
# Using XML Tools

Version 2019.2
2019-08-08

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

*Using XML Tools*
InterSystems IRIS Data Platform Version 2019.2 2019-08-08






For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

# Table of Contents

# List of Tables

# About This Book

This book describes, to programmers who are familiar with XML, how to use the InterSystems XML tools. It includes the following sections:

- Introduction to InterSystems XML Tools
- Writing XML Output from Objects
- Importing XML into Objects
- Representing an XML Document as a DOM
- Encrypting XML Documents
- Signing XML Documents
- Using %XML.TextReader
- Evaluating XPath Expressions
- Performing XSLT Transformations
- Customizing How the InterSystems SAX Parser Is Used
- Generating Classes from XML Schemas
- Generating XML Schemas from Classes
- Examining Namespaces and Classes
- XML Background

For a detailed outline, see the table of contents.

Also see the following sources:

- *Projecting Objects to XML* describes how to XML-enable your InterSystems IRIS™ classes so that you can use them with the tools described in this book.
- *Securing Web Services* describes how to secure InterSystems IRIS web services and web clients. This book contains information on prerequisites for digital signatures and XML encryption, as well as additional options not listed in this book.

# 1

# Introduction to InterSystems XML Tools

This book describes how to use InterSystems IRIS XML tools.

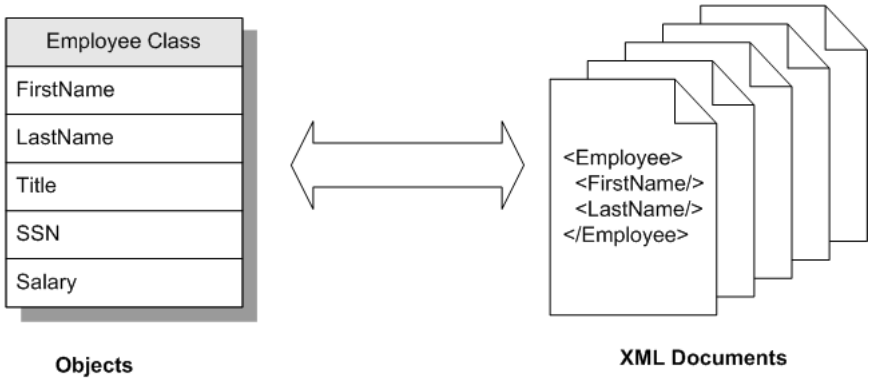
InterSystems IRIS brings the power of objects to XML processing — you can use objects as a direct representation of XML documents and vice versa. Because InterSystems IRIS includes a native object database, you can use such objects directly with a database. Furthermore, InterSystems IRIS provides tools for working with XML documents and DOMs (Document Object Model), even if these are *not* related to any InterSystems IRIS classes.

This chapter discusses the following topics:

- How to represent object data in XML documents and DOMs
- Creating arbitrary XML documents and DOMs
- Accessing data from XML documents and DOMs
- Modifying XML documents and DOMs
- The SAX Parser
- Additional XML tools in InterSystems IRIS
- Considerations to remember when using XML tools
- Standards supported in InterSystems IRIS

## 1.1 Representing Object Data in XML

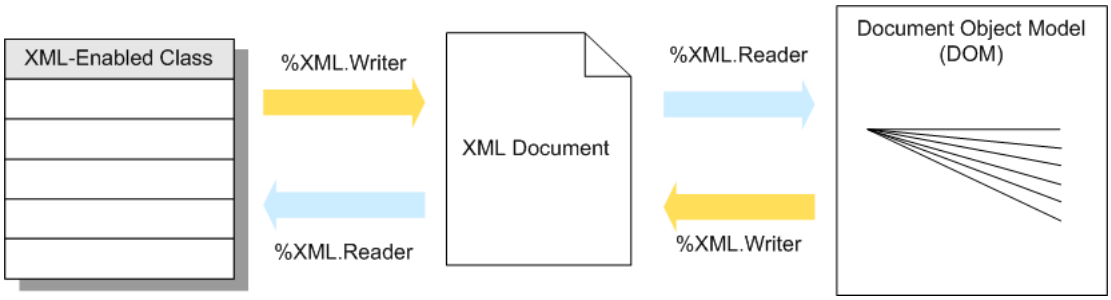
Some of the InterSystems IRIS XML tools are intended for use mainly with XML-enabled classes. To XML-enable a class, you add %XML.Adaptor to its superclass list. The %XML.Adaptor class enables you to represent instances of that class as XML documents. You add class parameters and property parameters to fine-tune the projection. Because there are so many options, an entire book is devoted to them: *Projecting Objects to XML*.




For an XML-enabled class, your data can be available in all the following forms:

- Contained in class instances. Depending on the class, the data can also possibly be saved to disk, where it is available in all the same ways as other persistent classes.
- Contained in XML documents, which could be files, streams, or other documents.
- Contained in a DOM (Document Object Model).

The following figure provides an overview of the tools that you use to convert your data among these forms:




The %XML.Writer class enables you to create XML documents. The output destination is typically a file or a stream. You identify the objects to include in the output, and the system generates output based on the rules established within the class definitions. For information, see "Writing XML Output from Objects."

The %XML.Reader class enables you to import a suitable XML document into a class instance. The source is typically a file or a stream. To use this class, you specify a correlation between a class name and an element contained in the XML document. The given element must have the structure that is expected by the corresponding class. Then you read the document, node by node. When you do so, the system creates in-memory instances of that class, containing the data found in the XML document. For information, see "Importing XML into InterSystems IRIS Objects."

A DOM is also a useful way to work with an XML document. You can use the %XML.Reader class to read an XML document and create a DOM that represents it. In this representation, the DOM is a series of nodes, and you navigate among them as needed. Specifically, you create an instance of %XML.Document, which represents the document itself and which contains the nodes. Then you use %XML.Node to examine and manipulate the nodes. You can use %XML.Writer to write the XML document again, if needed. For information, see "Representing an XML Document as a DOM."

The InterSystems IRIS XML tools provide many ways to access data in and modify both XML documents and DOMs.

# 1.2 Creating Arbitrary XML

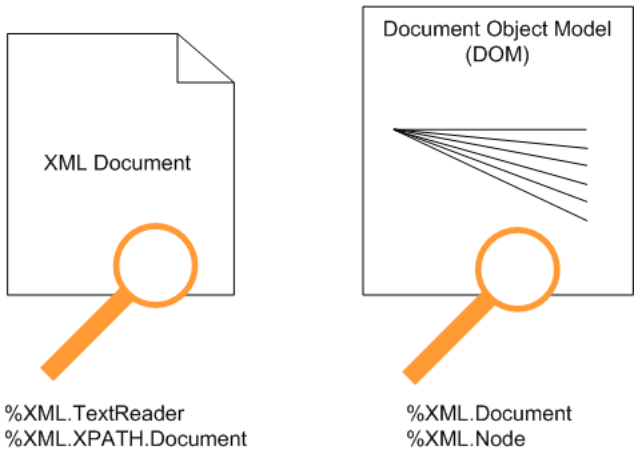
You can also use InterSystems IRIS XML tools to create and work with arbitrary XML — that is, XML that does not map to any InterSystems IRIS class. To create an arbitrary XML document, use %XML.Writer. This class provides methods for adding elements, adding attributes, adding namespace declarations, and so on.

To create an arbitrary DOM, use %XML.Document. This class provides a class method that returns a DOM with a single empty node. Then use instance methods of that class to add nodes as needed.

Or use %XML.Reader to read an arbitrary XML document and then create a DOM from that document.

# 1.3 Accessing Data

The InterSystems IRIS XML tools provide several ways to access data in XML form. The following figure shows a summary:



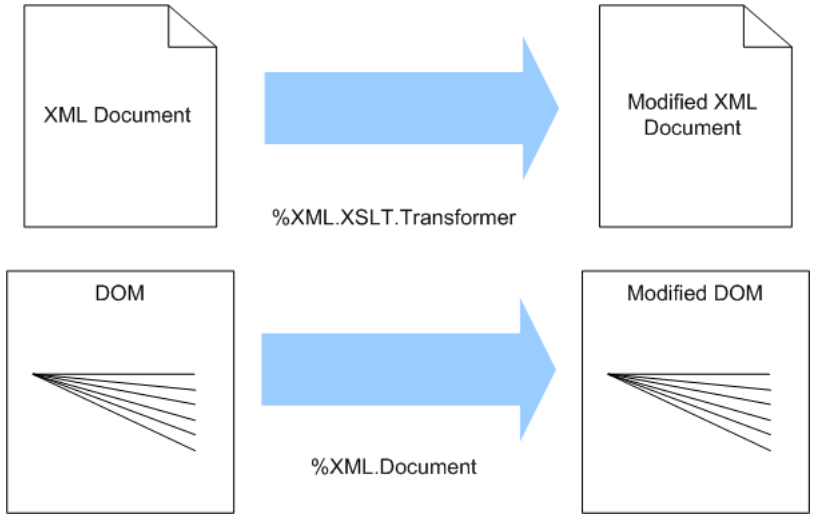

For any well-formed XML document, you can use the following classes to work with data in that document:

- %XML.TextReader — You can use this to read and parse a document, node by node. See "Using %XML.TextReader."
- %XML.XPATH.Document — You can use this to obtain data, by using an XPATH expression that refers to a specific node in the document. See "Evaluating XPath Expressions."

In InterSystems IRIS, a DOM is an instance of %XML.Document. This instance represents the document itself and contains the nodes. You use the properties and methods of this class to retrieve values from the DOM. You use %XML.Node to examine and manipulate the nodes. For information, see "Representing an XML Document as a DOM."

# 1.4 Modifying XML

The InterSystems IRIS XML tools also provide ways to modify data in XML form. The following figure shows a summary:




For an XML document, you can use class methods in %XML.XSLT.Transformer to perform XSLT transformations and obtain a modified version of the document. See "Performing XSLT Transformations."

For a DOM, you can use methods of %XML.Document to modify the DOM. For example, you can add or remove elements or attributes.

# 1.5 The SAX Parser

InterSystems IRIS XML Tools use the InterSystems IRIS SAX (Simple API for XML) Parser. This is a built-in SAX XML validating parser using the standard Xerces library. SAX is a parsing engine that provides complete XML validation and document parsing. InterSystems IRIS SAX communicates with an InterSystems IRIS process using a high-performance, in-process call-in mechanism. Using this parser, you can process XML documents using either the built-in InterSystems IRIS XML support or by providing your own custom SAX interface classes within InterSystems IRIS.

For special applications, you can create custom entity resolvers and content handlers; see "Customizing How the SAX Parser Is Used."

You can validate any incoming XML using industry-standard XML DTD or schema validation, and you can specify which XML items to parse. See "Customizing How the SAX Parser Is Used."

# 1.6 Additional XML Tools

InterSystems IRIS XML support includes the following additional tools:

- The XML Schema Wizard reads an XML schema document and generates a set of XML-enabled classes that correspond to the types defined in the schema. You specify a package to contain the classes, as well as various options that control the details of the class definitions. See "Generating Classes from XML Schemas."
- The %XML.Schema class enables you to generate an XML schema from a set of XML-enabled classes. See "Generating XML Schemas from Classes."
- The %XML.Namespaces class enables you to examine the XML namespaces and the classes in them, for an InterSystems IRIS namespace. See "Examining Namespaces and Classes."

- The %XML.Security.EncryptedData class and other classes enable you to encrypt XML documents, as well as decrypt encrypted documents. See "Encrypting XML Documents."
- The %XML.Security.Signature class and other classes enable you to digitally sign XML documents, as well validate digital signatures. See "Signing XML Documents."

# 1.7 Considerations When Using the XML Tools

When you work with XML tools of any kind, there are at least three general points to consider:

- Any XML document has a character encoding
- There are different ways to map an XML document to a class (literal or SOAP-encoded)
- You should be aware of the default behavior of the SAX Parser

## 1.7.1 Character Encoding of Input and Output

When you export an XML document, you can specify the *character encoding* to use; otherwise, InterSystems IRIS chooses the encoding, depending on the destination:

- If the output destination is a file or a binary stream, the default is `"UTF-8"`.
- If the output destination is a string or a character stream, the default is `"UTF-16"`.

For any XML document read by InterSystems IRIS, the XML declaration of the document should indicate the character encoding of that file, and the document should be encoded as declared. For example:

```
<?xml version="1.0" encoding="UTF-16"?>
```

However, if the character encoding is not declared in the document, InterSystems IRIS assumes the following:

- If the document is a file or a binary stream, InterSystems IRIS assumes that the character set is `"UTF-8"`.
- If the document is a string or a character stream, InterSystems IRIS assumes the character set is `"UTF-16"`.

For background information on character translation in InterSystems IRIS, see "Localization Support" in the *Orientation Guide for Server-Side Programming*.

## 1.7.2 Choosing a Document Format

When you work with an XML document, you must know the *format* to use when mapping the document to InterSystems IRIS classes. Similarly, when you create an XML document, you specify the document format to use when writing the document. The XML document formats are as follows:

- *Literal* means that the document is a literal copy of the object instance. In most cases, you use literal format, even when working with SOAP.

  Except where otherwise noted, the examples in the documentation use literal format.

- *Encoded* means encoded as described in the SOAP 1.1 standard or the SOAP 1.2 standard. For links to these standards, see "Standards Supported in InterSystems IRIS," later in this chapter.

  The details are slightly different for SOAP 1.1 and SOAP 1.2.

The following subsections show the differences between these document formats.

### 1.7.2.1 Literal Format

The following sample shows an XML document in literal format:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <Person>
      <Name>Klingman,Julie G.</Name>
      <DOB>1946-07-21</DOB>
      <GroupID>W897</GroupID>
      <Address>
         <City>Bensonhurst</City>
         <Zip>60302</Zip>
      </Address>
      <Doctors>
         <DoctorClass>
            <Name>Jung,Kirsten K.</Name>
         </DoctorClass>
         <DoctorClass>
            <Name>Xiang,Charles R.</Name>
         </DoctorClass>
         <DoctorClass>
            <Name>Frith,Terry R.</Name>
         </DoctorClass>
      </Doctors>
   </Person>
</Root>
```

### 1.7.2.2 Encoded Format

In contrast, the following example shows the same data in encoded format:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
   <DoctorClass id="id2" xsi:type="DoctorClass">
      <Name>Jung,Kirsten K.</Name>
   </DoctorClass>
...
   <DoctorClass id="id3" xsi:type="DoctorClass">
      <Name>Quixote,Umberto D.</Name>
   </DoctorClass>
...
   <DoctorClass id="id8" xsi:type="DoctorClass">
      <Name>Chadwick,Mark L.</Name>
   </DoctorClass>
...
   <Person>
      <Name>Klingman,Julie G.</Name>
      <DOB>1946-07-21</DOB>
      <GroupID>W897</GroupID>
      <Address href="#id17" />
      <Doctors SOAP-ENC:arrayType="DoctorClass[3]">
         <DoctorClass href="#id8" />
         <DoctorClass href="#id2" />
         <DoctorClass href="#id3" />
      </Doctors>
   </Person>
   <AddressClass id="id17" xsi:type="s_AddressClass">
      <City>Bensonhurst</City>
      <Zip>60302</Zip>
   </AddressClass>
...
</Root>
```

Note the following differences in the encoded version:

- The root element of the output includes declarations for the SOAP encoding namespace and other standard namespaces.
- This document includes person, address, and doctor elements all at the same level. The address and doctor elements are listed with unique IDs that are used by the person elements that refer to them. Each object-valued property is treated this way.
- The names of the top-level address and doctor elements are named the same as the respective *classes*, rather than being named the same as the property that refers to them.

- Encoded format does not include any attributes. The `GroupID` property is mapped as an attribute in the `Person` class. In literal format, this property is projected as an attribute. In the encoded version, however, the property is projected as an element.
- Collections are treated differently. For example, the list element has the attribute `ENC:arrayType`.
- Each element has a value for the `xsi:type` attribute.

**Note:** For SOAP 1.2, the encoded version is slightly different. To easily distinguish the versions, check the declaration for the SOAP encoding namespace:

- For SOAP 1.1, the SOAP encoding namespace is `"http://schemas.xmlsoap.org/soap/encoding/"`
- For SOAP 1.2, the SOAP encoding namespace is `"http://schemas.xmlsoap.org/wsdl/soap12/"`

### 1.7.3 Parser Behavior

The InterSystems IRIS SAX Parser is used whenever InterSystems IRIS reads an XML document, so it is useful to know its default behavior. Among other tasks, the parser does the following:

- It verifies whether the XML document is well-formed.
- It attempts to validate the document, using the given schema or DTD.

  Here it is useful to remember that a schema can contain <import> and <include> elements that refer to other schemas. For example:

  ```
  <xsd:import namespace="target-namespace-of-the-importing-schema"
                  schemaLocation="uri-of-the-schema"/>

  <xsd:include schemaLocation="uri-of-the-schema"/>
  ```

  The validation fails unless these other schemas are available to the parser. Especially with WSDL documents, it is sometimes necessary to download all the schemas and edit the primary schema to use the corrected locations.
- It attempts to resolve all entities, including all external entities. (Other XML parsers do this as well.) This process can be time-consuming, depending on their locations. In particular, Xerces uses a network accessor to resolve some URLs, and the implementation uses blocking I/O. Consequently, there is no timeout and network fetches can hang in error conditions, which have been rare in practice.

  Also, Xerces does not support https; that is, it cannot resolve entities that are at https locations.

  If needed, you can create custom entity resolvers and you can disable entity resolution; see "Customizing How the SAX Parser Is Used."

## 1.8 Standards Supported in InterSystems IRIS

InterSystems IRIS XML support follows these standards:

- XML 1.0 (http://www.w3.org/TR/REC-xml/)
- Namespaces in XML 1.0 (http://www.w3.org/TR/REC-xml-names/)
- XML Schema 1.0 (http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/)
- XPath 1.0 as specified by http://www.w3.org/TR/xpath

- SOAP 1.1 encoding as specified by section 5 of the SOAP 1.1 standard.
- SOAP 1.2 encoding as specified by section 3 Part 2: Adjuncts (http://www.w3.org/TR/soap12-part2/) of the SOAP 1.2 standard.

  For more information on SOAP, see the W3 web site (for example, http://www.w3.org/TR/2003/REC-soap12-part1-20030624/).
- XML Canonicalization Version 1.0 (also known as *inclusive canonicalization*), as specified by http://www.w3.org/TR/xml-c14n.
- XML Exclusive Canonicalization Version 1.0 as specified by http://www.w3.org/TR/xml-exc-c14n/, including the InclusiveNamespaces PrefixList feature (http://www.w3.org/TR/xml-exc-c14n/#def-InclusiveNamespaces-PrefixList)
- XML Encryption (http://www.w3.org/TR/xmlenc-core/)

  InterSystems IRIS supports key encryption using RSA-OAEP or RSA-1.5 and data encryption of the message body using AES-128, AES-192, or AES-256.
- XML Signature using Exclusive XML Canonicalization and RSA SHA-1 (http://www.w3.org/TR/xmldsig-core/)

The InterSystems IRIS SAX Parser uses the standard Xerces-C++ library, which complies with the XML 1.0 recommendation. For a list of these standards, see http://xml.apache.org/xerces-c/.

InterSystems IRIS provides two XSLT processors:

- The Xalan processor supports XSLT 1.0.
- The Saxon processor supports XSLT 2.0.

For information on additional standards related to web services and clients, see *Creating Web Services and Web Clients* and *Securing Web Services*.

For information on the character sets expected in XML, see the W3 web site (http://www.w3.org/TR/2006/REC-xml-20060816/#charsets).

**Note:** InterSystems IRIS does not support, within one element, multiple attributes with the same name, each in a different namespace.

# 2

# Writing XML Output from Objects

This chapter describes how you can generate XML output from InterSystems IRIS objects. It discusses the following topics:

- Overview and a simple example
- How to create an output method
- How to specify the character set of the output
- How to generate the prolog
- How to specify a default namespace for classes without a namespace
- How to add namespace declarations
- How to create the root element
- How to generate an element within the XML output
- How to control the use of namespaces
- How to control the appearance of namespace assignments
- How to control the output for string properties that equal ""
- How to write type information
- How to generate SOAP-encoded XML
- How to control unswizzling after export
- How to control the closing form for elements
- Other options provided by the writer
- Additional example, for use when experimenting with %XML.Writer

Also see "Writing XML Output from a DOM" in the chapter "Representing an XML Document as a DOM."

## 2.1 Overview of Creating an XML Writer

InterSystems IRIS provides tools for generating XML output for InterSystems IRIS objects. You specify the details of the XML projection, as described in *Projecting Objects to XML*. Then you create a writer method that specifies the overall structure of the XML output: the character encoding, the order in which objects are presented, whether processing instructions are included, and so on.

The essential requirements are as follows:

- If you need output for a particular object, the class definition for that object must extend %XML.Adaptor. With a few exceptions, the classes to which that object refers must also extend %XML.Adaptor. See *Projecting Objects to XML*.
- The output method must create an instance of %XML.Writer and then use methods of that instance.

The following Terminal session shows a simple example, in which we access an XML-enabled object and generate output for it:

```
GXML>Set p=##class(GXML.Person).%OpenId(1)

GXML>Set w=##class(%XML.Writer).%New()

GXML>Set w.Indent=1

GXML>Set status=w.RootObject(p)
<?xml version="1.0" encoding="UTF-8"?>
<Person GroupID="R9685">
  <Name>Zimmerman,Jeff R.</Name>
  <DOB>1961-10-03</DOB>
  <Address>
    <City>Islip</City>
    <Zip>15020</Zip>
  </Address>
  <Doctors>
    <Doctor>
      <Name>Sorenson,Chad A.</Name>
    </Doctor>
    <Doctor>
      <Name>Sorenson,Chad A.</Name>
    </Doctor>
    <Doctor>
      <Name>Uberoth,Roger C.</Name>
    </Doctor>
  </Doctors>
</Person>
```

The following sections provide details.

# 2.2 Creating an Output Method

Your output method constructs an XML document piece by piece, in the order you specify. The overall structure of your output method depends on whether you need to output a complete XML document or simply a fragment. This section discusses the following topics:

- Overall structure of your output method
- Error checking
- Inserting comments
- A basic example

## 2.2.1 Overall Method Structure

Your method should do some or all of the following, in this order:

1. If you are working with an object that might be invalid, call the **%ValidateObject()** method of that object and check the returned status. If the object is not valid, the XML would also not be valid.

   %XML.Writer does not validate the object before exporting it. This means that if you have just created an object and have not yet validated it, the object (and hence the XML) could be invalid (because, for example, a required property is missing).

2. Create an instance of the %XML.Writer class and optionally set its properties.

   In particular, you might want to set the following properties:

   - Indent — Controls whether the output is generated within indentation and line wrapping (if Indent equals 1) or as a single long line (if Indent equals 0). The latter is the default.

     See the later subsection "Details on Indent Option."

   - IndentChars — Specifies the characters used for indentation. The default is a string of two spaces. This property has no effect if Indent is 0.

   - Charset — Specifies the character set to use. See "Specifying the Character Set of the Output."

   For readability, the examples in this documentation use Indent equal to 1.

   Also see the sections "Properties That Affect the Prolog" and "Other Properties of the Writer" later in this chapter.

3. Specify an output destination.

   By default, output is written to the current device. To specify the output destination, call one of the following methods before starting to write the document:

   - **OutputToDevice()** — Directs the output to the current device.
   - **OutputToFile()** — Directs the output to the specified file. You can specify an absolute path or a relative path. Note that the directory path must already exist.
   - **OutputToString()** — Directs the output to a string. Later you can use another method to retrieve this string.
   - **OutputToStream()** — Directs the output to the specified stream.

4. Start the document. You can use the **StartDocument()** method. Note that the following methods implicitly start a document if you have not started a document via **StartDocument()**: **Write()**, **WriteDocType()**, **RootElement()**, **WriteComment()**, and **WriteProcessingInstruction()**.

5. Optionally write lines of the prolog of the document. You can use the following methods:

   - **WriteDocType()** — Writes the DOCTYPE declaration. For details, see "Generating a Document Type Declaration."
   - **WriteProcessingInstruction()** — Writes a processing instruction. For details, see "Writing Processing Instructions."

6. Optionally specify the default namespace. The writer uses this for classes that do not have a defined XML namespace. See "Specify the Default Namespace."

7. Optionally add namespace declarations to the root element. To do so, you can invoke several utility methods before starting the root element. See "Adding Namespace Declarations."

8. Start the root element of the document. The details depend on whether the root element of that document corresponds to an InterSystems IRIS object. There are two possibilities:

   - The root element might correspond directly to an InterSystems IRIS object. This is typically the case if you are generating output for a single object.

     In this case, you use the **RootObject()** method, which writes the specified XML-enabled object as the root element.

   - The root element might be merely a wrapper for a set of elements, and those elements are InterSystems IRIS objects.

     In this case, you use the **RootElement()** method, which inserts the root-level element with the name you specify.

   Also see "Writing the Root Element."

9. If you used the **RootElement()** method, call methods to generate the output for one or more elements inside the root element. You can write any elements within the root element, with any order or logic you choose. There are several ways that you can write an individual element, and you can combine these techniques:

   - You can use the **Object()** method, which writes an XML-enabled object. You specify the name of this element, or you can use the default, which is defined by the object.
   - You can use the **Element()** method, which writes the start tag for an element, with the name you provide. Then you can write content, attributes, and child elements, by using methods such as **WriteAttribute()**, **WriteChars()**, **WriteCData()**, and others. A child element could be another **Element()** or could be an **Object()**. You use the **EndElement()** method to indicate the end of the element.
   - You can use the %XML.Element class and construct an element manually.

   See "Constructing an Element Manually" for further details.

10. If you used the **RootElement()** method, call the **EndRootElement()** method. This method closes the root element of the document and decreases the indentation (if any), as appropriate.
11. If you started the document with **StartDocument()**, call the **EndDocument()** method to close the document.
12. If you directed the output to a string, use the **GetXMLString()** method to retrieve that string.

There are many other possible organizations, but note that some methods can be called only in some contexts. Specifically, once you start a document, you cannot start another document until you end the first document. If you try to do so, the writer method returns the following status:

```
#6275: Cannot output a new XML document or change %XML.Writer properties
until the current document is completed.
```

The **StartDocument()** method explicitly starts a document. As noted earlier, other methods implicitly start a document: **Write()**, **WriteDocType()**, **RootElement()**, **WriteComment()**, and **WriteProcessingInstruction()**.

**Note:** The methods described here are designed to enable you to write specific units to the XML document, but in some cases, you may need more control. The %XML.Writer class provides an additional method, **Write()**, which you can use to write an arbitrary string into any place in the output.

Also, you can use the **Reset()** method to reinitialize the writer properties and output method. This is useful if you have generated an XML document and want to generate another one without creating a new writer instance.

## 2.2.2 Error Checking

Most of the methods of %XML.Writer return a status. You should check the status after each step and quit if appropriate.

## 2.2.3 Inserting Comment Lines

As noted earlier, you use the **WriteComment()** method to insert a comment line. You can use this method anywhere in your document. If you have not yet started the XML document, this method implicitly starts the document.

## 2.2.4 Example

The following example method generates XML output for a given XML-enabled object.

```
ClassMethod Write(obj) As %Status
{
    set writer=##class(%XML.Writer).%New()
    set writer.Indent=1

    //these steps are not really needed because
```

```
    //this is the default destination
    set status=writer.OutputToDevice()
    if $$$ISERR(status) {
        do $System.Status.DisplayError(status)
        quit $$$ERROR($$$GeneralError, "Output destination not valid")
    }

    set status=writer.RootObject(obj)
    if $$$ISERR(status) {
        do $System.Status.DisplayError(status)
        quit $$$ERROR($$$GeneralError, "Error writing root object")
    }

    quit status
}
```

Notice that this method uses the **OutputToDevice()** method to direct output to the current device, which is the default destination. This is not necessary but is shown for demonstration purposes.

Suppose that we execute this method as follows:

```
 set obj=##class(GXML.Person4).%OpenId(1)
 do ##class(MyWriters.BasicWriter).Write(obj)
```

The output depends on what class is used, of course, but could look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person4>
  <Name>Tesla,Alexandra L.</Name>
  <DOB>1983-11-16</DOB>
  <GroupID>Y9910</GroupID>
  <Address>
    <City>Albany</City>
    <Zip>24450</Zip>
  </Address>
  <Doctors>
    <Doc>
      <Name>Schulte,Frances T.</Name>
    </Doc>
    <Doc>
      <Name>Smith,Albert M.</Name>
    </Doc>
  </Doctors>
</Person4>
```

### 2.2.5 Details on Indent Option

As noted earlier, you can use the *Indent* property of your writer to get output that contains additional line breaks, for greater readability. There is no formal specification for this formatting. This section describes the rules used by %XML.Writer if *Indent* equals 1:

- Any element that contains only whitespace characters is converted to an empty element.
- Each element is placed on its own line.
- If an element is a child of a preceding element, the element is indented relative to that parent element. The indentation is determined by the *IndentChars* property, which defaults to two spaces.

## 2.3 Specifying the Character Set of the Output

To specify the character set to use in the output document, you can set the Charset property of your writer instance. The options include `"UTF-8"`, `"UTF-16"`, and other character sets supported by InterSystems IRIS.

For information on the default encoding, see “Character Encoding of Input and Output,” earlier in this book.

# 2.4 Writing the Prolog

The prolog of an XML file (the section before the root element) can contain a documentation type declaration, processing instructions, and comments. This section discusses the following:

- Properties that affect the prolog
- How to generate the document type declaration, if needed
- How to generate processing instructions

## 2.4.1 Properties That Affect the Prolog

In your writer instance, the following properties affect the prolog:

**Charset**

Controls two things: the character set declaration in the XML declaration and (correspondingly) the character set encoding used in the output. See "Specifying the Character Set of the Output."

**NoXmlDeclaration**

Controls whether the output contains the XML declaration. In most cases, the default is 0, which means that the declaration is written. If the character set is not specified and if the output is directed to a string or character stream, the default is 1 and no declaration is written.

## 2.4.2 Generating a Document Type Declaration

Before the root element, you can include a document type declaration, which declares the schema used in the document. To generate a document type declaration, you use the **WriteDocType()** method, which has one required argument and three optional arguments. For the purpose of this documentation, a document type declaration consists of the following possible parts:

```
<!DOCTYPE doc_type_name external_subset [internal_subset]>
```

As shown here, the document type has a name, which must be the name of the root element, according to the rules of XML. The declaration can include an external subset, an internal subset, or both.

The *external_subset* section points to DTD files elsewhere. The structure of this section is any of the following:

```
PUBLIC public_literal_identifier
PUBLIC public_literal_identifier system_literal_identifier
SYSTEM system_literal_identifier
```

Here *public_literal_identifier* and *system_literal_identifier* are strings that contain the URIs of a DTD. Note that a DTD can have both a public identifier and a system identifier. The following shows an example document type declaration that contains an external subset that uses both a public and a system identifier:

```
<!DOCTYPE hatches <!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">>
```

The *internal_subset* section is a set of entity declarations. The following shows an example document type declaration that contains only an internal set of declarations:

```
<!DOCTYPE teams [ <!ENTITY baseball team (name,city,player*)>
                  !ENTITY name (#PCDATA)>
                  !ENTITY city (#PCDATA)>
                  !ENTITY player (#PCDATA)>
  ] >
```

The **WriteDocType()** method has four arguments:

- The first argument specifies the name of the document type, for use within this XML document. This is required and must be a valid XML identifier. You also must use this same name as the name of root level element in this document.
- The optional second and third arguments specify the external part of the declaration, as follows:

  ***Table 2–1: WriteDocType Arguments***

| Second Argument | Third Argument | Resulting External Part |
|---|---|---|
| "*publicURI*" | *null* | PUBLIC "*publicURI*" |
| "*publicURI*" | "*systemURI*" | PUBLIC "*publicURI*" "*systemURI*" |
| *null* | "*systemURI*" | SYSTEM "*systemURI*" |

- The optional fourth argument specifies the internal part of the declaration. If this argument is non-null, then it is wrapped in square brackets `[]` and placed appropriately at the end of the declaration. No other characters are added.

### 2.4.3 Writing Processing Instructions

To write a processing instruction into the XML, you use the **WriteProcessingInstruction()** method, which takes two arguments:

1. The name of the processing instruction (also known as the target).
2. The instructions themselves as a string.

The method writes the following into the XML:

```
<?name instructions?>
```

For example, suppose that you wanted to write the following processing instruction:

```
<?xml-stylesheet type="text/css" href="mystyles.css"?>
```

You could do this by calling the **WriteProcessingInstruction()** method as follows:

```
set instructions="type=""text/css"" href=""mystyles.css"""
set status=writer.WriteProcessingInstruction("xml-stylesheet", instructions)
```

## 2.5 Specifying a Default Namespace

In your writer instance, you can specify a default namespace, which is applied only to classes that do not have a setting for the *NAMESPACE* parameter. You have a couple of options:

- You can specify a default namespace within an output method. The four primary output methods (**RootObject()**, **RootElement()**, **Object()**, or **Element()**) all accept a namespace as an argument. The relevant element is assigned to the namespace only if the *NAMESPACE* parameter is not set in the class definition.
- You can specify an overall default namespace for the writer instance. To do so, specify a value for the DefaultNamespace property of your writer instance.

### 2.5.1 Example

Consider the following class:

```
Class Writers.BasicDemoPerson Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLNAME="Person";

Property Name As %Name;

Property DOB As %Date;

}
```

By default, if we simply export an object of this class, we see output like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person>
  <Name>Persephone MacMillan</Name>
  <DOB>1976-02-20</DOB>
</Person>
```

In contrast, if we set DefaultNamespace equal to `"http://www.person.org"` in our writer instance and then export an object, we receive output like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person xmlns="www.person.org">
  <Name>Persephone MacMillan</Name>
  <DOB>1976-02-20</DOB>
</Person>
```

In this case, the default namespace is used for the `<Person>` element, which otherwise would not be assigned to a namespace.

## 2.6 Adding Namespace Declarations

### 2.6.1 Default Behavior

The %XML.Writer class automatically inserts the namespace declarations, generates namespace prefixes, and applies the prefixes where appropriate. For example, consider the following class definition:

```
Class ResearchWriters.PersonNS Extends (%Persistent, %XML.Adaptor)
{

Parameter NAMESPACE = "http://www.person.com";

Parameter XMLNAME = "Person";

Property Name As %Name;

Property DOB As %Date;

}
```

If you export multiple objects of this class, you see something like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Person xmlns="http://www.person.com">
    <Name>Persephone MacMillan</Name>
    <DOB>1975-09-15</DOB>
  </Person>
  <Person xmlns="http://www.person.com">
    <Name>Joseph White</Name>
    <DOB>2003-01-31</DOB>
  </Person>
...
```

The namespace declaration is added to each `<Person>` element automatically. You might prefer to add it only to the root of the document.

