Projecting Objects to XML

Version 2018.1
2018-12-07

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

Projecting Objects to XML
Caché Version 2018.1 2018-12-07

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 project Caché objects to XML and how to control that projection to XML elements, attributes, and schemas. It includes the following sections:

- Introduction to Object-XML Projections
- Projecting Caché Objects to XML
- Controlling Transformations of Values
- Handling Empty Strings and Null Values
- Controlling the XML Element and Attribute Names
- Controlling the Namespaces for Elements and Attributes
- Controlling the Projection to XML Schemas
- Advanced Options for XML Schemas
- Special Topics
- Summary of XML Projection Parameters

For a detailed outline, see the table of contents.

Also see the following books:

- *Using Caché XML Tools* describes how to use Caché tools to work with XML-enabled objects and with general XML documents and DOMs (Document Object Model).
- *Creating Web Services and Web Clients in Caché* describes how to create Caché web services and web clients.

For general information, see the *InterSystems Documentation Guide*.

1

Introduction to Object-XML Projections

This book describes how to project Caché objects to XML and how to control that projection to XML elements, attributes, and types.

This chapter introduces how to project objects to XML and explains why you might want to do this.

For information on XML standards supported in Caché, see "Standards Supported by Caché" in *Using Caché XML Tools*.

1.1 The Basics

The phrase *projecting an object to XML* means defining how that object can be used as an XML document. To project an object to XML, you add %XML.Adaptor to the superclass list of the class that defines the object, as well as any other object classes used by that class, with minor exceptions.

This activity is also called *defining the XML projection* of the class that defines the object or *XML-enabling* that class.

1.2 How It Works

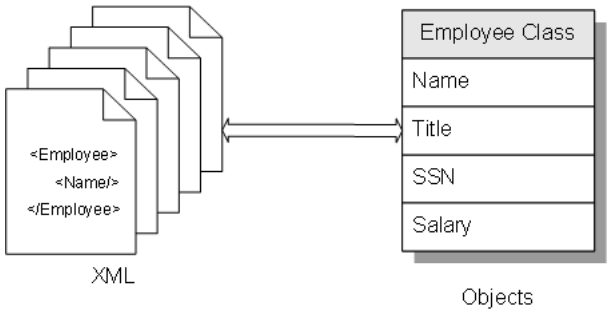
When you define the XML projection for a class:

- The system generates additional INT code for the class that enables you to use class instances as XML documents. (This code is generated when you compile the class, and you should not edit it.)
- Each property of your class automatically inherits from %XML.PropertyParameters.
- %XML.Adaptor adds XML-related class parameters to your class.
- %XML.PropertyParameters adds XML-related property parameters to properties in your class.
- Data type properties define the **LogicalToXSD()** and **XSDToLogical()** methods, which control how data is transformed when output to XML or input from XML.

Then, if the default projection is not suitable for your needs, you edit the XML-related parameters in your class as needed.

1.3 Projection Options

The XML projection for a given class determines how instances of that class correspond to XML documents and vice versa. For example:



You can control the XML projection in many ways, including the following:

- Controlling the structure to which a property is projected. For example, simple properties can be projected as either elements or attributes — or they cannot be projected at all, as shown in the preceding example.
- Controlling the XML element and attribute names.
- Controlling the XML namespaces to which elements and attributes are assigned.
- Controlling the details of how the Caché class is mapped to an XML schema.

Most of these parameters affect all the scenarios in which you use the XML-enabled class. A few parameters apply in certain scenarios; this book notes these exceptions.

1.4 Related Tools in Caché

When you define the XML projection for a class, you have access to a large set of Caché tools, which are suitable for many practical applications. You can use these tools to work with your class in any of the following ways:

- Export objects of that class to XML documents.
- Import XML documents into Caché, which creates new instances of that class, which you can then save.
- Use objects of that class as arguments for web services and web clients.
- Generate XML schemas. Caché implicitly defines an XML type for the class and uses that for validation when using objects of that class in any of the preceding ways.

Caché provides additional tools for working with XML documents, including arbitrary XML documents that do not correspond to Caché classes. These tools include support for DOM, XPath, and XSLT.

Caché uses the Caché SAX Parser to validate and parse inbound and outbound XML documents (SAX means "Simple API for XML"). The Caché SAX Parser is a built-in SAX XML validating parser using the standard Xerces library. Caché SAX communicates with a Caché process using a high-performance, in-process call-in mechanism. You can fine-tune the parser or provide your own custom SAX interface classes.

For information on using any of the XML tools described here, see *Using Caché XML Tools*.

For information on web services and clients, see *Creating Web Services and Web Clients in Caché*.

1.5 Possible Applications for XML Documents

You can use XML documents in a variety of practical applications, including:

- As a standard format in messaging applications. This includes industry-standard protocols as well as homegrown solutions.
- As a standard format for data exchange between applications and users.
- As a standard representation for external data storage. This may include traditional database records or it may include more complex content such as documentation.
- As the payload of SOAP messages sent between web services and web clients.
- As the contents of an XData block in a class definition. See "XData Blocks" in *Using Caché Objects*.

