saves the LONGCHAR variable in UTF-8.

If the LONGCHAR variable's code page is fixed (that is, set using the FIX-CODEPAGE statement) and the fixed code page is not equivalent to the character encoding you specify in the encoding option, the WRITE-JSON( ) method generates an error and returns FALSE. The JSON string is not saved to the LONGCHAR.

**formatted**

An optional LOGICAL expression where TRUE directs the AVM to format the JSON string in a hierarchical manner using extra white space, carriage returns, and line feeds. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

**encoding**

An optional CHARACTER expression that specifies the name of the character encoding the AVM uses to write the JSON string. The default encoding is "UTF-8".
The encoding name must specify a Unicode transformation format. Valid values are "UTF-8", "UTF-16", "UTF-16BE", "UTF-16LE", "UTF-32", "UTF-32BE", and "UTF-32LE".

**Note:** If you specify the empty string ('"') or the Unknown value ('?'), the AVM uses the default encoding of "UTF-8".

**omit-initial-values**

An optional LOGICAL expression where TRUE directs the AVM to exclude temp-table fields containing their initial values from the JSON string, and FALSE directs the AVM to include all temp-table field data in the JSON. The default value is FALSE. If you specify the Unknown value ('?'), the method uses the default value of FALSE.

**omit-outer-object**

This argument is not valid when target-format is "XML". If set to TRUE, SERIALIZE-ROW generates an error and returns FALSE.

---

**Writing a ProDataSet to XML**

The following code sample writes a ProDataSet to an XML file. This is a snippet of the setup code showing the ProDataSet definition (note the NESTED options):

```plaintext
/* pi-tfx-writeSetup-4.i */
/* Definition of a ProDataSet. */
...
DEFINE DATASET DSET FOR ttDepartment, ttEmployee, ttFamily, ttBenefits
    DATA-RELATION DeptEmp FOR ttDepartment, ttEmployee
        RELATION-FIELDS (deptcode, deptcode) NESTED
    DATA-RELATION EmpFam FOR ttEmployee, ttFamily
        RELATION-FIELDS (empnum, empnum) NESTED
    DATA-RELATION EmpBene FOR ttEmployee, ttBenefits
        RELATION-FIELDS (empnum, empnum) NESTED.
...
```

The NESTED option cannot be used when min-xmlschema optional parameter is TRUE. The following is a code sample for a simple XML write from a static ProDataSet:

```plaintext
/* pi-tfx-write-4.p */
/* Writes the data from a static ProDataSet to an XML file. */
{pi-tfx-parameterVarDefs.i}
{pi-tfx-writeSetup-4.i}

DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
ASSIGN
    cTargetType   = "FILE"
    cFile         = "Dept400.xml"
    lFormatted    = TRUE
    cEncoding     = ?
    cSchemaLocation = ?
    lWriteSchema  = FALSE
    lMinSchema    = FALSE
    lWriteBeforeImage = FALSE.
```
Return = DATASET DSET:WRITE-XML(cTargetType, cFile, lFormatted, cEncoding, cSchemaLocation, lMinSchema, lWriteBeforeImage).

This example uses the NESTED option, and is a snippet of the resulting XML demonstrating what it does:

```xml
<?xml version="1.0"?>
<DSET xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ttDepartment><DeptCode>400</DeptCode>
    <DeptName>Sales</DeptName>
    <ttEmployee>
      <EmpNum>3</EmpNum>
      <LastName>Smith</LastName>
      <FirstName>Justine</FirstName>
      <Address>1342 Atlantic Ave</Address>
      <Address2>Apt 345b</Address2>
      <City>Boston</City>
      <State>MA</State>
      <PostalCode>01834</PostalCode>
      <HomePhone>617 333-3334</HomePhone>
      <WorkPhone>800 787-8484</WorkPhone>
      <DeptCode>400</DeptCode>
      <Position>Sales Manager</Position>
      <Birthdate>1960-01-08</Birthdate>
      <StartDate>1997-06-19</StartDate>
      <VacationDaysLeft>5</VacationDaysLeft>
      <SickDaysLeft>4</SickDaysLeft>
      <ttFamily>
        <EmpNum>3</EmpNum>
        <RelativeName>Kelly Smith</RelativeName>
        <Relation>Daughter</Relation>
        <Birthdate>1993-08-18</Birthdate>
        <CoveredOnBenefits>true</CoveredOnBenefits>
        <BenefitDate>1997-06-19</BenefitDate>
      </ttFamily>
      <ttFamily>
        <EmpNum>3</EmpNum>
        <RelativeName>Mark Smith</RelativeName>
        <Relation>Spouse</Relation>
        <Birthdate>1960-01-08</Birthdate>
        <CoveredOnBenefits>true</CoveredOnBenefits>
        <BenefitDate>1998-02-21</BenefitDate>
      </ttFamily>
    </ttEmployee>
  </ttDepartment>
</DSET>
```

Note that the ttEmp row associated with employee number 3 of department 400 is nested within ttDept row for department 400 and that the rows from the ttFam table associated with employee number 3 are included within the ttEmp element.

Remove the NESTED keywords from the setup code and run it again. The resulting XML file now listed all rows from one table then all rows from the next table, and so on. The following is a condensed version of the XML highlighting this point:

```xml
<ttDepartment>
  <ttEmployee>
    <EmpNum>3</EmpNum>
    ...
  </ttEmployee>
  <ttEmployee>
    <EmpNum>6</EmpNum>
    ...
  </ttEmployee>
</ttDepartment>
```
Minimizing XML document size

When using XML documents to handle large amounts of data, it is always worthwhile to design your schema to minimize the size of XML documents. Smaller XML documents reduce memory, network bandwidth, and disk usage. The resulting resource savings can be significant in a large XML application.

There are two features of ABL XML support that can further reduce the size of XML documents:

- For ProDataSets with nested data relations, you can choose to omit the duplicate entries of foreign key fields that appear in each child record nested within a parent record.
- For any ProDataSet or temp-table, you can choose to omit fields from the XML whose current value is the same as the initial value of that field.

Omitting foreign key fields in nested child records

When working with a nested data relation in a ProDataSet, the fields that define the relationship appear in both the parent record and the child records. Because the child records are contained within the parent record, the foreign key fields in the child records are redundant.

You can choose to omit foreign key fields by:

- Specifying the FOREIGN-KEY-HIDDEN option on a DEFINE DATA-RELATION statement
- Specifying the foreign-key-hidden argument on the ADD-RELATION( ) method of a Data-relation object
- Setting the FOREIGN-KEY-HIDDEN attribute of a Data-relation object to TRUE

In all three cases, the NESTED option must also be specified.

Care must be taken when deciding to use this feature. The READ-XML( ) method automatically populates foreign keys in nested child records with the value in the outer parent record when the foreign key is omitted from the XML document. Unless you are sure that a non-ABL consumer of the XML document will do the same, you should not use this option in your nested data-relations.
For example, while .NET can read this XML document and populate an ADO.NET Dataset, it will create rows in the child DataTable with a null value for the foreign key field.

Omitting fields with initial values

When a field is defined, it gets an initial value either by using the default initial value for the field's data type or from the INITIAL option in the definition statement. The following table shows the default initial values for ABL data types.

Table 25: Default initial values for ABL data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>”” (empty string)</td>
</tr>
<tr>
<td>CLOB</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>INT64</td>
<td>0</td>
</tr>
<tr>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>No</td>
</tr>
<tr>
<td>RAW</td>
<td>Zero-length sequence of bytes</td>
</tr>
<tr>
<td>RECID</td>
<td>Unknown value (?)</td>
</tr>
<tr>
<td>ROWID</td>
<td>Unknown value (?)</td>
</tr>
</tbody>
</table>

Omitting initial value fields from your XML can be useful if:

- Fields with initial values are not important to the business or application logic of your XML consumer
- The XML consumer knows how to recreate missing fields and populate them with initial values

To omit these fields, specify TRUE for the optional `omit-initial-values` argument of the `WRITE-XML()` and `WRITE-XMLSCHEMA()` methods.
The ABL extensions to XML Schema specify that an omitted field is optional and what its initial value should be. Therefore, if you generate an XML Document from ABL, the `READ-XML( )` method will always correctly restore omitted fields and their initial values.

A non-ABL consumer may correctly restore omitted fields and values. For example, .NET can read an XML Schema and XML document into a dynamic ADO.NET Dataset using its `ReadXmlSchema( )` and `ReadXml( )` APIs, but missing elements are always interpreted as a null (unknown in ABL terms) DataColumn in the DataTable. They do recognize the initial value definition in the XML Schema when creating rows in the DataTable directly (initial value in the XML Schema gets translated to the DefaultValue property on the DataColumn), but do not with `ReadXml( )`.