## 2.6.2 Manually Adding the Declarations

You can control when a namespace is introduced into the XML output. The following methods all affect the next element that is written (but do not affect any elements after that one). For convenience, several of these methods add standard W3 namespaces.

Typically you use these methods to add namespace declarations to the root element of the document; that is, you call one or more of these methods before calling **RootObject()** or **RootElement()**.

**Note:** None of these methods assign any elements to namespaces, and these namespaces are never added as the default namespace. When you generate a specific element, you indicate the namespace that it uses, as noted later in "Writing the Root Element" and "Generating an XML Element."

### AddNamespace()

```
method AddNamespace(namespace As %String,
                    prefix As %String,
                    schemaLocation As %String) as %Status
```

Adds the specified namespace. Here *namespace* is the namespace to add, *prefix* is an optional prefix for this namespace, and *schemaLocation* is the optional URI that indicates the location of the corresponding schema.

If you do not specify a prefix, a prefix is automatically generated (of the form `s01`, `s02`, and so on).

The following example shows the effect of this method. First, suppose that the `Person` class is assigned to a namespace (by means of the *NAMESPACE* class parameter). If you generate output for an instance of this class without first calling the **AddNamespace()** method, you might receive output like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
  <Person xmlns="http://www.person.org">
    <Name>Love,Bart Y.</Name>
...
```

Alternatively, suppose that you call the **AddNamespace()** method as follows before you write the root element:

```
 set status=writer.AddNamespace("http:///www.person.org","p")
```

If you then generate the root element, the output is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns:p="http:///www.person.org">
  <Person xmlns="http://www.person.org">
...
```

Or suppose that when you call the **AddNamespace()** method, you specify the third argument, which gives the location of the associated schema:

```
 set
status=writer.AddNamespace("http:///www.person.org","p","http://www.MyCompany.com/schemas/person.xsd")
```

In this case, if you then generate the `Root` element, the output is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns:p="http:///www.person.org"
xsi:schemaLocation="http:///www.person.org http://www.MyCompany.com/schemas/person.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Person xmlns="http://www.person.org">
...
```

### AddInstanceNamespace()

```
method AddInstanceNamespace(prefix As %String) as %Status
```

Adds the W3 schema instance namespace. Here *prefix* is the optional prefix to use for this namespace. The default prefix is `xsi`.

For example:

```
<Root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
```

### AddSchemaNamespace()

```
method AddSchemaNamespace(prefix As %String) as %Status
```

Adds the W3 schema namespace. Here *prefix* is the optional prefix to use for this namespace. The default prefix is `s`.

For example:

```
<Root xmlns:s="http://www.w3.org/2001/XMLSchema">
...
```

### AddSOAPNamespace()

```
method AddSOAPNamespace(soapPrefix As %String,
                        schemaPrefix As %String,
                        xsiPrefix As %String) as %Status
```

Adds the W3 SOAP encoding namespace, SOAP schema namespace, and SOAP schema instance namespace. This method has three optional arguments: the prefixes to use for these namespaces. The default prefixes are `SOAP-ENC`, `s`, and `xsi`, respectively.

For example:

```
<Root xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
```

Also see “Generating SOAP-Encoded XML,” later in this chapter.

### AddSOAP12Namespace()

```
method AddSOAP12Namespace(soapPrefix As %String,
                          schemaPrefix As %String,
                          xsiPrefix As %String) as %Status
```

Adds the W3 SOAP 1.2 encoding namespace, SOAP schema namespace, and SOAP schema instance namespace.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns:SOAP-ENC="http://www.w3.org/2003/05/soap-encoding"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...
```

You can use more than one of these methods. If you use more than one of them, the affected element contains the declarations for all the specified namespaces.

For an example, see "More Complex Schema Example,"' in the chapter "Generating XML Schemas from Classes."

# 2.7 Writing the Root Element

Every XML document must contain exactly one root element. There are two ways you can create this element:

- The root element might correspond directly to a single InterSystems IRIS XML-enabled object.

  In this case, you use the **RootObject()** method, which writes the specified XML-enabled object as the root element. The output includes all object references contained in that object. The root element acquires the structure of that object, and you cannot insert additional elements. You can specify a name for the root element, or you use the default, which is defined by the XML-enabled object.

  Previous examples used this technique.

- The root element might be merely a wrapper for a set of elements (perhaps for a set of XML-enabled objects).

  In this case, you use the **RootElement()** method, which inserts the root-level element with the name you specify. This method also increases the indentation level for succeeding operations, if this document is indented.

  Then you call additional methods to generate the output for one or more elements inside the root element. Inside the root, you can include elements you need, with any order or logic you choose. See "Constructing an Element Manually" for further details. After this, you call the **EndRootElement()** method to close the root element.

  For an example, see the next section.

In either case, you can specify a namespace to use for the root element, which is applied only if the XML-enabled class does not have a value for the *NAMESPACE* parameter. See "Summary of Namespace Assignment."

Remember that if the document includes a document type declaration, the name of that DTD must be the same as the name of the root element.

# 2.8 Generating an XML Element

If you use **RootElement()** to start the root element of a document, you are responsible for generating each element inside that root element. You have three choices:

- Generate an element from an InterSystems IRIS object
- Construct an element manually with the **Element()** method
- Construct an element manually with %XML.Element

For information on assigning the exported objects to namespaces, see "Controlling the Use of Namespaces," later in this chapter.

## 2.8.1 Generating an Object as an Element

You can generate output from an InterSystems IRIS object as the element. In this case, you use the **Object()** method, which writes an XML-enabled object. The output includes all object references contained in that object. You specify the name of this element, or you can use the default, which is defined within the object.

You can use the **Object()** method only between the **RootElement()** and **EndRootElement()** methods.

### 2.8.1.1 Example

This example generates output for all saved instances of a given XML-enabled class:

```
ClassMethod WriteAll(cls As %String="", directory As %String = "c:\")
{
    if '##class(%Dictionary.CompiledClass).%ExistsId(cls) {
        Write !, "Class does not exist or is not compiled"
        Quit
    }
    Set check=$classmethod(cls,"%Extends","%XML.Adaptor")
    If 'check {
        Write !, "Class does not extend %XML.Adaptor"
        Quit
    }

    set filename=directory_cls_".xml"
    set writer=##class(%XML.Writer).%New()
    set writer.Indent=1
    set status=writer.OutputToFile(filename)
    if $$$ISERR(status) { do $System.Status.DisplayError(status) quit  }

    set status=writer.RootElement("SampleOutput")
    if $$$ISERR(status) { do $System.Status.DisplayError(status) quit  }

    //Get IDs of objects in the extent for the given class
    set stmt=##class(%SQL.Statement).%New()
    set status = stmt.%PrepareClassQuery(cls,"Extent")
    if $$$ISERR(status) { do $System.Status.DisplayError(status) quit  }

    set rset=stmt.%Execute()
    while (rset.%Next()) {
        //for each ID, write that object
        set objid=rset.%Get("ID")
        set obj=$CLASSMETHOD(cls,"%OpenId",objid)
        set status=writer.Object(obj)
        if $$$ISERR(status) {Do $System.Status.DisplayError(status) Quit}}

    do writer.EndRootElement()
    do writer.EndDocument()
}
```

The output from this method contains all saved objects of the given class, nested within the root element. For Sample.Person, the output is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<SampleOutput>
  <Person>
    <Name>Tillem,Robert Y.</Name>
    <SSN>967-54-9687</SSN>
    <DOB>1961-11-27</DOB>
    <Home>
      <Street>3355 First Court</Street>
      <City>Reston</City>
      <State>WY</State>
      <Zip>11090</Zip>
    </Home>
    <Office>
      <Street>4922 Main Drive</Street>
      <City>Newton</City>
      <State>NM</State>
      <Zip>98073</Zip>
    </Office>
    <FavoriteColors>
      <FavoriteColorsItem>Red</FavoriteColorsItem>
    </FavoriteColors>
    <Age>47</Age>
  </Person>
  <Person>
    <Name>Waters,Ed X.</Name>
    <SSN>361-66-2801</SSN>
    <DOB>1957-05-29</DOB>
    <Home>
      <Street>5947 Madison Drive</Street>
...
</SampleOutput>
```

## 2.8.2 Constructing an Element Manually

You can construct an XML element manually. In this case, you use the **Element()** method, which writes the start tag for an element, with the name you provide. Then you can write content, attributes, and child elements. You use the **EndElement()** method to indicate the end of the element.

The relevant methods are as follows:

### Element()

```
method Element(tag, namespace As %String) as %Status
```

Writes the start tag. You can provide a namespace for the element, which is applied only if the XML-enabled class does not have a value for the *NAMESPACE* parameter. See "Summary of Namespace Assignment."

### WriteAttribute()

```
method WriteAttribute(name As %String,
                      value As %String  = "",
                      namespace As %String,
                      valueNamespace As %String  = "",
                      global As %Boolean  = 0) as %Status
```

Writes an attribute. You must specify the attribute name and value. The argument *namespace* is the namespace for the attribute name. The argument *valueNamespace* is the namespace for the attribute value; this is used when the values are defined in an XML schema namespace.

For *global*, specify true if the attribute is global in the associated XML schema and thus should have a prefix.

If you use this method, you must use it directly after **Element()** (or after **RootElement()**).

### WriteChars()

```
method WriteChars(text) as %Status
```

Writes a string, performing any necessary escaping needed to make the string suitable as the content of an element. The argument must be of type %String or %CharacterStream.

### WriteCData()

```
method WriteCData(text) as %Status
```

Writes a CDATA section. The argument must be of type %String or %CharacterStream.

### WriteBase64()

```
method WriteBase64(binary) as %Status
```

Encodes the specified binary bytes as base-64 and writes the resulting text as the content of the element. The argument must be of type %Binary or %BinaryStream.

### WriteBinHex()

```
method WriteBinHex(binary) as %Status
```

Encodes the specified binary bytes as binhex and writes the resulting text as the content of the element. The argument must be of type %Binary or %BinaryStream.

#### EndElement()

```
method EndElement() as %Status
```

Ends the element to which it can be matched.

You can use these methods only between the **RootElement()** and **EndRootElement()** methods.

**Note:** The methods described here are designed to enable you to write specific, logical pieces to the XML document, but in some cases, you may need more control. The %XML.Writer class provides an additional method, **Write()**, which you can use to write an arbitrary string. It is your responsibility to ensure that the result is a well-formed XML document; no validation is provided.

### 2.8.2.1 Example

An example routine follows:

```
 //write output to current device
 //simple demo of Element & related methods

 set writer=##class(%XML.Writer).%New()
 set writer.Indent=1

 set status=writer.OutputToDevice()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.StartDocument()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.RootElement("root")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.Element("SampleElement")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.WriteAttribute("Attribute","12345")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.Element("subelement")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.WriteChars("Content")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.EndElement()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.Element("subelement")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.WriteChars("Content")
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.EndElement()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.EndElement()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.EndRootElement()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

 set status=writer.EndDocument()
 if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
```

The output for this routine is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <SampleElement Attribute="12345">
    <subelement>Content</subelement>
    <subelement>Content</subelement>
  </SampleElement>
</root>
```

### 2.8.3 Using %XML.Element

In the previous section, we used the **Element()** and specified the element to generate; we could have also specified a namespace. In some cases, you might want to use an instance of the %XML.Element class instead of an element name. This class has the following properties:

- The Local property specifies whether this element is local to its parent element, which affects the control of namespaces. For details, see "Controlling Whether an Element Is Local to Its Parent," ahead.
- The Namespace property specifies the namespace for this element.
- The Tagname property specifies the name for this element.

Here you can also use the **WriteAttribute()** method, described earlier.

## 2.9 Controlling the Use of Namespaces

As described in *Projecting Objects to XML*, you can assign a class to a namespace so that the corresponding XML element belongs in that namespace, and you can control whether the properties of the class belong to that namespace as well.

When you export objects in the class to XML, the %XML.Writer class provides additional options such as specifying whether an element is local to its parent. This section includes the following topics:

- How the %XML.Writer class handles namespaces by default
- How to specify whether a local element is qualified
- How to specify whether an element is local to its parent
- How to specify whether an attribute is qualified
- A summary of how namespaces are assigned

**Note:** In InterSystems IRIS XML support, you specify namespaces on a class-by-class basis. Typically each class has its own namespace declaration; however, usually only one or a small number of namespaces are needed. You specify related information on a class-by-class basis as well (rather than in some global manner). This includes settings that control whether an element is local to its parent and whether subelements are qualified. It is recommended that you use a consistent approach, for simplicity.

### 2.9.1 Default Handling of Namespaces

To assign an XML-enabled class to a namespace, set the *NAMESPACE* parameter of the class, as described in *Projecting Objects to XML*. The %XML.Writer class automatically inserts the namespace declarations, generates namespace prefixes, and applies the prefixes where appropriate. For example, consider the following class definition:

```
Class GXML.Objects.WithNamespaces.Person Extends (%Persistent, %Populate, %XML.Adaptor)
{
Parameter NAMESPACE = "http://www.person.com";
Property Name As %Name [ Required ];
Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h") [ Required ];
Property GroupID As %Integer(MAXVAL=10,MINVAL=1,XMLPROJECTION="ATTRIBUTE");
}
```

If you export an object of this class, you see something like the following:

```
<Person xmlns="http://www.person.com" GroupID="4">
  <Name>Uberoth,Amanda Q.</Name>
  <DOB>1952-01-13</DOB>
</Person>
```

Note the following:

- The namespace declaration is added to each <Person> element.
- Local elements (<Name> and <DOB>) of the <Person> element are qualified by default. The namespace is added as the default namespace and thus applies to these elements.
- The attribute (GroupID) of the <Person> element is not qualified by default. This attribute has no prefix and thus is considered unqualified.
- The prefixes shown here were automatically generated. (Remember that when you assign an object to a namespace, you specify only the namespace, not the prefix.)
- This output occurs without setting any namespace-related properties in the writer and without using any namespace-related methods in the writer.

### 2.9.1.1 The Effect of Context of Namespace Assignment

The namespace to which an XML-enabled object is assigned depends upon whether the object is exported at the top level or as a property of another object.

Consider a class named Address. Suppose that you use the *NAMESPACE* parameter to assign the Address class to the namespace "http://www.address.org". If you exported an object of the Address class directly, you might receive output like the following:

```
<Address xmlns="http://www.address.org">
  <Street>8280 Main Avenue</Street>
  <City>Washington</City>
  <State>VT</State>
  <Zip>15355</Zip>
</Address>
```

Notice that the <Address> element and all its elements are in the same namespace ("http://www.address.org").

In contrast, suppose that the Person class that has a property that is an Address object. Also suppose that you use the *NAMESPACE* parameter to assign the Person class to the namespace "http://www.person.org". If you exported an object of the Person class, you would receive output like the following:

```
<Person xmlns="http://www.person.org">
  <Name>Zevon,Samantha H.</Name>
  <DOB>1964-05-24</DOB>
  <Address xmlns="http://www.address.org">
     <Street>8280 Main Avenue</Street>
     <City>Washington</City>
     <State>VT</State>
     <Zip>15355</Zip>
  </Address>
</Person>
```

Notice that the <Address> element is in the namespace of its parent object ("http://www.person.org"). The elements of <Address>, however, are within the namespace "http://www.address.org".

## 2.9.2 Controlling Whether Local Elements Are Qualified

When you export an object at the top level, it is usually treated as a global element. Then its local elements are handled according to the setting of the *ELEMENTQUALIFIED* parameter of the XML-enabled object. If this class parameter is not set, the value of the writer property ElementQualified is used instead; by default, this is 1 for literal format or 0 for encoded format.

The following example shows an object with the default setting of ElementQualified, which is 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonNS xmlns="http://www.person.com" GroupID="M9301">
  <Name>Pybus,Gertrude X.</Name>
  <DOB>1986-10-19</DOB>
</PersonNS>
```

The namespace is added to the <PersonNS> element as the default namespace and thus applies to the elements <Name> and <DOB> subelements.

Suppose that we altered the writer definition and set the ElementQualified property to 0. In this case, the same object would appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<s01:PersonNS xmlns:s01="http://www.person.com" GroupID="M9301">
  <Name>Pybus,Gertrude X.</Name>
  <DOB>1986-10-19</DOB>
</s01:PersonNS>
```

In this case, the namespace is added to the <PersonNS> element with a prefix, which is used for the <PersonNS> element but not for its subelements.

## 2.9.3 Controlling Whether an Element Is Local to Its Parent

By default, when you use the **Object()** method to generate an element and that element has a namespace, the element is not local to its parent. You can instead force the element to belong to the namespace of its parent. To do so, you use the optional *local* argument of the **Object()** method; this is the fourth argument.

### 2.9.3.1 Local Argument As 0 (Default)

In the examples here, consider a Person class that specifies the *NAMESPACE* class parameter as "http://www.person.com".

If you open the root element and then use **Object()** to generate a Person, the <Person> element is in the "http://www.person.com" namespace. Consider the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns="http://www.rootns.org">
   <Person xmlns="http://www.person.com">
      <Name>Adam,George L.</Name>
      <DOB>1947-06-29</DOB>
   </Person>
</Root>
```

A similar result would occur if we nested the <Person> element more deeply inside other elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns="www.rootns.org">
   <GlobalEl xmlns="globalns">
      <Inner xmlns="innerns">
         <Person xmlns="http://www.person.com">
            <Name>Adam,George L.</Name>
            <DOB>1947-06-29</DOB>
         </Person>
      </Inner>
   </GlobalEl>
</Root>
```

### 2.9.3.2 Local Argument Set to 1

To force the <Person> element to be local to its parent, we set the *local* argument equal to 1. If we do so and generate the previous output again, we would receive the following for the less nested version:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns="http://www.rootns.org">
   <s01:Person xmlns="http://www.person.com"
xmlns:s01="http://www.rootns.org">
      <Name>Adam,George L.</Name>
      <DOB>1947-06-29</DOB>
   </s01:Person>
</Root>
```

Note that now the <Person> element is in the "http://www.rootns.org" namespace, the namespace of its parent element. Similarly, the more highly nested version would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root xmlns="www.rootns.org">
   <GlobalEl xmlns="globalns">
      <Inner xmlns="innerns">
         <s01:Person xmlns="http://www.person.com" xmlns:s01="innerns">
            <Name>Adam,George L.</Name>
            <DOB>1947-06-29</DOB>
         </s01:Person>
      </Inner>
   </GlobalEl>
</Root>
```

## 2.9.4 Controlling Whether Attributes Are Qualified

When you export an object, by default its attributes are not qualified. To make them qualified, set the writer property AttributeQualified equal to 1. The following example shows output generated with a writer for which AttributeQualified equals 0 (or has not been set):

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <s01:Person xmlns:s01="http://www.person.com" GroupID="E8401">
      <Name>Leiberman,Amanda E.</Name>
      <DOB>1988-10-28</DOB>
   </s01:Person>
</Root>
```

In contrast, the following example shows the same object generated with a writer for which AttributeQualified equals 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <s01:Person xmlns:s01="http://www.person.com" s01:GroupID="E8401">
      <Name>Leiberman,Amanda E.</Name>
      <DOB>1988-10-28</DOB>
   </s01:Person>
</Root>
```

In both cases, the elements are unqualified.

## 2.9.5 Summary of Namespace Assignment

This section describes how the namespace is determined for any given element in your XML output.

### 2.9.5.1 Top-Level Elements

For an element that corresponds to an InterSystems IRIS class that is exported at the top level, the following rules apply:

1. If you specified the *NAMESPACE* parameter for the class, the element is in that namespace.
2. If that parameter is not specified, the element is in the namespace you specified in the output method that generated the element (**RootObject()**, **RootElement()**, **Object()**, or **Element()**).
3. If you did not specify a namespace in the output method, the element is in the namespace specified by the DefaultNamespace property of the writer.
4. If the DefaultNamespace property is null, the element is not in any namespace.

### 2.9.5.2 Lower-Level Elements

The subelements of the class that you are exporting are affected by the *ELEMENTQUALIFIED* parameter of that class. If *ELEMENTQUALIFIED* is not set, the value of the writer property ElementQualified is used instead; by default, this is 1 for literal format or 0 for encoded format.

If elements are qualified for a given class, the subelements of that class are assigned to namespaces as follows:

1. If you specified the *NAMESPACE* parameter for the *parent object*, the subelements are explicitly assigned to that namespace.
2. If that parameter is not specified, the subelements are explicitly assigned to the namespace you specified in the output method that generated the element (**RootObject()**, **RootElement()**, **Object()**, or **Element()**).
3. If you did not specify a namespace in the output method, the subelements are explicitly assigned to the namespace given by the DefaultNamespace property of the writer.
4. If the DefaultNamespace property is null, the subelements are not explicitly assigned to any namespace.

# 2.10 Controlling the Appearance of Namespace Assignments

In addition to controlling namespace assignments, you can control how the namespace assignments appear in the XML output. Specifically you can control the following:

- Explicit versus implicit namespace assignment
- Prefixes for namespaces

## 2.10.1 Explicit versus Implicit Namespace Assignment

When you assign elements and attributes to a namespace, there are two equivalent representations in XML, controlled by the SuppressXmlns property of your writer instance.

Suppose that we generate XML output for an object named `Person` which is assigned to the namespace `"http://www.person.org"` (by means of the *NAMESPACE* class parameter, discussed earlier). An example of the default output (with SuppressXmlns equal to 0) is as follows:

```
<Person xmlns="http://www.person.com" GroupID="4">
  <Name>Uberoth,Amanda Q.</Name>
  <DOB>1952-01-13</DOB>
</Person>
```

Another possible form, which is entirely equivalent, is as follows. This was generated with SuppressXmlns equal to 1, which ensures that every element that is assigned explicitly to a namespace is shown with the prefix for that namespace.

```
<s01:Person xmlns:s01="http://www.person.com" GroupID="4">
  <s01:Name>Uberoth,Amanda Q.</s01:Name>
  <s01:DOB>1952-01-13</s01:DOB>
</s01:Person>
```

Note that this property affects only the way in which the namespace assignment is *shown*; it does not control *how* any namespace is assigned. This parameter has no effect if you do not use namespaces.

## 2.10.2 Specifying Custom Prefixes for Namespaces

When you generate XML output for an object, the system generates namespace prefixes as needed. The first namespace prefix is `s01`, the next is `s02`, and so on. You can specify different prefixes. To do so, set the *XMLPREFIX* parameter in the class definitions for the XML-enabled objects themselves. This parameter has two effects:

- It ensures that the prefix you specify is declared in the XML output. That is, it is declared even if doing so is not necessary.
- It uses that prefix rather than the automatically generated prefix that you would otherwise see.

For details, see *Projecting Objects to XML*.

# 2.11 Controlling How Empty Strings ("") Are Exported

When you XML-enable an object, you specify how null values and empty strings are projected to XML (see *Projecting Objects to XML*).

One of the options is to set *XMLIGNORENULL* equal to `"RUNTIME"` (not case-sensitive) in the XML-enabled class. In this case, when you use %XML.Writer to generate output, InterSystems IRIS uses the value of the RuntimeIgnoreNull property of the writer to determine how to handle any property that equals `""`, as follows:

- If the RuntimeIgnoreNull property of the writer is 0 (the default), the *XMLNIL* parameter controls how to export the property. *XMLNIL* is a class parameter and a property parameter; the property parameter takes precedence.
  - If *XMLNIL* is 0 (the default), the property is not projected. That is, it is not included in the XML document.
  - If *XMLNIL* is 1 and if the property is used as an element, the property is exported as follows:

    ```
    <PropName xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
    ```

  - If *XMLNIL* is 1 and if the property is used as an attribute, the property is not exported.
- If the RuntimeIgnoreNull property of the writer is 1, the property is exported as an empty element or an empty attribute (it is exported the same way as the value `$char(0)`, which is always exported as an empty element or an empty exported).

The RuntimeIgnoreNull property of the writer has no effect unless the *XMLIGNORENULL* is `"RUNTIME"` in the XML-enabled class.

## 2.11.1 Example: RuntimeIgnoreNull Is 0 (Default)

First, consider the following class:

```
Class EmptyStrings.Export Extends (%Persistent, %XML.Adaptor)
{

Parameter XMLNAME="Test";


Parameter XMLIGNORENULL = "RUNTIME";


///project this one as an element
///XMLNIL is 0, the default
Property Property1 As %String;


///project this one as an attribute
```

```
///XMLNIL is 0, the default
Property Property2 As %String(XMLPROJECTION = "ATTRIBUTE");


///project this one as an element with XMLNIL=1
Property Property3 As %String(XMLNIL = 1);


///project this one as an attribute with XMLNIL=1
Property Property4 As %String(XMLNIL=1,XMLPROJECTION="ATTRIBUTE");

}
```

If you create a new instance of this class (and do not set the values of any properties), and you then use %XML.Writer to generate output for it, you see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Test>
  <Property3 xsi:nil="true"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</Test>
```

### 2.11.2 Example: RuntimeIgnoreNull Is 1

In this example, we set the RuntimeIgnoreNull property of the writer equal to 1. When we generate output for the same object that we used in the last example, we see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Test Property2="" Property4="">
  <Property1></Property1>
  <Property3></Property3>
</Test>
```

In this case, because RuntimeIgnoreNull is 1, the *XMLNIL* parameter is not used. Instead, `""` is exported as an empty attribute or an empty element.

## 2.12 Exporting Type Information

An XML writer does not write type information by default. There are two options you can use to include type information in the output:

- The OutputTypeAttribute property of the writer. If this property is 1, the writer includes XML type information for all the elements within the objects that it writes (but not for the objects themselves). For example:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <Root xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Person>
      <Name xsi:type="s:string">Petersburg,Greta U.</Name>
      <DOB xsi:type="s:date">1949-05-15</DOB>
    </Person>
  </Root>
  ```

  Note that the appropriate namespace is added to the root of the XML document.

- The *className* argument of the **Object()** and **RootObject()** methods. You use this argument to specify the expected ObjectScript type of the object (the name of the class).

  If the argument is the same as the actual type, the writer does not include type information for the object.

  If the argument is different from the actual type, the writer includes the actual XML type for the object (which defaults to the class name). For example, suppose that you write output for an instance of `Test2.PersonWithAddress`, and suppose that you specify the *className* argument as `MyPackage.MyClass`. Because `MyPackage.MyClass` is not the same as the actual class name, the writer generates the following output:

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonWithAddress xsi:type="PersonWithAddress"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Name>Avery,Robert H.</Name>
  <Address>
    <City>Ukiah</City>
    <Zip>82281</Zip>
  </Address>
</PersonWithAddress>
```

For the complete argument list for the **Object()** and **RootObject()** methods, see the class reference.

# 2.13 Generating SOAP-Encoded XML

For the %XML.Writer class, the Format property controls the overall format of the output. This is one of the following:

- `"literal"`, the default, which is used in most of the examples in this book.
- `"encoded"`, encoded as described in the SOAP 1.1 standard.
- `"encoded12"`, encoded as described in the SOAP 1.2 standard.

## 2.13.1 Creating Inline References

In encoded format, any object-valued property is included as a reference, and the referenced object is exported as a separate element.

To export such properties inline instead of as separate elements, set the ReferencesInline property (of the writer) equal to 1.

The ReferencesInline property has no effect if Format is `"literal"`.

# 2.14 Controlling Unswizzling After Export

When you export a persistent XML-enabled object, the system automatically swizzles all needed information into memory as usual; this information includes object-valued properties. After exporting the object, InterSystems IRIS unswizzles any lists of objects but does not (by default) unswizzle single object references. In the case of large objects, this can result in <STORE> errors.

To cause any single object references to be unswizzled in this scenario, set the *XMLUNSWIZZLE* parameter in your XML-enabled class as follows:

```
Parameter XMLUNSWIZZLE = 1;
```

The default for this parameter is 0.

# 2.15 Controlling the Closing of Elements

An element that contains only attributes can be represented in either of the following ways:

```
<myelementname attribute="value" attribute="value" attribute="value"></myelementname>
<myelementname attribute="value" attribute="value" attribute="value"/>
```

The **Object()** method always exports elements with the first syntax. If you need to close an element with the second syntax shown here, write the object manually, as described in "Constructing an Element Manually," earlier in this chapter.

# 2.16 Other Options of the Writer

This section discusses other options provided by %XML.Writer:

- Canonicalize() method
- Shallow property
- Summary property
- Base64LineBreaks property
- CycleCheck property

## 2.16.1 Canonicalize() Method

The **Canonicalize()** method writes an XML node in canonicalized form. This method has the following signature:

```
method Canonicalize(node As %XML.Node, ByRef PrefixList, formatXML As %Boolean = 0) as %Status
```

Where

- *node* is a subtree of the document, as an instance of %XML.Node, which is discussed in the chapter "Representing an XML Document as a DOM."
- *PrefixList* is one of the following:
  - For *inclusive canonicalization*, specify *PrefixList* as `"c14n"`. In this case, the output is in the form given by XML Canonicalization Version 1.0, as specified by http://www.w3.org/TR/xml-c14n.
  - For *exclusive canonicalization*, specify *PrefixList* as a multidimensional array with the following nodes:

    | Node | Value |
    |---|---|
    | *PrefixList(prefix)* where *prefix* is a namespace prefix | Namespace used with this namespace prefix |

    In this case, the output is in the form given by Exclusive XML Canonicalization Version 1.0, as specified by http://www.w3.org/TR/xml-exc-c14n/.
- *formatXML* controls the format. If *formatXML* is true, then the writer uses the formatting specified for the writer instance rather than the formatting specified by the XML Canonicalization specification. The output is then not canonical XML, but has done the namespace processing for canonical XML. This option is useful for outputting a fragment of an XML document, such as the SOAP body in the **ProcessBodyNode()** callback from a web service, while still having some control of the format.

## 2.16.2 Shallow Property

The Shallow property of your writer instance affects the output of properties that have object values. This class property lets you force all such output to be shallow, that is, to force the output to contain IDs of the referenced objects, rather than the details of the objects. This property interacts with the *XMLDEFAULTREFERENCE* class parameter and *XMLREFERENCE*

property parameter of the XML-enabled object, as shown in the following table. This table shows the resulting output, for each case:

***Table 2–2: Effect of Shallow = 1***

| Values of *XMLREFERENCE* and *XMLDEFAULTREFERENCE* | Output if Shallow=1 |
|---|---|
| Property parameter *XMLREFERENCE* is `"SUMMARY"` or `"COMPLETE"` | No output is generated for this property |
| Property parameter *XMLREFERENCE* is `"ID"`, `"OID"`, or `"GUID"` | Output is generated for this property and is of the type ID, OID, or GUID, as appropriate |
| Property parameter *XMLREFERENCE* is not set, but class parameter *XMLDEFAULTREFERENCE* is `"SUMMARY"` or `"COMPLETE"` | No output is generated for this property |
| Property parameter *XMLREFERENCE* is not set, but class parameter *XMLDEFAULTREFERENCE* is `"ID"`, `"OID"`, or `"GUID"` | Output is generated for this property and is of the type ID, OID, or GUID, as appropriate |
| Neither the property parameter *XMLREFERENCE* nor the class parameter *XMLDEFAULTREFERENCE* is set | No output is generated for this property |

The Shallow property does not affect properties whose values are serial objects or properties that have non-object values.

Also see "Controlling the Form of the Projection for Object-valued Properties" in *Projecting Objects to XML*.

## 2.16.3 Summary Property

The Summary property of your writer instance controls whether to export the entire XML-enabled object or just its summary; this can have one of the following values:

- The value 0 exports the entire object; this is the default.
- The value 1 exports just the properties that are listed as the summary.

As noted in *Projecting Objects to XML*, the summary of an object is specified by its *XMLSUMMARY* class parameter; it is a comma-separated list of properties.