2

Projecting Caché Objects to XML

This chapter describes how to project Caché objects as XML documents. This chapter includes the following topics:

- How to project an object to XML
- How to ensure that the properties have XML projections
- Summary of the default XML projection
- Example XML projection
- How to specify the available format options (literal, encoded, or both)
- How to control the projection of a simple property
- How to control the projection of an object-valued property
- How to control the projection of a collection property
- How to control the projection of a relationship property
- How to control the projection of a stream property
- How to control the availability of a projected property (import, export, or both)
- How to disable the XML projection
- A note on methods in the %XML.Adaptor class

Class and Property Parameters Discussed in This Chapter
• *ELEMENTTYPE* • *XMLPROJECTION* • *XMLFORMAT* • *XMLENABLED* • *XMLSUMMARY* • *XMLDEFAULTREFERENCE* • *GUIDENABLED* • *XMLIO*

Later chapters discuss the details of controlling the element and attribute names and controlling the associated XML schema.

2.1 Projecting a Caché Object to XML

To project an object to XML (alternatively, to define the XML projection of that object), do the following:

1. Add %XML.Adaptor to the superclass list of the class that defines the object.

   For example:

   ```
   Class MyApp.MyClass Extends (%Persistent, %XML.Adaptor)
   {
   //class details
   }
   ```

   This step *XML-enables* the class.

   Alternatively, if the object you are projecting is an instance of a system class, create and use a subclass instead. In the subclass, add %XML.Adaptor to the superclass list.

   Also, Caché provides many specialized XML-enabled classes in its class library. These include classes in the %Zen, %CSP, %Net, %SOAP, and other packages. All these XML-enabled classes inherit from %XML.Adaptor.

2. If you are creating an XML-enabled subclass of %ListOfDataTypes, %ArrayOfDataTypes, %ListOfObjects, or %ArrayOfObjects, then in the subclass, specify the *ELEMENTTYPE* class parameter. For example:

   ```
   Class MyApp.MyIntegerCollection Extends %ListOfDataTypes
   {
   Parameter ELEMENTTYPE="%Library.Integer";

   }
   ```

   For *ELEMENTTYPE*, specify the complete package and class name of the class used in the collection. If you do not specify this parameter, the type is assumed to be a string.

   This step is necessary only if you need a complete XML schema.

3. Ensure that each property of the XML-enabled class has an XML projection if suitable, as described in the next section.

   In most cases, if the property has an object value, you must XML-enable the class that defines the property. The exceptions are collections and streams *when used as properties*, as described in the next section.

   No work is necessary for data type classes.

4. Recompile the changed classes.

Now you can, for example, write the object to an XML document, as described in *Using Caché XML Tools*.

2.1.1 Exceptions for Objects Used with Web Methods

If you use %ListOfDataTypes, %ListOfObjects, %ArrayOfDataTypes, %ArrayOfObjects as input or output for a web method, it is not necessary to create an XML-enabled subclass. It is necessary to specify *ELEMENTTYPE*, but you can do this in the method signature.

If you use a stream class as input or output for a web method, it is not necessary to create an XML-enabled subclass.

For further information, see *Creating Web Services and Web Clients in Caché*.

2.2 Ensuring That Properties Have Projections to XML

To ensure that each property of the object has an XML projection:

- For each simple (non-object) property of this object, no work is necessary. Each Caché data type already has an XML projection.
- For each stream property, no work is necessary. The XML tools treat stream objects specially when used as properties of an XML-enabled class.
- For object-valued properties other than collections, add %XML.Adaptor to the superclass list of the referenced class.

  This includes relationship properties.
- For information on collections, see the following subsections.

For information on how Caché XML tools handle property values of different types, see "Controlling Transformations of Values," later in this book. For information on how property types are projected to XML types, see "Controlling the Projection to XML Schemas," later in this book.

2.2.1 Properties Defined with List of or Array of

For each property that is defined with the syntax `Property PropName As List of classname` or `Property PropName As Array of classname`, do the following:

- If *classname* is an object class, XML-enable that class. That is, add %XML.Adaptor to the superclass list of *classname*.
- If *classname* is a datatype class, no work is needed.
- If *classname* is a stream class, and if the property is a list, no work is needed. The streams are projected to XML as strings.

Note: Caché does not support projecting arrays of streams to XML. If your object has a property that is defined as an array of streams, include `XMLPROJECTION="none"` for that property.

For example:

```
Class MyApp.MyXMLObject Extends (%RegisteredObject, %XML.Adaptor)
{
Property MyListOfObjects As list Of MyApp.OtherXMLObject;

Property MyArrayOfObjects As array Of MyApp.OtherXMLObject;

Property MyListOfDT As list Of %String;

Property MyArrayOfDT As array Of %String;

Property MyListOfStreams As list Of %GlobalCharacterStream;

Property MyArrayOfStreams As array Of %GlobalCharacterStream(XMLPROJECTION = "NONE");
}
```

2.2.2 Properties of Type %ListOfDataTypes or %ArrayOfDataTypes

The XML tools automatically project properties of type %ListOfDataTypes or %ArrayOfDataTypes as containers, as shown later in this book. By default, the container includes string elements.