### Sample ProDataSet to XML round-trip

One of the most important use cases for the features described in this chapter is that of the XML round-trip. XML can be a persistent storage mechanism between OpenEdge sessions. For example, a mobile computing model could use XML as a persistent data store. Suppose a sales representative could work off-line from the server with a local copy of selected records, limited to his accounts, stored as XML. The mobile application could read the XML and work locally. The changes made off-line by the sales representative on the records would be stored locally as an updated XML file, storing before-image data as well. When the sales representative reconnects to the system, the XML would be sent back to the application server to synchronize with enterprise application. The application code could decide whether to apply the remote changes or overwrite with server changes. The before-image data could be used to compare changes that might have been made while the sales representative was working remotely.

All you need to do is create an XML write procedure to store your data and an XML read procedure to retrieve it. Of course, you must ensure that the `min-xmlschema` option is set to `FALSE`. That is, you want the AVM to write the ABL XML Schema extensions so that the ABL definitions are fully restored when the file is read back in.
The following include file sets up a pair of temp-tables and a ProDataSet for the two code samples to use:

```c
/* pi-tfx-writeSetup-6.i */
/* Creates two new static temp-tables and a ProDataSet. */

/* Definition for Temp-Table ttCustomer */
DEFINE TEMP-TABLE ttCustomer NO-UNDO BEFORE-TABLE ttCustBef
   FIELD CustNum LIKE Customer.CustNum
   FIELD Name LIKE Customer.Name COLUMN-LABEL "Loyal Customer"
      XML-NODE-TYPE "Attribute"
   FIELD Country LIKE Customer.Country
   FIELD Comments LIKE Customer.Comments FORMAT "x(40)"
   INDEX CustNum IS PRIMARY UNIQUE CustNum
   INDEX Name Name
   INDEX Comments IS WORD-INDEX Comments.

/* Definition for Temp-Table ttOrder */
DEFINE TEMP-TABLE ttOrder BEFORE-TABLE ttOrdBef
   FIELD OrderNum LIKE Order.OrderNum
   FIELD CustNum LIKE Order.CustNum
   FIELD OrderDate LIKE Order.OrderDate
   INDEX OrderNum IS PRIMARY UNIQUE OrderNum
   INDEX CustOrder IS UNIQUE CustNum OrderNum
   INDEX OrderDate OrderDate.

DEFINE DATASET dsCustomerOrders FOR ttCustomer, ttOrder
   DATA-RELATION custOrd FOR ttCustomer, ttOrder
      REPOSITION RELATION-FIELDS (CustNum, CustNum) NESTED.
```

The first code sample performs an XML write and informs you if it has been successful:

```c
/* pi-tfx-write-6a.p */
/* Writes data from a static ProDataSet to an XML file. This demonstrates
   the first half of a persistent storage mechanism through XML. */

{pi-tfx-parameterVarDefs.i}
{pi-tfx-writeSetup-6.i}

DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
DEFINE VARIABLE hPDS AS HANDLE NO-UNDO.

hPDS = DATASET dsCustomerOrders:HANDLE.

/* Before the method call, your application does work with its data. */

ASSIGN
   cTargetType = "FILE"
   cFile = "dsCustomerOrder.xml"
   lFormatted = YES
   cEncoding = ?
   cSchemaLocation = ?
   lWriteSchema = YES
   lMinSchema = FALSE
   lWriteBeforeImage = TRUE.

lReturn = hPDS:WRITE-XML (cTargetType, cFile, lFormatted, cEncoding,
IF lReturn = FALSE THEN DO:
   MESSAGE "WRITE-XML on ProDataSet failed!" VIEW-AS ALERT-BOX.
   RETURN.
END.
ELSE
   MESSAGE "Successful WRITE-XML on : " hPDS:NAME VIEW-AS ALERT-BOX.
```
The second code sample performs an XML read and informs you if it has been successful, as shown:

```abl
/* pi-tfx-write-6b.p */
/* Reads and writes the data to and from a static ProDataSet to an XML file. */

{pi-tfx-parameterVarDefs.i}
{pi-tfx-writeSetup-6.i}

DEFINE VARIABLE hPDS AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

hPDS = DATASET dsCustomerOrders:HANDLE.

ASSIGN
  cSourceType = "FILE"
  cFile = "dsCustomerOrders.xml"
  cReadMode = ?
  cSchemaLocation = ?
  lOverrideDefaultMapping = FALSE.

lReturn = hPDS:READ-XML(cSourceType, cFile, cReadMode, cSchemaLocation, lOverrideDefaultMapping).

IF NOT lReturn THEN DO:
  MESSAGE "READ-XML on ProDataSet failed!" VIEW-AS ALERT-BOX.
  RETURN.
END.
ELSE
  MESSAGE "Successful READ-XML on:" hPDS:NAME VIEW-AS ALERT-BOX.

/* After the method call, your application does work with its data. */
```

Mapping ABL names to different XML element or attribute names

If you have XML Documents or XML Schemas that you import and export through ABL, you can always adjust your ABL definitions or XML Schema to use identical names. In some cases, you may be working with pre-existing ABL definitions and XML Schema that cannot be easily changed. The `XML-NODE-NAME` or `SERIALIZE-NAME` attributes on a ProDataSet object handle, temp-table object, temp-table buffer object, or buffer field object allows you to specify an XML element (or attribute) name for the ABL object. For more information on the interaction of these attributes, see `XML-NODE-NAME` and `SERIALIZE-NAME` on page 123.

You can set these attributes on the `DEFINE BUFFER`, `DEFINE DATASET`, or `DEFINE TEMP-TABLE` statement by using the corresponding options. You can set them for a buffer field using the field definition options of the `DEFINE TEMP-TABLE` statement. You can also directly set the `XML-NODE-NAME` or `SERIALIZE-NAME` attributes on the object handle.

These attributes allow you to work around ABL names that use illegal XML characters or to work around XML element or attribute names that are reserved words in ABL.

The ABL `READ-XMLSCHEMA( )` method and the `xsdto4gl` utility create ABL temp-table or ProDataSet definitions with the correct ABL names.
The following example demonstrates the use of the attribute by mapping names illegal in ABL or XML to an acceptable substitute:

```abl
/* pi-tfx-write-12.p */
DEFINE VARIABLE hdset AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

DEFINE TEMP-TABLE ttCustomer% NO-UNDO XML-NODE-NAME "ttCustSafe"
FIELD Cust# AS INTEGER XML-NODE-NAME "custNo"
FIELD Name AS CHARACTER XML-NODE-TYPE "ATTRIBUTE" XML-NODE-NAME "Name1"
FIELD Country AS CHARACTER
INDEX CustNum IS UNIQUE PRIMARY Cust#.

DEFINE TEMP-TABLE ttOrd$ NO-UNDO XML-NODE-NAME "ttOrdSafe"
FIELD Order# AS INTEGER XML-NODE-NAME "OrderNum"
FIELD Cust# AS INTEGER XML-NODE-NAME "custNo"
INDEX OrderNum IS UNIQUE PRIMARY Order#.

DEFINE DATASET dsCustOrd& NAMESPACE-URI "urn:myds"
XML-NODE-NAME "dsCO" FOR ttCustomer%, ttOrd$
DATA-RELATION custOrd FOR ttCustomer%, ttOrd$
   RELATION-FIELDS (Cust#, Cust#) NESTED.

hdset = DATASET dsCustOrd&:HANDLE.
CREATE ttCustomer%.
ASSIGN ttCustomer%.Cust# = 10
   ttCustomer%.Name = "Round Trip Inc".
CREATE ttOrd$.
ASSIGN ttOrd$.Cust# = 10
   ttOrd$.Order# = 95.

```

This is the XML Document written by the code:

```
<?xml version="1.0"?>
<dsCO xmlns="urn:myds" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ttCustSafe xmlns="urn:myds" Name1="Round Trip Inc">
      <custNo>10</custNo>
      <Country/>
   </ttCustSafe>
   <ttOrdSafe>
      <OrderNum>95</OrderNum>
      <custNo>10</custNo>
   </ttOrdSafe>
</dsCO>
```

Using XML Schema

You can write XML data and XML Schema from an ABL application and read it back into the application at a later time. In this scenario, the provided ABL methods shield you from having to learn much about XML Schema. However, more complex use cases may require you to understand the interaction of ABL and XML Schema.
During a read XML operation, the AVM acts differently depending on the following:

- If no ABL definition exists, which means the object is dynamic, the AVM can provide a definition for the dynamic object from XML Schema. For more information, see Creating dynamic ABL objects from XML Schema on page 165.