For example, suppose that you are generating output for the `Person` class, which is XML-enabled and suppose that your default output looks like this:

```
<Persons>
 <Person>
  <Name>Xenia,Yan T.</Name>
  <DOB>1986-10-21</DOB>
 </Person>
 <Person>
  <Name>Vivaldi,Ashley K.</Name>
  <DOB>1981-01-25</DOB>
 </Person>
</Persons>
```

Suppose you have set *XMLSUMMARY* equal to `"Name"` for the `Person` class. In this case, if your writer sets the Summary property to 1, the output would look like the following:

```
<Persons>
 <Person>
  <Name>Xenia,Yan T.</Name>
 </Person>
 <Person>
  <Name>Vivaldi,Ashley K.</Name>
 </Person>
</Persons>
```

### 2.16.4 Base64LineBreaks Property

You can include automatic line breaks for properties of type %Binary or %xsd.base64Binary. To do so, set the Base64LineBreaks property to 1 for your writer instance. In this case, the writer inserts an automatic line feed/carriage return after every 76 characters. The default value for this property is 0.

### 2.16.5 CycleCheck Property

The CycleCheck property of your writer instance controls whether the writer checks for any cycles (endless loops) within the referenced objects that could result in errors. The default is 1, which means that the writer does check for cycles.

If you are sure that there are no cycles, set CycleCheck to 0 for a slight improvement in performance.

## 2.17 Additional Example: Writer with Choice of Settings

The following method may be useful for people who want to experiment with properties of %XML.Writer. It accepts an input argument, which is a string that names a writer "version." Each writer version corresponds to specific settings of the properties of the writer instance.

```
Class Utils.Writer
{

/// given a "name", return a writer with those properties
ClassMethod CreateWriter(wname) As %XML.Writer
{
 set w=##class(%XML.Writer).%New()
 set w.Indent=1
 set w.IndentChars="   "
 if wname="DefaultWriter" {
  set w.Indent=0  ; set back to default
  }
 elseif wname="EncodedWriter" {
  set w.Format="encoded"
  }
 elseif wname="EncodedWriterRefInline" {
  set w.Format="encoded"
  set w.ReferencesInline=1
  }
 elseif wname="AttQualWriter" {
  set w.AttributeQualified=1
  }
 elseif wname="AttUnqualWriter" {
  set w.AttributeQualified=0 ; default
  }
 elseif wname="ElQualWriter" {
  set w.ElementQualified=1 ; default
  }
 elseif wname="ElUnqualWriter" {
  set w.ElementQualified=0
  }
 elseif wname="ShallowWriter" {
  set w.Shallow=1
  }
 elseif wname="SOAPWriter1.1" {
  set w.Format="encoded"
  set w.ReferencesInline=1
  }
 elseif wname="SOAPWriter1.2" {
  set w.Format="encoded12"
  set w.ReferencesInline=1
  }
 elseif wname="SummaryWriter" {
  set w.Summary=1
  }
 elseif wname="WriterNoXmlDecl" {
  set w.NoXMLDeclaration=1
  }
 elseif wname="WriterRefInline" {
  set w.ReferencesInline=1
```

```
  }
 elseif wname="WriterRuntimeIgnoreNull" {
  set w.RuntimeIgnoreNull=1
  }
 elseif wname="WriterSuppressXmlns" {
  set w.SuppressXmlns=1
  }
 elseif wname="WriterUTF16" {
  set w.Charset="UTF-16"
  }
 elseif wname="WriterWithDefNS" {
  set w.DefaultNamespace="www.Def.org"
  }
 elseif wname="WriterWithDefNSSuppressXmlns" {
  set w.DefaultNamespace="www.Def.org"
  set w.SuppressXmlns=1
  }
 elseif wname="WriterWithDtdSettings" {
  set w.DocType ="MyDocType"
  set w.SystemID = "http://www.mysite.com/mydoc.dtd"
  set w.PublicID = "-//W3C//DTD XHTML 1.0 Transitional//EN"
  set w.InternalSubset = ""
  }
 elseif wname="WriterXsiTypes" {
  set w.OutputTypeAttribute=1
  }
 quit w
}

}
```

The following fragment shows how this method was used to help generate examples for the documentation:

```
/// method to write one to a file
ClassMethod WriteOne(myfile,cls,element,wname,ns,local,rootns)
{
    set writer=..CreateWriter(wname)
    set mydir=..#MyDir
    set comment="Output for the class: "_cls
    set comment2="Writer settings: "_wname

    if $extract(mydir,$length(mydir))'="/" {set mydir=mydir_"/"}
    set file=mydir_myfile
    set status=writer.OutputToFile(file)
    if $$$ISERR(status) { do $System.Status.DisplayError(status) quit }

    set status=writer.WriteComment(comment)
    if $$$ISERR(status) { do $System.Status.DisplayError(status) quit }

    set status=writer.WriteComment(comment2)
...
}
```

Notice that the output will include two comment lines. One indicates the name of the XML-enabled class shown in the file. The other indicates the name of the writer settings used to generate the file. The output directory is controlled centrally (via a parameter), and this generic method includes arguments that are passed to both the **RootElement()** method and the **Object()** method.

# 3

# Importing XML into Objects

This chapter describes how to use the %XML.Reader class to import XML documents into InterSystems IRIS objects. It discusses the following topics:

- An overview and a simple example
- How to create an import method and the basic structure of such a method, with an example
- How to check for required elements and attributes
- How to handle unexpected elements and attributes
- How to handle empty elements and attributes
- How to skip earlier parts of the input document
- Other methods that you use less frequently
- Other properties that you can set to control the overall behavior of the reader
- How to customize how the reader behaves when it finds a correlated object
- Additional examples

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

You can also use %XML.Reader to read an arbitrary XML document and return a DOM (Document Object Model); see the chapter "Representing an XML Document as a DOM," later in this book.

## 3.1 Overview of Creating an XML Reader

InterSystems IRIS provides tools for reading an XML document and creating one or more instances of XML-enabled InterSystems IRIS objects that correspond to elements of that document. The essential requirements are as follows:

- The class definition for that object must extend %XML.Adaptor. With a few exceptions, the classes to which that object refers must also extend %XML.Adaptor. See *Projecting Objects to XML*.

  **Tip:** If the corresponding XML schema is available, you can use that to generate the class (and any supporting classes). See the chapter "Generating Classes from XML Schemas," later in this book.

- To import the XML document, create an instance of %XML.Reader and then invoke methods of that instance. These methods specify the XML source document, associate an XML element with an XML-enabled class, and read elements from the source into objects.

The %XML.Reader class works with the methods provided by the %XML.Adaptor class to do the following:

- It parses and validates the incoming XML document using the InterSystems IRIS SAX interface. The validation can include either DTD or XML Schema validation.
- It determines if any XML-enabled objects are correlated with the elements contained within the XML document and creates in-memory instances of these objects as it reads the document.

Note that the object instances created by %XML.Reader are not stored within the database; they are in-memory objects. If you want to store objects within the database, you must call the **%Save()** method (for persistent objects) or copy the relevant property values to a persistent object and save it. The application must also decide when to insert new data and when to update existing data; %XML.Reader has no way of making this distinction.

The following Terminal session shows a simple example. Here we read an XML file into a new object, examine that object, and then save that object:

```
GXML>Set reader = ##class(%XML.Reader).%New()

GXML>Set file="c:\sample-input.xml"

GXML>Set status = reader.OpenFile(file)

GXML>Write status
1
GXML>Do reader.Correlate("Person","GXML.Person")

GXML>Do reader.Next(.object,.status)

GXML>Write status
1
GXML>Write object.Name
Worthington,Jeff R.
GXML>Write object.Doctors.Count()
2
GXML>Do object.%Save()
```

This example uses the following sample XML file:

```
<Person GroupID="90455">
  <Name>Worthington,Jeff R.</Name>
  <DOB>1976-11-03</DOB>
  <Address>
    <City>Elm City</City>
    <Zip>27820</Zip>
  </Address>
  <Doctors>
    <Doctor>
      <Name>Best,Nora A.</Name>
    </Doctor>
    <Doctor>
      <Name>Weaver,Dennis T.</Name>
    </Doctor>
  </Doctors>
</Person>
```

# 3.2 Creating an Import Method

## 3.2.1 Overall Method Structure

Your method should do some or all of the following, in this order:

1. Create an instance of the %XML.Reader class.
2. Optionally specify the Format property of this instance, to specify the format of the file that you are importing; see "Reader Properties."

   By default, InterSystems IRIS assumes that XML files are in literal format. If your file is in SOAP-encoded format, you must indicate this so that the file can be read correctly.
3. Optionally set other properties of this instance; see "Reader Properties."
4. Open the source. To do so, use one of the following methods of %XML.Reader:

   - **OpenFile()** — Opens a file.
   - **OpenStream()** — Opens a stream.
   - **OpenString()** — Opens a string.
   - **OpenURL()** — Opens a URL.

     If the URL uses HTTPS, see "Accessing a Document at an HTTPS URL," later in this chapter.

   In each case, you can optionally specify a second argument for the method, to override the value of the Format property.
5. Correlate one or more XML element names in this file with InterSystems IRIS XML-enabled classes that have the corresponding structures. There are two ways to do this:

   - Use the **Correlate()** method, which has the following signature:

     ```
     method Correlate(element As %String,
                      class As %String,
                      namespace As %String)
     ```

     Where *element* is an XML element name, *class* is an InterSystems IRIS class name (with package), and *namespace* is an optional namespace URI.

     If you use the *namespace* argument, the match is restricted to the specified element name in the specified namespace. If you specify the *namespace* argument as "", that matches the default namespace given in the **Next()** method. If you do not use the *namespace* argument, then only the element name is used for the match.

     **Tip:** You can call the **Correlate()** method repeatedly to correlate more than one element, although the examples in this chapter show only one correlation.

   - Use the **CorrelateRoot()** method, which has the following signature:

     ```
     method CorrelateRoot(class As %String)
     ```

     Where *class* is an InterSystems IRIS class name (with package). This method specifies that the root element of the XML document is correlated to the specified class.
6. Instantiate the class instances as follows:

   - If you used **Correlate()**, loop through the correlated elements in the file, one element at a time. Within the loop, use the **Next()** method, which has the following signature:

     ```
     method Next(ByRef oref As %ObjectHandle,
                 ByRef sc As %Status,
                 namespace As %String = "") as %Integer
     ```

     Where *oref* is the object created by the method, *sc* is the status, and *namespace* is the default namespace for the file.

   - If you used **CorrelateRoot()**, call the **Next()** method once, which causes the correlated class to be instantiated.

The **Next()** method returns 0 when it reaches the end of the file. If you call **Next()** again after this, you will loop through the objects in the file again, starting at the top of the file. (The correlations you specified are still in effect.)

## 3.2.2 Error Checking

Most of the methods mentioned in the previous section return a status. You should check the status after each step and quit if appropriate.

## 3.2.3 Basic Import Example

Consider the following XML file called test.xml:

```
<Root>
   <Person>
      <Name>Elvis Presley</Name>
   </Person>
   <Person>
      <Name>Johnny Carson</Name>
   </Person>
</Root>
```

We start by defining an XML-enabled class, MyApp.Person, that is an object representation of a person:

```
Class MyApp.Person Extends (%Persistent,%XML.Adaptor)
{
Parameter XMLNAME = "Person";

Property Name As %String;
}
```

To import this file into an instance of a MyAppPerson class, we could write the following method:

```
ClassMethod Import()
{
    // Create an instance of %XML.Reader
    Set reader = ##class(%XML.Reader).%New()

    // Begin processing of the file
    Set status = reader.OpenFile("c:\test.xml")
    If $$$ISERR(status) {do $System.Status.DisplayError(status)}

    // Associate a class name with the XML element name
    Do reader.Correlate("Person","MyApp.Person")

    // Read objects from xml file
    While (reader.Next(.object,.status)) {
        Write object.Name,!
    }

    // If error found during processing, show it
    If $$$ISERR(status) {do $System.Status.DisplayError(status)}

}
```

This method performs several tasks:

- It parses the input file using the InterSystems IRIS SAX interface. This includes validating the document against its DTD or schema, if specified.
- The **Correlate()** method associates the class MyApp.MyPerson with the XML element `<Person>`; each child element in `<Person>` becomes a property of MyPerson.
- It reads each `<Person>` element from the input file until there are none left.
- Finally, if the loop terminates because of an error, the error is displayed on the current output device.

As mentioned above, this example does not store objects to the database. Because MyPerson is a persistent object, you could do this by adding the following lines within the While loop:

```
Set savestatus = object.%Save()
If $$$ISERR(savestatus) {do $System.Status.DisplayError(savestatus)}
```

### 3.2.4 Accessing a Document at an HTTPS URL

For the **OpenURL()** method, if the document is at a URL that requires SSL/TLS, do the following:

1. Use the Management Portal to create an SSL/TLS configuration that contains the details of the needed connection. For information, see the chapter "Using SSL/TLS with InterSystems IRIS" in the *Security Administration Guide*.

   This is a one-time step.

2. When you use %XML.Reader, set the SSLConfiguration property of the reader instance. For the value, specify the name of the SSL/TLS configuration that you created in the previous step.

Or, when you use %XML.Reader, also do the following:

1. Create an instance of %Net.HttpRequest.
2. Set the SSLConfiguration property of that instance equal to the configuration name of the SSL/TLS configuration that you created in the Management Portal.
3. Use that instance of %Net.HttpRequest as the third argument to **OpenURL()**.

For example:

```
set reader=##class(%XML.Reader).%New()
set httprequest=##class(%Net.HttpRequest).%New()
set httprequest.SSLConfiguration="mysslconfigname"
set status=reader.OpenURL("https://myurl",,httprequest)
```

#### 3.2.4.1 Accessing a Document When the Server Requires Authentication

If the server requires authentication, create an instance of %Net.HttpRequest and set the Username and Password properties of that instance. Also use SSL as described above (so also set the SSLConfiguration property). Then use that instance of %Net.HttpRequest as the third argument to **OpenURL()** as shown in the example above.

For more details on %Net.HttpRequest and authentication, see "Providing Login Credentials" in the chapter "Sending HTTP Requests" in *Using Internet Utilities*.

## 3.3 Checking for Required Elements and Attributes

By default, the **Next()** method does not check for the existence of elements and attributes that correspond to properties that are marked as Required. To cause the reader to check for the existence of such elements and attributes, set the CheckRequired property of the reader to 1 before calling **Next()**. The default value for this property is 0, for compatibility reasons.

If you set CheckRequired to 1, and if you call **Next()** and the imported XML is missing a required element or attribute, the **Next()** method sets the *sc* argument to an error code. For example:

```
SAMPLES>set next= reader.Next(.object,.status)

SAMPLES>w next
0
SAMPLES>d $system.Status.DisplayError(status)

ERROR #6318: Property required in XML document: ReqProp
```

# 3.4 Handling Unexpected Elements and Attributes

Because the source XML documents might contain unexpected elements and attributes, the %XML.Adaptor class provides parameters to specify how to react when you import such a document. For details, see the chapter "Special Topics" in *Projecting Objects to XML*.

# 3.5 Controlling How Empty Elements and Attributes Are Imported

When you XML-enable an object, you specify how null values and empty strings are projected to XML (see *Projecting Objects to XML*).

One of the options is to set *XMLIGNORENULL* equal to `"RUNTIME"` (not case-sensitive) in the XML-enabled class. In this case, when you use %XML.Reader to read an XML file and create InterSystems IRIS objects, InterSystems IRIS uses the value of the IgnoreNull property of the reader to determine how to handle empty elements or attributes, as follows:

- If the IgnoreNull property of the reader is 0 (the default), and an element or attribute is empty, the corresponding property is set equal to `$char(0)`
- If the IgnoreNull property of the reader is 1, and an element or attribute is empty, the corresponding property is not set and therefore equals `""`

The IgnoreNull property of the reader has no effect unless the *XMLIGNORENULL* is `"RUNTIME"` in the XML-enabled class; the other possible values for *XMLIGNORENULL* are 0 (the default), 1, and `"INPUTONLY"`; see *Projecting Objects to XML*.

## 3.5.1 Example: IgnoreNull Is 0 (Default)

First, consider the following class:

```
Class EmptyStrings.Import Extends (%Persistent, %XML.Adaptor)
{

Parameter XMLNAME="Test";

///Reader will set IgnoreNull property
Parameter XMLIGNORENULL = "RUNTIME";

Property PropertyA As %String;

Property PropertyB As %String;

Property PropertyC As %String;

Property PropertyD As %String(XMLPROJECTION = "ATTRIBUTE");

Property PropertyE As %String(XMLPROJECTION = "ATTRIBUTE");
}
```

Also consider the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Test PropertyD="">
  <PropertyA></PropertyA>
  <PropertyB xsi:nil="true"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</Test>
```

If you create an instance of %XML.Reader and use it to import this file into the previous class, you get the following result:

- `PropertyA` and `PropertyD` equal `$char(0)`.
- All other properties equal `""`.

For example, if you wrote a routine to examine each property, inspect its value, and write output, you might get something like the following:

```
PropertyA is $char(0)
PropertyB is null
PropertyC is null
PropertyD is $char(0)
PropertyE is null
```

### 3.5.2 Example: IgnoreNull Is 1

In a variation of the preceding example, we set the IgnoreNull property of the reader equal to 1. In this case, all properties equal `""`.

For example, if you wrote a routine to examine each property, inspect its value, and write output, you might get something like the following:

```
PropertyA is null
PropertyB is null
PropertyC is null
PropertyD is null
PropertyE is null
```

## 3.6 Skipping Earlier Parts of the Input Document

An XML document consists of a set of nodes; the root node is numbered 0, the first element within that is numbered 1, and so on. You can specify the node at which to start reading; this is particularly useful for large documents. To do so, set the Node property of the reader. For the value, specify an integer.

An example method follows:

```
ClassMethod DemoSkippingAhead(nodenumber as %Integer=0)
{
    Set cls="GXML.Person"
    Set filename="c:\GXML.Person.xml"
    Set element="Person"

    // Create an instance of %XML.Reader
    Set reader = ##class(%XML.Reader).%New()

    // Begin processing of the file
    Set status = reader.OpenFile(filename)
    If $$$ISERR(status) {Do $System.Status.DisplayError(status)}

    // Associate a class name with the XML element name
    Do reader.Correlate(element,cls)

    //skip ahead in the file
    Set reader.Node=nodenumber

    // Read objects from xml file
    While (reader.Next(.object,.status)) {
        Write "Node number "_reader.Node_" contains "_object.Name,!
    }
}
```

This method assumes a specific input file, class name, and element name. By default, this method starts at the beginning of the file. For example:

```
GXML>d ##class(Readers.ReadFile).DemoSkippingAhead()
Node number 3 contains Emerson,Chad I.
Node number 30 contains O'Rielly,Patricia L.
Node number 63 contains Hanson,Brendan T.
Node number 120 contains Orwell,Tara H.
Node number 195 contains Gold,Elvis O.
Node number 234 contains Klein,Brenda U.
Node number 252 contains Yezek,Kristen Q.
Node number 327 contains Quine,Angelo B.
Node number 378 contains Vivaldi,Milhouse J.
Node number 453 contains Yezek,Vincent D.
Node number 471 contains Xander,Juanita D.
Node number 522 contains Winters,Kyra R.
Node number 555 contains Woo,Michelle J.
Node number 636 contains Ihringer,Yan A.
Node number 654 contains West,Hannah N.
Node number 729 contains Xiang,Bob G.
Node number 762 contains Ximines,Howard H.
Node number 789 contains Quixote,Jocelyn P.
Node number 864 contains Hills,Charles E.
Node number 897 contains Evans,Milhouse R.
```

In contrast, we could skip ahead as follows:

```
GXML>d ##class(Readers.ReadFile).DemoSkippingAhead(700)
Node number 729 contains Xiang,Bob G.
Node number 762 contains Ximines,Howard H.
Node number 789 contains Quixote,Jocelyn P.
Node number 864 contains Hills,Charles E.
Node number 897 contains Evans,Milhouse R.
```

# 3.7 Other Useful Methods

On occasion, you may need to use the following additional methods of %XML.Reader:

- Use the **Rewind()** method if you need to restart reading from the beginning of the XML source document. This method clears all correlations.
- Use the **Close()** method if you want to explicitly close and clean up the import handler. The import handler is cleaned up automatically; this method is included for backward compatibility.

# 3.8 Reader Properties

You can set the following properties of %XML.Reader to control the overall behavior of your method:

- Use the UsePPGHandler property to specify whether the instance of %XML.Reader uses process-private globals when it parses the document. If this property is true, the instance uses process-private globals. If this property is false, the instance uses memory. If this property is not set (or equals an empty string), the instance uses the default, which is normally memory.

  Use the Format property to specify the overall format of the XML document. Specify one of the following values:

  – `"literal"`, the default, which is used in most of the examples in this chapter.

  – `"encoded"`, encoded as described in the SOAP 1.1 standard.

  – `"encoded12"`, encoded as described in the SOAP 1.2 standard.

  Note that you can override the Format property within the **OpenFile()**, **OpenStream()**, **OpenString()**, and **OpenURL()** methods.

  This property has no effect unless you use **Correlate()** and **Next()**.

- Use the Summary property to force the reader to import only the summary fields of the XML-enabled objects. As noted in *Projecting Objects to XML*, the summary of an object is specified by its *XMLSUMMARY* class parameter, which you specify as a comma-separated list of properties.
- Use the IgnoreSAXWarnings property to specify whether the reader should report warnings issued by the SAX parser.

%XML.Reader also provides properties that you can use to examine the document you are reading:

- The Document property contains an instance of %XML.Document that represents the entire, parsed document that you are reading. For information on %XML.Document, see the class reference.
- The Node property is a string that represents the current node of the XML document. Note that 0 means the document, that is, the parent of the root element.

For information on the EntityResolver, SAXFlags, and SAXSchemaSpec properties, see the chapter "Customizing How the SAX Parser Is Used." Note that the SAXMask property is ignored.

# 3.9 Redefining How the Reader Handles Correlated Objects

When an instance of %XML.Reader finds an XML element that is correlated to an XML-enabled class, the reader calls the **XMLNew()** method of that class, which in turn calls **%New()** by default. That is, when a reader finds a correlated element, it creates a new object of the correlated class. The new object is populated with the data that you read from the XML document.

You can customize this behavior by redefining **XMLNew()** in your XML-enabled class (or perhaps in your own custom XML adaptor). For example, this method can instead open an existing instance of that class. The existing instance then receives the data that you read from the XML document.

The following examples show how you can modify **XMLNew()** to update existing instances with new data from an XML document.

In both examples, for simplicity, we assume that a node in the XML document contains an ID that we can compare to the IDs in the extent for the class. We could of course compare the XML document to the existing objects in other ways.

## 3.9.1 When %XML.Reader Calls XMLNew()

For reference, %XML.Reader calls the **XMLNew()** method automatically in two circumstances:

- %XML.Reader calls **XMLNew()** when you call the **Next()** method of %XML.Reader, after you have correlated XML elements (in an external document) with an XML-enabled class. The **Next()** method gets the next element from the document, calls **XMLNew()** to create an instance of the appropriate object, and then imports the element into the object.
- Similarly, %XML.Reader calls **XMLNew()** for any object-valued properties of the correlated XML elements.

## 3.9.2 Example 1: Modifying XMLNew() in an XML-Enabled Class

First consider the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Person>
    <IRISID>4</IRISID>
    <Name>Quine,Maria K.</Name>
    <DOB>1964-11-14</DOB>
    <Address>
      <City>Hialeah</City>
      <Zip>94999</Zip>
    </Address>
    <Doctors>
      <Doctor>
        <Name>Vanzetti,Debra B.</Name>
      </Doctor>
    </Doctors>
  </Person>
...
```

This file maps to the following InterSystems IRIS class (partly shown):

```
Class GXML.PersonWithXMLNew Extends (%Persistent,%Populate,%XML.Adaptor)
{

Parameter XMLNAME="Person";

/// make sure this is the same as the XMLNAME of the property
/// in this class that is of type %XML.Id
Parameter NAMEOFEXPORTID As %String = "IRISID";

Property IdForExport
As %XML.Id(XMLNAME="IRISID",XMLPROJECTION="ELEMENT") [Private,Transient];

Property Name As %Name;

Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h");

Property Address As GXML.Address;

Property Doctors As list Of GXML.Doctor;

}
```

In this class, the purpose of the IdForExport property is to project the InterSystems IRIS internal ID to an element (`IRISID`) when objects of this class are exported. (In this particular example, this enables us to easily generate suitable files for import. Your class does not have to contain such a property.)

The *NAMEOFEXPORTID* parameter is used to indicate the element that we use for the InterSystems IRIS ID when we export objects of this class. This is included only for the convenience of the customized **XMLNew()** method that we have also added to this class. This method is defined as follows:

```
ClassMethod XMLNew(doc As %XML.Document, node As %Integer,
contOref As %RegisteredObject = "") As GXML.PersonWithXMLNew
{
    Set id=""
    Set tmpnode=doc.GetNode(node)
    Do tmpnode.MoveToFirstChild()
    Do {
        //compare data node to the string given by the NAMEOFEXPORTID parameter
        //which indicates the XMLNAME of the ids for this object
        If tmpnode.NodeData=..#NAMEOFEXPORTID {
            //get the text from this node; this corresponds to an id in the database
            Do tmpnode.GetText(.id)}
        } While tmpnode.MoveToNextSibling()

    //if there is no id in the given node, create a new object
    If id="" {
        Write !, "Creating a new object..."
        Quit ..%New()}

    //open the given object
    Set result=..%OpenId(id)

    //if the id doesn't correspond to an existing object, create a new object
    If result=$$$NULLOREF {
        Write !, "Creating a new object..."
        Quit ..%New()}

    Write !, "Updating an existing object..."
```

```
    Quit result
}
```

%XML.Reader calls this method when it reads an XML document and correlates a node to GXML.PersonWithXMLNew. This method looks at the value of the *NAMEOFEXPORTID* parameter in this class, which is `IRISID`. It then examines the node in the document with the element `IRISID` and gets its value.

If this ID corresponds to an existing object of this class, the method opens that instance. Otherwise, the method opens a new instance of this class. In both cases, the instance receives the properties as specified in the XML document.

Finally, the following utility class includes a method that opens the XML file and calls %XML.Reader:

```
Class GXML.DemoXMLNew Extends %RegisteredObject
{

ClassMethod ReadFile(filename As %String = "c:\temp\sample.xml")
{
    Set reader=##class(%XML.Reader).%New()
    Set sc=reader.OpenFile(filename)
    If $$$ISERR(sc) {Do $system.OBJ.DisplayError(sc) Quit  }

    Do reader.Correlate("Person","GXML.PersonWithXMLNew")

    //loop through elements in file
    While reader.Next(.person,.sc) {
        Write !,person.Name,!
        Set sc=person.%Save()
        If $$$ISERR(sc) {Do $system.OBJ.DisplayError(sc) Quit  }
    }

    Quit
}

}
```

When you run the preceding method, one of the following occurs for each <Person> element in the file:

- An existing object is opened, updated with details from the file, and saved.
- Or a new object is created, with details from the file.

For example:

```
GXML>d ##class(GXML.DemoXMLNew).ReadFile()

Updating an existing object...
Zampitello,Howard I.

Creating a new object...
Smyth,Linda D.

Creating a new object...
Vanzetti,Howard I.
```

## 3.9.3 Example 2: Modifying XMLNew() in a Custom XML Adaptor

In the second example, we create a custom XML adaptor to perform the same actions as shown in the first example. The adaptor class is as follows:

```
Class GXML.AdaptorWithXMLNew Extends %XML.Adaptor
{

/// make sure this is the same as the XMLNAME of the property in this class
/// that is of type %XML.Id
Parameter NAMEOFEXPORTID As %String = "IRISID";

Property IdForExport
As %XML.Id(XMLNAME="IRISID",XMLPROJECTION="ELEMENT") [Private,Transient];

ClassMethod XMLNew(document As %XML.Document, node As %Integer,
containerOref As %RegisteredObject = "") As %RegisteredObject
[CodeMode=objectgenerator,GenerateAfter=%XMLGenerate,ServerOnly=1]
```

```
{
 If %compiledclass.Name'="GXML.AdaptorWithXMLNew" {
  Do %code.WriteLine(" Set id="""")
  Do %code.WriteLine(" Set tmpnode=document.GetNode(node)")
  Do %code.WriteLine(" Do tmpnode.MoveToFirstChild()")
  Do %code.WriteLine(" Do {")
  Do %code.WriteLine("     If tmpnode.NodeData=..#NAMEOFEXPORTID ")
  Do %code.WriteLine("       {Do tmpnode.GetText(.id)}")
  Do %code.WriteLine(" } While tmpnode.MoveToNextSibling() ")
  Do %code.WriteLine(" If id="""" {")
  Do %code.WriteLine("  Write !,""Creating new object...""")
  Do %code.WriteLine("  Quit ##class("_%class.Name_").%New()}")
  Do %code.WriteLine(" set result=##class("_%class.Name_").%OpenId(id)")
  Do %code.WriteLine(" If result=$$$NULLOREF {")
  Do %code.WriteLine("     Write !,""Creating new object...""")
  Do %code.WriteLine("     Quit ##class("_%class.Name_").%New() }")
  Do %code.WriteLine(" Write !,""Updating existing object ...""")
  Do %code.WriteLine(" Quit result")

  }

 QUIT $$$OK
}

}
```

The IdForExport property and the *NAMEOFEXPORTID* parameter establish a convention for how we project the InterSystems IRIS internal ID to an element when objects of subclasses are exported. The intent is that if you redefine IdForExport in a subclass, you redefine *NAMEOFEXPORTID* correspondingly.

In this class, the **XMLNew()** method is a method generator. When this class (or any subclass) is compiled, InterSystems IRIS writes the code shown here into the body of this method. See the chapter "Method Generators" in *Defining and Using Classes*.

The following class extends our custom adaptor:

```
Class GXML.PersonWithXMLNew2
Extends (%Persistent, %Populate, GXML.AdaptorWithXMLNew)
{

Parameter XMLNAME = "Person";

Property Name As %Name;

Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h");

Property Address As GXML.Address;

Property Doctors As list Of GXML.Doctor;

}
```

When you run the sample `ReadFile` method shown earlier, for each `<Person>` element in the file, the method either creates and saves a new record or opens and updates an existing record.

# 3.10 Additional Examples

This section contains some additional examples.

## 3.10.1 Flexible Reader Class

The following example shows a more flexible reader method that accepts the filename, directory, class, and element as input arguments. The method saves each object that it reads.

```
Class Readers.BasicReader Extends %RegisteredObject
{

ClassMethod Read(mydir, myfile, class, element)
{
```

```
   set reader=##class(%XML.Reader).%New()
   if $extract(mydir,$length(mydir))'="/" {set mydir=mydir_"/"}
   set file=mydir_myfile
   set status=reader.OpenFile(file)
   if $$$ISERR(status) {do $System.Status.DisplayError(status)}

   do reader.Correlate(element,class)

   while reader.Next(.object,.status)
   {
      if $$$ISERR(status) {do $System.Status.DisplayError(status)}
      set status=object.%Save()
      if $$$ISERR(status) {do $System.Status.DisplayError(status)}
      }
}

}
```

Notice that when you read in a `Person` object, you automatically read in its corresponding `Address` object. (And when you save the `Person` object, you automatically save the corresponding `Address` object as well.) This is a fairly crude technique that would be suitable only for bulk load of data; it does not make any attempt to compare to or update existing data.

In order to use this method, you would need an XML-enabled class whose projection matched the incoming XML document. Suppose that you had XML-enabled classes named `MyApp.PersonWithAddress` and `MyApp.Address`. Also suppose that you had an XML document as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <Person>
      <Name>Able, Andrew</Name>
      <DOB>1977-10-06</DOB>
      <Address>
         <Street>6218 Clinton Drive</Street>
         <City>Reston</City>
         <State>TN</State>
         <Zip>87639</Zip>
      </Address>
   </Person>
</Root>
```

To read the objects in this file and save them to disk, you would do something like the following:

```
 set dir="C:\XMLread-these"
 set file="PersonData.txt"
 set cls="MyApp.PersonWithAddress"
 set element="Person"
 do ##class(Readers.BasicReader).Read(dir,file,cls,element)
```

## 3.10.2 Reading a String

The following method accepts an XML string, class, and element as input arguments. It saves each object that it reads.

```
Class Readers.BasicReader Extends %RegisteredObject
{

ClassMethod ReadString(string, class, element)
{
   set reader=##class(%XML.Reader).%New()
   set status=reader.OpenString(string)
   if $$$ISERR(status) {do $System.Status.DisplayError(status)}

   do reader.Correlate(element,class)

   while reader.Next(.object,.status)
   {
      if $$$ISERR(status) {do $System.Status.DisplayError(status)}
      set status=object.%Save()
      if $$$ISERR(status) {do $System.Status.DisplayError(status)}
      }
}

}
```

To use this method, you would do something like the following:

```
set cls="MyApp.Person"
set element="Person"
do ##class(Readers.BasicReader).ReadString(string,cls,element)
```

# 4

# Representing an XML Document as a DOM

The %XML.Document and %XML.Node classes enable you to represent an arbitrary XML document as a DOM (document object model). You can then navigate this object and modify it. You can also create a new DOM and add to it. This chapter discusses the following topics:

- How to open an existing XML document as a DOM
- How to get information about the namespaces used in a DOM
- How to navigate to nodes of the DOM
- Types of nodes in a DOM
- How to get information about the current node
- How to access basic attribute data of the current node
- Additional methods for accessing attributes
- How to create or modify a DOM
- How to generate XML output from a DOM or a node of a DOM

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

## 4.1 Opening an XML Document as a DOM

To open an existing XML document for use as a DOM, do the following:

1. Create an instance of the %XML.Reader class.
2. Optionally specify the Format property of this instance, to specify the format of the file that you are importing.

   By default, InterSystems IRIS assumes that XML files are in literal format. If your file is in SOAP-encoded format, you must indicate this so that the file can be read correctly.

   See "Reader Properties" in the chapter "Importing XML into Objects."

   This property has no effect unless you use **Correlate()** and **Next()**.
3. Open the source. To do so, use one of the following methods of %XML.Reader:

- **OpenFile()** — Opens a file.
- **OpenStream()** — Opens a stream.
- **OpenString()** — Opens a string.
- **OpenURL()** — Opens a URL.

In each case, you can optionally specify a second argument for the method to override the value of the Format property.

4. Access the Document property, which is a DOM. This property is an instance of %XML.Document and it provides methods that you can use to find information about the document as a whole. For example, **CountNamespace()** returns the total number of namespaces used by the DOM.

Or, if you have a stream that contains an XML document, call the **GetDocumentFromStream()** method of %XML.Document. This returns an instance of %XML.Document.

## 4.1.1 Example 1: Converting a File to a DOM

For example, the following method reads an XML file and returns an instance of %XML.Document that represents that document:

```
ClassMethod GetXMLDocFromFile(file) As %XML.Document
{
    set reader=##class(%XML.Reader).%New()

    set status=reader.OpenFile(file)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}

    set document=reader.Document
    quit document
}
```

## 4.1.2 Example 2: Converting an Object to a DOM

The following method accepts an OREF and returns an instance of %XML.Document that represents that object. The method assumes that the OREF is an instance of an XML-enabled class:

```
ClassMethod GetXMLDoc(object) As %XML.Document
{
    //make sure this is an instance of an XML-enabled class
    if '$IsObject(object){
        write "Argument is not an object"
        quit $$$NULLOREF
        }
    set classname=$CLASSNAME(object)
    set isxml=$CLASSMETHOD(classname,"%Extends","%XML.Adaptor")
    if 'isxml {
        write "Argument is not an instance of an XML-enabled class"
        quit $$$NULLOREF
        }

    //step 1 - write object as XML to a stream
    set writer=##class(%XML.Writer).%New()
    set stream=##class(%GlobalCharacterStream).%New()
    set status=writer.OutputToStream(stream)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}
    set status=writer.RootObject(object)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}

    //step 2 - extract the %XML.Document from the stream
    set status=##class(%XML.Document).GetDocumentFromStream(stream,.document)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}

   quit document
}
```

# 4.2 Getting the Namespaces of the DOM

When InterSystems IRIS reads an XML document and creates a DOM, it identifies all the namespaces used in the document and assigns an index number to each.

Your instance of %XML.Document class provides the following methods that you can use to find information about the namespaces in the document:

**CountNamespace()**

Returns the number of namespaces in the document.

**FindNamespace()**

Returns the index that corresponds to the given namespace.

**GetNamespace()**

Returns the XML namespace URI for the given index.

The following example method displays a report showing the namespaces used in a document:

```
ClassMethod ShowNamespaces(doc As %XML.Document)
{
    Set count=doc.CountNamespace()
    Write !, "Number of namespaces in document: "_count
    For i=1:1:count {
        Write !, "Namespace "_i_" is "_doc.GetNamespace(i)
}

}
```

Also see “Getting Information about the Current Node,” later in this chapter.

# 4.3 Navigating Nodes of the DOM

To access nodes of the document, you can use two different techniques:

- Use the **GetNode()** method of your instance of %XML.Document. This method accepts an integer, which indicates the node number, starting with 1.
- Call the **GetDocumentElement()** method of your instance of %XML.Document.

  This method returns an instance of %XML.Node, which provides properties and methods that you use to access information about the root node and to move to other nodes. The following subsections provide details on working with %XML.Node.

## 4.3.1 Moving to Child or Sibling Nodes

To move to child nodes or sibling nodes, use the following methods of your instance of %XML.Node:

- **MoveToFirstChild()**
- **MoveToLastChild()**
- **MoveToNextSibling()**
- **MoveToPreviousSibling()**

Each of these methods tries to moves to another node (as indicated by the name of the method). If this is possible, the method returns true. If not, it returns false and the focus is the same as it was before the method was called.

Each of these methods has one optional argument, *skipWhitespace*. If this argument is true, the method ignores any whitespace. The default for *skipWhitespace* is false.

### 4.3.2 Moving to the Parent Node

To move to the parent of the current node, use the **MoveToParent()** method of your instance of %XML.Node.

This method takes an optional argument, *restrictDocumentNode*. If this argument is true, the method does not move to the document node (the root). The default for *restrictDocumentNode* is false.

### 4.3.3 Moving to a Specific Node

To move to a specific node, you can set the NodeId property of your instance of %XML.Node. For example:

```
set saveNode = node.NodeId
//..... lots of processing
//...
// restore position
set node.NodeId=saveNode
```

### 4.3.4 Using the id Attribute

In some cases, the XML document may include an attribute named `id`, which is used to identify different nodes in the document. For example:

```
<?xml version="1.0"?>
<team>
<member id="alpha">Jack O'Neill</member>
<member id="beta">Samantha Carter</member>
<member id="gamma">Daniel Jackson</member>
</team>
```

If (like this example) the document uses an attribute named `id`, you can use it to navigate to that node. To do so, you use the **GetNodeById()** method of the document, which returns an instance of %XML.Node positioned at that node. (Notice that unlike most other navigation methods, this method is available from %XML.Document, rather than %XML.Node.)

## 4.4 DOM Node Types

The %XML.Document and %XML.Node classes recognize the following DOM node types:

- Element (`$$$xmlELEMENTNODE`)

  Note that these macros are defined in the %xml.DOM.inc include file.
- Text (`$$$xmlTEXTNODE`)
- Whitespace (`$$$xmlWHITESPACENODE`)

Other types of DOM nodes are simply ignored.

Consider the following XML document:

```
<?xml version="1.0"?>
<team>
<member id="alpha">Jack O'Neill</member>
<member id="beta">Samantha Carter</member>
<member id="gamma">Daniel Jackson</member>
</team>
```

When viewed as a DOM, this document consists of the following nodes:

***Table 4–1: Example of Document Nodes***

| NodeID | NodeType | LocalName | Notes |
|---|---|---|---|
| 0,29 | $$$xmlELEMENTNODE | team | |
| 1,29 | $$$xmlWHITESPACENODE | | This node is a child of the <team> node |
| 1,23 | $$$xmlELEMENTNODE | member | This node is a child of the <team> node |
| 2,45 | $$$xmlTEXTNODE | Jack O'Neill | This node is a child of the first <member> node |
| 1,37 | $$$xmlWHITESPACENODE | | This node is a child of the <team> node |
| 1,41 | $$$xmlELEMENTNODE | member | This node is a child of the <team> node |
| 3,45 | $$$xmlTEXTNODE | Samantha Carter | This node is a child of the second <member> node |
| 1,45 | $$$xmlWHITESPACENODE | | This node is a child of the <team> node |
| 1,49 | $$$xmlELEMENTNODE | member | This node is a child of the <team> node |
| 4,45 | $$$xmlTEXTNODE | Daniel Jackson | This node is a child of the third <member> node |
| 1,53 | $$$xmlWHITESPACENODE | | This node is a child of the <team> node |

For information on accessing the node type, localname, and other details, see the next section.

# 4.5 Getting Information about the Current Node

The following string properties of %XML.Node provide information about the current node. In all cases, an error is thrown if there is no current node.

**LocalName**

Local name of the current element node. An error is thrown if you try to access this property for another type of node.

**Namespace**

Namespace URI of the current element node. An error is thrown if you try to access this property for another type of node.

**NamespaceIndex**

Index of the namespace of the current element node.

When InterSystems IRIS reads an XML document and creates a DOM, it identifies all the namespaces used in the document and assigns an index number to each.

An error is thrown if you try to access this property for another type of node.

#### Nil

Equals true if `xsi:nil` or `xsi:null` is true or 1 for this element node. Otherwise, this property equals false.

#### NodeData

Value of a character node.

#### NodeId

ID of the current node. You can set this property in order to navigate to another node.

#### NodeType

Type of the current node, as discussed in the previous section.

#### QName

Qname of the element node. Only used for output as XML when the prefix is valid for the document.

The following methods provide additional information about the current node:

### GetText()

```
method GetText(ByRef text) as %Boolean
```

Gets the text contents of an element node. This method returns true if text is returned; in this case, the actual text is appended to the first argument, which is returned by reference.

### HasChildNodes()

```
method HasChildNodes(skipWhitespace As %Boolean = 0) as %Boolean
```

Returns true if the current node has child nodes; otherwise it returns false.

### GetNumberAttributes()

```
method GetNumberAttributes() as %Integer
```

Returns the number of the attributes of the current element. Attributes are discussed further later in this chapter.

## 4.5.1 Example

The following example method writes a report that gives information about the current node:

```
ClassMethod ShowNode(node as %XML.Node)
{
    Write !,"LocalName="_node.LocalName
    If node.NodeType=$$$xmlELEMENTNODE  {
        Write !,"Namespace="_node.Namespace
    }
    If node.NodeType=$$$xmlELEMENTNODE {
        Write !,"NamespaceIndex="_node.NamespaceIndex
     }
    Write !,"Nil="_node.Nil
    Write !,"NodeData="_node.NodeData
    Write !,"NodeId="_node.NodeId
    Write !,"NodeType="_node.NodeType
    Write !,"QName="_node.QName
    Write !,"HasChildNodes returns "_node.HasChildNodes()
```

```
    Write !,"GetNumberAttributes returns "_node.GetNumberAttributes()
    Set status=node.GetText(.text)
    If status {
        Write !, "Text of the node is "_text
        } else {
            Write !, "GetText does not return text"
        }
}
```

Example output might be as follows:

```
LocalName=staff
Namespace=
NamespaceIndex=
Nil=0
NodeData=staff
NodeId=1
NodeType=e
QName=staff
HasChildNodes returns 1
GetNumberAttributes returns 5
GetText does not return text
```

# 4.6 Basic Methods for Examining Attributes

You can use the following methods of %XML.Node to examine the attributes of the current node. Also see "Additional Methods for Examining Attributes," later in this chapter, for additional methods.

- **AttributeDefined()** — Returns nonzero (true) if the current element has an attribute with the given name.
- **FirstAttributeName()** — Returns the attribute name for the first attribute of the current element.
- **GetAttributeValue()** — Returns the value of the given attribute. If the element does not have the attribute, the method returns null.
- **GetNumberAttributes()** — Returns the number of the attributes of the current element.
- **LastAttributeName()** — Returns the attribute name of the last attribute of the current element.
- **NextAttributeName()** — Given an attribute name, this method returns the name of the next attribute in collation order, whether the specified attribute is valid or not.
- **PreviousAttributeName()** — Given an attribute name, this method returns the name of the previous attribute in collation order, whether the specified attribute is valid or not.

The following example walks through the attributes in a given node and writes a simple report:

```
ClassMethod ShowAttributes(node as %XML.Node)
{
    Set count=node.GetNumberAttributes()
    Write !, "Number of attributes: ", count
    Set first=node.FirstAttributeName()
    Write !, "First attribute is: ", first
    Write !, "   Its value is: ",node.GetAttributeValue(first)
    Set next=node.NextAttributeName(first)

    For i=1:1:count-2 {
        Write !, "Next attribute is: ", next
        Write !, "   Its value is: ",node.GetAttributeValue(next)
        Set next=node.NextAttributeName(next)
        }
    Set last=node.LastAttributeName()
    Write !, "Last attribute is: ", last
    Write !, "   Its value is: ",node.GetAttributeValue(last)
}
```

Consider the following sample XML document:

```
<?xml version="1.0"?>
<staff attr1="first" attr2="second" attr3="third" attr4="fourth" attr5="fifth">
  <doc>
    <name>David Marston</name>
  </doc>
</staff>
```

If you pass the first node of this document to the example method, you see the following output:

```
Number of attributes: 5
First attribute is: attr1
   Its value is: first
Next attribute is: attr2
   Its value is: second
Next attribute is: attr3
   Its value is: third
Next attribute is: attr4
   Its value is: fourth
Last attribute is: attr5
   Its value is: fifth
```

# 4.7 Additional Methods for Examining Attributes

This section discusses methods that you can use to get the name, value, namespace, QName, and value namespace for any attribute. These methods are grouped as follows:

- Methods that use only the attribute name
- Methods that use the attribute name and a namespace

**Note:** In the XML standard, an element can include multiple attributes with the same name, each in a different namespace. In InterSystems IRIS XML, however, this is not supported.

Also see "Basic Methods for Examining Attributes."

## 4.7.1 Methods That Use Only the Attribute Name

Use the following methods to obtain information about attributes.

### GetAttribute()

```
method GetAttribute(attributeName As %String,
                    ByRef namespace As %String,
                    ByRef value As %String,
                    ByRef valueNamespace As %String)
```

Returns data for the given attribute. This method returns the following values by reference:

- *namespace* is the namespace URI from the QName of the attribute.
- *value* is the attribute value.
- *valueNamespace* is the namespace URI to which the value belongs. For example, consider the following attribute:

  ```
  xsi:type="s:string"
  ```

  The value of this attribute is `string`, and this value is in the namespace that is declared elsewhere with the prefix `s`. Suppose that an earlier part of this document included the following namespace declaration:

  ```
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  ```

  In this case, *valueNamespace* would be `"http://www.w3.org/2001/XMLSchema"`.

#### GetAttributeNamespace()

```
method GetAttributeNamespace(attributeName As %String) as %String
```

Returns the namespace URI from QName of the attribute named *attributeName* for the current element.

#### GetAttributeQName()

```
method GetAttributeQName(attributeName As %String) as %String
```

Returns the QName of the given attribute.

#### GetAttributeValue()

```
method GetAttributeValue(attributeName As %String) as %String
```

Returns the value of the given attribute.

#### GetAttributeValueNamespace()

```
method GetAttributeValueNamespace(attributeName As %String) as %String
```

Returns the namespace of the value of the given attribute.

## 4.7.2 Methods That Use the Attribute Name and Namespace

To get information about attributes by using both their names and their namespaces, use the following methods:

#### GetAttributeNS()

```
method GetAttributeNS(attributeName As %String,
                      namespace As %String,
                      ByRef value As %String,
                      ByRef valueNamespace As %String)
```

Returns data for the given attribute, where *attributeName* and *namespace* specify the attribute of interest. This method returns the following data by reference:

- *value* is the attribute value.
- *valueNamespace* is the namespace URI to which the value belongs. For example, consider the following attribute:

  ```
  xsi:type="s:string"
  ```

  The value of this attribute is `string`, and this value is in the namespace that is declared elsewhere with the prefix `s`. Suppose that an earlier part of this document included the following namespace declaration:

  ```
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  ```

  In this case, *valueNamespace* would be `"http://www.w3.org/2001/XMLSchema"`.

#### GetAttributeQNameNS()

```
method GetAttributeQNameNS(attributeName As %String,
                           namespace As %String)
                           as %String
```

Returns the QName of the given attribute, where *attributeName* and *namespace* specify the attribute of interest.

### GetAttributeValueNS()

```
method GetAttributeValueNS(attributeName As %String,
                           namespace As %String)
                           as %String
```

Returns the value of the given attribute, where *attributeName* and *namespace* specify the attribute of interest.

### GetAttributeValueNamespaceNS

```
method GetAttributeValueNamespaceNS(attributeName As %String,
                                    namespace As %String)
                                    as %String
```

Returns the namespace of the value of the given attribute, where *attributeName* and *namespace* specify the attribute of interest.

## 4.8 Creating or Editing a DOM

To create a DOM or to modify an existing one, you use the following methods of %XML.Document:

### CreateDocument()

```
classmethod CreateDocument(localName As %String,
                           namespace As %String)
                           as %XML.Document
```

Returns a new instance of %XML.Document that consists of only a root element.

### AppendCharacter()

```
method AppendCharacter(text As %String)
```

Appends new character data node to the list of children of this element node. The current node pointer does not change; this node is still the parent of the appended child.

### AppendChild()

```
method AppendChild(type As %String)
```

Appends new node to the list of children of this node. The current node pointer does not change; this node is still the parent of the appended child.

### AppendElement()

```
method AppendElement(localName As %String,
                     namespace As %String,
                     text As %String)
```

Appends a new element node to the list of children of this node. If the text argument is specified, then character data is added as the child of the new element. The current node pointer does not change; this node is still the parent of the appended child.

### AppendNode()

```
method AppendNode(sourceNode As %XML.Node) as %Status
```

Appends a copy of the specified node as a child of this node. The node to copy may be from any document. The current node pointer does not change. This node is still the parent of the appended child.

### AppendTree()

```
method AppendTree(sourceNode As %XML.Node) as %Status
```

Appends a copy of the specified node, including all its children, as a child of this node. The tree to copy may be from any document, but this node may not be a descendant of the source node. The current node pointer does not change. This node is still the parent of the appended child.

### InsertNamespace()

```
method InsertNamespace(namespace As %String)
```

Adds the given namespace URI to the document.

### InsertCharacter()

```
method InsertCharacter(text as %String, ByRef child As %String, Output sc As %Status) as %String
```

Inserts a new character data node as a child of this node. The new character data is inserted just before the specified child node. The *child* argument is the node ID of the child node; it is passed by reference so that it may be updated after the insert. The nodeId of the inserted node is returned. The current node pointer does not change.

### InsertNode()

```
method InsertNode(sourceNode As %XML.Node, ByRef child As %String, Output sc As %Status) as
%String
```

Inserts a copy of the specified node as a child of this node. The node to copy may be from any document. The new node is inserted just before the specified child node. The *child* argument is the node ID of the child node; it is passed by reference so that it may be updated after the insert. The nodeId of the inserted node is returned. The current node pointer does not change.

### InsertTree()

```
method InsertTree(sourceNode As %XML.Node, ByRef child As %String, Output sc As %Status) as
%String
```

Inserts a copy of the specified node, including its children, as a child of this node. The tree to copy may be from any document, but this node may not be a descendant of the source node. The new node is inserted just before the specified child node. The *child* argument is the node ID of the child node; it is passed by reference so that it may be updated after the insert. The nodeId of the inserted node is returned. The current node pointer does not change.

### Remove()

```
method Remove()
```

Removes the current node and make its parent the current node.

### RemoveAttribute()

```
method RemoveAttribute(attributeName As %String)
```

Removes the given attribute.

### RemoveAttributeNS()

```
method RemoveAttributeNS(attributeName As %String,
                         namespace As %String)
```

Removes the given attribute, where *attributeName* and *namespace* specify the attribute of interest.

### ReplaceNode()

```
method ReplaceNode(sourceNode As %XML.Node) as %Status
```

Replaces the node with a copy of the specified node. The node to copy may be from any document. The current node pointer does not change.

### ReplaceTree()

```
method ReplaceTree(sourceNode As %XML.Node) as %Status
```

Replaces the node with a copy of the specified node, including all its children. The tree to copy may be from any document, but may not be a descendant of the source node. The current node pointer does not change.

### SetAttribute()

```
method SetAttribute(attributeName As %String,
                    namespace As %String = "",
                    value As %String = "",
                    valueNamespace As %String = "")
```

Sets data for an attribute of the current element. Here:

- *attributeName* is the name of the attribute.
- *namespace* is the namespace URI from QName of the attribute named *attributeName* for this element.
- *value* is the attribute value.
- *valueNamespace* is the namespace URI corresponding to the prefix when the attribute value is of the form "prefix:value".

# 4.9 Writing XML Output from a DOM

You can serialize a DOM or a node of a DOM and generate XML output. To do this, you use the following methods of %XML.Writer:

### Document()

```
method Document(document As %XML.Document) as %Status
```

Given an instance of %XML.Document, this method writes the document to the currently specified output destination.

### DocumentNode()

```
method DocumentNode(document As %XML.Node) as %Status
```

Given an instance of %XML.Node, this method writes the node to the currently specified output destination.

### Tree()

```
method Tree(node As %XML.Node) as %Status
```

Given an instance of %XML.Node, this method writes the node and its tree of descendants to the currently specified output destination.

For information on specifying the output destination and setting properties of %XML.Writer, see the chapter "Writing XML Output from Objects."

# 5

# Encrypting XML Documents

This chapter describes how to encrypt XML documents. It discusses the following topics:

- Overview
- How to create an encrypted XML document
- How to decrypt an encrypted XML document

**Tip:** You might find it useful to enable SOAP logging in this namespace so that you receive more information about any errors; see "InterSystems IRIS SOAP Log" in "Troubleshooting SOAP Problems in InterSystems IRIS" in the book *Creating Web Services and Web Clients*.

## 5.1 About Encrypted XML Documents

An encrypted XML document includes the following elements:

- An `<EncryptedData>` element, which includes encrypted data encrypted by a randomly generated symmetric key. (It is more efficient to encrypt with a symmetric key than with a public key.)
- At least one `<EncryptedKey>` element. Each `<EncryptedKey>` element carries an encrypted copy of the symmetric key that was used to encrypt the data; it also includes an X.509 certificate with a public key. The recipient who has the matching private key can decrypt the symmetric key and then decrypt the `<EncryptedData>` element.
- (Optional) Other elements in clear text.

The following shows an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Container xmlns="http://www.w3.org/2001/04/xmlenc#">
  <EncryptedKey>
    <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
      <DigestMethod xmlns="http://www.w3.org/2000/09/xmldsig#"
                    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
    </EncryptionMethod>
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <X509Data>
        <X509Certificate>MIICnDCCAYQCAWUwDQYJKo... content omitted</X509Certificate>
      </X509Data>
    </KeyInfo>
    <CipherData>
      <CipherValue>J2DjVgcB8vQx3UCy5uejMB ... content omitted</CipherValue>
    </CipherData>
    <ReferenceList>
      <DataReference URI="#Enc-E0624AEA-9598-4436-A154-F746B07A2C55"></DataReference>
    </ReferenceList>
  </EncryptedKey>
```

```
  <EncryptedData Id="Enc-E0624AEA-9598-4436-A154-F746B07A2C55"
                 Type="http://www.w3.org/2001/04/xmlenc#Content">
    <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
    </EncryptionMethod>
    <CipherData>
      <CipherValue>LmoBK7+nDelTOsC3 ... content omitted</CipherValue>
    </CipherData>
  </EncryptedData>
</Container>
```

To create an encrypted document, you use the classes %XML.Security.EncryptedData and %XML.Security.EncryptedKey. These XML-enabled classes project to valid <EncryptedData> and <EncryptedKey> elements in the appropriate namespace.

# 5.2 Creating an Encrypted XML Document

The easiest way to create an encrypted XML document is as follows:

1. Define and use a general-purpose container class that can be projected directly to the desired XML document.
2. Create a stream that contains the XML that you will encrypt.
3. Encrypt that stream and write it, along with the corresponding encryption keys, to the appropriate properties of the container class.
4. Generate XML output for your container class.

## 5.2.1 Prerequisites for Encryption

Before you can encrypt a document, you must create an InterSystems IRIS credential set that contains the certificate of the entity to whom you are sending the encrypted document. In this case, you do not need (and should not have) the associated private key. For details, see the chapter "Setup and Other Common Activities" in *Securing Web Services*.

## 5.2.2 Requirements of the Container Class

A general-purpose container class must include the following:

- A property of type %XML.Security.EncryptedData that is projected as the <EncryptedData> element.

  This property will carry the encrypted data.
- At least one property of type %XML.Security.EncryptedKey that is projected as the <EncryptedKey> element.

  These properties will carry the corresponding key information.

The following shows an example:

```
Class XMLEncryption.Container Extends (%RegisteredObject, %XML.Adaptor)
{

Property Data As %XML.Security.EncryptedData (XMLNAME="EncryptedData");

Property Key As %XML.Security.EncryptedKey (XMLNAME="EncryptedKey");

Parameter NAMESPACE = "http://www.w3.org/2001/04/xmlenc#";

//methods

}
```

## 5.2.3 Generating an Encrypted XML Document

To generate and write an encrypted document, do the following:

1. Create a stream that contains an XML document.

   To do this, you typically use %XML.Writer to write output for an XML-enabled object to a stream.

2. Create at least one instance of %SYS.X509Credentials that accesses the InterSystems IRIS credential set of the entity to whom you are going to give the encrypted document. To do so, call the **GetByAlias()** class method of this class. For example:

   ```
   set credset=##class(%SYS.X509Credentials).GetByAlias("recipient")
   ```

   To run this method, you must be logged in as a user included in the OwnerList for that credential set, or the OwnerList must be null. Also see "Retrieving a Credential Set Programmatically" in *Securing Web Services*.

3. Create at least one instance of %XML.Security.EncryptedKey. To create an instance of this class, use the **CreateX509()** class method of this class. For example:

   ```
   set enckey=##class(%XML.Security.EncryptedKey).Createx509(credset,encryptionOptions,referenceOption)
   ```

   - *credset* is the instance of %SYS.X509Credentials that you just created.
   - *encryptionOptions* is `$$$SOAPWSIncludeNone` (there are other options, but they do not apply in this scenario).

     This macro is defined in the %soap.inc include file.
   - *referenceOption* specifies the nature of the reference to the encrypted element. For permitted values, see "Reference Options for X.509 Certificates" in *Securing Web Services*.

   The macros used here are defined in the %soap.inc include file.

4. Create an instance of %Library.ListOfObjects and use its **Insert()** method to insert the instances of %XML.Security.EncryptedKey that you just created.

5. Create an instance of %XML.Security.EncryptedData by using the **%New()** method. For example:

   ```
   set encdata=##class(%XML.Security.EncryptedData).%New()
   ```

6. Use the **EncryptStream()** instance method of %XML.Security.EncryptedData to encrypt the stream that you created in step 2. For example:

   ```
   set status=encdata.EncryptStream(stream,encryptedKeys)
   ```

   - *stream* is the stream that you created in step 1.
   - *encryptedKeys* is the list of keys that you created in step 4.

7. Create and update an instance of your container class.

   - Write the list of keys to the appropriate property of this class.
   - Write the instance of %XML.Security.EncryptedData to the appropriate property of this class.

   The details depend on your class.

8. Use %XML.Writer to generate output for your container class. See the chapter "Writing XML Output from Objects."

For example, the container class shown previously also includes the following method:

```
ClassMethod Demo(filename = "",obj="")
{
#Include %soap

    if (obj="") {
        set obj=##class(XMLEncryption.Person).GetPerson()
    }

    //create stream from this XML-enabled object
    set writer=##class(%XML.Writer).%New()
    set stream=##class(%GlobalCharacterStream).%New()
    set status=writer.OutputToStream(stream)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }
    set status=writer.RootObject(obj)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }
    do stream.Rewind()

    set container=..%New()             ; this is the object we will write out
    set cred=##class(%SYS.X509Credentials).GetByAlias("servercred")
    set parts=$$$SOAPWSIncludeNone
    set ref=$$$KeyInfoX509Certificate
    set key=##class(%XML.Security.EncryptedKey).CreateX509(cred,parts,ref)
    set container.Key=key        ; this detail depends on the class

    //need to create a list of keys (just one in this example)
    set keys=##class(%Collection.ListOfObj).%New()
    do keys.Insert(key)

    set encdata=##class(%XML.Security.EncryptedData).%New()

    set status=encdata.EncryptStream(stream,keys)
    set container.Data=encdata   ; this detail depends on the class

    // write output for the container
    set writer=##class(%XML.Writer).%New()
    set writer.Indent=1
    if (filename'="") {
        set status=writer.OutputToFile(filename)
        if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}
        }
    set status=writer.RootObject(container)
    if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}
}
```

This method can accept the OREF of any XML-enabled class; if none is provided, a default is used.

# 5.3 Decrypting an Encrypted XML File

## 5.3.1 Prerequisites for Decryption

Before you can decrypt an encrypted XML document, you must provide both of the following:

- Trusted certificates for InterSystems IRIS to use.
- An InterSystems IRIS credential set whose private key matches the public key used in the encryption.

For details, see the chapter "Setup and Other Common Activities" in *Securing Web Services*.

## 5.3.2 Decrypting the Document

To decrypt an encrypted XML document, do the following:

1. Create an instance of %XML.Reader and use it to open the document.

   See the chapter "Importing XML into Objects," earlier in this book.

2. Get the Document property of your reader. This is an instance of %XML.Document that contains the XML document as DOM.

3. Use the **Correlate()** method of your reader to correlate the <EncryptedKey> element or elements with the class %XML.Security.EncryptedKey. For example:

```
do reader.Correlate("EncryptedKey","%XML.Security.EncryptedKey")
```

4. Iterate through the document to read the <EncryptedKey> element or elements. To do this, you use the **Next()** method of the reader, which returns an imported object, if any, by reference. For example:

```
if 'reader.Next(.ikey,.status) {
    write !,"Unable to import key",!
    do $system.OBJ.DisplayError(status)
    quit
    }
```

   The imported object is an instance of %XML.Security.EncryptedKey.

5. Create an instance of %Library.ListOfObjects and use its **Insert()** method to insert the instances of %XML.Security.EncryptedKey that you just obtained from the document.

6. Call the **ValidateDocument()** method of the class %XML.Security.EncryptedData.

```
set status=##class(%XML.Security.EncryptedData).ValidateDocument(.doc,keys)
```

   The first argument, returned by reference, is the modified version of the DOM that you retrieved in step 2. The second argument is the list of keys from the previous step.

7. Optionally use %XML.Writer to generate output for the modified DOM. See the chapter "Writing XML Output from Objects."

For example, the container class shown earlier contains the following class method:

```
ClassMethod DecryptDoc(filename As %String)
{
#Include %soap
    set reader=##class(%XML.Reader).%New()
    set status=reader.OpenFile(filename)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }

    set doc=reader.Document
    //get <Signature> element
    do reader.Correlate("EncryptedKey","%XML.Security.EncryptedKey")
    if 'reader.Next(.ikey,.status) {
        write !,"unable to import key",!
        do $system.OBJ.DisplayError(status)
     quit
    }

    set keys=##class(%Collection.ListOfObj).%New()
    do keys.Insert(ikey)
    // the following step returns the decrypted document
    set status=##class(%XML.Security.EncryptedData).ValidateDocument(.doc,keys)

    set writer=##class(%XML.Writer).%New()
    set writer.Indent=1
    do writer.Document(doc)
    quit $$$OK
}

}
```

# 6

# Signing XML Documents

This chapter describes how to add digital signatures to XML documents. It discusses the following topics:

- Overview
- How to digitally sign a document
- How to validate digital signatures

**Tip:** You might find it useful to enable SOAP logging in this namespace so that you receive more information about any errors; see "InterSystems IRIS SOAP Log" in "Troubleshooting SOAP Problems in InterSystems IRIS" in the book *Creating Web Services and Web Clients*.

For information on alternative digest, signature, and canonicalization methods, see "Adding Digital Signatures" in *Securing Web Services*.

## 6.1 About Digitally Signed Documents

A digitally signed XML document includes one or more `<Signature>` elements, each of which is a digital signature. Each `<Signature>` element signs a specific element in the document as follows:

- Each signed element has an `Id` attribute, which equals some unique value. For example:

  ```
  <Person xmlns="http://mynamespace" Id="123456789">
  ```

- A `<Signature>` element includes a `<Reference>` element that points to that `Id` as follows:

  ```
  <Reference URI="#123456789">
  ```

  The `<Signature>` element is signed by a private key. This element includes an X.509 certificate signed by a signing authority. If the recipient of the signed document trusts this signing authority, the recipient can then validate the certificate and use the contained public key to validate the signature.

**Note:** InterSystems IRIS also supports a variation in which the signed elements have an attribute named `ID` rather than `Id`. For information, see the last section in this chapter.