- If an ABL definition exists and XML Schema is also specified, the AVM verifies that the XML Schema matches the ABL definition. For more information, see Verifying XML Schema against ABL definitions on page 166.

- If no ABL definition exists and no XML Schema is specified, the AVM attempts to construct a schema from XML data. This is called inferring schema. For more information, see Inferring definitions from XML data on page 166.

Creating dynamic ABL objects from XML Schema

Both the READ-XML( ) and READ-XMLSCHEMA( ) methods can take an XML Schema and create a temp-table or ProDataSet definition from it.

The READ-XML( ) method can find and use XML Schema from the following sources, in the following order:

1. The XML Schema specified with the schema-location option of the READ-XML( ) method. If this option is used, XML Schema references embedded in XML data are ignored.

2. A <schema> child element of the root element of the XML data document. Provided the method does not specify a schema-location, then any one or combination of this and the remaining list item are used.

3. An xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute on an instance data in the XML data document.

In the following example, the READ-XMLSCHEMA( ) method will produce a temp-table named Person with fields name, height, weight, and gender:

```xml
<xsd:schema
    <!-- namespace declarations go here -->
    <xsd:element name="Person">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="PersonRow" minOccurs="0" maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="name" type="xsd:string" nillable="true"/>
                            <xsd:element name="height" type="xsd:int" nillable="true"/>
                            <xsd:element name="weight" type="xsd:int" nillable="true"/>
                            <xsd:attribute name="gender" type="xsd:string"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```
In the next example, the READ-XMLSCHEMA( ) method produces a ProDataSet named dsGarden with two temp-tables: one for holding information about the plots in the garden and another for holding information about vegetables in the garden:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns=""
xmlns:prodata="urn:schemas-progress-com:xml-prodata:0001">
  <xsd:element name="dsGarden" prodata:proDataSet="true">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ttPlots" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="plotId" type="xsd:int" nillable="true"/>
              <xsd:element name="plotDesc" type="xsd:string" nillable="true"/>
              <xsd:element name="isFallow" type="xsd:boolean" nillable="true"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="ttVeggies" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="veggieId" type="xsd:int" nillable="true"/>
              <xsd:element name="veggieDesc" type="xsd:string" nillable="true"/>
              <xsd:element name="plotId" type="xsd:int" nillable="true"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Verifying XML Schema against ABL definitions

If the temp-table or ProDataSet already contains an ABL definition, READ-XML( ) verifies the XML Schema specified with the schema-location option, or any schema defined or referenced in the XML against the temp-table or ProDataSet definition. Verification is different from the XML notion of validation. Validation refers to whether or not the XML document conforms to a particular XML Schema. Verification is the ABL-specific process of comparing ABL definitions to XML Schema and determining if they are compatible. If verification fails, the method fails.

You have some control over the verification process. The verify-schema-mode option lets you specify a verification mode, which are described in.

Inferring definitions from XML data

If the temp-table or ProDataSet has no definition, and the READ-XML( ) method does not specify a schema-location, and the XML document does not contain references to XML Schema, READ-XML( ) will attempt to construct XML Schema from the XML data. This is known as inferring schema.
If an element in the instance data contains the "xsi:type" attribute, the field is assigned an ABL data type based on XML Schema mapping rules. See XML Schema and ABL Data Type Mappings on page 185 if the element is not typed, the field will be CHARACTER. All attributes and text elements will become CHARACTER fields.

Inferring schema for a temp-table

Suppose you have an XML document for a single temp-table, but you have no XML Schema for it. You can either call the READ-XML( ) method on a temp-table handle and create a dynamic temp-table object in ABL; or, you can call the READ-XML( ) method on a ProDataSet handle and create a dynamic ProDataSet object with a single temp-table buffer in ABL.

For example, examine this XML document:

```
<?xml version="1.0"?>
<Rows>
  <ttFam RelativeName="Ken Koberlein">
    <Relation>Spouse</Relation>
    <IncludedOnBenefits>true</IncludedOnBenefits>
    <EmpNum>1</EmpNum>
  </ttFam>
  <ttFam RelativeName="Kelly Smith">
    <Relation>Daughter</Relation>
    <IncludedOnBenefits>true</IncludedOnBenefits>
    <EmpNum>3</EmpNum>
  </ttFam>
  <ttFam RelativeName="Mark Smith">
    <Relation>Spouse</Relation>
    <IncludedOnBenefits>true</IncludedOnBenefits>
    <EmpNum>3</EmpNum>
  </ttFam>
  <ttFam RelativeName="Sam Smith">
    <Relation>Son</Relation>
    <IncludedOnBenefits>false</IncludedOnBenefits>
    <EmpNum>3</EmpNum>
  </ttFam>
</Rows>
```

Using READ-XML( ) on a temp-table object with this XML document will create a temp-table named ttFam, with four CHARACTER fields: RelativeName, Relation, IncludedOnBenefits and EmpNum.

Using READ-XML( ) on a ProDataSet object with this XML will create a ProDataSet named Rows, with one temp-table buffer, ttFam, with all its buffer-fields.

Inferring schema for a ProDataSet

Suppose you have an XML document for a dataset, but you have no XML Schema for it. You can call the READ-XML( ) method on a ProDataSet handle and create a dynamic ProDataSet object in ABL.
This XML document is an example:

```xml
<?xml version="1.0"?>
<myDataSet>
  <Department>
    <DeptCode>400</DeptCode>
    <DeptName>Sales</DeptName>
  </Department>
  <Employee EmpNum="3">
    <DeptCode>400</DeptCode>
    <LastName>Smith</LastName>
    <FirstName>Lee</FirstName>
    <Address>1342 Atlantic Ave</Address>
    <City>Boston</City>
  </Employee>
  <Family RelativeName="Ken Koberlein">
    <Relation>Spouse</Relation>
    <IncludedOnBenefits>true</IncludedOnBenefits>
    <EmpNum>3</EmpNum>
  </Family>
</myDataSet>
```

Using `READ-XML( )` on a ProDataSet object will create a ProDataSet named `myDataSet` with three temp-table buffers: Department, Employee and Family. The Department temp-table has two CHARACTER fields: DeptCode and DeptName. The Employee temp-table has six CHARACTER fields: EmpNum, DeptCode, LastName, FirstName, Address and City. The Family temp-table has four CHARACTER fields: RelativeName, Relation, IncludedOnBenefits, and EmpNum.

Using `READ-XML( )` on a temp-table object will fail, since there is more than one temp-table represented in the XML document.

### Inferring data-relations from nested XML and XML Schema

ABL supports the mapping of complex XML and XML Schema to a ProDataSet definition. When temp-tables within the XML are nested, a relationship is required between the parent table and its nested child in order to preserve the round-trip of the schema and data serialization.

When there is no explicit relationship defined in the XML Schema, or no implicit relationship can be found based a matching field in the parent and nested child tables, ABL provides a PARENT-ID-RELATION for the ProDataSet. A PARENT-ID-RELATION relates the parent and child based on the RECID of the parent record.

The PARENT-ID-RELATION allows ABL to expand the set of XML formats that can map to a ProDataSet. This expanded support includes the following:

- `READ-XML( ), READ-XMLSCHEMA( ),` and the `bproxsdo4gl` utility will create a PARENT-ID-RELATION when reading XML or XML Schema if a DATA-RELATION cannot be used to represent the nested XML. These methods will also add a RECID field to the child table with XML-NODE-TYPE "HIDDEN" to hold the RECID of the parent record. When creating records, `READ-XML( )` will automatically populate the added field in the child table with the RECID of the parent record.

- `WRITE-XML( )` and `WRITE-XMLSCHEMA( )` recognize the PARENT-ID-RELATION and serialize the data and schema correctly.
• The PARENT-ID-RELATION phrase is available on the DEFINE DATASET statement, and the ADD-PARENT-ID-RELATION method is available on a dynamic ProDataSet object.

• PARENT-FIELDS-BEFORE and PARENT-FIELDS-AFTER options are available on a PARENT-ID-RELATION, which specify which parent fields are to be serialized before and after the nested child records.

• XML-NODE-TYPE and SERIALIZE-HIDDEN are available at the ProDataSet level so that a ProDataSet can be serialized with the top-level temp-table as the root node of the XML document.

Examples of additional XML support

This section provides various examples to demonstrate support for mapping complex XML to a ProDataSet.

Simple example of a PARENT-ID-RELATION

The following is a sample XML file with nested child tables.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CustomerOrders>
  <Customer>
    <CustNum>1</CustNum>
    <Name>LiftTours</Name>
    <Order>
      <OrderNum>100</OrderNum>
      <OrderTotal>1234.89</OrderTotal>
    </Order>
    <Order>
      <OrderNum>150</OrderNum>
      <OrderTotal>999.99</OrderTotal>
    </Order>
  </Customer>
  <Customer>
    <CustNum>3</CustNum>
    <Name>Hoops</Name>
    <Order>
      <OrderNum>200</OrderNum>
      <OrderTotal>1899.99</OrderTotal>
    </Order>
  </Customer>
</CustomerOrders>
```