The following shows an example, with whitespace added for readability:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person xmlns="http://mynamespace" Id="123456789">
  <Name>Persephone MacMillan</Name>
  <DOB>1976-02-20</DOB>
```

```
  <s01:Signature xmlns="http://www.w3.org/2000/09/xmldsig#"
                 xmlns:s01="http://mynamespace"
                 s02:Id="Id-BC0B1674-758D-40B9-84BF-F7BAA3AA19F4"
xmlns:s02="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
      </CanonicalizationMethod>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
      </SignatureMethod>
      <Reference URI="#123456789">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature">
          </Transform>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml1317c14n-20010315">
          </Transform>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
        <DigestValue>FHwW2U58bztLI4cIE/mp+nsBNZg=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>MTha3zLoj8Tg content omitted</SignatureValue>
    <KeyInfo>
      <X509Data>
        <X509Certificate>MIICnDCCAYQCAWUwDQYJ content omitted</X509Certificate>
      </X509Data>
    </KeyInfo>
  </s01:Signature>
</Person>
```

To create digital signatures, you use the class %XML.Security.Signature. This is an XML-enabled class whose projection is a valid `<Signature>` element in the appropriate namespace.

# 6.2 Creating a Digitally Signed XML Document

To create a digitally signed XML document, you use %XML.Writer to generate output for one or more appropriately defined XML-enabled objects.

Before you generate output for the objects, you must create the needed signatures and write them to the objects, so that the information is available to be written to the destination.

## 6.2.1 Prerequisites for Signing

Before you can sign a document, you must create at least one InterSystems IRIS credential set. An InterSystems IRIS credential set is an alias for the following set of information, stored in the system manager's database:

- A certificate, which contains a public key. The certificate should be signed by a signing authority that is trusted by the recipient of the document.
- The associated private key, which InterSystems IRIS uses when needed but never sends.

  The private key is needed for signing.
- (Optional) The password for the private key, which InterSystems IRIS uses when needed but never sends. You can either load the private key or you can supply it at runtime.

For details, see the chapter "Setup and Other Common Activities" in *Securing Web Services*.

## 6.2.2 Requirements of the XML-Enabled Class

The XML-enabled class must include the following:

- A property that is projected as the `Id` attribute.

- At least one property of type %XML.Security.Signature that is projected as the <Signature> element. (An XML document can contain multiple <Signature> elements.)

Consider the following class:

```
Class XMLSignature.Simple Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter NAMESPACE = "http://mynamespace";

Parameter XMLNAME = "Person";

Property Name As %String;

Property DOB As %String;

Property PersonId As %String(XMLNAME = "Id", XMLPROJECTION = "ATTRIBUTE");

Property MySig As %XML.Security.Signature(XMLNAME = "Signature");

//methods
}
```

## 6.2.3 Generating and Adding the Signature

To generate and add the digital signature, do the following:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Create an instance of %SYS.X509Credentials that accesses the appropriate InterSystems IRIS credential set. To do so, call the **GetByAlias()** class method of %SYS.X509Credentials.

   ```
   classmethod GetByAlias(alias As %String, pwd As %String) as %SYS.X509Credentials
   ```

   - *alias* is the alias for the certificate.
   - *pwd* is the private key password. The private key password is needed only if the associated private key is encrypted and if the password was not loaded when the private key file was loaded.

   To run this method, you must be logged in as a user included in the OwnerList for that credential set, or the OwnerList must be null. Also see "Retrieving a Credential Set Programmatically" in *Securing Web Services*.
3. Create an instance of %XML.Security.Signature that uses the given credential set. To do so, call the **CreateX509()** class method of that class:

   ```
   classmethod CreateX509(credentials As %SYS.X509Credentials,
   signatureOption As %Integer,              referenceOption As %Integer) as
   %XML.Security.Signature
   ```

   - *credentials* is the instance of %SYS.X509Credentials that you just created.
   - *signatureOption* is $$$SOAPWSIncludeNone (there are other options, but they do not apply in this scenario)
   - *referenceOption* specifies the nature of the reference to the signed element. For permitted values, see "Reference Options for X.509 Certificates" in *Securing Web Services*.

   The macros used here are defined in the %soap.inc include file.
4. Get the value of the Id attribute, for the Id to which this signature will point.

   This detail depends on the definition of your XML-enabled object.

5. Create an instance of %XML.Security.Reference to point to that `Id`. To do so, call the **Create()** class method of that class:

```
ClassMethod Create(id As %String,                    algorithm As %String,
            prefixList As %String)
```

   *id* is the Id to which this reference should point.

   *algorithm* should be one of the following:

   - `$$$SOAPWSEnvelopedSignature_","_$$$SOAPWSexcc14n` — Use this version for exclusive canonicalization.
   - `$$$SOAPWSEnvelopedSignature` — This is equivalent to the preceding option.
   - `$$$SOAPWSEnvelopedSignature_","_$$$SOAPWSexcc14n` — Use this version for inclusive canonicalization.

6. For your signature object, call the **AddReference()** method to add this reference to the signature:

```
Method AddReference(reference As %XML.Security.Reference)
```

7. Update the appropriate property of your XML-enabled class to contain the signature.

   This detail depends on your XML-enabled class. For example:

```
set object.MySig=signature
```

8. Create an instance of %XML.Document that contains your XML-enabled object serialized as XML.

   This is necessary because the signature must include information about the signed document.

   See "Example 2: Converting an Object to a DOM," earlier in this book.

   **Note:** This document does not contain whitespace.

9. Call the **SignDocument()** method of your signature object:

```
Method SignDocument(document As %XML.Document) As %Status
```

   The argument for this method is the instance of %XML.Document that you just created. The **SignDocument()** method uses information in that instance to update the signature object.

10. Use %XML.Writer to generate output for the object. See the chapter "Writing XML Output from Objects."

    **Note:** The output that you generate must include the same whitespace (or lack of whitespace) as contained in the document used in the signature. The signature contains a digest of the document, and the digest will not match the document if you set the Indent property to 1 in the writer.

For example:

```
Method WriteSigned(filename As %String = "")
{
#Include %soap
    //create a signature object
    set cred=##class(%SYS.X509Credentials).GetByAlias("servercred")
    set parts=$$$SOAPWSIncludeNone
    set ref=$$$KeyInfoX509Certificate

    set signature=##class(%XML.Security.Signature).CreateX509(cred,parts,ref,.status)
    if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}

    // get the Id attribute of the element we will sign;
    set refid=$this.PersonId ; this detail depends on structure of your classes
```

```
    // then create a reference to that Id within the signature object
    set algorithm=$$$SOAPWSEnvelopedSignature_","_$$$SOAPWSc14n
    set reference=##class(%XML.Security.Reference).Create(refid,algorithm)
    do signature.AddReference(reference)

    //set the MySig property so that $this has all the information needed
    //when we generate output for it
    set $this.MySig=signature ; this detail depends on structure of your classes

    //in addition to $this, we need an instance of %XML.Document
    //that contains the object serialized as XML
    set document=..GetXMLDoc($this)

    //use the serialized XML object to sign the document
    //this updates parts of the signature
    set status=signature.SignDocument(document)
    if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}

    // write output for the object
    set writer=##class(%XML.Writer).%New()
    if (filename'="") {
        set status=writer.OutputToFile(filename)
        if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}
    }
    do writer.RootObject($this)
}
```

The preceding instance method uses the following generic class method, which can be used with any XML-enabled object:

```
ClassMethod GetXMLDoc(object) As %XML.Document
{
    //step 1 - write object as XML to a stream
    set writer=##class(%XML.Writer).%New()
    set stream=##class(%GlobalCharacterStream).%New()
    set status=writer.OutputToStream(stream)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}
    set status=writer.RootObject(object)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}

    //step 2 - extract the %XML.Document from the stream
    set status=##class(%XML.Document).GetDocumentFromStream(stream,.document)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit $$$NULLOREF}
    quit document
}
```

### 6.2.3.1 Variation: Digital Signature with `URI=""` in Reference

As a variation, the `<Reference>` element for a signature can have `URI=""`, which is a reference to the root node of the XML document that contains the signature. To create a digital signature this way:

1. Optionally include the %soap.inc include file, which defines macros you might need to use.
2. Create an instance of %SYS.X509Credentials that accesses the appropriate InterSystems IRIS credential set. To do so, call the **GetByAlias()** class method of %SYS.X509Credentials, as described in the preceding steps.
3. Create an instance of %XML.Security.Signature that uses the given credential set. To do so, call the **CreateX509()** class method of that class, as described in the preceding steps.
4. Create an instance of %XML.Security.X509Data, as follows:

   ```
   set valuetype=$$$KeyInfoX509SubjectName_","_$$$KeyInfoX509Certificate
   set x509data=##class(%XML.Security.X509Data).Create(valuetype,cred)
   ```

   Where *cred* is the instance of %SYS.X509Credentials that you previously created. These steps create an `<X509Data>` element that contains an `<X509SubjectName>` element and an `<X509Certificate>` element.
5. Add the `<X509Data>` element to the `<KeyInfo>` element of the signature, as follows:

   ```
   do signature.KeyInfo.KeyInfoClauseList.Insert(x509data)
   ```

   Where *signature* is the instance of %XML.Security.Signature, and *x509data* is the instance of %XML.Security.X509Data.
6. Create an instance of %XML.Security.Reference as follows:

```
set algorithm=$$$SOAPWSEnvelopedSignature
set reference=##class(%XML.Security.Reference).Create("",algorithm)
```

7. Continue the preceding steps at step 6 (calling **AddReference()**).

# 6.3 Validating a Digital Signature

For any digitally signed document that you receive, you can validate the signatures. You do not need to have an XML-enabled class that matches the document contents.

## 6.3.1 Prerequisites for Validating Signatures

To validate digital a signature, you must first provide a trusted certificate to InterSystems IRIS for the signer. InterSystems IRIS can validate a signature if it can verify the signer's certificate chain from the signer's own certificate to a self-signed certificate from a certificate authority (CA) that is trusted by InterSystems IRIS, including intermediate certificates (if any).

For details, see the chapter "Setup and Other Common Activities" in *Securing Web Services*.

## 6.3.2 Validating a Signature

To validate the signatures in a digitally signed XML document, do the following:

1. Create an instance of %XML.Reader and use it to open the document.

   This class is discussed in the chapter "Importing XML into Objects," earlier in this book.

2. Get the Document property of your reader. This is an instance of %XML.Document that contains the XML document as DOM.

3. Use the **Correlate()** method of your reader to correlate the `<Signature>` element or elements with the class %XML.Security.Signature. For example:

   ```
   do reader.Correlate("Signature","%XML.Security.Signature")
   ```

4. Iterate through the document to read the `<Signature>` element or elements. To do this, you use the **Next()** method of the reader, which returns an imported object, if any, by reference. For example:

   ```
   if 'reader.Next(.isig,.status) {
       write !,"Unable to import signature",!
       do $system.OBJ.DisplayError(status)
       quit
       }
   ```

   The imported object is an instance of %XML.Security.Signature.

5. Call the **ValidateDocument()** method of the imported signature. The argument to this method must be the instance of %XML.Document that you retrieved earlier.

   ```
   set status=isig.ValidateDocument(document)
   ```

   For more validation options, see the class reference for this method in %XML.Security.Signature.

For example:

```
ClassMethod ValidateDoc(filename As %String)
{
    set reader=##class(%XML.Reader).%New()
    set status=reader.OpenFile(filename)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }

    set document=reader.Document
    //get <Signature> element
    //assumes there is only one signature
    do reader.Correlate("Signature","%XML.Security.Signature")
    if 'reader.Next(.isig,.status) {
        write !,"Unable to import signature",!
        do $system.OBJ.DisplayError(status)
        quit
    }
    set status=isig.ValidateDocument(document)
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }
}
```

# 6.4 Variation: Digital Signature That References an ID

In the typical case, a `<Signature>` element includes a `<Reference>` element that points to a unique `Id` elsewhere in the document. InterSystems IRIS also supports a variation in which the `<Reference>` element points to an attribute named `ID` rather than `Id`. In this variation, extra work is needed to sign the document *and* to validate the document.

To digitally sign the document, follow the steps in "Creating a Digitally Signed XML Document," with the following changes:

- For the XML-enabled class, include a property that is projected as the `ID` attribute rather than the `Id` attribute.
- When you generate and add the signature, call the **AddIDs()** method of the %XML.Document instance. Do this after you obtain the serialized XML document and before you call the **SignDocument()** method of the signature object. For example:

  ```
  //set the MySig property so that $this has all the information needed
  //when we generate output for it
  set $this.MySig=signature    ; this detail depends on structure of your classes

  //in addition to $this, we need an instance of %XML.Document
  //that contains the object serialized as XML
  set document=..GetXMLDoc($this)

  //***** added step when signature references an ID attribute *****
  do document.AddIDs()

  //use the serialized XML object to sign the document
  //this updates parts of the signature
  set status=signature.SignDocument(document)
  if $$$ISERR(status) {do $system.OBJ.DisplayError(status) quit}
  ```

- When you validate the document, include the following steps just before you call **Correlate()**:

  1. Call the **AddIDs()** method of the %XML.Document instance
  2. Call the **Rewind()** method of the XML reader.

  For example:

  ```
  set document=reader.Document

  //added steps when signature references an ID attribute
  do document.AddIDs()
  do reader.Rewind()

  //get <Signature> element
  do reader.Correlate("Signature","%XML.Security.Signature")
  ```

# 7

# Using %XML.TextReader

The %XML.TextReader class offers a simple, easy way to read arbitrary XML documents that may or may not map directly to InterSystems IRIS objects. Specifically, this class provides ways to navigate a well-formed XML document and view the information in it (elements, attributes, comments, namespace URIs, and so on). This class also provides complete document validation, based on either a DTD or an XML schema. Unlike %XML.Reader, however, %XML.TextReader does not provide a way to return a DOM. If you require a DOM, see the chapter "Importing XML into Objects," earlier in this book.

This chapter discusses the following topics:

- An overview of how to create a text reader method, with a couple of examples
- Details on the types of document nodes
- Available properties for each type of node
- A summary of the argument lists for the parse methods
- More information on how to navigate the document
- How to validate a document based on either a DTD or an XML schema document
- Examples that read files or objects and report on their namespace usage

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

## 7.1 Creating a Text Reader Method

To read an arbitrary XML document that does not necessarily have any relationship to an InterSystems IRIS object class, you invoke methods of the %XML.TextReader class, which opens the document and loads it into temporary storage as a *text reader object*. The text reader object contains a navigable tree of nodes, each of which contains information about the source document. Your method can then navigate the document and find out information about it. Properties of the object give you information about the document that depend on your current location within the document. If there are validation errors, those errors are also available as nodes in the tree.

### 7.1.1 Overall Structure

Your method should do some or all of the following:

1. Specify a document source, via the first argument of one of the following methods:

| Method | First Argument |
|---|---|
| **ParseFile()** | A file name, with complete path. Note that the filename and path must contain only ASCII characters. |
| **ParseStream()** | A stream |
| **ParseString()** | A string |
| **ParseURL()** | A URL |

   In any case, the source document must be a well-formed XML document; that is, it must obey the basic rules of XML syntax. Each of these methods returns a status (`$$$OK` or a failure code) to indicate whether the result was successful. You can test the status with the usual mechanisms; in particular, you can use **$System.Status.DisplayError(***status***)** to see the text of the error message.

   For each of these methods, if the method returns `$$$OK`, it returns by reference (its second argument) the text reader object that contains the information in the XML document.

   Additional arguments let you control entity resolution, validation, which items are found, and so on. These are described later in this chapter, in "Argument Lists for the Parse Methods."

2. Check the status returned by the parse method and quit if appropriate.

   If the parse method returned `$$$OK`, you have an text reader object that corresponds to the source XML document. You can navigate this object.

   Your document is likely to contain nodes such as `"element"`, `"endelement"`, `"startprefixmapping"`, and so on. The node types are listed in "Node Types," later in this chapter.

   **Important:** In the case of any validation errors, your document contains `"error"` or `"warning"` nodes. Your code should check for such nodes. See "Performing Validation."

3. Use one of the following instance methods to start reading the document.

   - Use **Read()** to navigate to the first node of the document.
   - Use **ReadStartElement()** to navigate to the first element of a specific type.
   - Use **MoveToContent()** to navigate to the first node of type `"chars"`.

   See "Navigating the Document," later in this chapter.

4. Get the values of the properties of interest for this node, if any. Available properties include Name, Value, Depth, and so on. See "Node Properties," later in this chapter.

5. Continue to navigate through the document as needed and get property values.

   If the current node is an element, you can use the **MoveToAttributeIndex()** or **MoveToAttributeName()** methods to move the focus to attributes of that element. To return to the element, if applicable, use **MoveToElement()**.

6. If needed, use the **Rewind()** method to return to the start of the document (before the first node). This is the only method that can go backward in the source.

After your method runs, the text reader object is destroyed and all related temporary storage is cleaned up.

## 7.1.2 Example 1

Here is a simple method that reads any XML file and shows the sequence number, type, name, and value of every node:

```
ClassMethod WriteNodes(myfile As %String)
{
    set status=##class(%XML.TextReader).ParseFile(myfile,.textreader)
    //check status
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
    //iterate through document, node by node
    while textreader.Read()
    {
        Write !, "Node ", textreader.seq, " is a(n) "
        Write textreader.NodeType," "
        If textreader.Name'=""
        {
            Write "named: ", textreader.Name
            }
            Else
            {
                Write "and has no name"
                }
        Write !, "    path: ",textreader.Path
        If textreader.Value'=""
        {
            Write !, "    value: ", textreader.Value
            }
        }
}
```

This example does the following:

1. It calls the **ParseFile()** class method. This reads the source file, creates a text reader object, and returns that in the variable doc by reference.
2. If **ParseFile()** is successful, the method then invokes the **Read()** method to find each successive node within the document.
3. For each node, the method writes output lines that contain the sequence number of the node, the node type, the node name (if any), the node path, and the node value (if any). Output is written to the current device.

Consider the following example source document:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="mystyles.css"?>
<Root>
   <s01:Person xmlns:s01="http://www.root.org">
      <Name attr="xyz">Willeke,Clint B.</Name>
      <DOB>1925-10-01</DOB>
   </s01:Person>
</Root>
```

For this source document, the preceding method generates the following output:

```
Node 1 is a(n) processinginstruction named: xml-stylesheet
    path:
    value: type="text/css" href="mystyles.css"
Node 2 is a(n) element named: Root
    path: /Root
Node 3 is a(n) startprefixmapping named: s01
    path: /Root
    value: s01 http://www.root.org
Node 4 is a(n) element named: s01:Person
    path: /Root/s01:Person
Node 5 is a(n) element named: Name
    path: /Root/s01:Person/Name
Node 6 is a(n) chars and has no name
    path: /Root/s01:Person/Name
    value: Willeke,Clint B.
Node 7 is a(n) endelement named: Name
    path: /Root/s01:Person/Name
Node 8 is a(n) element named: DOB
    path: /Root/s01:Person/DOB
Node 9 is a(n) chars and has no name
    path: /Root/s01:Person/DOB
    value: 1925-10-01
Node 10 is a(n) endelement named: DOB
    path: /Root/s01:Person/DOB
Node 11 is a(n) endelement named: s01:Person
    path: /Root/s01:Person
Node 12 is a(n) endprefixmapping named: s01
    path: /Root
```

```
    value: s01
Node 13 is a(n) endelement named: Root
    path: /Root
```

Notice that the comment has been ignored; by default, the %XML.TextReader class ignores comments. For information on changing this, see "Argument Lists for the Parse Methods," later in this chapter.

### 7.1.3 Example 2

The following example reads an XML file and lists every element in it:

```
ClassMethod ShowElements(myfile As %String)
{
    set status = ##class(%XML.TextReader).ParseFile(myfile,.textreader)
    //check status
    if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
    //iterate through document, node by node
    while textreader.Read()
    {
        if (textreader.NodeType = "element")
        {
            write textreader.Name,!
            }
        }
}
```

This method checks the type of each node, by using the NodeType property. If the node is an element, the method prints its name to the current device. For the XML source document shown earlier, this method generates the following output:

```
Root
s01:Person
Name
DOB
```

## 7.2 Node Types

Each node of a document is one of the following types:

***Table 7–1: Node Types in a Text Reader Document***

| Type | Description |
|---|---|
| `"attribute"` | An XML attribute. |
| `"chars"` | A set of characters (such as content of an element). The %XML.TextReader class recognizes other node types (`"CDATA"`, `"EntityReference"`, and `"EndEntity"`) but automatically converts them to `"chars"`. |
| `"comment"` | An XML comment. |
| `"element"` | The start of an XML element. |
| `"endelement"` | The end of an XML element. |
| `"endprefixmapping"` | End of the context where a namespace is declared. |
| `"entity"` | An XML entity. |
| `"error"` | A validation error found by the parser. See "Performing Validation." |
| `"ignorablewhitespace"` | The white space between markup in a mixed content model. |

| Type | Description |
|---|---|
| "processinginstruction" | An XML processing instruction. |
| "startprefixmapping" | An XML namespace declaration, which may or may not include a namespace. |
| "warning" | A validation warning found by the parser. See "Performing Validation." |

Notice that an XML element consists of multiple nodes. For example, consider the following XML fragment:

```
<Person>
   <Name>Willeke,Clint B.</Name>
   <DOB>1925-10-01</DOB>
</Person>
```

The SAX parser views this XML as the following set of nodes:

***Table 7–2: Example of Document Nodes***

| Node Number | Type of Node | Name of Node, If Any | Value of Node, If Any |
|---|---|---|---|
| 1 | element | Person | |
| 2 | element | Name | |
| 3 | chars | | Willeke,Clint B. |
| 4 | endelement | Name | |
| 5 | element | DOB | |
| 6 | chars | | 1925-10-01 |
| 7 | endelement | DOB | |
| 8 | endelement | Person | |

For example, notice that the <DOB> element is considered to be three nodes: an element node, a chars node, and an endelement node. Also notice that the contents of this element are available only as the value of the chars node.

# 7.3 Node Properties

As mentioned earlier, the %XML.TextReader class parses an XML document and creates an text reader object that consists of a set of nodes that correspond to the components of the document; the node types are described in "Document Nodes," earlier in this chapter.

When you change focus to a different node, the properties of the text reader object are updated to contain information about the node that you are currently examining. This section describes all the properties of the %XML.TextReader class.

### AttributeCount

If the current node is an element or an attribute, this property indicates the number of attributes of the element. Within a given element, the first attribute is numbered 1.

For any other type of node, this property is 0.

### Depth

Indicates the depth of the current node within the document. The root element is at depth 1; items outside the root element are at depth 0. Note that an attribute is at the same depth as the element to which it belongs. Similarly, an error or warning is at the same depth as the item that caused the error or warning.

### EOF

True if the reader has reached the end of the source document; false otherwise.

### HasAttributes

If the current node is an element, this property is true if that element has attributes (or false if it does not). If the current node is an attribute, this property is true.

For any other type of node, this property is false.

### HasValue

True if the current node is a type of node that has a value (even if that value is null). Otherwise this property is false. Specifically, this property is true for the following types of nodes:

- attribute
- chars
- comment
- entity
- ignorablewhitespace
- processinginstruction
- startprefixmapping

Note that HasValue is false for nodes of type error and warning, even though those node types have values.

### IsEmptyElement

True if the current node is an element and is empty. Otherwise this property is false.

### LocalName

For nodes of type attribute, element, or endelement, this is the name of the current element or attribute, without the namespace prefix. For all other types of nodes, this property is null.

### Name

Fully qualified name of the current node, as appropriate for the type of node. The following table gives the details:

*Table 7–3: Names for Nodes, by Type*

| Node Type | Name and Example |
|---|---|
| attribute | The name of the attribute. For example, if an attribute is: `groupID="GX078"` then Name is: `groupID` |

| Node Type | Name and Example |
|---|---|
| element<br>or<br>endelement | The name of the element. For example, if an element is:<br>`<s01:Person groupID="GX078">...</s01:Person>`<br>then Name is:<br>`s01:Person` |
| entity | The name of the entity. |
| startprefixmapping<br>or<br>endprefixmapping | The prefix, if any. For example, if a namespace declaration is as follows:<br>xmlns:s01="http://www.root.org"<br>then Name is:<br>`s01`<br>For another example, if a namespace declaration is as follows:<br>xmlns="http://www.root.org"<br>then Name is null. |
| processinginstruction | The target of the processing instruction. For example, if a processing instruction is:<br>`<?xml-stylesheet type="text/css" href="mystyles.css"?>`<br>then Name is:<br>`xml-stylesheet` |
| all other types | null |

### NamespaceUri

For nodes of type attribute, element, or endelement, this is the namespace to which attribute or element belongs, if any. For all other types of nodes, this property is null.

### NodeType

Type of the current node. See “Document Nodes,” earlier in this chapter.

### Path

Path to the element. For example, consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="mystyles.css"?>
<s01:Root xmlns:s01="http://www.root.org" xmlns="www.default.org">
   <Person>
      <Name>Willeke,Clint B.</Name>
      <DOB>1925-10-01</DOB>
      <GroupID>U3577</GroupID>
      <Address xmlns="www.address.org">
         <City>Newton</City>
         <Zip>56762</Zip>
      </Address>
   </Person>
</s01:Root>
```

For the `City` element, the Path property is `/s01:Root/Person/Address/City`. Other elements are treated similarly.

### ReadState

Indicates the overall state of the text reader object, one of the following:

- `"Initial"` means that the **Read()** method has not yet been called.
- `"Interactive"` means that the **Read()** method has been called at least once.
- `"EndOfFile"` means that the end of the file has been reached.

### Value

Value, if any, of the current node, as appropriate for the type of node. The following table gives the details:

*Table 7–4: Values for Nodes, by Type*

| Node Type | Value and Example |
|---|---|
| attribute | The value of the attribute. For example, if an attribute is: `groupID="GX078"` then Value is: `GX078` |
| chars | The content of the text node. For example, if an element is: `<DOB>1925-10-01</DOB>` then for the chars node, Value is: `1925-10-01` |
| comment | The content of the comment. For example, if a comment is: `<!--Comment here-->` then Value is: `Comment here` |
| entity | The definition of the entity. |
| error | The error message. For an example, see "Performing Validation," later in this chapter. |
| ignorablewhitespace | The content of the white space. |
| processinginstruction | The entire content of the processing instruction, excluding the target. For example, if a processing instruction is: `<?xml-stylesheet type="text/css" href="mystyles.css"?>` then Value is: `type="text/css" href="mystyles.css"?` |
| startprefixmapping | The prefix, followed by a space, followed by the URI. For example, if a namespace declaration is as follows: xmlns:s01="http://www.root.org" then Value is: `s01 http://www.root.org` |

| Node Type | Value and Example |
|---|---|
| warning | The warning message. For an example, see "Performing Validation," later in this chapter. |
| all other types (including element) | null |

**seq**

The sequence number of this node within the document. The first node is numbered 1. Note that an attribute has the same sequence number as the element to which it belongs.

# 7.4 Argument Lists for the Parse Methods

To specify a document source, you use the **ParseFile()**, **ParseStream()**, **ParseString()**, or **ParseURL()** method of your text reader. In any case, the source document must be a well-formed XML document; that is, it must obey the basic rules of XML syntax. For these methods, only the first two arguments are required. For reference, these methods have the following arguments, in order:

1. *Filename*, *Stream*, *String*, or *URL* — Document source.

   Note that for **ParseFile()**, the *Filename* argument must contain only ASCII characters.

2. *TextReader* — Text reader object, returned as an output parameter if the method returns `$$$OK`.

3. *Resolver* — An entity resolver to use when parsing the source. See "Performing Custom Entity Resolution" in the chapter "Customizing How the SAX Parser Is Used."

4. *Flags* — A flag or combination of flags to control the validation and processing performed by the SAX parser. See "Setting the Parser Flags" in the chapter "Customizing How the SAX Parser Is Used."

5. *Mask* — A mask to specify which items are of interest in the XML source. See "Specifying the Event Mask" in the chapter "Customizing How the SAX Parser Is Used."

   **Tip:** For the parsing methods of %XML.TextReader, the default mask is `$$$SAXCONTENTEVENTS`. Note that this ignores comments. To parse all possible types of nodes, use `$$$SAXALLEVENTS` for this argument. Note that these macros are defined in the %occSAX.inc include file.

6. *SchemaSpec* — A schema specification, against which to validate the document source. This argument is a string that contains a comma-separated list of namespace/URL pairs:

   ```
   "namespace URL,namespace URL"
   ```

   Here *namespace* is the XML namespace used for the schema and *URL* is a URL that gives the location of the schema document. There is a single space character between the namespace and URL values.

7. *KeepWhiteSpace* — An option to keep white space or not.

8. *pHttpRequest* — (For the **ParseURL()** method only) A request for the web server, as an instance of %Net.HttpRequest. By default, the system creates a new instance of %Net.HttpRequest and uses that, but you can instead make a request with a different instance of %Net.HttpRequest. This is useful in the case where you have a pre-existing %Net.HttpRequest with proxy and other properties already set. This option applies only to URLs of type http (not file or ftp, for example).

   For details on %Net.HttpRequest, see the book *Using Internet Utilities*. Or see the class documentation for %Net.HttpRequest.

# 7.5 Navigating the Document

To navigate through the document, you use the following methods of your text reader: **Read()**, **ReadStartElement()**, **MoveToAttributeIndex()**, **MoveToAttributeName()**, **MoveToElement()**, **MoveToContent()**, and **Rewind()**.

## 7.5.1 Navigating to the Next Node

To move to the next node in a document, use the **Read()** method. The **Read()** method returns a true value until there are no more nodes to read (that is, until the end of the document is reached). The previous examples used this method in a loop like the following:

```
 While (textreader.Read()) {
...
 }
```

## 7.5.2 Navigating to the First Occurrence of a Specific Element

You can move to the first occurrence of a specific element within a document. To do so, use the **ReadStartElement()** method. This method returns true unless the element is not found. If the element is not found, the method reaches the end of the file.

The **ReadStartElement()** method takes two arguments: the name of the element and (optionally) the namespace URI. Note that the %XML.TextReader class does not do any processing of namespace prefixes. Therefore the **ReadStartElement()** method regards the following two elements as having different names:

```
<Person>Smith,Ellen W. xmlns="http://www.person.org"</Person>
<s01:Person>Smith,Ellen W. xmlns:s01="http://www.person.org"</s01:Person>
```

## 7.5.3 Navigating to an Attribute

When you navigate to an element, if that element has attributes, you can navigate to them, in either of two ways:

- Use the **MoveToAttributeIndex()** method to move to a specific attribute by index (ordinal position of the attribute within the element). This method takes one argument: the index number of the attribute. Note that you can use the AttributeCount property to learn how many attributes a given element has; see "Node Properties," later in this chapter, for a list of all properties.
- Use the **MoveToAttributeName()** method to move to a specific attribute by name. This method takes two arguments: the name of the attribute and (optionally) the namespace URI. Note that the %XML.TextReader class does not do any processing of namespace prefixes; if an attribute has a prefix, that prefix is considered part of the attribute name.

When you are finished with the attributes for the current element, you can move to the next element in the document by invoking one of the navigation methods such as **Read()**. Alternatively, you can invoke the **MoveToElement()** method to return to the element that contains the current attribute.

For example, the following code lists all the attributes for the current node by index number:

```
If (textreader.NodeType = "element") {
    // list attributes for this node
    For a = 1:1:textreader.AttributeCount {
        Do textreader.MoveToAttributeIndex(a)
        Write textreader.LocalName," = ",textreader.Value,!
    }
}
```

The following code finds the value of the color attribute for the current node:

```
If (textreader.NodeType = "element") {
    // find color attribute for this node
    If (textreader.MoveToAttributeName("color")) {
        Write "color = ",textreader.Value,!
    }
}
```

### 7.5.4 Navigating to the Next Node with Content

The **MoveToContent()** method helps you find content. Specifically:

- If the node is of any type other than `"chars"`, this method advances to the next node of type `"chars"`.
- If the node is of type `"chars"`, this method does not advance in the file.

### 7.5.5 Rewinding

All the methods described here go forward in a document, except for the **Rewind()** method, which navigates to the start of the document and resets all properties.

## 7.6 Performing Validation

By default, the source document is validated against any DTD or schema document provided. If the document includes a DTD section, the document is validating against that DTD. To validate against a schema document instead, specify the schema within the argument list for **ParseFile()**, **ParseStream()**, **ParseString()**, or **ParseURL()**, as described in "Argument Lists for the Parse Methods."

Most types of validation issues are nonfatal and cause either an error or a warning. Specifically, nodes of type `"error"` or `"warning"` are automatically added to the document tree, at the location where the error occurred. You can navigate to and inspect these nodes in the same way as any other type of node.

For example, consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Root [
  <!ELEMENT Root (Person)>
  <!ELEMENT Person (#PCDATA)>
]>
<Root>
   <Person>Smith,Joe C.</Person>
</Root>
```

In this case, we do not expect any validation errors. Recall the example method `WriteNodes()` shown earlier in this chapter. If we used that method to read this document, the output would be as follows:

```
Node 1 is a(n) element named: Root
    and has no value
Node 2 is a(n) ignorablewhitespace and has no name
    with value:

Node 3 is a(n) element named: Person
    and has no value
Node 4 is a(n) chars and has no name
    with value: Smith,Joe C.
Node 5 is a(n) endelement named: Person
    and has no value
Node 6 is a(n) ignorablewhitespace and has no name
    with value:

Node 7 is a(n) endelement named: Root
    and has no value
```

In contrast, suppose that the file looked like this instead:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Root [
  <!ELEMENT Root (Person)>
  <!ELEMENT Person (#PCDATA)>
]>
<Root>
   <Employee>Smith,Joe C.</Employee>
</Root>
```

In this case, we expect errors because the `<Employee>` element is not declared in the DTD section. Here, if we use the example method `WriteNodes()` to read this document, the output would be as follows:

```
Node 1 is a(n) element named: Root
    and has no value
Node 2 is a(n) ignorablewhitespace and has no name
    with value:

Node 3 is a(n) error and has no name
    with value: Unknown element 'Employee'
while processing c:/TextReader/docwdtd2.txt at line 7 offset 14
Node 4 is a(n) element named: Employee
    and has no value
Node 5 is a(n) chars and has no name
    with value: Smith,Joe C.
Node 6 is a(n) endelement named: Employee
    and has no value
Node 7 is a(n) ignorablewhitespace and has no name
    with value:

Node 8 is a(n) error and has no name
    with value: Element 'Employee' is not valid for content model: '(Person)'
while processing c:/TextReader/docwdtd2.txt at line 8 offset 8
Node 9 is a(n) endelement named: Root
    and has no value
```

Also see "Setting the Parser Flags" in the chapter "Customizing How the SAX Parser Is Used."

# 7.7 Examples: Namespace Reporting

The following example method reads an arbitrary XML file and indicates the namespaces to which each element and attribute belongs:

```
ClassMethod ShowNamespacesInFile(filename As %String)
{
  Set status = ##class(%XML.TextReader).ParseFile(filename,.textreader)

  //check status
  If $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

  //iterate through document, node by node
  While textreader.Read()
  {
    If (textreader.NodeType = "element")
    {
       Write !,"The element ",textreader.LocalName
       Write " is in the namespace ",textreader.NamespaceUri
       }
    If (textreader.NodeType = "attribute")
    {
       Write !,"The attribute ",textreader.LocalName
       Write " is in the namespace ",textreader.NamespaceUri
       }
     }
}
```

When used in the Terminal, this method produces output like the following:

```
The element Person is in the namespace www://www.person.com
The element Name is in the namespace www://www.person.com
```

The following variation accepts an XML-enabled object, writes it to a stream, and then uses that stream to generate the same type of report:

```
ClassMethod ShowNamespacesInObject(obj)
{
  set writer=##class(%XML.Writer).%New()

  set str=##class(%GlobalCharacterStream).%New()
  set status=writer.OutputToStream(str)
  if $$$ISERR(status) {do $System.Status.DisplayError(status) quit ""}

  //write to the stream
  set status=writer.RootObject(obj)
  if $$$ISERR(status) {do $System.Status.DisplayError(status) quit }

  Set status = ##class(%XML.TextReader).ParseStream(str,.textreader)

  //check status
  If $$$ISERR(status) {do $System.Status.DisplayError(status) quit}

  //iterate through document, node by node
  While textreader.Read()
  {
    If (textreader.NodeType = "element")
    {
       Write !,"The element ",textreader.LocalName
       Write " is in the namespace ",textreader.NamespaceUri
       }
    If (textreader.NodeType = "attribute")
    {
       Write !,"The attribute ",textreader.LocalName
       Write " is in the namespace ",textreader.NamespaceUri
       }
     }
  }
```

# 8

# Evaluating XPath Expressions

XPath (XML Path Language) is an XML-based expression language for obtaining data from an XML document. With the %XML.XPATH.Document class, you can easily evaluate XPath expressions, given an arbitrary XML document that you provide. This chapter discusses the following topics:

- An overview of how to use %XML.XPATH.Document to evaluate XPath expressions
- How to use the **CreateFromFile()**, **CreateFromStream()**, and **CreateFromString()** methods
- How to use the **EvaluateExpression()** method
- How to use the results returned by the **EvaluateExpression()** method, including a suggested generic approach to examining the results
- Examples

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

## 8.1 Overview of Evaluating XPath Expressions in InterSystems IRIS

To use InterSystems IRIS XML support to evaluate XPath expressions using an arbitrary XML document, you do the following:

1. Create an instance of %XML.XPATH.Document. To do so, use one of the following class methods: **CreateFromFile()**, **CreateFromStream()**, or **CreateFromString()**. With any of these methods, you specify the input XML document as the first argument and receive an instance of %XML.XPATH.Document as an output parameter.

   This step parses the XML document, using a built-in XSLT processor.

2. Use the **EvaluateExpression()** method of your instance of %XML.XPATH.Document. For this method, you specify the node context and an expression to evaluate.

   - The node context specifies the context in which to evaluate the expression. This uses XPath syntax to express the path to that desired node. For example:

     ```
     "/staff/doc"
     ```

- The expression to evaluate also uses XPath syntax. For example:

  ```
  "name[@last='Marston']"
  ```

You receive the results as an output parameter (as the third argument).

The following sections provide details of all these methods, as well as examples.

**Note:** If you are iterating through a large set of documents and evaluating XPath expressions for each of them, InterSystems recommends that you set the OREF for a document equal to null when you are done processing it, before you open the next document. This works around a limitation in third-party software. This limitation causes slightly increased CPU usage as you process large numbers of documents in a loop.

# 8.2 Argument Lists When Creating an XPATH Document

To create an instance of %XML.XPATH.Document, use the **CreateFromFile()**, **CreateFromStream()**, or **CreateFromString()** class method of that class. For these class methods, the complete argument list is as follows, in order:

1. *pSource*, *pStream*, or *pString* — The source document.
   - For **CreateFromFile()**, this argument is the filename.
   - For **CreateFromStream()**, this argument is a binary stream.
   - For **CreateFromString()**, this argument is a string.
2. *pDocument* — The result, which is returned as an output parameter. This is an instance of %XML.XPATH.Document.
3. *pResolver* — An optional entity resolver to use when parsing the source. See "Performing Custom Entity Resolution," in the chapter "Customizing How the SAX Parser Is Used."
4. *pErrorHandler* — An optional custom error handler. See "Customizing the Error Handling," in the next chapter. If you do not specify a custom error handler, the method uses a new instance of %XML.XSLT.ErrorHandler.
5. *pFlags* — Optional flags to control the validation and processing performed by the SAX parser. See "Setting the Parser Flags" in the chapter "Customizing How the SAX Parser Is Used."
6. *pSchemaSpec* — An optional schema specification, against which to validate the document source. This argument is a string that contains a comma-separated list of namespace/URL pairs:

   ```
   "namespace URL,namespace URL"
   ```

   Here *namespace* is the XML namespace used for the schema and *URL* is a URL that gives the location of the schema document. There is a single space character between the namespace and URL values.
7. *pPrefixMappings* — An optional prefix mappings string. For details, see the subsection "Adding Prefix Mappings for Default Namespaces."

The **CreateFromFile()**, **CreateFromStream()**, and **CreateFromString()** methods return a status, which should be checked. For example:

```
Set tSC=##class(%XML.XPATH.Document).CreateFromFile("c:\sample.xml",.tDocument)
If $$$ISERR(tSC) Do $System.OBJ.DisplayError(tSC)
```

### 8.2.1 Adding Prefix Mappings for Default Namespaces

When an XML document uses default namespaces, that poses a problem for XPath. Consider the following example:

```
<?xml version="1.0"?>
<staff xmlns="http://www.staff.org">
  <doc type="consultant">
    <name first="David" last="Marston">Mr. Marston</name>
    <name first="David" last="Bertoni">Mr. Bertoni</name>
    <name first="Donald" last="Leslie">Mr. Leslie</name>
    <name first="Emily" last="Farmer">Ms. Farmer</name>
  </doc>
</staff>
```

In this case, the `<staff>` element belongs to a namespace but does not have a namespace prefix. XPath does not provide an easy way to access the `<doc>` element.

So that you can easily access nodes that have default namespaces, the %XML.XPATH.Document class provides a prefix mappings feature, which you can use in two ways:

- You can set the PrefixMappings property of your instance of %XML.XPATH.Document. This property is meant to provide a unique prefix for each default namespace in the source document, so that your XPath expressions can use those prefixes rather than the full namespace URIs.

  The PrefixMappings property is a string that consists of a comma-separated list; each list item is a prefix, followed by a space, followed by a namespace URI.

- When you call **CreateFromFile()**, **CreateFromStream()**, or **CreateFromString()**, you can specify the *pPrefixMappings* argument. This string must be of the same form as previously described.

  See "Argument Lists When Creating an XPATH Document," earlier in this chapter.

Then use these prefixes in the same way you use any namespace prefixes.

For example, suppose that when you read the preceding XML into an instance of %XML.XPATH.Document, you specified PrefixMappings as follows:

```
"s http://www.staff.org"
```

In this case you could use `"/s:staff/s:doc"` to access the `<doc>` element.

Note that you can use the instance method **GetPrefix()** to obtain the prefix that you previously specified for a given path in the document.

## 8.3 Evaluating XPath Expressions

To evaluate XPath expressions, you use the **EvaluateExpression()** method of your instance of %XML.XPATH.Document. For this method, you specify the following arguments, in order:

1. *pContext* — The node context, which specifies the context in which to evaluate the expression. Specify a string that contains the XPath syntax for the path to that desired node. For example:

   ```
   "/staff/doc"
   ```

2. *pExpression* — A predicate that selects particular results. Specify a string that contains the desired XPath syntax. For example:

   ```
   "name[@last='Marston']"
   ```

**Note:** With other technologies, it is common practice to concatenate the predicate to the end of the node path. The %XML.XPATH.Document class does not support this syntax, because the underlying XSLT processor requires the node context and the predicate as separate arguments.

3. *pResults* — The results, which are returned as an output parameter. For information on the results, see "Using the XPath Results," later in this chapter.

The **EvaluateExpression()** method returns a status, which should be checked. For example:

```
Set tSC=tDoc.EvaluateExpression("/staff/doc","name[@last='Smith']",.tRes)
If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC)}
```

# 8.4 Using the XPath Results

An XPath expression can return a subtree of the XML document, multiple subtrees, or a scalar result. The **EvaluateExpression()** method of %XML.XPATH.Document is designed to handle all these cases. Specifically, it returns a list of results. Each item in that list has a Type property that has one of the following values:

- $$$XPATHDOM — Indicates that this item contains a subtree of the XML document. This item is an instance of %XML.XPATH.DOMResult, which provides methods to navigate and examine the subtree. For information, see "Examining an XML Subtree."
- $$$XPATHVALUE — Indicates that this item is a single scalar result. This item is an instance of %XML.XPATH.ValueResult. For information, see "Examining a Scalar Result."

These macros are defined in the %occXSLT.inc include file.

The following subsections provide details on these classes, as well as a summary and example of the overall approach you might take.

## 8.4.1 Examining an XML Subtree

This section describes how to navigate the XML subtree represented by %XML.XPATH.DOMResult and how to get information about your current location in that subtree.

### 8.4.1.1 Navigating the Subtree

To navigate an instance of %XML.XPATH.DOMResult, you can use the following methods of the instance: **Read()**, **MoveToAttributeIndex()**, **MoveToAttributeName()**, **MoveToElement()**, and **Rewind()**.

To move to the next node in a document, use the **Read()** method. The **Read()** method returns a true value until there are no more nodes to read (that is, until the end of the document is reached).

When you navigate to an element, if that element has attributes, you can navigate to them, by using the following methods:

- Use the **MoveToAttributeIndex()** method to move to a specific attribute by index (ordinal position of the attribute within the element). This method takes one argument: the index number of the attribute. Note that you can use the AttributeCount property to learn how many attributes a given element has.
- Use the **MoveToAttributeName()** method to move to a specific attribute by name. This method takes two arguments: the name of the attribute and (optionally) the namespace URI.

When you are finished with the attributes for the current element, you can move to the next element in the document by invoking one of the navigation methods such as **Read()**. Alternatively, you can invoke the **MoveToElement()** method to return to the element that contains the current attribute.

All the methods described here go forward in a document, except for the **Rewind()** method, which navigates to the start of the document and resets all properties.

### 8.4.1.2 Properties of Nodes

In addition to the Type property, the following properties of %XML.XPATH.DOMResult provide information about your current location.

#### AttributeCount

If the current node is an element, this property indicates the number of attributes of the element.

#### EOF

True if the reader has reached the end of the source document; false otherwise.

#### HasAttributes

If the current node is an element, this property is true if that element has attributes (or false if it does not). If the current node is an attribute, this property is true.

For any other type of node, this property is false.

#### HasValue

True if the current node is a type of node that has a value (even if that value is null). Otherwise this property is false.

#### LocalName

For nodes of type attribute or element, this is the name of the current element or attribute, without the namespace prefix. For all other types of nodes, this property is null.

#### Name

Fully qualified name of the current node, as appropriate for the type of node.

#### NodeType

Type of the current node, one of the following: `attribute`, `chars`, `cdata`, `comment`, `document`, `documentfragment`, `documenttype`, `element`, `entity`, `entityreference`, `notation`, or `processinginstruction`.

#### Path

For nodes of type element, this is the path to the element. For all other types of nodes, this property is null.

#### ReadState

Indicates the overall read state, one of the following:

- `"initial"` means that the **Read()** method has not yet been called.
- `"cursoractive"` means that the **Read()** method has been called at least once.
- `"eof"` means that the end of the file has been reached.

#### Uri

The URI of the current node. The value returned depends on the type of node.

**Value**

Value, if any, of the current node, as appropriate for the type of node. If the value is less than 32 KB in size, this is a string. Otherwise, it is a character stream. For details, see "Examining a Scalar Result."

## 8.4.2 Examining a Scalar Result

This section describes use the XPath result that is represented by the %XML.XPATH.ValueResult class. In addition to the Type property, this class provides the Value property.

Note that if the value is larger than 32 KB in length, it is automatically placed in a stream object. Unless you are certain of the kinds of results that you will receive, you should check whether Value is a stream object. To do so, you can use the **$IsObject** function. (That is, if this value is an object, it is a stream object, because that is the only kind of object it can be.)

The following fragment shows a possible test:

```
// Value can be a stream if result is greater than 32 KB in length
Set tValue=tResult.Value

If $IsObject(tValue){
    Write ! Do tValue.OutputToDevice()
} else {
    Write tValue
}
```

## 8.4.3 General Approach

Unless you can be certain of the kinds of results that you will receive when you evaluate XPath expressions, you should write your code to handle the most general possible case. A possible organization of the code is as follows:

1. Find the number of elements in the list of returned results. Iterate through this list.
2. For each list item, check the Type property.
   - If Type is $$$XPATHDOM, use the methods of the %XML.XPATH.DOMResult class to navigate this XML subtree and examine it.
   - If Type is $$$XPATHVALUE, check whether the Value property is a stream object. If it is a stream object, use the usual stream interface to access the data. Otherwise, the Value property is a string.

For an example, see the ExampleDisplayResults() class method of the %XML.XPATH.Document. This method examines the results as described in the preceding list and then writes them to the Terminal. The following section shows what the output from this method looks like.

# 8.5 Examples

The examples in this section evaluate XPath expressions against the following XML document:

```
<?xml version="1.0"?>
<staff>
  <doc type="consultant">
    <name first="David" last="Marston">Mr. Marston</name>
    <name first="David" last="Bertoni">Mr. Bertoni</name>
    <name first="Donald" last="Leslie">Mr. Leslie</name>
    <name first="Emily" last="Farmer">Ms. Farmer</name>
  </doc>
  <doc type="GP">
    <name first="Myriam" last="Midy">Ms. Midy</name>
    <name first="Paul" last="Dick">Mr. Dick</name>
    <name first="Scott" last="Boag">Mr. Boag</name>
    <name first="Shane" last="Curcuru">Mr. Curcuru</name>
    <name first="Joseph" last="Kesselman">Mr. Kesselman</name>
    <name first="Stephen" last="Auriemma">Mr. Auriemma</name>
  </doc>
</staff>
```

These examples were adapted from more extensive examples contained in the %XML.XPATH.Document class; see the source code for a closer look at those.

## 8.5.1 Evaluating an XPath Expression That Has a Subtree Result

The following class method reads an XML file and evaluates an XPath expression that returns an XML subtree:

```
/// Evaluates an XPath expression that returns a DOM Result
ClassMethod Example1()
{
    Set tSC=$$$OK
    do {

    Set tSC=##class(%XML.XPATH.Document).CreateFromFile(filename,.tDoc)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC) Quit}

    Set context="/staff/doc"
    Set expr="name[@last='Marston']"
    Set tSC=tDoc.EvaluateExpression(context,expr,.tRes)
    If $$$ISERR(tSC) Quit

        Do ##class(%XML.XPATH.Document).ExampleDisplayResults(tRes)

    } while (0)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC)}
    Quit
}
```

This example selects any nodes whose <name> element has a `last` attribute equal to `Marston`. This expression is evaluated in the <doc> node of the <staff> element.

Notice that this example uses the ExampleDisplayResults() class method of the %XML.XPATH.Document.

When you execute the `Example1()` method, providing the previous XML file as input, you see the following output:

```
XPATH DOM
element: name
         attribute: first Value: David
         attribute: last  Value: Marston

chars : #text Value: Mr. Marston
```

## 8.5.2 Evaluating an XPath Expression That Has a Scalar Result

The following class method reads an XML file and evaluates an XPath expression that returns a scalar result:

```
/// Evaluates an XPath expression that returns a VALUE Result
ClassMethod Example2()
{
    Set tSC=$$$OK
    do {

    Set tSC=##class(%XML.XPATH.Document).CreateFromFile(filename,.tDoc)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC) Quit}

    Set tSC=tDoc.EvaluateExpression("/staff","count(doc)",.tRes)
        If $$$ISERR(tSC) Quit

        Do ##class(%XML.XPATH.Document).ExampleDisplayResults(tRes)

    } while (0)
    If $$$ISERR(tSC) {Do $System.OBJ.DisplayError(tSC)}
    Quit
}
```

This example counts <doc> subnodes. This expression is evaluated in the <staff> element.

When you execute the Example2() method, providing the previous XML file as input, you see the following output:

```
XPATH VALUE
2
```

# 9

# Performing XSLT Transformations

XSLT (Extensible Stylesheet Language Transformations) is an XML-based language that you use to describe how to transform a given XML document into another XML or other "human-readable" document. You can use classes in the %XML.XSLT and %XML.XSLT2 packages to perform XSLT 1.0 and 2.0 transforms. This chapter discusses the following topics:

- An overview
- How to configure, start, and stop the XSLT 2.0 Gateway
- How to create a compiled style sheet
- How to perform an XSLT transform
- Examples
- How to reuse an XSLT Gateway Server connection (XSLT 2.0)
- How to create and use a custom error handler
- How to specify a parameter list for use by the style sheet
- How to add and use XSLT extension functions
- How to use the XSL Transform Wizard in Studio

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

## 9.1 Overview of Performing XSLT Transformations in InterSystems IRIS

InterSystems IRIS provides two XSLT processors, each with its own API:

- The Xalan processor supports XSLT 1.0. The %XML.XSLT package provides the API for this processor.
- The Saxon processor supports XSLT 2.0. The %XML.XSLT2 package provides the API for this processor.

The %XML.XSLT2 API sends requests to Saxon over a connection to the XSLT 2.0 Gateway. The gateway allows multiple connections. This means that, for example, you could have two separate InterSystems IRIS processes connected to the gateway, each with its own set of compiled stylesheets, sending transform requests at the same time.

With the Saxon processor, compiled stylesheets and the isc:evaluate cache are connection-specific; you must manage your own connection to take advantage of either feature. If you open a connection and create a compiled stylesheet or evaluate a transform that populates the isc:evaluate cache, all other transforms evaluated on that connection will have access to the compiled stylesheet and isc:evaluate cache entries. If you open a new connection, other connections (and their compiled stylesheets and caches) are ignored.

The APIs are similar for the two processors, except that methods in %XML.XSLT2 use an additional argument, to specify the gateway connection to use.

To perform XSLT transformations, do the following:

1. If you are using the Saxon processor, configure the XSLT Gateway Server, as described in the next section. Or use the default configuration.

   The gateway is not needed if you are using the Xalan processor.

   The system automatically starts the gateway when needed. Or you can manually start it.

2. If you are using the Saxon processor, optionally create an instance of %Net.Remote.Gateway, which represents a single connection to the XSLT Gateway.

   Note that this step is required to take advantage of compiled stylesheets and the isc:evaluate cache, when you are using the Saxon processor.

3. Optionally create a compiled stylesheet and load it into memory. See "Creating a Compiled Stylesheet," later in this chapter. If you are using the Saxon processor, be sure to specify the *gateway* argument when you create the compiled stylesheet.

   This step is useful if you intend to use the same stylesheet repeatedly. This step, however, also consumes memory. Be sure to remove the compiled stylesheet when you no longer need it.

4. Call one of the transform methods of the applicable API. If you are using the Saxon processor, optionally specify the *gateway* argument when you call the transform method.

   See "Performing an XSLT Transform," later in this chapter.

5. Optionally call additional transform methods. If you are using the Saxon processor, optionally specify the *gateway* argument when you call the transform method; this enables you to evaluate another transform using the same connection. This transformation will have access to all compiled stylesheets and isc:evaluate cache entries associated with this connection. If you open a new connection, other connections (and their compiled stylesheets and caches) are ignored.

Studio also provides a wizard that you can use to test XSLT transformations; this is described later in this chapter.

# 9.2 Configuring, Starting, and Stopping the XSLT 2.0 Gateway

When you use the Saxon processor (to perform XSLT 2.0 transformations), InterSystems IRIS uses the XSLT 2.0 Gateway (which in turn uses Java). To configure this gateway:

1. In the Management Portal, select **System Administration** > **Configuration** > **Connectivity** > **XSLT 2.0 Gateway Server**.
2. Select **Go**.

The system displays the XSLT Gateway Server page.

The left area displays configuration details, and the right area displays recent activity.

3. In the left area, optionally specify the following settings:

   - **Port Number** — TCP port number for *exclusive* use by the XSLT 2.0 Gateway. This port number must not conflict with any other local TCP port on the server.

     The default is the InterSystems IRIS superserver port number plus 3000. If this number is greater than 65535, then the system uses 54773.

   - **Java Version** — Java version to use.

   - **Log File** — File pathname for a log file. If you omit this setting, no logging is performed. If you specify a filename but omit the directory, the log file is written to the system manager's directory.

   - **Java Home Directory** — Path of the directory that *contains* the Java bin directory. Specify this if there is no default Java on the server, or if you want to use a different Java.

     To see the default Java, execute the following command in a shell on the server:

     ```
     java -version
     ```

   - **JVM Arguments** — Any additional arguments for the Java Virtual Machine to use.

   This area also displays the current value of the JAVA_HOME environment variable.

   Note that you cannot edit any of these values while the gateway is running.

4. If you have made changes, select **Save** to save them. Or select **Reset** to discard them.
5. Optionally select **Test** to test your changes.

On this page, you can also do the following:

- Start the gateway. To do so, select **Start** in the right area.

  Note that InterSystems IRIS automatically starts the gateway when needed. It is not necessary to start the gateway manually.

- Stop the gateway. To do so, select **Stop** in the right area.

# 9.3 Reusing an XSLT Gateway Server Connection (XSLT 2.0)

If you are using the Saxon processor, InterSystems IRIS uses the XSLT 2.0 Gateway, which you have previously configured. To communicate with this gateway, InterSystems IRIS internally creates an *XSLT gateway connection* (an instance of %Net.Remote.Gateway). By default, the system creates a connection, uses it for the transformation, and then discards the connection. There is overhead associated with opening a new connection, so maintaining a single connection for multiple transforms gives the best performance. Also, you must maintain your own connection in order to take advantage of compiled stylesheets and the isc:evaluate cache.

To reuse an XSLT gateway connection:

1. Call the **StartGateway()** method of %XML.XSLT2.Transformer:

   ```
   set status=##class(%XML.XSLT2.Transformer).StartGateway(.gateway)
   ```

This method starts the XSLT 2.0 Gateway (if it is not already running) and returns, as output, an instance of %Net.Remote.Gateway. Note that the method also returns a status.

The instance of %Net.Remote.Gateway represents the connection to the gateway.

**StartGateway()** has an optional second argument, *useSharedMemory*. If this argument is true (the default), connections to localhost or to 127.0.0.1 will use shared memory, if that is possible. To force the connections to only use TCP/IP, set this argument to false.

2. Check the status returned by the previous step:

```
if $$$ISERR(status) {
    quit
}
```

3. Create any compiled stylesheets. When you do so, specify the *gateway* argument as the instance of instance of %Net.Remote.Gateway that you created in step 1.
4. Call the transform methods of %XML.XSLT2.Transformer as needed (**TransformFile()**, **TransformFileWithCompiledXSL()**, **TransformStream()**, and **TransformStreamWithCompiledXSL()**). When you do so, specify the *gateway* argument as the instance of %Net.Remote.Gateway that you created in step 1.
5. When you no longer need a given compiled stylesheet, call the **ReleaseFromServer()** method of %XML.XSLT2.CompiledStyleSheet:

```
Set status=##class(%XML.XSLT2.CompiledStyleSheet).ReleaseFromServer(compiledStyleSheet,,gateway)
```

   **Important:** Be sure to use this method when you no longer need the compiled stylesheet.

6. When you no longer need the XSLT gateway connection, call the **StopGateway()** method of %XML.XSLT2.Transformer, passing the gateway connection as the argument:

```
set status=##class(%XML.XSLT2.Transformer).StopGateway(gateway)
```

   This method discards the connection and resets the current device. It does not stop the XSLT 2.0 Gateway.

   **Important:** Be sure to use this method when you no longer need the connection.

For an example, see the method **Example10()** in XSLT2.Examples in the SAMPLES namespace.

# 9.4 Creating a Compiled Stylesheet

If you intend to use the same style sheet repeatedly, you may want to compile it to improve speed. Note that this step consumes memory. Be sure to remove the compiled style sheet when you no longer need it.

To create a compiled style sheet:

- If you are using the Xalan processor (for XSLT 1.0), use one of the following class methods of %XML.XSLT.CompiledStyleSheet:
  - **CreateFromFile()**
  - **CreateFromStream()**
- If you are using the Saxon processor (for XSLT 2.0), use one of the following class methods of %XML.XSLT2.CompiledStyleSheet:
  - **CreateFromFile()**

  - **CreateFromStream()**

  Also note that you will need to create an XSLT Gateway connection; see "Reusing an XSLT Gateway Server Connection (XSLT 2.0)."

For all these methods, the complete argument list is as follows, in order:

1. *source* — The style sheet.

   For **CreateFromFile()**, this argument is the filename. For **CreateFromStream()**, this argument is a stream.

2. *compiledStyleSheet* — The compiled style sheet, returned as an output parameter.

   This is an instance of the style sheet class (%XML.XSLT.CompiledStyleSheet or %XML.XSLT2.CompiledStyleSheet, as appropriate).

3. *errorHandler* — An optional custom error handler to use when compiling the style sheet. See "Customizing the Error Handling," later in this chapter.

   For methods in both classes, this is an instance of %XML.XSLT.ErrorHandler.

4. (Only for %XML.XSLT2.CompiledStyleSheet) *gateway* — An instance of %Net.Remote.Gateway. See "Reusing an XSLT Gateway Server Connection (XSLT 2.0)."

The **CreateFromFile()** and **CreateFromStream()** methods return a status, which should be checked.

For example:

```
//set tXSL equal to the OREF of a suitable stream
Set tSC=##class(%XML.XSLT.CompiledStyleSheet).CreateFromStream(tXSL,.tCompiledStyleSheet)
If $$$ISERR(tSC) Quit
```

# 9.5 Performing an XSLT Transform

To perform an XSLT transform:

- If you are using the Xalan processor (for XSLT 1.0), use one of the following class methods of %XML.XSLT.Transformer:
  - **TransformFile()** — Transforms a file, given an XSLT style sheet.
  - **TransformFileWithCompiledXSL()** — Transforms a file, given a compiled XSLT style sheet.
  - **TransformStream()** — Transforms a stream, given an XSLT style sheet.
  - **TransformStreamWithCompiledXSL()** — Transforms a stream, given a compiled XSLT style sheet.
  - **TransformStringWithCompiledXSL()** — Transforms a string, given a compiled XSLT style sheet.

  For these methods, use an instance of %XML.XSLT.CompiledStyleSheet as the compiled stylesheet. See "Creating a Compiled Stylesheet."

- If you are using the Saxon processor (for XSLT 2.0), use one of the following class methods of %XML.XSLT2.Transformer:
  - **TransformFile()** — Transforms a file, given an XSLT style sheet.
  - **TransformFileWithCompiledXSL()** — Transforms a file, given a compiled XSLT style sheet.
  - **TransformStream()** — Transforms a stream, given an XSLT style sheet.
  - **TransformStreamWithCompiledXSL()** — Transforms a stream, given a compiled XSLT style sheet.

For these methods, use an instance of %XML.XSLT2.CompiledStyleSheet as the compiled stylesheet. See "Creating a Compiled Stylesheet."

These methods have similar signatures. The argument lists for these methods are as follows, in order:

1. *pSource* — The source XML, which is to be transformed. See the table after this list.
2. *pXSL* — The style sheet or compiled style sheet. See the table after this list.
3. *pOutput* — The resulting XML, returned as an output parameter. See the table after this list.
4. *pErrorHandler* — An optional custom error handler. See "Customizing the Error Handling," later in this chapter. If you do not specify a custom error handler, the method uses a new instance of %XML.XSLT.ErrorHandler (for both classes).
5. *pParms* — An optional InterSystems IRIS multidimensional array that contains parameters to be passed to the style sheet. See "Specifying Parameters for Use by the Stylesheet," later in this chapter.
6. *pCallbackHandler* — An optional callback handler that defines XSLT extension functions. See "Adding and Using XSLT Extension Functions," later in this chapter.
7. *pResolver* — An optional entity resolver. See "Performing Custom Entity Resolution," later in this book.
8. (Only for %XML.XSLT2.Transformer) *gateway* — An optional instance of %Net.Remote.Gateway. Specify this argument if you want to reuse an XSLT Gateway connection for better performance; see "Reusing an XSLT Gateway Server Connection (XSLT 2.0)."

For reference, the following table shows the first three arguments of these methods, compared side by side:

***Table 9–1: Comparison of XSLT Transform Methods***

| Method | *pSource* (Input XML) | *pXSL* (Stylesheet) | *pOutput*(Output XML) |
|---|---|---|---|
| **TransformFile()** | String that gives a file name | String that gives a file name | String that gives a file name |
| **TransformFileWithCompiledXSL()** | String that gives a file name | Compiled style sheet | String that gives a file name |
| **TransformStream()** | Stream | Stream | Stream, *returned by reference* |
| **TransformStreamWithCompiledXSL()** | Stream | Compiled style sheet | Stream, *returned by reference* |
| **TransformStringWithCompiledXSL()** | String | Compiled style sheet | String that gives a file name |

# 9.6 Examples

This section shows a couple of transformations, using the following code (but different input files):

```
Set in="c:\0test\xslt-example-input.xml"
Set xsl="c:\0test\xslt-example-stylesheet.xsl"
Set out="c:\0test\xslt-example-output.xml"
Set tSC=##class(%XML.XSLT.Transformer).TransformFile(in,xsl,.out)
Write tSC
```

## 9.6.1 Example 1: Simple Substitution

In this example, we start with the following input XML:

```
<?xml version="1.0" ?>
<s1 title="s1 title attr">
  <s2 title="s2 title attr">
    <s3 title="s3 title attr">Content</s3>
  </s2>
</s1>
```

And we use the following style sheet:

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml" indent="yes"/>

<xsl:template match="//@* | //node()">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates select="node()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="/s1/s2/s3">
<xsl:apply-templates select="@*"/>
<xsl:copy>
Content Replaced
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

In this case, the output file would be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<s1 title="s1 title attr">
  <s2 title="s2 title attr">
    <s3>
Content Replaced
</s3>
  </s2>
</s1>
```

## 9.6.2 Example 2: Extraction of Contents

In this example, we start with the following input XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<MyRoot>
   <MyElement No="13">Some text</MyElement>
   <MyElement No="14">Some more text</MyElement>
</MyRoot>
```

And we use the following style sheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsl:output method="text"
            media-type="text/plain"/>
<xsl:strip-space elements="*"/>

<!-- utilities not associated with specific tags -->
<!-- emit a newline -->
<xsl:template name="NL">
    <xsl:text>&#xa;</xsl:text>
</xsl:template>

<!-- beginning of processing -->
```

```
<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="MyElement">
    <xsl:value-of select="@No"/>
    <xsl:text>: </xsl:text>
    <xsl:value-of select="."/>
    <xsl:call-template name="NL"/>
</xsl:template>

</xsl:stylesheet>
```

In this case, the output file would be as follows:

```
13: Some text
14: Some more text
```

### 9.6.3 Additional Examples

InterSystems IRIS provides the following additional examples:

- For XSLT 1.0, see the **Example()**, **Example2()**, and other methods in %XML.XSLT.Transformer.
- For XSLT 2.0, see the class XSLT2.Examples in the SAMPLES namespace.

## 9.7 Customizing the Error Handling

When an error occurs, either XSLT processor (Xalan or Saxon) executes the **error()** method of the current error handler, sending a message as an argument to that method. Similarly, when a fatal error or warning occurs, the XSLT processor executes the **fatalError()** or **warning()** method, as appropriate.

For all three of these methods, the default behavior is to write the message to the current device.

To customize the error handling, you do the following:

1. For either the Xalan or the Saxon processor, create a subclass of %XML.XSLT.ErrorHandler. In this subclass, implement the **error()**, **fatalError()**, and **warning()** methods as needed.

   Each of these methods accepts a single argument, a string that contains the message sent by the XSLT processor.

   These methods do not return values.

2. Then:
   - To use this error handler when compiling a stylesheet, create an instance of your subclass and use it in the argument list when you compile the stylesheet. See "Creating a Compiled Stylesheet."
   - To use this error handler when performing an XSLT transform, create an instance of your subclass and use it in the argument list of the transform method you use. See "Performing an XSLT Transform."

## 9.8 Specifying Parameters for Use by the Stylesheet

To specify parameters for use by the stylesheet:

1. Create an instance of %ArrayOfDataTypes.

2. Call the **SetAt()** method of this instance to add parameters and their values to this instance. For **SetAt()**, specify the first argument as the parameter value and the second argument as the parameter name.

   Add as many parameters as needed.

   For example:

   ```
   Set tParameters=##class(%ArrayOfDataTypes).%New()
   Set tSC=tParameters.SetAt(1,"myparameter")
   Set tSC=tParameters.SetAt(2,"anotherparameter")
   ```

3. Use this instance as the *pParms* argument of the transform method.

Instead of an %ArrayOfDataTypes, you can use an InterSystems IRIS multidimensional array, which can have any number of nodes with following structure and value:

| Node | Value |
|---|---|
| *arrayname*("*parameter_name*") | Value of the parameter named by *parameter_name* |

# 9.9 Adding and Using XSLT Extension Functions

You can create XSLT extension functions in InterSystems IRIS and then use them within your stylesheet, as follows:

- For XSLT 2.0 (the Saxon processor), you can use the **evaluate** function in the namespace `com.intersystems.xsltgateway.XSLTGateway` or the **evaluate** function in the namespace `http://extension-functions.intersystems.com`
- For XSLT 1.0 (the Xalan processor), you can only use the **evaluate** function in the namespace `http://extension-functions.intersystems.com`

By default (and as an example), the latter function reverses the characters that it receives. Typically, however, the default behavior is not used, because you implement some other behavior. To simulate multiple separate functions, you pass a selector as the first argument and implement a switch that uses that value to choose the processing to perform.

Internally, the **evaluate** function is implemented as a method (**evaluate()**) in the XSLT callback handler.

To add and use XSLT extension functions, do the following:

1. For either the Xalan or the Saxon processor, create a subclass of %XML.XSLT.CallbackHandler. In this subclass, implement the **evaluate()** method as needed. See the following subsection.
2. In the style sheet, declare the namespace to which the **evaluate** function belongs and use the **evaluate** function as needed. See the following subsection.
3. When performing an XSLT transform, create an instance of your subclass and use it in the argument list of the transform method you use. See "Performing an XSLT Transform."

## 9.9.1 Implementing the evaluate() Method

Internally, the code that calls the XSLT processor can pass any number of positional arguments to the **evaluate()** method of the current callback handler, which receives them as an array that has the following structure:

| Node | Value |
|---|---|
| *Args* | Number of arguments |
| *Args*(*index*) | Value of the argument in the position *index* |

The method has a single return value. The return value can be either:

- A scalar variable (such as a string or number).
- A stream object. This allows you to return an extremely long string, one that exceeds the string length limit. The stream has to be wrapped in an instance of %XML.XSLT.StreamAdapter which enables the XSLT processor to read the stream. The following shows a partial example:

```
Method evaluate(Args...) As %String
{
 //create stream
 ///...

 // create instance of %XML.XSLT.StreamAdapter to
 // contain the stream
 Set return=##class(%XML.XSLT.StreamAdapter).%New(tStream)

 Quit return

}
```

## 9.9.2 Using evaluate in a Stylesheet

To use XSLT extension functions in an XSLT, you must declare the namespace of the extension functions in the XSLT stylesheet. For the InterSystems **evaluate** function, this namespace is `http://extension-functions.intersystems.com` or `com.intersystems.xsltgateway.XSLTGateway`, as discussed previously.

The following example shows a style sheet that uses **evaluate**:

```
<?xml version="1.0"?>

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
xmlns:isc="http://extension-functions.intersystems.com">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="//@* | //node()">
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
      <xsl:apply-templates select="node()"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="/s1/s2/s3">
  <xsl:apply-templates select="@*"/>
  <xsl:choose>
  <xsl:when test="function-available('isc:evaluate')">
  <xsl:copy>
  <xsl:value-of select="isc:evaluate(.)" disable-output-escaping="yes"/>
  </xsl:copy>
  </xsl:when>
  <xsl:otherwise>
  <xsl:value-of select="."/>
  </xsl:otherwise>
  </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

For a closer look at this example, see the source code for the **Example3()** method of %XML.XSLT.Transformer.

### 9.9.3 Working with the isc:evaluate Cache

The XSLT 2.0 gateway caches evaluate function calls in the *isc:evaluate cache*. The default maximum size of the cache is 1000 items, but you can set the size to a different value. Also, you can clear the cache, you can dump the cache, and you can pre-populate the cache from a %List with the following format:

- Total number of cache entries
- For each entry:
    1. Total number of evaluate arguments
    2. All evaluate arguments
    3. Evaluate value

The cache also includes a filter list of function names that can be cached. Note the following:

- Function names can be added to or removed from the filter list.
- The filter list can be cleared.
- The filter list can be overridden by setting a boolean that will cache every evaluate call.

Adding a function name to the filter list does not limit the size of the evaluate cache. There may be any number of calls to the same function, but with different arguments and return values. Each combination of function name and arguments is a separate entry in the evaluate cache.

You can use methods from the %XML.XSLT2.Transformer class to manipulate the evaluate cache. For details, see the class reference.

## 9.10 Using the XSL Transform Wizard

Studio provides a wizard that performs an XSLT transformation, which is useful when you want to quickly test a style sheet or your custom XSLT extension functions. To use this schema wizard:

1. Select **Tools** > **Add-Ins** > **XSLT Schema Wizard**.
2. Specify the following required details:
    - For **XML File**, select **Browse** to select the XML file to transform.
    - For **XSL File**, select **Browse** to select the XSL style sheet to use.
    - For **Render As**, select either **Text** or **XML** to control how the transformation is displayed.
3. If you have created a subclass of %XML.XSLT.CallbackHandler that you want to use in this transformation, specify the following details:
    - For the first drop-down list in **XSLT Helper Class**, select a namespace.
    - For the second drop-down list in **XSLT Helper Class**, select that class.

    See "Adding and Using XSLT Extension Functions," earlier in this chapter.
4. Select **Finish**.

    The bottom of the dialog box displays the transformed file. You can copy and paste from this area.

5. To close this dialog box, select **Cancel**.

# 10

# Customizing How the InterSystems SAX Parser Is Used

Whenever InterSystems IRIS reads an XML document, it uses the InterSystems IRIS SAX (Simple API for XML) Parser. This chapter describes your options for controlling the InterSystems IRIS SAX Parser. It discusses the following topics:

- Overview of where the parser is used
- A summary of the available parser options
- How to specify parser options in general
- How to set parser flags for validation and so on
- How to specify the mask, which indicates the document events in which you are interested (such as start of element, start of DTD, and so on)
- How to specify a schema specification for use in validation
- How to disable entity resolution
- How to specify a custom entity resolver
- How to specify a custom content handler
- How to use HTTPS

## 10.1 About the InterSystems IRIS SAX Parser

The InterSystems IRIS SAX Parser is used whenever InterSystems IRIS reads an XML document.

It is an event-driven XML parser that reads an XML file and issues callbacks when it finds items of interest, such as the start of an XML element, start of a DTD, and so on.

(More accurately, the parser works in conjunction with a content handler, and the content handler issues the callbacks. This distinction is important only if you are customizing the SAX interface, as described in "Creating a Custom Content Handler," later in this chapter.)

The parser uses the standard Xerces-C++ library, which complies with the XML 1.0 recommendation and many associated standards. For a list of these standards, see http://xml.apache.org/xerces-c/.

# 10.2 Available Parser Options

You can control the behavior of the SAX parser in the following ways:

- You can set flags to specify the kinds of validation and processing to perform.

  Note that the parser always checks whether the document is a well-formed XML document.
- You can specify the events in which you are interested (that is, the items you want the parser to find). To do this, you specify a mask that indicates the events of interest.
- You can provide a schema specification against which to validate the document.
- You can disable entity resolution by using a special-purpose entity resolver.
- You can specify a timeout period for entity resolution.
- You can specify a more general custom entity resolver, if you need to control how the parser finds the definitions for any entities in the document.
- If the source document is accessed at a URL, you can specify the request sent to the web server, as an instance of %Net.HttpRequest.

  For details on %Net.HttpRequest, see the book *Using Internet Utilities*. Or see the class documentation for %Net.HttpRequest.
- You can specify a custom content handler.
- You can use HTTPS.

The available options depend on how you are using the InterSystems IRIS SAX Parser, as summarized in the following table:

***Table 10–1: SAX Parser Options in %XML Classes***

| Option | %XML.Reader | %XML.TextReader | %XML.XPATH.Document | %XML.SAX.Parser |
|---|---|---|---|---|
| Specifying parser flags | supported | supported | supported | supported |
| Specifying which parsing events are interesting (for example, start of element, end of element, comments) | *not supported* | supported | *not supported* | supported |
| Specifying a schema specification | supported | supported | supported | supported |
| Disabling entity resolution or otherwise customizing entity resolution | supported | supported | supported | supported |
| Specifying a custom HTTP request (if parsing a URL) | *not supported* | supported | *not supported* | supported |
| Specifying the content handler | *not supported* | *not supported* | *not supported* | supported |
| Parse documents at HTTPS locations | supported | *not supported* | *not supported* | supported |
| Resolve entities at HTTPS locations | *not supported* | *not supported* | *not supported* | supported |

# 10.3 Specifying the Parser Options

You specify the parser behavior differently depending on *how* you are using the InterSystems IRIS SAX Parser:

- If you are using %XML.Reader, you can set the Timeout, SAXFlags, SAXSchemaSpec, and EntityResolver properties of the reader instance. For example:

```
#include %occInclude
#include %occSAX
// set the parser options we want
Set flags = $$$SAXVALIDATION
            + $$$SAXNAMESPACES
            + $$$SAXNAMESPACEPREFIXES
            + $$$SAXVALIDATIONSCHEMA

Set reader=##class(%XML.Reader).%New()
Set reader.SAXFlags=flags
```

  These macros are defined in the %occSAX.inc include file.

- In other cases, you specify arguments of the method you are using. For example:

```
#include %occInclude
#include %occSAX

//set the parser options we want
Set flags = $$$SAXVALIDATION
            + $$$SAXNAMESPACES
            + $$$SAXNAMESPACEPREFIXES
            + $$$SAXVALIDATIONSCHEMA

Set status=##class(%XML.TextReader).ParseFile(myfile,.doc,,flags)
```

  For details on the argument lists for the relevant methods, see the following sections:

  – For %XML.Reader, see the chapter "Importing XML into InterSystems IRIS Objects."

  – For %XML.TextReader, see the chapter "Using %XML.TextReader."

  – For %XML.XPATH.Document, see the chapter "Evaluating XPath Expressions."

  – For %XML.SAX.Parser, see "Creating a Custom Content Handler," later in this chapter.

  Or see the class documentation.

# 10.4 Setting the Parser Flags

The %occSAX.inc include file lists the flags that you can use to control the validation performed by the Xerces parser. The basic flags are as follows:

- $$$SAXVALIDATION — Specifies whether to perform schema validation. If this flag is on (the default), all validation errors are reported.
- $$$SAXNAMESPACES — Specifies whether to recognize namespaces. If this flag is on (the default), the parser processes namespaces. If this flag is off, InterSystems IRIS causes the localname of the element to be an empty string on the **startElement()** callback of the %XML.SAX.ContentHandler.
- $$$SAXNAMESPACEPREFIXES — Specifies whether to process namespace prefixes. If this flag is on, the parser reports the original prefixed names and attributes used for namespace declarations. By default, this flag is off.

- $$$SAXVALIDATIONDYNAMIC — Specifies whether to perform validation dynamically. If this flag is on (the default), validation is performed only if a grammar is specified.
- $$$SAXVALIDATIONSCHEMA — Specifies whether to perform validation against a schema. If this flag is on (the default), validation is performed against the given schema, if any.
- $$$SAXVALIDATIONSCHEMAFULLCHECKING — Specifies whether to perform full schema constraint checking, including time-consuming or memory-intensive checking. If this flag is on, all constraint checking is performed. By default, this flag is off.
- $$$SAXVALIDATIONREUSEGRAMMAR — Specifies whether to cache the grammar for reuse in later parses within the same InterSystems IRIS process. By default, this flag is off.
- $$$SAXVALIDATIONPROHIBITDTDS — Special flag that causes the parser to throw an error if it encounters a DTD. Use this flag if you need to prevent processing of DTDs. To use this flag, you must explicitly add the value $$$SAXVALIDATIONPROHIBITDTDS to the parse flags passed to the various parsing methods of %XML.SAX.Parser.

The following additional flags provide useful combinations of the basic flags:

- $$$SAXDEFAULTS — Equivalent to the SAX defaults.
- $$$SAXFULLDEFAULT — Equivalent to the SAX defaults, plus the option to process namespace prefixes.
- $$$SAXNOVALIDATION — Do not perform schema validation but do recognize namespaces and namespace prefixes. Note that the SAX parser always checks whether the document is a well-formed XML document.

For details, see %occSAX.inc, which also provides links to further details on these kinds of validation.

The following fragment shows how you can combine parser options:

```
...
#include %occInclude
#include %occSAX
...
 ;; set the parser options we want
 set opt = $$$SAXVALIDATION
               + $$$SAXNAMESPACES
               + $$$SAXNAMESPACEPREFIXES
               + $$$SAXVALIDATIONSCHEMA
...
  set status=##class(%XML.TextReader).ParseFile(myfile,.doc,,opt)
  //check status
  if $$$ISERR(status) {do $System.Status.DisplayError(status) quit}
```

# 10.5 Specifying the Event Mask

The %occSAX.inc include file also lists the flags that you use to specify which event callbacks to process. For performance reasons, it is desirable to process only the callbacks that you need. You may or may not need to specify the mask, depending on which class you use to call the InterSystems IRIS SAX Parser.

- For %XML.TextReader, the default is $$$SAXCONTENTEVENTS. All event callbacks except for comments are processed.
- For %XML.SAX.Parser, the default is 0, which means that the parser calls the **Mask()** method of the content handler. In turn, this method computes the mask by detecting all the event callbacks that you have customized. Only those events are processed. You would use %XML.SAX.Parser if you had created a custom content handler; see "Creating a Custom Content Handler," later in this chapter.

## 10.5.1 Basic Flags

The basic flags are as follows:

- $$$SAXSTARTDOCUMENT — Instructs the parser to issue a callback when it starts the document.
- $$$SAXENDDOCUMENT — Instructs the parser to issue a callback when it ends the document.
- $$$SAXSTARTELEMENT — Instructs the parser to issue a callback when it finds the start of an element.
- $$$SAXENDELEMENT — Instructs the parser to issue a callback when it finds the end of an element.
- $$$SAXCHARACTERS — Instructs the parser to issue a callback when it finds characters.
- $$$SAXPROCESSINGINSTRUCTION — Instructs the parser to issue a callback when it finds a processing instruction.
- $$$SAXSTARTPREFIXMAPPING — Instructs the parser to issue a callback when it finds the start of a prefix mapping.
- $$$SAXENDPREFIXMAPPING — Instructs the parser to issue a callback when it finds the end of a prefix mapping.
- $$$SAXIGNORABLEWHITESPACE — Instructs the parser to issue a callback when it finds ignorable whitespace. This applies only if the document has a DTD and validation is enabled.
- $$$SAXSKIPPEDENTITY — Instructs the parser to issue a callback when it finds a skipped entity.
- $$$SAXCOMMENT — Instructs the parser to issue a callback when it finds a comment.
- $$$SAXSTARTCDATA — Instructs the parser to issue a callback when it finds the start of a CDATA section.
- $$$SAXENDCDATA — Instructs the parser to issue a callback when it finds the end of a CDATA section.
- $$$SAXSTARTDTD — Instructs the parser to issue a callback when it finds the start of a DTD.
- $$$SAXENDDTD — Instructs the parser to issue a callback when it finds the end of a DTD.
- $$$SAXSTARTENTITY — Instructs the parser to issue a callback when it finds the start of an entity.
- $$$SAXENDENTITY — Instructs the parser to issue a callback when it finds the end of an entity.

## 10.5.2 Convenient Combination Flags

The following additional flags provide useful combinations of the basic flags:

- $$$SAXCONTENTEVENTS — Instructs the parser to issue a callback for any event that contains "content."
- $$$SAXLEXICALEVENT — Instructs the parser to issue a callback for any lexical event.
- $$$SAXALLEVENTS — Instructs the parser to issue callbacks for all events.

## 10.5.3 Combining Flags into a Single Mask

The following fragment shows how you can combine multiple flags into a single mask:

```
...
#include %occInclude
#include %occSAX
...
 // set the mask options we want
 set mask = $$$SAXSTARTDOCUMENT
              + $$$SAXENDDOCUMENT
              + $$$SAXSTARTELEMENT
              + $$$SAXENDELEMENT
              + $$$SAXCHARACTERS
...
 // create a TextReader object (doc) by reference
 set status = ##class(%XML.TextReader).ParseFile(myfile,.doc,,,mask)
```

# 10.6 Specifying a Schema Document

You can specify a schema specification against which to validate the document source. Specify a string that contains a comma-separated list of namespace/URL pairs:

```
"namespace URL,namespace URL,namespace URL,..."
```

Here *namespace* is the XML namespace (not a namespace prefix) and *URL* is a URL that gives the location of the schema document for that namespace. There is a single space character between the namespace and URL values. For example, the following shows a schema specification with a single namespace:

```
"http://www.myapp.org http://localhost/myschemas/myapp.xsd"
```

The following shows a schema specification with two namespaces:

```
"http://www.myapp.org http://localhost/myschemas/myapp.xsd,http://www.other.org
http://localhost/myschemas/other.xsd"
```

# 10.7 Disabling Entity Resolution

Even when you set SAX flags to disable validation, the SAX parser still attempts to resolve external entities, which can be time-consuming, depending on their locations.

The class %XML.SAX.NullEntityResolver implements an entity resolver that always returns an empty stream. Use this class if you want to disable entity resolution. Specifically, when you read the XML document, use an instance of %XML.SAX.NullEntityResolver as the entity resolver. For example:

```
Set resolver=##class(%XML.SAX.NullEntityResolver).%New()
Set reader=##class(%XML.Reader).%New()
Set reader.EntityResolver=resolver

Set status=reader.OpenFile(myfile)
...
```

**Important:** Because this change disables all resolution of external entities, this technique also disables all external DTD and schema references in your XML document.

# 10.8 Performing Custom Entity Resolution

Your XML document may contain references to external DTDs or other entities. By default, InterSystems IRIS attempts to find the source documents for these entities and resolve them. To control how InterSystems IRIS resolves external entities, use the following procedure:

1. Define an entity resolver class.

   This class must extend the %XML.SAX.EntityResolver class and must implement the **resolveEntity()** method, which has the following signature:

   ```
   method resolveEntity(publicID As %Library.String, systemID As %Library.String) as %Library.Integer
   ```

   This method is invoked each time the XML processor finds a reference to an external entity (such as a DTD); here *publicID* and *systemID* are the Public and System identifier strings for that entity.

The method should fetch the entity or document, return it as a stream, and then wrap the stream in an instance of %XML.SAX.StreamAdapter. This class provides the necessary methods that are used to determine characteristics of the stream.

If the entity cannot be resolved, the method should return $$$NULLOREF to indicate to the SAX parser that the entity cannot be resolved).

**Important:** Despite the fact that the method signature indicates that the return value is %Library.Integer, the method should return an instance of %XML.SAX.StreamAdapter or a subclass of that class.

Also, identifiers that reference external entities are always passed to the **resolveEntity()** method *as specified in the document*. Particularly, if such an identifier uses a relative URL, the identifier is passed as a relative URL, which means that the actual location of the referencing document is not passed to the **resolveEntity()** method, and the entity cannot be resolved. In such scenarios, use the default entity resolver rather than a custom one.

For an example of an entity resolver class, see the source code for %XML.SAX.EntityResolver.

2. When you read an XML document, do the following:
   a. Create an instance of your entity resolver class.
   b. Use that instance when you read the XML document, as described in "Specifying the Parser Options," earlier in this chapter.

Also see the previous section, "Disabling Entity Resolution"; note that %XML.SAX.NullEntityResolver (discussed in that section) is a subclass of %XML.SAX.EntityResolver.

## 10.8.1 Example 1

For example, consider the following XML document:

```
<?xml version="1.0" ?>
<!DOCTYPE html SYSTEM  "c://temp/html.dtd">
<html>
<head><title></title></head>
<body>
<p>Some < xhtml-content > with custom entities &entity1; and &entity2;.</p>
<p>Here is another paragraph with &entity1; again.</p>
</body></html>
```

This document uses the following DTD:

```
<!ENTITY entity1
         PUBLIC "-//WRC//TEXT entity1//EN"
         "http://www.intersystems.com/xml/entities/entity1">
<!ENTITY entity2
         PUBLIC "-//WRC//TEXT entity2//EN"
         "http://www.intersystems.com/xml/entities/entity2">
<!ELEMENT html (head, body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT body (p)>
<!ELEMENT p (#PCDATA)>
```

To read this document, you would need a custom entity resolver like the following:

```
Class CustomResolver.Resolver Extends %XML.SAX.EntityResolver
{

Method resolveEntity(publicID As %Library.String, systemID As %Library.String) As %Library.Integer
{
    Try {
        Set res=##class(%Stream.TmpBinary).%New()
        //check if we are here to resolve a custom entity
        If systemID="http://www.intersystems.com/xml/entities/entity1"
        {
```

```
            Do res.Write("Value for entity1")
            Set return=##class(%XML.SAX.StreamAdapter).%New(res)
            }
            Elseif systemID="http://www.intersystems.com/xml/entities/entity2"
            {
                Do res.Write("Value for entity2")
                Set return=##class(%XML.SAX.StreamAdapter).%New(res)
            }
            Else //otherwise call the default resolver
            {
                Set res=##class(%XML.SAX.EntityResolver).%New()
                Set return=res.resolveEntity(publicID,systemID)
            }
    }
    Catch
    {
        Set return=$$$NULLOREF
    }
    Quit return
}

}
```

The following class contains a demo method that parses the file shown earlier and uses this custom resolver:

```
Include (%occInclude, %occSAX)

Class CustomResolver.ParseFileDemo
{

ClassMethod ParseFile()
{
    Set res= ##class(CustomResolver.Resolver).%New()
    Set file="c:/temp/html.xml"
    Set parsemask=$$$SAXALLEVENTS+$$$SAXERROR
    Set status=##class(%XML.TextReader).ParseFile(file,.textreader,res,,parsemask,,0)
    If $$$ISERR(status) {Do $system.OBJ.DisplayError(status) Quit }

    Write !,"Parsing the file ",file,!
    Write "Custom entities in this file:"
    While textreader.Read()
    {
        If textreader.NodeType="entity"{
            Write !, "Node:", textreader.seq
            Write !,"    name: ", textreader.Name
            Write !,"    value: ", textreader.Value
        }
    }

}

}
```

The following shows the output of this method, in a Terminal session:

```
GXML>d ##class(CustomResolver.ParseFileDemo).ParseFile()

Parsing the file c:/temp/html.xml
Custom entities in this file:
Node:13
    name: entity1
    value: Value for entity1
Node:15
    name: entity2
    value: Value for entity2
Node:21
    name: entity1
    value: Value for entity1
```

## 10.8.2 Example 2

For example, suppose that you need to read an XML document that contains the following:

```
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
 "c:\test\doctypes\docbook\docbookx.dtd">
```

In this case, the **resolveEntity** method would be invoked with *publicId* set to `-//OASIS//DTD DocBook XML V4.1.2//EN` and *systemId* set to `c:\test\doctypes\docbook\docbookx.dtd`.

The **resolveEntity** method determines the correct source for the external entity, returns it as a stream, and wraps it in an instance of %XML.StreamAdaptor. The XML parser reads the entity definition from this specialized stream.

For an example, refer to the %XML.Catalog and %XML.CatalogResolver classes included in the InterSystems IRIS library. The %XML.Catalog class defines a simple database that associates public and system identifiers with URLs. The %XML.CatalogResolver class is an entity resolver class that uses this database to find the URL for a given identifier. The %XML.Catalog class can load its database from an SGML-style catalog file; this file maps identifiers to URLs in a standard format.

# 10.9 Creating a Custom Content Handler

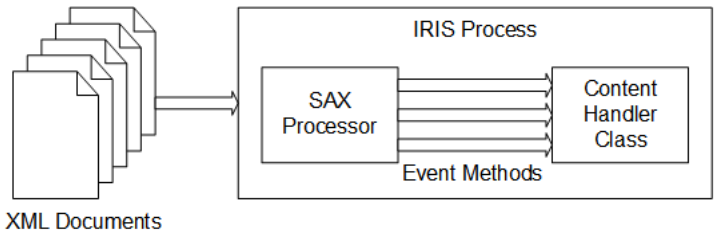
You can create a custom content handler for your own needs, if you call the InterSystems IRIS SAX Parser directly. This section discusses the following topics:

- Overview
- Description of the methods to customize in your content handler
- A summary of the argument lists of the parsing methods in the %XML.SAX.Parser class
- Example

## 10.9.1 Overview of Creating Custom Content Handlers

To customize how the InterSystems IRIS SAX Parser imports and handles XML, create and use a custom SAX content handler. Specifically, create a subclass of %XML.SAX.ContentHandler. Then, in the new class, override any of the default methods to perform the actions that are required. Use the new content handler as an argument when you parse an XML document; to do this, you use the parsing methods of the %XML.SAX.Parser class.

This operation is illustrated in the following diagram:




The process for creating and using a custom import mechanism is as follows:

1. Create a class that extends %XML.SAX.ContentHandler.
2. In that class, include the methods that you wish to override and provide new definitions as needed.
3. Write a class method that reads an XML document by using one of the parsing methods of the %XML.SAX.Parser class, namely **ParseFile()**, **ParseStream()**, **ParseString()**, or **ParseURL()**.

   When you call the parsing method, specify your custom content handler as an argument.

## 10.9.2 Customizable Methods of the SAX Content Handler

The %XML.SAX.ContentHandler class automatically executes certain methods at specific times. By overriding them, you can customize the behavior of your content handler.

### 10.9.2.1 Responding to Events

The %XML.SAX.ContentHandler class parses an XML file and generates events when it reaches particular points in the XML file. Depending on the event, a different method is executed. These methods are as follows:

- **OnPostParse()** — Triggered when XML parsing is complete.
- **characters()** — Triggered by character data.
- **comment()** — Triggered by comments.
- **endCData()** — Triggered by the end of a CDATA section.
- **endDocument()** — Triggered by the end of the document.
- **endDTD()** — Triggered by the end of a DTD.
- **endElement()** — Triggered by the end of an element.
- **endEntity()** — Triggered by the end of an entity.
- **endPrefixMapping()** — Triggered by the end of a namespace prefix mapping.
- **ignorableWhitespace()** — Triggered by ignorable whitespace in element content.
- **processingInstruction()** — Triggered by an XML processing instruction.
- **skippedEntity()** — Triggered by a skipped entity.
- **startCData()** — Triggered by the beginning of a CDATA section.
- **startDocument()** — Triggered by the beginning of the document.
- **startDTD()** — Triggered by the beginning of a DTD.
- **startElement()** — Triggered by the start of an element.
- **startEntity()** — Triggered by the start of an entity.
- **startPrefixMapping()** — Triggered by the start of an namespace prefix mapping.

These methods are empty by default, and you can override them in your custom content handler. For information on their expected argument lists and return values, see the class documentation for %XML.SAX.ContentHandler.

### 10.9.2.2 Handling Errors

The %XML.SAX.ContentHandler class also executes methods when it encounters certain errors:

- **error()** — Triggered by a recoverable parser error.
- **fatalError()** — Triggered by a fatal XML parsing error.
- **warning()** — Triggered by notification of a parser warning.

These methods are empty by default, and you can override them in your custom content handler. For information on their expected argument lists and return values, see the class documentation for %XML.SAX.ContentHandler.

### 10.9.2.3 Computing the Event Mask

When you call the InterSystems IRIS SAX Parser (via the %XML.SAX.Parser class), you can specify a mask argument that indicates which callbacks are interesting. If you do not specify a mask argument, the parser calls the **Mask()** method of the content handler. This method returns an integer that specifies the composite mask that corresponds to your overridden methods of the content handler.

For example, suppose that you create a custom content handler that contains new versions of the **startElement()** and **endElement()** methods. In this case, the **Mask()** method returns a numeric value that is equivalent to the sum of `$$$SAXSTARTELEMENT` and `$$$SAXENDELEMENT`, the flags that corresponding to these two events. If you do not specify a mask argument to the parsing method, the parser calls the **Mask()** method of your content handler and thus processes only those two events.

### 10.9.2.4 Other Useful Methods

The %XML.SAX.ContentHandler class provides other methods that are useful in special situations:

- **LocatePosition()** — Returns, by reference, two arguments that indicate the current position in the parsed document. The first indicates the line number, and the second indicates the line offset.
- **PushHandler()** — Pushes a new content handler on the stack. All subsequent callbacks from SAX go to this new content handler, until this handler is finished processing.

  You use this method if you are parsing a document of one type and you encounter a segment of XML that you want to parse in a different way. In this case, when you detect the segment that you want to handle differently, you call the **PushHandler()** method, which creates a new content handler instance. All callbacks go to this content handler until you call **PopHandler()** to return the previous content handler.
- **PopHandler()** — Returns to the previous content handler on the stack.

These methods are final and cannot be overridden.

## 10.9.3 Argument Lists for the SAX Parsing Methods

To specify a document source, you use the **ParseFile()**, **ParseStream()**, **ParseString()**, or **ParseURL()** method of the %XML.SAX.Parser class. In any case, the source document must be a well-formed XML document; that is, it must obey the basic rules of XML syntax. The complete argument list is as follows, in order:

1. *pFilename*, *pStream*, *pString*, or *pURL* — The document source.
2. *pHandler* — A content handler, which is an instance of the %XML.SAX.ContentHandler class.
3. *pResolver* — An entity resolver to use when parsing the source. See "Performing Custom Entity Resolution," earlier in this chapter.
4. *pFlags* — Flags to control the validation and processing performed by the SAX parser. See "Setting the Parser Flags," earlier in this chapter.
5. *pMask* — A mask to specify which items are of interest in the XML source. Usually you do not need to specify this argument, because for the parsing methods of %XML.SAX.Parser, the default mask is 0. This means that the parser calls the **Mask()** method of the content handler. That method computes the mask by detecting (during compilation) all the event callbacks that you customized in the event handler. Only those event callbacks are processed. However, if you want to specify the mask, see "Specifying the Event Mask," earlier in this chapter.
6. *pSchemaSpec* — A schema specification, against which to validate the document source. This argument is a string that contains a comma-separated list of namespace/URL pairs:

   ```
   "namespace URL,namespace URL"
   ```

Here *namespace* is the XML namespace used for the schema and *URL* is a URL that gives the location of the schema document. There is a single space character between the namespace and URL values.

7. *pHttpRequest* (For the **ParseURL()** method only) — The request to the web server, as an instance of %Net.HttpRequest.

   For details on %Net.HttpRequest, see the book *Using Internet Utilities*. Or see the class documentation for %Net.HttpRequest.

8. *pSSLConfiguration* — Configuration name of a client SSL/TLS configuration.

   See "Using HTTPS," later in this chapter.

**Note:** Notice that this argument list is slightly different from that of the parse methods of the %XML.TextReader class. For one difference, %XML.TextReader does not provide an option to specify a custom content handler.

## 10.9.4 A SAX Handler Example

Suppose you want a list of all the XML elements that appear in a file. To do this, you need simply to note every start element. Then the process is as follows:

1. Create a class, here called MyApp.Handler, which extends %XML.SAX.ContentHandler:

```
Class MyApp.Handler Extends %XML.SAX.ContentHandler
{
}
```

2. Override the **startElement()** method with the following content:

```
Class MyApp.MyHandler extends %XML.SAX.ContentHandler
{
// ...

Method startElement(uri as %String, localname as %String,
             qname as %String, attrs as %List)
{
    //we have found an element
    write !,"Element: ",localname
}

}
```

3. Add a class method to the Handler class that reads and parses an external file:

```
Class MyApp.MyHandler extends %XML.SAX.ContentHandler
{
// ...
ClassMethod ReadFile(file as %String) as %Status
{
    //create an instance of this class
    set handler=..%New()

    //parse the given file using this instance
    set status=##class(%XML.SAX.Parser).ParseFile(file,handler)

    //quit with status
    quit status
}
}
```

   Note that this is a class method because it is invoked in an application to perform its processing. This method does the following:

   a. It creates an instance of a content handler object:

```
set handler=..%New()
```

   b. It invokes the **ParseFile()** method of the %XML.SAX.Parser class. This validates and parses the document (specified by *filename*) and invokes the various event handling methods of the content handler object:

```
set status=##class(%XML.SAX.Parser).ParseFile(file,handler)
```

Each time an event occurs while the parser parses the document (such as a start or end element), the parser invokes the appropriate method in the content handler object. In this example, the only overridden method is **startElement()**, which then writes out element names. For other events, such as reaching end elements, nothing happens (the default behavior).

c. When the **ParseFile()** method reaches the end of the file, it returns. The handler object goes out of scope and is automatically removed from memory.

4. At the appropriate point in the application, invoke the **ReadFile()** method, passing it the file to parse:

```
Do ##class(Samples.MyHandler).ReadFile(filename)
```

Where *filename* is the path of the file being read.

For instance, if the content of the file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
  <Person>
    <Name>Edwards,Angela U.</Name>
    <DOB>1980-04-19</DOB>
    <GroupID>K8134</GroupID>
    <HomeAddress>
      <City>Vail</City>
      <Zip>94059</Zip>
    </HomeAddress>
    <Doctors>
      <Doctor>
        <Name>Uberoth,Wilma I.</Name>
      </Doctor>
      <Doctor>
        <Name>Wells,George H.</Name>
      </Doctor>
    </Doctors>
  </Person>
</Root>
```

Then the output of this example is as follows:

```
Element: Root
Element: Person
Element: Name
Element: DOB
Element: GroupID
Element: HomeAddress
Element: City
Element: Zip
Element: Doctors
Element: Doctor
Element: Name
Element: Doctor
Element: Name
```

# 10.10 Using HTTPS

%XML.SAX.Parser supports HTTPS. That is, you can use this class to do the following:

- (For **ParseURL()**) Parse XML documents served at HTTPS locations.
- (For all parsing methods) Resolve entities at HTTPS locations.