In the above XML file, CustomerOrders is a DataSet with two member tables:

• Customer with two fields, CustNum and Name

• Order with two fields, OrderNum and OrderTotal

Although there is no matching field between the nested Customer and Order tables, the PARENT-ID-RELATION phrase provides syntax for specifying a relationship between Customer and Order records based on the RECID of the parent Customer record.
The following is a static ProDataSet definition for the above XML document:

```abl
DEFINE TEMP-TABLE Customer NO-UNDO
FIELD CustNum as INTEGER
FIELD Name as CHARACTER.

DEFINE TEMP-TABLE Order NO-UNDO
FIELD OrderNum AS INTEGER
FIELD OrderTotal AS DECIMAL
FIELD Customer_id AS RECID XML-NODE-TYPE "HIDDEN".

DEFINE DATASET CustomerOrders FOR Customer, Order
PARENT-ID-RELATION Relation1 FOR Customer,
Order PARENT-ID-FIELD Customer_id.
```

The ABL code to create the data for the CustomerOrders ProDataSet and write out the XML is as follows:

```abl
CREATE Customer.
ASSIGN CustNum = 1
   Name = "Lift Tours".

CREATE Order.
ASSIGN OrderNum = 100
   OrderTotal = 1234.89
   Customer_id = RECID(Customer).

CREATE Order.
ASSIGN OrderNum = 150
   OrderTotal = 999.99
   Customer_id = RECID(Customer).

CREATE Customer.
ASSIGN CustNum = 3
   Name = "Hoops".

CREATE Order.
ASSIGN OrderNum = 200
   OrderTotal = 1899.99
   Customer_id = RECID(Customer).

```

**XML schema with root element mapping to a temp-table**

The following example illustrates the functionality of creating a DATASET definition from an XML schema where the root element is mapped to a temp-table.

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="person">
    <complexType>
      <sequence>
        <element name="name" type="xsd:string" />
        <element name="children">
          <complexType>
            <sequence>
              <element name="child" type="xsd:string" maxOccurs="unbounded"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>  
</schema>
```
In the above XML Schema:

- The root element, person, is mapped to a temp-table with a single field, name.
- The nested child element, children, has no fields defined.
- The children element has a nested element, child, which maps to a temp-table with a single TEXT field.

The following is the static ProDataSet definition for the XML Schema:

```abl
DEFINE TEMP-TABLE person NO-UNDO
   FIELD name AS CHARACTER.

DEFINE TEMP-TABLE children NO-UNDO
   FIELD person_id as RECID XML-NODE-TYPE "Hidden".

DEFINE TEMP-TABLE child NO-UNDO
   FIELD child_Text AS CHARACTER XML-NODE-TYPE "Text"
   FIELD children_id AS RECID XML-NODE-TYPE "Hidden".

DEFINE DATASET personDset XML-NODE-TYPE "Hidden"
   FOR person, children, child
   PARENT-ID-RELATION Relation1 FOR person, children PARENT-ID-FIELD person_id
```

The bproxsdt04gl utility defines ProDataSet named `personDset`, with temp-tables `person` and `children`. `READ-XML()` and `READ-XMLSCHEMA()` creates the dynamic equivalent of this static definition.

Since `personDset` is XML-NODE-TYPE "Hidden", the `<personDset>` element is not written during `WRITE-XML()` call. The root element of the XML document is `<person>`.

The ABL code required to create the data for `personDset` ProDataSet and generate the `person.xml` document is as follows:

```ABL
CREATE person.
   ASSIGN name = "Ken".

CREATE children.
   ASSIGN person_id = RECID(person).

CREATE child.
   ASSIGN child_Text = "Adam"
      Children_id = RECID(children).

CREATE child.
   ASSIGN child_Text = "Elana"
      Children_id = RECID(children).

```
The `person.xml` XML file generated is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <name>Ken</name>
  <children>
    <child>Adam</child>
    <child>Elana</child>
  </children>
</person>
```

**XML schema with no explicit field definition for a parent table**

The following example demonstrates the functionality of creating a DATASET definition from an XML schema where a parent temp-table contains no field definitions. The following XML schema is structured as a ProDataSet:

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="Air_FlightInfo">
    <complexType>
      <sequence>
        <element name="generalFlightInfo">
          <complexType>
            <sequence>
              <element name="flightDate" minOccurs="0">
                <complexType>
                  <sequence>
                    <element name="departureDate" type="xsd:date"/>
                    <element name="departureTime" type="xsd:string"/>
                    <element name="arrivalDate" type="xsd:date"/>
                    <element name="arrivalTime" type="xsd:string"/>
                  </sequence>
                </complexType>
              </element>
              <element name="flightIdentification" minOccurs="0">
                <complexType>
                  <sequence>
                    <element name="flightNumber" type="xsd:string"/>
                    <element name="operationalSuffix" type="xsd:string"/>
                  </sequence>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

This XML schema is structured like a ProDataSet, `Air_FlightInfo` containing three temp-tables:

- `generalFlightInfo`, with no fields defined
- `flightDate` nested with `generalFlightInfo`, with 4 fields: `departureDate`, `departureTime`, `arrivalDate`, and `arrivalTime`
• flightIdentification nested within generalFlightInfo, with two fields: flightNumber, and operationalSuffix

The following is the static ProData definition for the XML schema:

```
DEFINE TEMP-TABLE generalFlightInfo NO-UNDO
   FIELD generalFlightInfo_field AS INTEGER XML-NODE-TYPE "Hidden".

DEFINE TEMP-TABLE flightDate NO-UNDO
   FIELD departureDate AS DATE
   FIELD departureTime AS CHARACTER
   FIELD arrivalDate AS DATE
   FIELD arrivalTime AS CHARACTER
   FIELD generalFlightInfo_id AS RECID XML-NODE-TYPE "Hidden".

DEFINE TEMP-TABLE flightIdentification NO-UNDO
   FIELD flightNumber AS CHARACTER
   FIELD operationalSuffix AS CHARACTER
   FIELD generalFlightInfo_id AS RECID XML-NODE-TYPE "Hidden".

DEFINE DATASET Air_FlightInfo
   FOR generalFlightInfo, flightDate, flightIdentification
   PARENT-ID-RELATION Relation1 FOR generalFlightInfo, flightDate
      PARENT-ID-FIELD generalFlightInfo_id
   PARENT-ID-RELATION Relation2 FOR generalFlightInfo, flightIdentification
      PARENT-ID-FIELD generalFlightInfo_id.
```

The top-level table, generalFlightInfo, does not have fields defined. Therefore, bproxsdo4gl and READ-XMLSHEMA( ) adds an integer <tablename_field>, in this case generalFlightInfo_field, to the generalFlightInfo temp-table definition. READ-XML() or READ-XMLSHEMA() methods on the XML Schema will create the dynamic equivalent of this static definition.

The following is the ABL code required to create the data for the Air_FlightInfo dataset:

```
CREATE generalFlightInfo.
CREATE flightDate.
ASSIGN flightDate.departureDate = 12/2/2010
   flightDate.departureTime = "13:15"
   flightDate.arrivalDate = 12/2/2010
   flightDate.arrivalTime = "16:45"
   flightDate.generalFlightInfo_id = RECID(generalFlightInfo).

CREATE flightIdentification.
ASSIGN flightIdentification.flightNumber = "1024"
   flightIdentification.operationalSuffix = "ABC"
   flightIdentification.generalFlightInfo_id = RECID(generalFlightInfo).

```

The following is the XML output:

```
<?xml version="1.0" encoding="UTF-8"?>
<Air_FlightInfo>
   <generalFlightInfo>
      <flightDate>
         <departureDate>2010-12-02</departureDate>
         <departureTime>13:15</departureTime>
         <arrivalDate>2010-12-02</arrivalDate>
      </flightDate>
   </generalFlightInfo>
</Air_FlightInfo>
```
XML schema with parent table fields after nested child tables

The following example demonstrates the functionality provided to create a DATASET definition from an XML schema where the parent fields of a temp-table are defined after the nested child temp-tables:

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="ClientInfo">
    <complexType>
      <sequence>
        <element name="Individual" minOccurs="0">
          <complexType>
            <sequence>
              <element name="LastName" type="xsd:string"/>
              <element name="FirstName" type="xsd:string"/>
              <element name="BirthDate" type="xsd:date"/>
              <element name="Gender" type="xsd:string"/>
            </sequence>
          </complexType>
        </element>
        <element name="Organization" minOccurs="0">
          <complexType>
            <sequence>
              <element name="OrgName" type="xsd:string"/>
              <element name="FedBN" type="xsd:string"/>
              <element name="ProvBN" type="xsd:string"/>
            </sequence>
          </complexType>
        </element>
        <element name="Language" type="xsd:string"/>
        <element name="Email" type="xsd:string"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

In the above schema:
- The root element, ClientInfo, maps to a temp-table with a two fields, Language and Email. These fields are in the sequence after the nested child definitions.
- ClientInfo has two nested child temp-table definitions
  - Individual, with four fields: LastName, FirstName, BirthDate, and Gender.
  - Organization, with three fields: OrgName, FedBN, and ProvBN.
The following is the static ProDataSet definition for the XML schema:

```
DEFINE TEMP-TABLE ClientInfo NO-UNDO
FIELD Language AS CHARACTER
FIELD Email AS CHARACTER.

DEFINE TEMP-TABLE Individual NO-UNDO
FIELD LastName AS CHARACTER
FIELD FirstName AS CHARACTER
FIELD BirthDate AS CHARACTER
FIELD Gender AS CHARACTER
FIELD ClientInfo_id AS RECID XML-NODE-TYPE "HIDDEN".

DEFINE TEMP-TABLE Organization NO-UNDO
FIELD OrgName AS CHARACTER
FIELD FedBN AS CHARACTER
FIELD ProvBN AS CHARACTER
FIELD ClientInfo_id AS RECID XML-NODE-TYPE "HIDDEN".

DEFINE DATASET ClientInfoDset XML-NODE-TYPE "HIDDEN"
FOR ClientInfo, Individual, Organization
PARENT-ID-RELATION Relation1 FOR ClientInfo, Individual
    PARENT-ID-FIELD ClientInfo_id
PARENT-ID-RELATION Relation2 FOR ClientInfo, Organization
    PARENT-ID-FIELD ClientInfo_id
    PARENT-FIELDS-AFTER (Language, Email).
```

The `READ-XML()` and `READ-XMLSCHEMA()` methods create the dynamic equivalent of this static definition.
XML schema with parent table fields interspersed among child tables

The following example demonstrates the functionality provided to create a DATASET definition from an XML schema where the parent fields of a temp-table are defined both before and after the nested child temp-tables:

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="Buy">
    <complexType>
      <sequence>
        <element name="TrxnTyp" type="xsd:string"/>
        <element name="Amt">
          <complexType>
            <sequence>
              <element name="AmtType" type="xsd:string"/>
              <element name="AmtValue" type="xsd:string"/>
            </sequence>
          </complexType>
        </element>
        <element name="TaxCode" type="xsd:string"/>
        <element name="ClientComm" type="xsd:string"/>
        <element name="Payment" minOccurs="0">
          <complexType>
            <sequence>
              <element name="PayType" type="xsd:string"/>
              <element name="Currency" type="xsd:string"/>
            </sequence>
          </complexType>
        </element>
        <element name="Comments" type="xsd:string"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

In the above schema:

- The root element, `Buy`, maps to a temp-table with four fields: `TrxnTyp`, `TaxCode`, `ClientComm`, and `Comments`.
- `Buy` has two nested child temp-table definitions:
  - Amt, with two fields: `AmtType` and `AmtValue`
  - Payment, with two fields: `PayType` and `Currency`
- The `TrxnTyp` field from the `Buy` temp-table is defined before the nested Amt temp-table definition.
- The `TaxCode` and `ClientComm` fields from the `Buy` temp-table are defined after the nested Amt temp-table.
- The `Comments` field from the `Buy` temp-table is defined after the nested Payment temp-table definition.
The following is the static ProDataSet definition for the XML schema:

```abl
DEFINE TEMP-TABLE Buy NO-UNDO
FIELD TrxnTyp AS CHARACTER
FIELD TaxCode AS CHARACTER
FIELD ClientComm AS CHARACTER
FIELD Comments AS CHARACTER.

DEFINE TEMP-TABLE Amt NO-UNDO
FIELD AmtType AS CHARACTER
FIELD AmtValue AS CHARACTER
FIELD Buy_id AS RECID XML-NODE-TYPE "HIDDEN".

DEFINE TEMP-TABLE Payment NO-UNDO
FIELD PayType AS CHARACTER
FIELD Currency AS CHARACTER
FIELD Buy_id AS RECID XML-NODE-TYPE "HIDDEN".

DEFINE DATASET BuyDset XML-NODE-TYPE "HIDDEN"
FOR Buy, Amt, Payment
PARENT-ID-RELATION Relation1 FOR Buy, Amt
   PARENT-ID-FIELD Buy_id
   PARENT-FIELDS-BEFORE (TrxnTyp)
   PARENT-FIELDS-AFTER (TaxCode, ClientComm)
PARENT-ID-RELATION Relation2 FOR Buy, Payment
   PARENT-ID-FIELD Buy_id
   PARENT-FIELDS-AFTER (Comments).
```

READ-XML() or READ-XMLSCHEMA() creates the dynamic equivalent of this static definition.

Since the Relation1 PARENT-ID-RELATION contains a PARENT-FIELDS-BEFORE and PARENT-FIELDS-AFTER phrase, the TrxnTyp fields of a Buy temp-table record is written to XML before any Amt child records, and TaxCode and ClientComm fields are written to XML after all Amt child records.

Since the Relation2 PARENT-ID-RELATION contains a PARENT-FIELDS-AFTER phrase, the Comments field of a Buy temp-table record is written to XML after all Payment child records.

**ABL representation of temp-tables and ProDataSets as XML Schema**

**Note:** The following information is advanced and optional. You do not need to understand any of this to successfully use the features described in this chapter.

The XML Schema for a temp-table consists of a complex type definition for the fields (a sequence of field definitions), followed by any index definitions. The XML Schema for a ProDataSet consists of a complex type definition for the member temp-table buffers (a sequence of temp-table complex type definitions) followed by any index and data-relation definitions. ABL provides XML Schema extensions in order to fully describe the definitions of the temp-table or ProDataSet in XML Schema.

By default, ABL XML Schema extensions are present in temp-table and ProDataSet XML Schema documents. If you do not want the ABL XML Schema extensions, set the min-xmlschema option on the methods to TRUE. The XML Schema will then omit most of the ABL-specific XML Schema attributes.
Namespaces

The following table lists important namespaces that are used in ABL-generated XML Schema documents.

Table 26: Important namespaces

<table>
<thead>
<tr>
<th>ABL-used prefix</th>
<th>Namespace URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>The namespace for the standard XML Schema 1.0 tags, which is often referred to as the XML Schema namespace</td>
</tr>
<tr>
<td>xsi</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
<td>The namespace for XML Schema 1.0 instance attributes, which is often referred to as XML Schema-Instance namespace</td>
</tr>
<tr>
<td>prodata</td>
<td>urn:schemas-progress-com:xml-prodata:0001</td>
<td>The namespace for ABL-specific XML Schema extensions associated with the XML features described in this chapter</td>
</tr>
</tbody>
</table>

ABL-specific attributes on temp-table and ProDataSet definitions

The following attributes will appear on the outer element definition for a temp-table or ProDataSet:

- **datasetName** — Identifies the ProDataSet name when the corresponding XML element name is different, as defined by the ABL XML-NODE-NAME or SERIALIZE-NAME attributes.
- **fieldName** — Identifies the ABL temp-table field name when the corresponding XML element name is different, as defined by the ABL XML-NODE-NAME or SERIALIZE-NAME attributes.
- **notActive** — Identifies that the data relation is not active. This attribute supports ProDataSets that have multiple data relations defined with the intent that certain data relations are active for certain contexts and not active for others.
- **prefix** — Identifies the NAMESPACE-PREFIX for the temp-table or ProDataSet object.
- **proDataSet** — Identifies the XML Schema as a ProDataSet XML Schema.
- **proTempTable** — Identifies the XML Schema as a temp-table XML Schema.
- **recursive** — Identifies that the data relation is recursive.
- **tableName** — Identifies the ABL temp-table name or temp-table buffer name when the corresponding XML element name is different, as defined by the ABL XML-NODE-NAME or SERIALIZE-NAME attributes.
Representing temp-tables in XML Schema

Temp-tables are represented with an XML Schema element definition, followed by a complex type definition defining the fields. The XML Schema name attribute identifies the name of the temp-table. The following ABL-specific XML Schema attributes further describe the temp-table:

- **undo** — Identifies the temp-table as UNDO
- **beforeTable** — Specifies the name of any before-image table for this temp-table
- **bufferName** — If the XML Schema is written for a non-default buffer of a temp-table, this attribute specifies the buffer name

Temp-table fields can be represented as elements, attributes, or text in an XML Document. The XML-NODE-TYPE attribute on the buffer-field object lets you control how fields are represented in the XML document and the XML Schema.