In all cases, if any of these items are served at an HTTPS location, do the following:

1. Use the Management Portal to create an SSL/TLS configuration that contains the details of the needed connection. For information, see the chapter "Using SSL/TLS with InterSystems IRIS" in the *Security Administration Guide*.

This is a one-time step.

2. When you invoke the applicable parsing method of %XML.SAX.Parser, specify the *pSSLConfiguration* argument.

By default, InterSystems IRIS uses the Xerces entity resolution. %XML.SAX.Parser uses its own entity resolution *only* in the following cases:

- The *pSSLConfiguration* argument is non-null.
- A proxy server is configured.

  See "Using a Proxy Server" in the chapter "Sending HTTP Requests" in *Using Internet Utilities*.

# 11

# Generating Classes from XML Schemas

Studio provides a wizard that reads an XML schema (from a file or URL) and generates a set of XML-enabled classes that correspond to the types defined in the schema. All the classes extend %XML.Adaptor. You specify a package to contain the classes, as well as various options that control the details of the class definitions.

The wizard is also available as a class method, which you can also use. Internally, the SOAP Wizard uses this method when it reads a WSDL document and generates web clients or web services; see *Creating Web Services and Web Clients*.

This chapter includes information on the following topics:

- How to use the XML Schema Wizard
- How to generate the classes programmatically
- InterSystems IRIS data types assigned for each XSD type
- Property keywords set by the wizard
- Property parameters set by the wizard
- When and how to adjust the generated classes to accommodate extremely long strings

This wizard does not load data into InterSystems IRIS. If you need to load data after generating classes, see the chapter "Importing XML into Objects."

**Note:** The XML declaration of any XML document that you use should indicate the character encoding of that document, and the document should be encoded as declared. If the character encoding is not declared, InterSystems IRIS uses the defaults described in "Character Encoding of Input and Output," earlier in this book. If these defaults are not correct, modify the XML declaration so that it specifies the character set actually used.

## 11.1 Using the Wizard

To use the XML Schema Wizard:

1. Select **Tools** > **Add-Ins** > **XML Schema Wizard**.
2. On the first screen, you specify the XML schema to use. Do one of the following:
   - For **Schema File**, select **Browse** to select an XML schema file.
   - For **URL**, specify the URL of the schema.
3. Select **Next**.

The next screen displays the schema so that you can verify that you have chosen the correct one.

4. Optionally select the following options:

   - **Keep Empty Classes**, which specifies whether to keep unused classes that have no properties. If you select this option, such classes are not removed at the end of the wizard; otherwise, they are removed.
   - **Create No Array Properties**, which controls whether the wizard generates array properties. If you select this option, the wizard does not generate array properties but instead generates another form. See "Creation of Array Properties" in the appendix "Details of the Generated Classes" in *Creating Web Services and Web Clients*.
   - **Generate XMLNIL property parameter for nillable elements**, which controls whether the wizard specifies the *XMLNIL* property parameter for applicable properties in the generated classes.

     This option applies to each property that corresponds to an XML element that is specified with `nillable="true"`. If you select this option, the wizard adds `XMLNIL=1` to the property definition. Otherwise, it does not add this parameter. For details on this parameter, see "Handling Empty Strings and Null Values" in *Projecting Objects to XML*.
   - **Generate XMLNILNOOBJECT property parameter for nillable elements**, which controls whether the wizard specifies the *XMLNILNOOBJECT* property parameter for applicable properties in the generated classes.

     This option applies to each property that corresponds to an XML element that is specified with `nillable="true"`. If you select this option, the wizard adds `XMLNILNOOBJECT=1` to the property definition. Otherwise, it does not add this parameter. For details on this parameter, see "Handling Empty Strings and Null Values" in *Projecting Objects to XML*.

5. Select **Next**.

   The next screen displays some basic information on options about the classes to generate.

6. On this screen, specify the following options:

   - Optionally select **Compile generated classes** if you want the wizard to compile the generated classes.
   - Optionally select **Add NAMESPACE Class Parameter** to specify the *NAMESPACE* parameter. In this case, *NAMESPACE* is set to the value of the targetNamespace in the schema.

     If you leave this option clear, *NAMESPACE* is not specified.

     It is recommended to select this option in all cases, because every XML-enabled class should be assigned to an XML namespace. (For backward compatibility, however, it is possible to leave this option clear.)
   - If you want the generated classes to be persistent classes, select **Create Persistent Classes**. Then the classes extend %Persistent.

     You can change this later in the wizard for individual classes.
   - If you generate persistent classes, you have options to determine how to handle a <complexType> A that consists of a <sequence> of another <complexType> B. When the wizard generates a persistent class containing property A, there are three possible forms for this property. It can be defined as a list of objects, as a one-to-many relationship (the default), or as a parent-child relationship. The options are summarized in the following table:

| Use Relationships for Collection Properties in Persistent Classes | Add index to many-one relationship | Use parent-child relationship | Form of the generated property A |
|---|---|---|---|
| selected (default) | not selected | not selected | One-to-many relationship with no index |
| selected (default) | selected | not selected | One-to-many relationship with index on the many side |
| selected (default) | This option is ignored if you select Use parent-child relationship | selected | Parent-child relationship |
| not selected | not selected | not selected | List of objects |

Also, if **Use parent-child relationship** is *not* selected, optionally select the option **Add %OnDelete method to classes in order to cascade deletes**. If you select this option, when the wizard generates class definitions, it includes an implementation of the **%OnDelete()** callback method in those classes. The generated **%OnDelete()** method deletes all persistent objects that are referenced by the class. Do not select this option if you do select **Use parent-child relationship**; a parent-child relationship already provides similar logic.

**Note:** If you modify the generated classes, be sure to modify the **%OnDelete()** callback method as needed.

- If you generate persistent classes, the wizard can add a transient property to each object-type class so that you can project an InterSystems IRIS internal identifier for the objects. The choices are as follows:
  - **None** — If you select this option, the wizard does not add any of the properties described here.
  - **Use Id** — If you select this option, the wizard adds the following property to each object-type class:

    ```
    Property %identity As %XML.Id (XMLNAME="_identity", XMLPROJECTION="ATTRIBUTE") [Transient];
    ```

  - **Use Oid** — If you select this option, the wizard adds the following property to each object-type class:

    ```
    Property %identity As %XML.Oid (XMLNAME="_identity", XMLPROJECTION="ATTRIBUTE") [Transient];
    ```

  - **Use GUID** — If you select this option, the wizard adds the following property to each object-type class:

    ```
    Property %identity As %XML.GUID (XMLNAME="_identity", XMLPROJECTION="ATTRIBUTE") [Transient];
    ```

  See also the chapter "Special Topics" in *Projecting Objects to XML*.
- The table at the bottom lists XML namespaces in the schema. Here, specify the package to contain the classes for the XML namespace shown in that row. To do so, specify a package name in the **Package Name** field for that row.

7. Select **Next**.
8. On the next screen, specify the following options:
   - **Java Enabled** — If you select this option, each class includes a Java projection.
   - **Data Population** — If you select this option, each class extends %Populate in addition to %XML.Adaptor.
   - **SQL Column Order** — If you select this option, each property specifies a value for the SqlColumnNumber keyword, so that the properties have the same order in SQL that they have in the schema.

- **No Sequence Check** — If you check this option, the wizard sets the *XMLSEQUENCE* parameter to 0 in the generated classes. This option is useful in some situations in which your XML files do not have elements in the same order as the XML schema.

  By default, the *XMLSEQUENCE* parameter is set to 1 in the generated classes. This ensures that the properties are included within the class definition in the same order as in the schema. Also see the chapter "Special Topics" in *Projecting Objects to XML*.

- **XMLIGNORENULL** — If you select this option, the wizard adds `XMLIGNORENULL=1` to the class definitions. Otherwise, it does not add this parameter.

  For details on this class parameter, see "Handling Empty Strings and Null Values" in *Projecting Objects to XML*.

- **Use Streams for Binary** — If you select this option, the wizard generates a property is of type %Stream.GlobalBinary for any element of type `xsd:base64Binary`. If this option is clear, the property is of type %xsd.base64Binary instead.

  Note that the wizard ignores any *attributes* of type `xsd:base64Binary`.

- Below the check boxes, the table lists the classes that the wizard will generate. For each class, make sure that **Extends/Type** is set appropriately. Here you choose one of the following:

  – **Persistent** — If you select this option, the class is a persistent class.

  – **Serial** — If you select this option, the class is a serial class.

  – **Registered Object** — If you select this option, the class is a registered object class.

  All generated classes also extend %XML.Adaptor.

- In the right column of the table, select **Index** for each property that should be indexed.

9. Select **Finish**.

The wizard then generates the classes and, if requested, compiles them.

For properties of these classes, if the corresponding element in the schema has a name that starts with an underscore (_), the name of the property starts with a percent sign (%).

# 11.2 Generating the Classes Programmatically

The XML Schema Wizard is also available as the **Process()** method of the %XML.Utils.SchemaReader class. To use this method:

1. Create an instance of %XML.Utils.SchemaReader.
2. Optionally set properties of the instance to control its behavior. For details, see the class documentation for %XML.Utils.SchemaReader.
3. Optionally create an InterSystems IRIS multidimensional array to contain information about additional settings. For details see the **Process()** method in the class documentation for %XML.Utils.SchemaReader.
4. Invoke the **Process()** method of your instance:

```
method Process(LocationURL As %String,
               Package As %String = "Test",
               ByRef Features As %String) as %Status
```

   - *LocationURL* must be the URL of the schema or the name of the schema file (including its complete path).

- *Package* is the name of the package in which to place the generated classes. If you do not specify a package, InterSystems IRIS uses the service name as the package name.
- *Features* is the multidimensional array that you optionally created in the previous step.

# 11.3 Default InterSystems IRIS Data Types for Each XSD Type

For each property that it generates, the XML Schema Wizard automatically uses an appropriate InterSystems IRIS data type class, depending on the XSD type specified in the schema. The following table lists the XSD types and the corresponding InterSystems IRIS data types:

***Table 11–1: InterSystems IRIS Data Types Used for XML Types***

| XSD Type in Source Document | Data Type in the Generated InterSystems IRIS Classes |
|---|---|
| anyURI | %xsd.anyURI |
| base64Binary | %xsd.base64Binary or %Stream.GlobalBinary, depending on options you chose. It is your responsibility to determine if each string might exceed the string length limit, and if so, to modify the generated property from %xsd.base64Binary to an appropriate stream class.) |
| boolean | %Boolean |
| byte | %xsd.byte |
| date | %Date |
| dateTime | %TimeStamp |
| decimal | %Numeric |
| double | %xsd.double |
| float | %xsd.float |
| hexBinary | %xsd.hexBinary |
| int | %xsd.int |
| integer | %Integer |
| long | %Integer |
| negativeInteger | %xsd.negativeInteger |
| nonNegativeInteger | %xsd.nonNegativeInteger |
| nonPositiveInteger | %xsd.nonPositiveInteger |
| positiveInteger | %xsd.positiveInteger |
| short | %xsd.short |

| XSD Type in Source Document | Data Type in the Generated InterSystems IRIS Classes |
|---|---|
| string | %String (Note: It is your responsibility to determine if each string might exceed the string length limit, and if so, to modify the generated type to an appropriate stream class.) |
| time | %Time |
| unsignedByte | %xsd.unsignedByte |
| unsignedInt | %xsd.unsignedInt |
| unsignedLong | %xsd.unsignedLong |
| unsignedShort | %xsd.unsignedShort |
| *no type given* | %String |

# 11.4 Property Keywords for the Generated Properties

For each property that it generates, the XML Schema Wizard also automatically sets the following keywords, using information in the schema:

- Description
- Required
- ReadOnly (if the corresponding element or attribute is defined with the `fixed` attribute)
- InitialExpression (the value is taken from the `fixed` attribute in the schema)
- Keywords related to relationships

# 11.5 Parameters for the Generated Properties

For each property that it generates, the XML Schema Wizard automatically sets *XMLNAME*, *XMLPROJECTION*, and all other XML-related parameters as needed. For information on these, see *Projecting Objects to XML*. It also sets other parameters such as *MAXVAL*, *MINVAL*, and *VALUELIST* as appropriate.

# 11.6 Adjusting the Generated Classes for the Extremely Long Strings

In rare cases, you might need to edit the generated classes to accommodate extremely long strings or binary values, beyond the string length limit.

For any string types, an XML schema does not contain any information to indicate how long the strings might be. The XML Schema Wizard maps any string values to the InterSystems IRIS %String class, and it maps any base64Binary values

to the %xsd.base64Binary class. These choices might not be appropriate, depending on the data that the class is intended to carry.

Before using the generated classes, you should do the following:

- Examine the generated classes and find the properties that are defined as %String or %xsd.base64Binary. Consider the contexts in which you will use these classes, and particularly these properties.
- If you think a %String property might need to contain string that exceeds the string length limit, redefine the property to an appropriate character stream. Similarly, if you think a %xsd.base64Binary property might need to contain a string that exceeds the same limit, redefine the property to an appropriate binary stream.
- Also note that for properties of type %String, %xsd.string, and %Binary, the MAXLEN property parameter is 50 characters, by default. You might need to specify a higher limit to have correct validation.

  (For properties of type %xsd.base64Binary, MAXLEN is "", which means the length is not checked by validation. The string length limit, however, does apply.)

# 12

# Generating XML Schemas from Classes

This chapter describes how to use %XML.Schema to generate XML schemas from your XML-enabled classes. It discusses the following:



## 12.1 Overview

To generate a complete schema that defines types for multiple classes in the same XML namespace, you build the schema by using %XML.Schema and then generate output for it by using %XML.Writer.

## 12.2 Building a Schema from Multiple Classes

To build an XML schema, do the following:

1. Create a new instance of %XML.Schema.
2. Optionally set the properties of your instance:
   - To specify the namespace for any otherwise unassigned types, specify the DefaultNamespace property. The default is null.
   - By default, the class documentation for the classes and their properties are included within <annotation> elements in the schema. To disable this, specify the IncludeDocumentation property as 0.

   **Note:** You must set these properties before calling the AddSchemaType() method.
3. Call the AddSchemaType() method of your instance.

```
method AddSchemaType(class As %String,
                     top As %String  = "",
                     format As %String,
                     summary As %Boolean  = 0,
                     input As %Boolean  = 0,
                     refOnly As %Boolean  = 0) as %Status
```

Here:

- *class* is the complete package and class name of an XML-enabled class.
- *top* is optional; if specified, it overrides the type name for this class.
- *format* specifies the format for this type. This must be `"literal"` (literal format, the default), `"encoded"` (for SOAP encoding), `"encoded12"` (for SOAP 1.2 encoding), or `"element"`. The value `"element"` is the same as literal format with an element at the top level.
- *summary*, if true, causes InterSystems IRIS to consider the *XMLSUMMARY* parameter of the XML-enabled class. If this parameter is specified, then the schema will contain only the properties listed by that parameter.
- *input*, if true, causes InterSystems IRIS to obtain the input schema, rather than the output schema. In most cases, the input schema and the output schema are the same; they are different if the *XMLIO* property parameter is specified for properties of the class.
- *refOnly*, if true, causes InterSystems IRIS to generate the schema only for the referenced types, rather than for the given class and all referenced types.

This method returns a status, which should be checked.

4. Repeat the previous step as necessary.
5. Optionally, to define the location of imported schemas, call the **DefineLocation()** method.

   ```
   method DefineLocation(namespace As %String, location As %String)
   ```

   Here *namespace* is a namespace used by one or more of the referenced classes, and location is a URL or path and filename of the corresponding schema (XSD file).

   You can call this method repeatedly to add locations for multiple imported schemas.

   If you do not use this method, the schema does include an <import> directive, but does not give the schema location.
6. Optionally, to define additional <import> directives, call the **DefineExtraImports()** method.

   ```
   method DefineExtraImports(namespace As %String, ByRef imports)
   ```

   Here *namespace* is the namespace into which the <import> directives should be added, and *imports* is a multidimensional array of the following form:

| Node | Value |
|---|---|
| *arrayname*(`"`*namespace URI*`"`) | String that gives the location of the schema (XSD file) for this namespace. |

# 12.3 Generating Output for the Schema

After you have created an instance of %XML.Schema as described in the previous section, do the following to generate output:

1. Call the **GetSchema()** method of your instance to return the schema as a node of a document object model (DOM).

   This method takes one argument: the URI of the target namespace for the schema. The method returns an instance of %XML.Node, which is described in the chapter "Representing an XML Document as a DOM."

   If the schema does not have a namespace, use `""` as the argument for **GetSchema()**.
2. Optionally modify this DOM.
3. To generate the schema, do the following:
   a. Create an instance of the %XML.Writer class and optionally set properties such as Indent.
   b. Optionally call the **AddNamespace()** method and other methods of the writer, to add namespace declarations to the `<schema>` element. See "Adding Namespace Declarations," earlier in this book.

      Because the schema probably refers to simple XSD types, it is useful to call **AddSchemaNamespace()** to add the XML schema namespace.
   c. Call the DocumentNode() or Tree() methods of the writer, using your schema as an argument.

      For information on using these methods, see "Writing XML Output from a DOM" in the chapter "Representing an XML Document as a DOM."

# 12.4 Examples

For additional examples, see the class reference for %XML.Schema.

## 12.4.1 Simple Example

The first example shows the basic steps:

```
Set schemawriter=##class(%XML.Schema).%New()

//add class and package (for example)
Set status=schemawriter.AddSchemaType("Facets.Test")

//retrieve schema by its URI
//which is null in this example
Set schema=schemawriter.GetSchema("")

//create writer
Set writer=##class(%XML.Writer).%New()
Set writer.Indent=1

//use writer
Do writer.DocumentNode(schema)
```

## 12.4.2 More Complex Schema Example

Consider the following scenario. The `Person` class is as follows:

```
Class SchemaWriter.Person Extends (%Persistent, %XML.Adaptor)
{

Parameter NAMESPACE = "http://www.myapp.com";

Property Name As %Name;

Property DOB As %Date(FORMAT = 5);

Property PatientID as %String;

Property HomeAddress as Address;

Property OtherAddress as AddressOtherNS ;
}
```

The `Address` class is defined to be in the same XML namespace (`"http://www.myapp.com"`), and the `OtherAddress` class is defined to be in a different XML namespace (`"http://www.other.com"`).

The `Company` class is also defined to be in the XML namespace `"http://www.myapp.com"`. It is defined as follows:

```
Class SchemaWriter.Company Extends (%Persistent, %XML.Adaptor)
{

Parameter NAMESPACE = "http://www.myapp.com";

Property Name As %String;

Property CompanyID As %String;

Property HomeOffice As Address;

}
```

Notice that there is no property relationship that connects the `Person` and `Company` classes.

To generate a schema for the namespace `"http://www.myapp.com"`, we could use the following:

```
ClassMethod Demo()
{
    Set schema=##class(%XML.Schema).%New()
    Set schema.DefaultNamespace="http://www.myapp.com"
    Set status=schema.AddSchemaType("SchemaWriter.Person")
    Set status=schema.AddSchemaType("SchemaWriter.Company")
    Do schema.DefineLocation("http://www.other.com","c:/other-schema.xsd")

    Set schema=schema.GetSchema("http://www.myapp.com")

    //create writer
    Set writer=##class(%XML.Writer).%New()
    Set writer.Indent=1

    Do writer.AddSchemaNamespace()
    Do writer.AddNamespace("http://www.myapp.com")
    Do writer.AddNamespace("http://www.other.com")

    Set status=writer.DocumentNode(schema)
    If $$$ISERR(status) {Do $system.OBJ.DisplayError() Quit }

}
```

The output is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:s01="http://www.myapp.com"
xmlns:s02="http://www.other.com"
elementFormDefault="qualified"
targetNamespace="http://www.myapp.com">
  <import namespace="http://www.other.com" schemaLocation="c:/other-schema.xsd"/>
  <complexType name="Person">
    <sequence>
      <element minOccurs="0" name="Name" type="s:string"/>
      <element minOccurs="0" name="DOB" type="s:date"/>
      <element minOccurs="0" name="PatientID" type="s:string"/>
      <element minOccurs="0" name="HomeAddress" type="s01:Address"/>
      <element minOccurs="0" name="OtherAddress" type="s02:AddressOtherNS"/>
    </sequence>
  </complexType>
```

```
  <complexType name="Address">
    <sequence>
      <element minOccurs="0" name="State">
        <simpleType>
          <restriction base="s:string">
            <maxLength value="2"/>
          </restriction>
        </simpleType>
      </element>
      <element minOccurs="0" name="Zip">
        <simpleType>
          <restriction base="s:string">
            <maxLength value="10"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
  <complexType name="Company">
    <sequence>
      <element minOccurs="0" name="Name" type="s:string"/>
      <element minOccurs="0" name="CompanyID" type="s:string"/>
      <element minOccurs="0" name="HomeOffice" type="s01:Address"/>
    </sequence>
  </complexType>
</schema>
```

Notice the following:

- The schema includes types for `Person` and all its referenced classes, as well as `Company` and all its referenced classes.
- The `<import>` directive imports the namespace used by the `OtherAddress` class; because we used **DefineLocation()**, this directive also indicates the location of the corresponding schema.
- Because we used **AddSchemaNamespace()** and **AddNamespace()** just before calling DocumentNode(), the `<schema>` element includes namespace declarations, which define prefixes for these namespaces.

  If we had not used **AddSchemaNamespace()** and **AddNamespace()**, `<schema>` would not include these namespace declarations, and the schema would instead have been as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="http://www.myapp.com">
  <import namespace="http://www.other.com" schemaLocation="c:/other-schema.xsd"/>
  <complexType name="Person">
    <sequence>
      <element minOccurs="0" name="Name" type="s01:string"
xmlns:s01="http://www.w3.org/2001/XMLSchema"/>
     <element minOccurs="0" name="DOB" type="s02:date" xmlns:s02="http://www.w3.org/2001/XMLSchema"/>

      <element minOccurs="0" name="PatientID" type="s03:string"
xmlns:s03="http://www.w3.org/2001/XMLSchema"/>
     <element minOccurs="0" name="HomeAddress" type="s04:Address" xmlns:s04="http://www.myapp.com"/>

      <element minOccurs="0" name="OtherAddress" type="s05:AddressOtherNS"
xmlns:s05="http://www.other.com"/>
    </sequence>
  </complexType>
  <complexType name="Address">
    <sequence>
      <element minOccurs="0" name="State">
        <simpleType>
          <restriction base="s06:string" xmlns:s06="http://www.w3.org/2001/XMLSchema">
...
```

# 13

# Examining Namespaces and Classes

The class %XML.Namespaces provides two class methods that you can use to examine XML namespaces and the classes contained in them:

### GetNextClass()

```
classmethod GetNextClass(namespace As %String,
           class As %String) as %String
```

Returns the next class (alphabetically) after the given class, in the given XML namespace. This method returns null when there are no more classes.

### GetNextNamespace()

```
classmethod GetNextNamespace(namespace As %String) as %String
```

Returns the next namespace (alphabetically) after the given namespace. This method returns null when there are no more namespaces.

In both cases, only the current InterSystems IRIS namespace is considered. Also, mapped classes are ignored.

For example, the following method lists the XML namespaces and their classes, for the current InterSystems IRIS namespace:

```
ClassMethod WriteNamespacesAndClasses()
{
  Set ns=""
  Set ns=##class(%XML.Namespaces).GetNextNamespace(ns)

  While ns '=""
  {
    Write !, "The namespace ",ns, " contains these classes:"
    Set cls=""
    Set cls=##class(%XML.Namespaces).GetNextClass(ns,cls)

    While cls '=""
    {
      Write !, "   ",cls
      Set cls=##class(%XML.Namespaces).GetNextClass(ns,cls)
      }

      Set ns=##class(%XML.Namespaces).GetNextNamespace(ns)
      }
}
```

When executed in the Terminal, this method generates output like the following:

```
The namespace http://www.address.org contains these classes:
   ElRef.NS.Address
   GXML.AddressNS
   MyApp4.Obj.Address
   MyAppNS.AddressNS
   Obj.Attr.Address
   Obj.Ns.Address
   Obj.Ns.AddressClass
The namespace http://www.doctor.com contains these classes:
   GXML.DoctorNS
The namespace http://www.one.org contains these classes:
   GXML.AddressNSOne
   GXML.DoctorNSOne
   GXML.PersonNSOne
...
```

# A

# XML Background

This appendix provides a quick summary of XML terms and concepts, as a refresher for users working with the InterSystems IRIS %XML classes. It is assumed that the reader is familiar with the basic syntax of XML, so this appendix is not intended as a primer on XML. It does not, for example, describe the syntax or list reserved characters. However, it does provide a list of the terms with short definitions, as a reference.

### attribute

A name-value pair of the following form:

```
ID="QD5690"
```

Attributes reside within elements, as shown below, and an element can have any number of attributes.

```
<Patient ID="QD5690">Cromley,Marcia N.</Patient>
```

### CDATA section

Denotes text that should not be validated, as follows:

```
<myelementname><![CDATA[
Non-validated data goes here.
You can even have stray "<" or ">" symbols in it.
]]></myelementname>
```

A CDATA (character data) section cannot contain the string `]]>` because this string marks the end of the section. This also means that CDATA sections cannot be nested.

Note that the contents of a CDATA section must conform to the encoding specified for the XML document, as does the rest of the XML document.

### comment

A parenthetical note that is not part of the main data of an XML document. A comment looks like this:

```
<!--Output for the class: GXML.PersonNS7-->
```

### content model

An abstract description of the possible contents of an XML element. The possible content models are as follows:

- Empty content model (no child elements or text nodes are permitted)
- Simple content model (only text nodes are permitted)
- Complex content model (only child elements)
- Mixed content model (both child elements and text nodes are permitted)

In all cases, the element may or may not have attributes; the phrase *content model* does not refer to the presence or absence of attributes in the element.

### default namespace

The namespace to which any unqualified elements belong, in a given context. A default namespace is added without a prefix. For example:

```
<Person xmlns="http://www.person.org">
  <Name>Isaacs,Rob G.</Name>
  <DOB>1981-01-29</DOB>
</Person>
```

Because this namespace declaration does not use a prefix, the `<Person>`, `<Name>`, and `<DOB>` elements all belong to this namespace.

Note that the following XML, which does not use a default namespace, is effectively equivalent to the preceding example:

```
<s01:Person s01:xmlns="http://www.person.org">
  <s01:Name>Isaacs,Rob G.</s01:Name>
  <s01:DOB>1981-01-29</s01:DOB>
</s01:Person>
```

### DOM

Document Object Model (DOM) is an object model for representing XML and related formats.

### DTD (document type definition)

A series of text directives contained within an XML document or in an external file. It defines all the valid elements and attributes that can be used within a document. DTDs do not themselves use XML syntax.

### element

An element typically consists of two tags (a start tag and an end tag), possibly surrounding text and other elements. The content of the element is everything between these two tags, including text and any child elements. The following is a complete XML element, with start tag, text content, and end tag:

```
<Patient>Cromley,Marcia N.</Patient>
```

An element can have any number of attributes and any number of child elements.

An empty element can either include a start tag and an end tag, or just a single tag. The following examples are equivalent:

```
<EndDate></EndDate>
```

```
<EndDate/>
```

In practice, elements are likely to refer to different parts of data records, such as:

```
<Student level="undergraduate">
   <Name>Barnes,Gerry</Name>
   <DOB>1981-04-23</DOB>
</Student>
```

### entity

A unit of text (within an XML file) that represents one or more characters. An entity has the following structure:

```
&characters;
```

### global element

The concepts of global and local elements apply to documents that use namespaces. The names of global elements are placed in a separate symbol space from those of local elements. A global element is an element whose type has global scope, that is, an element whose type is defined at the top level in the corresponding XML schema. Element declarations that appear as children of the `<xs:schema>` element are considered to be global declarations. Any other element declaration is a local element, unless it references a global declaration through the `ref` attribute, which effectively makes it a global element.

Attributes are global or local in the same way.

### local element

An XML element that is not global. Local elements do not belong explicitly to any namespace, unless elements are qualified. See *qualified* and *global element.*

### namespace

A namespace is a unique string that defines a domain for identifiers so that XML-based applications do not confuse one type of document with another. It is typically given as a URI (uniform resource indicator) in the form of a URL (uniform resource location), which may or may not correspond to an actual web address. For example, `"http://www.w3.org"` is a namespace.

You include a namespace declaration with one of the following syntaxes:

```
xmlns="your_namespace_here"
pre:xmlns="your_namespace_here"
```

In either case, the namespace is used only within the context where you inserted the namespace declaration. In the latter case, the namespace is associated with the given prefix (*pre*). Then an element or attribute belongs to this namespace if and only if the element or attribute also has this prefix. For example:

```
<s01:Person xmlns:s01="http://www.person.com">
   <Name>Ravazzolo,Roberta X.</Name>
   <DOB>1943-10-24</DOB>
</s01:Person>
```

The namespace declaration uses the `s01` prefix. The `<Person>` element uses this prefix as well, so this element belongs to this namespace. The `<Name>` and `<DOB>` elements, however, do not explicitly belong to any namespace.

### processing instructions (PI)

An instruction (within the prolog) intended to tell an application how to use an XML document or what to do with it. An example follows; this associates a stylesheet with the document.

```
<?xml-stylesheet type="text/css" href="mystyles.css"?>
```

### prolog

The part of the XML document that precedes the root element. The prolog starts with the XML declaration (which indicates the XML version used) and then may include a DTD declaration or schema declaration, as well as processing instructions. (Technically you do not need a DTD nor a schema. Also, technically, you can have both in the same file.)

### root, root element, document element

Each XML document is required to have exactly one element at the outermost level. This is known as the root, root element, or document element. The root element follows the prolog.

### qualified

An element or attribute is qualified if it is explicitly assigned to a namespace. Consider the following example, in which the elements and attribute of <Person> are not qualified:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <s01:Person xmlns:s01="http://www.person.com" GroupID="J1151">
      <Name>Frost,Sally O.</Name>
      <DOB>1957-03-11</DOB>
   </s01:Person>
</Root>
```

Here, the namespace declaration uses the s01 prefix. There is no default namespace. The <Person> element uses this prefix as well, so that element belongs to this namespace. There is no prefix for the <Name> and <DOB> elements or the <GroupID> attribute, so these do not explicitly belong to any namespace.

In contrast, consider the following case where the elements and attribute of <Person> are qualified:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <Person xmlns="http://www.person.com" GroupID="J1151">
      <Name>Frost,Sally O.</Name>
      <DOB>1957-03-11</DOB>
   </Person>
</Root>
```

In this case, the <Person> element defines a default namespace, which applies to the child elements and the attribute.

**Note:** The XML schema attributes elementFormDefault attribute and attributeFormDefault attribute control whether elements and attributes are qualified in a given schema. In InterSystems IRIS XML support, you use a class parameter to specify whether elements are qualified.

### schema

A document that specifies meta-information for a set of XML documents, as an alternative to a DTD. As with a DTD, you can use a schema to validate the contents of specific XML documents. XML schemas offer several advantages over DTDs for certain applications, including these:

- An XML schema is a valid XML document, making it easier to develop tools that operate on schemas.
- An XML schema can specify a richer set of features and includes type information for values.

Formally, a schema document is a XML document that complies with the W3 XML Schema specification (http://www.w3.org/XML/Schema). It obeys the rules of XML and uses some additional syntax. Typically the extension of the file is .xsd.

### style sheet

A document written in XSLT that describes how to transform a given XML document into another XML or other "human-readable" document.

### text node

One or more characters included between an opening element and the corresponding closing element. For example:

```
<SampleElement>
sample text node
</SampleElement>
```

### type

A constraint placed upon the interpretation of data. In an XML schema, the definition of each element and attribute corresponds to a type.

Types are either *simple* or *complex*.

Each attribute has a simple type. Simple types also represent elements that have no attributes and no child elements (only text nodes). Complex types represent other elements.

The following fragment of a schema shows some type definitions:

```
<s:complexType name="Person">
    <s:sequence>
        <s:element name="Name" type="s:string" minOccurs="0" />
        <s:element name="DOB" type="s:date" minOccurs="0" />
        <s:element name="Address" type="s_Address" minOccurs="0" />
    </s:sequence>
    <s:attribute name="GroupID" type="s:string" />
</s:complexType>
<s:complexType name="s_Address">
    <s:sequence>
        <s:element name="City" type="s:string" minOccurs="0" />
        <s:element name="Zip" type="s:string" minOccurs="0" />
    </s:sequence>
</s:complexType>
```

### unqualified

An element or attribute is unqualified if it is not explicitly assigned to a namespace. See *qualified*.

### well-formed XML

An XML document or fragment that obeys the rules of XML, such as having an end tag to match a start tag.

### XML declaration

A statement that indicates which XML version (and, optionally, which character set) is used in a given document. If included, it must be the first line in the document. An example follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

### XPath

XPath (XML Path Language) is an XML-based expression language for obtaining data from an XML document. The result can either be scalar or an XML subtree of the original document.

### XSLT

XSLT (Extensible Stylesheet Language Transformations) is an XML-based language that you use to describe how to transform a given XML document into another XML or other “human-readable” document.