**Note:** The XML-NODE-TYPE attribute interacts with the SERIALIZE-HIDDEN attribute. For more information, see XML-NODE-TYPE and SERIALIZE-HIDDEN on page 123.

XML Schema provides the following built-in attributes for minimally describing a temp-table field in an element definition (when the buffer-field has an XML-NODE-TYPE of "ELEMENT"):

- **name** — Name of the field.
- **type** — Data type of the field.
- **minOccurs** — If this is set to 0, then the field is optional and may not appear in the XML document. If the omit-initial-values argument on one of the write methods was used and the current value of an instance is the initial value, then the field is omitted. In this case, the initial value is stored in the xsd:default attribute.
- **maxOccurs** — Extent of array fields.
- **nillable** — Can be null or unknown (always has a value of "true" for temp-table fields). Unknown fields have the xsi:nil attribute set to "true" in the XML instance.
- **default** — The initial value for the field. If the field has the default initial value for it's data type (for example, 0 for integer), the attribute will not be present in the XML Schema.

XML Schema provides the following attributes for attribute definitions (when the buffer-field has a XML-NODE-TYPE of "ATTRIBUTE"):

- **name** — Name of the field.
- **type** — Data type of the field.
- **default** — The initial value for the field. If the field has the default initial value for it's data type (for example, 0 for integer), the attribute will not be present in the XML Schema.

XML Schema provides the following attribute for simple content definitions (when the Buffer-field has a XML-NODE-TYPE of "TEXT"):

- **type** — Data type of the field.
Note: If you want to be able to explicitly set a field to the Unknown value (?)(xsi:nil="true"), it must have XML-NODE-TYPE = "ELEMENT". Array fields must have XML-NODE-TYPE = "ELEMENT" or "HIDDEN“. Buffer fields with XML-NODE-TYPE="TEXT" cannot be part of a nested DATA-RELATION.

ABL provides the following XML Schema extensions for describing ABL-specific attributes of a temp-table field. These attributes, with the exception of dataType and initial, will only appear in the XML Schema when the min-xmlschema option for WRITE-XML( ) method or WRITE-XMLSCHEMA( ) method is FALSE.
### Table 27: ABL XML Schema extensions for temp-table fields

<table>
<thead>
<tr>
<th>XML Schema attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caseSensitive</td>
<td>Case-sensitive for CHARACTER fields.</td>
</tr>
<tr>
<td>columnCodepage</td>
<td>Column-codepage for CLOB fields.</td>
</tr>
<tr>
<td>columnLabel</td>
<td>Column-label for the field.</td>
</tr>
<tr>
<td>dataType</td>
<td>The ABL data type for fields where the XML Schema type attribute maps to multiple ABL data types (for example, string, base64Binary, dateTime). See XML Schema and ABL Data Type Mappings on page 185.</td>
</tr>
<tr>
<td>decimals</td>
<td>Number of decimal places for DECIMAL fields.</td>
</tr>
<tr>
<td>format</td>
<td>Display format for the field.</td>
</tr>
<tr>
<td>fieldName</td>
<td>Name of the field. It is used for fields with XML-NODE-TYPE of &quot;TEXT&quot;. It also identifies the ABL temp-table field name when the corresponding XML element name is different, as defined by the ABL XML-NODE-NAME or SERIALIZE-NAME attributes.</td>
</tr>
<tr>
<td>help</td>
<td>Help text for the field.</td>
</tr>
<tr>
<td>initial</td>
<td>Initial value for the field. Used for fields with XML-NODE-TYPE of &quot;TEXT&quot;, and for fields with initial value of TODAY, NOW or UNKNOWN.</td>
</tr>
<tr>
<td>label</td>
<td>Label for the field.</td>
</tr>
<tr>
<td>readOnly</td>
<td>Read-only attribute for the field.</td>
</tr>
<tr>
<td>userOrder</td>
<td>Position of the field in the temp-table. This attribute is used only when there's a mix of fields with XML-NODE-TYPE of &quot;ELEMENT&quot; and &quot;ATTRIBUTE&quot; in the temp-table.</td>
</tr>
</tbody>
</table>

**Note:** If any of the above ABL-specific field attributes have the ABL default value, WRITE-XML( ) method and WRITE-XMLSCHEMA( ) method will omit the attribute from the element or attribute definition when it writes the XML Schema.
Representing indexes in XML Schema

The following ABL-specific XML Schema attributes are added to help fully describe a temp-table index.

- **primaryIndex** — Indicates the primary index for the temp-table
- **wordIndex** — Identifies an index as a word index
- **descending** — Indicates descending order for a component of the index

Unique temp-table indexes are represented with the XML Schema unique definition, as shown in the syntax below:

```xml
<xsd:unique name="IndexName" prodata:primaryIndex="BooleanValue">
  <xsd:selector xpath="TableName" />
  <xsd:field xpath="FieldName" prodata:descending="BooleanValue"/>
  ...
  <xsd:field xpath="FieldName" prodata:descending="BooleanValue"/>
</xsd:unique>
```

Non-unique temp-table indexes are represented with the ABL-specific index definition, as shown in the following syntax:

```xml
<prodata:index name="IndexName" prodata:primaryIndex="BooleanValue">
  <prodata:table name="TableName" />
  <prodata:field name="FieldName" prodata:descending="BooleanValue"/>
  ...
  <prodata:field name="FieldName" prodata:descending="BooleanValue"/>
</prodata:index>
```

Representing data-relations in XML Schema

In a ProDataSet, you form an association between two member temp-tables by creating a DATA-RELATION between the parent and child temp-table. ABL-specific XML Schema attributes for data relations are present in the XML Schema whether the min-xmlschema option on the WRITE-XML() or WRITE-XMLSCHEMA() method is TRUE or FALSE.

The following ABL-specific XML Schema attributes are added to help fully describe data-relations:

- **foreignKeyHidden** — Indicates that child rows omit foreign key fields when the data-relation is defined as a nested relation.
- **nested** — Indicates that child rows are nested within parent rows in the XML document.
- **notActive** — Indicates a data relation that is not currently active. This attribute supports ProDataSets that have multiple data relations defined with the intent that certain data relations are active for certain contexts and not active for others.
- **recursive** — Indicates a recursive data-relation.
- **reposition** — Indicates a REPOSITION data-relation.
If the column or columns from the parent temp-table are also the components in a unique index, the data-relation is represented with the combination of XML Schema `unique` (unique index definition) and `keyref` definitions. Here is the syntax for `keyref`:

```xml
<xsd:key ref="RelationName" refer="UniqueIndexName"
    prodata:nested="BooleanValue">
    <xsd:selector xpath="ChildTableName" />
    <xsd:field xpath="FieldName" />
    ...
    <xsd:field xpath="FieldName" />
</xsd:keyref>
```

If the column or columns from the parent temp-table are not components in a unique index, the data-relation is represented with the ABL-specific relation definition. Here is the syntax for relation:

```xml
<prodata:relation name="RelationName"
    prodata:nested="BooleanValue"
    prodata:reposition="BooleanValue"
    prodata:parent="ParentTableName"
    prodata:child="ChildTableName"
    prodata:relationFields="RelationFields" />
```
XML Schema and ABL Data Type Mappings

This appendix contains reference tables for data type mappings. For details, see the following topics:

- Default mappings between ABL data types and XML Schema data types
- Default mappings from other XML Schema data types to ABL data types

Default mappings between ABL data types and XML Schema data types

The following table shows the default mapping between ABL data types and XML Schema data types used with the read and write XML methods of temp-tables and ProDataSets. The third column shows the ABL-specific XML Schema data type attribute which is an extension to the basic XML Schema. This attribute is used for ABL data types that are not the default for the XML Schema data type.

The table contains the most common XML Schema data types and their corresponding ABL data types.

Table 28: ABL data types mapped to XML Schema data types

<table>
<thead>
<tr>
<th>ABL type</th>
<th>XML Schema type attribute</th>
<th>ABL-specific XML Schema data type attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>base64Binary</td>
<td>prodata:blob</td>
</tr>
</tbody>
</table>
## Appendix A: XML Schema and ABL Data Type Mappings

<table>
<thead>
<tr>
<th>ABL type</th>
<th>XML Schema type attribute</th>
<th>ABL-specific XML Schema data type attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>string</td>
<td>—</td>
</tr>
<tr>
<td>CLOB</td>
<td>string</td>
<td>prodata:clob</td>
</tr>
<tr>
<td>COM-HANDLE</td>
<td>long</td>
<td>prodata:comHandle</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
<td>—</td>
</tr>
<tr>
<td>DATETIME</td>
<td>dateTime</td>
<td>prodata:dateTime</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>dateTime</td>
<td>—</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>decimal</td>
<td>—</td>
</tr>
<tr>
<td>INT64</td>
<td>long</td>
<td>—</td>
</tr>
</tbody>
</table>

**Note:** Prior to OpenEdge Version 10.1B, xsd:long mapped to ABL DECIMAL. This mapping is still treated as valid in ABL.

<table>
<thead>
<tr>
<th>ABL type</th>
<th>XML Schema type attribute</th>
<th>ABL-specific XML Schema data type attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER (32 bit)</td>
<td>int</td>
<td>—</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>boolean</td>
<td>—</td>
</tr>
<tr>
<td>RAW</td>
<td>base64Binary</td>
<td>—</td>
</tr>
<tr>
<td>RECID (32 or 64 bit)</td>
<td>long</td>
<td>prodata:recid</td>
</tr>
<tr>
<td>ROWID</td>
<td>base64Binary</td>
<td>prodata:rowid</td>
</tr>
<tr>
<td>HANDLE/ WIDGET-HANDLE</td>
<td>long</td>
<td>prodata:handle</td>
</tr>
</tbody>
</table>

**Note:** The -noint64 startup parameter forces ABL to use the pre-10.1B mapping of xsd:long to ABL DECIMAL. You have some control over default data type mappings. The `READ-XML()` and `READ-XMLSHEMA()` methods have `override-default-mapping` and `field-type-mapping` options. See Reading and Writing XML Data from Temp-Tables and ProDataSets on page 117.
Default mappings from other XML Schema data types to ABL data types

There are many other XML Schema data types that could appear in an XML Schema definition. ABL maps these data types but also preserves the original XML Schema type in the XML-DATA-TYPE attribute of the buffer-field object. When the WRITE-XML( ) and WRITE-XMLSCHEMA( ) methods encounter buffer-fields with these values, they write the original XML Schema data type out to the XML Schema.

The following table lists the other XML Schema data types and the ABL data type they map to.

**Table 29: Other XML Schema types and ABL data type mappings**

<table>
<thead>
<tr>
<th>XML Schema Type (type) Attribute</th>
<th>ABL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyURI</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>double</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>duration</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>ENTITY</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>float</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>gDay</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>gMonth</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>gYear</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>hexBinary</td>
<td>RAW</td>
</tr>
<tr>
<td>ID</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>IDREF</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>IDREFS</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>integer</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>language</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>XML Schema Type (type) Attribute</td>
<td>ABL Type</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Name</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>NCName</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NM_TOKEN</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>NM_TOKENS</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>normalizedString</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>NOTATION</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>QName</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>short</td>
<td>INTEGER</td>
</tr>
<tr>
<td>time</td>
<td>INTEGER</td>
</tr>
<tr>
<td>token</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
ABL to IANA Code Page Mappings

For details, see the following topics:

- ABL to IANA code page mappings

ABL to IANA code page mappings

When using ABL features that allow you to write XML documents, you may need to specify an encoding, which is the name of a codepage. In these cases, you must use the Internet Assigned Numbers Authority (IANA) name for the codepage and not the ABL name for the codepage. UTF-8 is the default code page. The following table lists the ABL and IANA equivalents.

Table 30: ABL code pages and corresponding IANA encodings

<table>
<thead>
<tr>
<th>ABL code page name</th>
<th>IANA encoding name</th>
</tr>
</thead>
<tbody>
<tr>
<td>620-2533</td>
<td>TIS-620</td>
</tr>
<tr>
<td>1250</td>
<td>windows-1250</td>
</tr>
<tr>
<td>1251</td>
<td>windows-1251</td>
</tr>
<tr>
<td>1252</td>
<td>windows-1252</td>
</tr>
<tr>
<td>1253</td>
<td>windows-1253</td>
</tr>
</tbody>
</table>
## Appendix B: ABL to IANA Code Page Mappings

<table>
<thead>
<tr>
<th>ABL code page name</th>
<th>IANA encoding name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1254</td>
<td>windows-1254</td>
</tr>
<tr>
<td>1255</td>
<td>windows-1255</td>
</tr>
<tr>
<td>1256</td>
<td>windows-1256</td>
</tr>
<tr>
<td>1257</td>
<td>windows-1257</td>
</tr>
<tr>
<td>1258</td>
<td>windows-1258</td>
</tr>
<tr>
<td>All</td>
<td>US-ASCII</td>
</tr>
<tr>
<td>Big5</td>
<td>Big5</td>
</tr>
<tr>
<td>CP936</td>
<td>GBK</td>
</tr>
<tr>
<td>EUCJIS</td>
<td>EUC-JP</td>
</tr>
<tr>
<td>GB18030</td>
<td>GB18030</td>
</tr>
<tr>
<td>GB2312</td>
<td>GB_2312-80</td>
</tr>
<tr>
<td>IBM037</td>
<td>IBM037</td>
</tr>
<tr>
<td>IBM273</td>
<td>IBM273</td>
</tr>
<tr>
<td>IBM277</td>
<td>IBM277</td>
</tr>
<tr>
<td>IBM278</td>
<td>IBM278</td>
</tr>
<tr>
<td>IBM284</td>
<td>IBM284</td>
</tr>
<tr>
<td>IBM297</td>
<td>IBM297</td>
</tr>
<tr>
<td>IBM437</td>
<td>IBM437</td>
</tr>
<tr>
<td>IBM500</td>
<td>IBM500</td>
</tr>
<tr>
<td>IBM850</td>
<td>IBM850</td>
</tr>
<tr>
<td>IBM851</td>
<td>IBM851</td>
</tr>
<tr>
<td>IBM852</td>
<td>IBM852</td>
</tr>
<tr>
<td>IBM857</td>
<td>IBM857</td>
</tr>
<tr>
<td>IBM858</td>
<td>IBM00858</td>
</tr>
<tr>
<td>IBM861</td>
<td>IBM861</td>
</tr>
<tr>
<td>ABL code page name</td>
<td>IANA encoding name</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>IBM862</td>
<td>IBM862</td>
</tr>
<tr>
<td>IBM866</td>
<td>IBM866</td>
</tr>
<tr>
<td>ISO8859-1</td>
<td>ISO-8859-1</td>
</tr>
<tr>
<td>ISO8859-2</td>
<td>ISO-8859-2</td>
</tr>
<tr>
<td>ISO8859-3</td>
<td>ISO-8859-3</td>
</tr>
<tr>
<td>ISO8859-4</td>
<td>ISO-8859-4</td>
</tr>
<tr>
<td>ISO8859-5</td>
<td>ISO-8859-5</td>
</tr>
<tr>
<td>ISO8859-6</td>
<td>ISO-8859-6</td>
</tr>
<tr>
<td>ISO8859-7</td>
<td>ISO-8859-7</td>
</tr>
<tr>
<td>ISO8859-8</td>
<td>ISO-8859-8</td>
</tr>
<tr>
<td>ISO8859-9</td>
<td>ISO-8859-9</td>
</tr>
<tr>
<td>ISO8859-15</td>
<td>ISO-8859-15</td>
</tr>
<tr>
<td>KOI8-R</td>
<td>KOI8-R</td>
</tr>
<tr>
<td>KSC5601</td>
<td>KS_C_5601-1987</td>
</tr>
<tr>
<td>ROMAN-8</td>
<td>hp-roman8</td>
</tr>
<tr>
<td>SHIFT-JIS</td>
<td>Shift_JIS</td>
</tr>
<tr>
<td>UTF-8</td>
<td>UTF-8</td>
</tr>
<tr>
<td>UTF-16</td>
<td>UTF-16</td>
</tr>
<tr>
<td>UTF-32</td>
<td>UTF-32</td>
</tr>
</tbody>
</table>
Appendix B: ABL to IANA Code Page Mappings
Command and Utility Reference

This appendix contains the standard reference documentation for utilities closely associated with topics found in this book. For details, see the following topics:

- bproxsdto4gl utility
bproxsdto4gl utility

The bproxsdto4gl tool is a command line utility that extracts the definition of a table or data set from an XML Schema file (.xsd file) and converts it to a static ABL temp-table or static ProDataSet definition. By default, the definition is saved as an ABL include file (.i file).

```
bproxsdto4gl
  [-h]
  { xml-schema-filename 
    [-output { output-directory-name | filename } ]
    [-element element-name ]
    [-overridemapping ]
    [-fieldtypemapping { field-mapping-file | string } ]
    [-noint64 ]
  }

-h

Displays a help message on the use of this utility.

xml-schema-filename

The name of the XML Schema file containing the schema to convert into an ABL definition. xml-schema-filename can contain absolute or relative path information.

-output { output-directory | filename }

Overrides the default behavior for the generated ABL include file. If not specified, the utility names the output file after the source .xsd file (xsd-filename.i) and saves the file to the current directory. To use the default name, but specify a different directory, supply an absolute or relative path. To change the name of the file, specify it and include any necessary absolute or relative path information.

-element element-name

The top-level element declaration (the element's name) from the XML Schema file that identifies the table or data set that will be the source of the conversion to an ABL temp-table or ProDataSet definition.

-overridemapping

Overrides the default data type mapping between XML Schema string and binary types when generating ABL field definitions from XML Schema. The XML Schema string data type maps to a CHARACTER field by default, and the XML Schema base64Binary and hexBinary data types map to a RAW field by default. If you specify overridemapping, XML Schema string will map to a CLOB field, and XML Schema base64Binary or hexBinary will map to a BLOB field.
-fieldtypemapping { field-mapping-file | field-mapping-string }

A string containing a comma-separated list of fieldname and data type pairs, or a file that contains the comma-separated list. This option allows you to specify the ABL data type for a specific field from the XML Schema. See XML Schema and ABL Data Type Mappings on page 185 for more information on the default data type mapping between XML Schema and ABL.

This is the syntax for the string or file containing the pairs:

```
fieldnamel, data-type1, fieldname2, data-type2 ...
```

If there are multiple buffers defined in the XML Schema, the `fieldname` must be qualified with the buffer name from the XML Schema. For example:

```
buffer-name.fieldname
```

`data-type` must be a valid ABL data type for a temp-table field. For example: CLOB, BLOB, DATETIME, and so on.

-noint64

Prior to OpenEdge Version 10.1B, the ABL INT64 data type did not exist and this utility mapped XML Schema types of xsd:long to the ABL DECIMAL data type. Use this option if you want to use the xsd:long to ABL DECIMAL mapping. Otherwise, xsd:long maps to INT64.
Index

A

ABL
  benefits of XML development 23
  developing XML-enabled applications 25
DOM interface 35
DOM support 25
SAX support 26–27
serializing XML data 27
ABL object names, mapping to XML names 163
ADD-SCHEMA-LOCATION method 52
APPEND-CHILD method 53
APPEND-CHILD method CREATE-NODE) method 52
ATTRIBUTE-NAMES attribute 53
attributes 45
attributes, setting 40
attributes, values 40
attributes, XML 34

B

bproxsdto4gl utility 194

C

callback procedures 26
Characters SAX callback 85
child element 33
CHILD-NUM attribute 53
CLONE-NODE method 54
Codepages
  encoding for serialization 155
CREATE-NODE method 48
CREATE-NODE-NAMESPACE method 48, 52

D

DELETE-NODE method 54
deleting objects 37
Document Object Model 25
Document object model (DOM)
  defined 35
Document Type Definition
  defined 33
Document type definition (DTD)
  defined 32
DOM
See also Document object model (DOM)
  ABL interface 35
  ABL support 25
  advantages 25

DOM (continued)
  and XML documents document 37
  attributes 40
  creating output 39
  defined 25
  error handling 50
  input example 46
  internationalization 50
  limits 25
  loading 43
  LONGCHAR 42
  MEMPTR 42
  namespaces 48
  node subtypes 38
  nodes 37
  output example 40
  prerequisites 35
  reading input 43
  root node 40
  stream 42
  validation 51
See also Document object model (DOM)
DTD, See Document type definition (DTD)>

E

element 33
empty element 35
ENCODING attribute 51
end tag 33
EndElement SAX callback 86
EndElement SAX callback 86
EndPrefixMapping SAX callback 87
entity 35
ENTITY-EXPANSION-LIMIT attribute 53
error messages
  SAX. 84
Error SAX callback 87
Extensible Markup Language, defined 21

F

FatalError SAX callback 87

G

GET-ATTRIBUTE method 53
GET-ATTRIBUTE-NODE method 53
GET-CHILD method 52–53
GET-DOCUMENT-ELEMENT method 52
GET-PARENT method 53
HANDLE attribute 53–54

I

IgnorableWhitespace SAX callback 88
IMPORT-NODE method 52
IN WIDGET-POOL option
  CREATE SAX-ATTRIBUTES statement 62
INITIALIZE-DOCUMENT-TYPE method 51
INSERT-BEFORE method 52–53
INSTANTIATING-PROCEDURE attribute 53–54

L

leaf element 33
LOAD method 52
LOCAL-NAME attribute 53
LONGCHAR-TO-NODE-VALUE method 54

M

Mapping ABL names to XML names 163
markup 21
MEMPTR-TO-NODE-VALUE method 54

N

NAME attribute 52, 54
NAMESPACE-PREFIX attribute 52–53
NAMESPACE-URI attribute 52–53
namespaces 35
NO-ERROR option
  CREATE SAX-ATTRIBUTES statement 62
NODE-VALUE attribute 54
NODE-VALUE-TO-LONGCHAR method 54
NODE-VALUE-TO-MEMPTR method 54
nodes
  accessing 44
  appending 40
  creating 40
  subtypes 38
NONAMESPACE-SCHEMA-LOCATION attribute 52
NORMALIZE method 54
NotationDecl SAX callback 88
NUM-CHILDREN attribute 52–53

O

OWNER-DOCUMENT attribute 54

P

parent element 33

parser 23, 35
ProcessingInstruction SAX callback 89
ProDataSets
  defining from XML Schema 194
ProDataSets and XML 27
PUBLIC-ID attribute 52

R

REMOVE-ATTRIBUTE method 53
REMOVE-CHILD method 52–53
REPLACE-CHILD method 52–53
ResolveEntity SAX callback 89
root element 32–33
root node 40

S

SAVE( ) method 53
SAX
  ABL support 27
  ABL support for 26
  advantages 26
  defined 26
  limits 27
  streaming 26
SAX callback procedures
  Characters 85
  EndDocument 86
  EndElement 86
  EndPrefixMapping 87
  Error 87
  FatalError 87
  IgnorableWhitespace 88
  NotationDecl 88
  ProcessingInstruction 89
  ResolveEntity 89
  StartDocument 91
  StartElement 91
  StartPrefixMapping 92
  UnparsedEntityDecl 92
  Warning 93
SAX error messages 84
SCHEMA-LOCATION attribute 52
SCHEMA-PATH attribute 52
Serialization
  codepages 155
  serialization use cases 28
service-oriented architecture 23
SET-ATTRIBUTE method 48, 53
SET-ATTRIBUTE-NODE method 48, 53
SOA 23
start tag 33
StartDocument SAX callback 91
StartElement SAX callback 91
StartPrefixMapping SAX callback 92
STRICT-ENTITY-RESOLUTION attribute 53
SUBTYPE attribute 52, 54
SUPPRESS-NAMESPACE-PROCESSING attribute 52
SYSTEM-ID attribute 52

T
tag 33
temp-tables
  defining from XML Schema 194
temp-tables and XML 27
TYPE attribute 53–54

U
UNIQUE-ID attribute 53–54
UnparsedEntityDecl SAX callback 92

W
W3C 22
Warning SAX callback 93
whitespace 33
WRITE-JSON( ) method
  syntax 155

X
X-document object
  and handles 37
  creating 37
  defined 37
  deleting 37
  lifecycle 37
X-noderef object
  accessing 44
  attributes and values 40, 45
  creating 38, 40
  defined 38
  deleting 38
  lifecycle 38
  root node 40
XML
  and messaging 23
  and ProDataSets 27
  and proxies 23

XML (continued)
  and SOA 23
  and temp-tables 27
  and Web services 23
  attributes 34
  basic learning resources 24
  benefits of 22
  body 32
  child 33
  defined 32
  definition of 21
  document structure 32
  documents 32
  element 33
  elements 33
  empty element 35
  end tag 33
  entity 35
  leaf 33
  markup 32
  namespaces 35
  nodes
    names and values 38
  parent 33
  Progress courses 24
  prolog 32
  role of 21
  root 33
  root element 32
  schema 22
  serialization features 28
  start tag 33
  tag 33
  terminology 32
  whitespace 33

XML file
  input 43
  loading 43
  output to MEMPTR 42
  output to stream 42

XML names, mapping to ABL names 163

XML Schema
  defined 22, 33
  XML Schema conversion 194
  XML Schema Definition Language 22
  XSD 22
  xsd conversion 194