If you need a correct XML schema, and if it is inappropriate to assume that the elements are strings, then create and use a subclass of the collection class. In the subclass, specify the *ELEMENTTYPE* class parameter. For example:

```
Class MyApp.MyIntegerCollection Extends %ListOfDataTypes
{
Parameter ELEMENTTYPE="%Library.Integer";
}
```

For *ELEMENTTYPE*, specify the complete package and class name of the class used in the collection.

2.2.3 Properties of Type %ListOfObjects or %ArrayOfObjects

The XML tools automatically project properties of the type %ListOfObjects or %ArrayOfObjects as containers, as shown later in this book. It is necessary, however, to XML-enable the class used in the collection.

For a property of type %ArrayOfObjects, the class used in the collection cannot be a stream class.

If you need a complete XML schema, then you must specify the element type for the collection. To do so, create and use a subclass of the collection class. In the subclass, specify the *ELEMENTTYPE* class parameter, as shown in the previous section.

2.2.4 Exceptions

If a given property is not projected to XML, there is no need to XML-enable the class to which it refers. The following properties are not projected to XML:

- Private properties
- Multidimensional properties
- Properties that specify the *XMLPROJECTION* parameter as `"NONE"`

You can project private properties if you set the *XMLPROJECTION* property parameter to `"ELEMENT"` or some other appropriate value, as described later in this chapter. The same applies to multidimensional properties. Also, you can project a multidimensional property only if its top node has a value; note that only the top node is included in the projection.

2.3 Summary of the Default Projection

The default XML projection is as follows:

- An object instance corresponds to a top-level XML element.
- Only properties are projected. No other class members are projected.

  Also, private properties and multidimensional properties are ignored.
- Properties are projected to XML in the same order in which they appear within Studio.
- Any property without a specified type is assumed to be a string.
- Each object-valued property corresponds to an XML element within the enclosing top-level XML element.

  Its properties are nested within this element.
- Collections are projected as nested elements. This is true for both collections of data types and collections of objects.

  The lower-level details are slightly different for lists and arrays.
- Relationships are treated in the same way as list properties.

  The XML projection contains only one side of the relationship; an error occurs if you attempt to project both sides.

- Character streams are projected as strings.
- Binary streams are projected using strings with base–64 encoding.

2.4 Example XML Projection

This section shows an XML-enabled class and its XML projection. Later sections in this chapter show additional example.

2.4.1 Example XML-Enabled Class

The following shows an XML-enabled class that includes the major structural property variations:

```
Class Basics.BasicDemo Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property SimpleProp As %String;

Property ObjProp As SimpleObject;

Property Collection1 As list Of %String;

Property Collection2 As list Of SimpleObject;

Property MultiDimProp As %String [ MultiDimensional ];

Property PrivateProp As %String [ Private ];

}
```

The *XMLTYPENAMESPACE* parameter is discussed later; this specifies the target namespace for types defined in this class.

The `SimpleObject` class is also XML-enabled:

```
Class Basics.SimpleObject Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property MyProp As %String;

Property AnotherProp As %String;

}
```

2.4.2 Example XML Document

The following shows an XML document generated from an instance of the `BasicDemo` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<BasicDemo>
  <SimpleProp>abc</SimpleProp>
  <ObjProp>
    <MyProp>12345</MyProp>
    <AnotherProp>67890</AnotherProp>
  </ObjProp>
  <Collection1>
    <Collection1Item>list item 1</Collection1Item>
    <Collection1Item>list item 2</Collection1Item>
  </Collection1>
  <Collection2>
    <SimpleObject>
      <MyProp>12345</MyProp>
      <AnotherProp>67890</AnotherProp>
    </SimpleObject>
    <SimpleObject>
      <MyProp>12345</MyProp>
```

```
      <AnotherProp>67890</AnotherProp>
    </SimpleObject>
  </Collection2>
</BasicDemo>
```

2.4.3 Example Schema

The following shows the schema for the XML type namespace used in the two sample classes:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="BasicDemo">
    <sequence>
      <element minOccurs="0" name="SimpleProp" type="s:string"/>
      <element minOccurs="0" name="ObjProp" type="s01:SimpleObject" xmlns:s01="mytypes"/>
      <element minOccurs="0" name="Collection1" type="s02:ArrayOfCollection1ItemString"
xmlns:s02="mytypes"/>
      <element minOccurs="0" name="Collection2" type="s03:ArrayOfSimpleObjectSimpleObject"
xmlns:s03="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="SimpleObject">
    <sequence>
      <element minOccurs="0" name="MyProp" type="s:string"/>
      <element minOccurs="0" name="AnotherProp" type="s:string"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfCollection1ItemString">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="Collection1Item" nillable="true"
type="s:string"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfSimpleObjectSimpleObject">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="SimpleObject"
               nillable="true" type="s04:SimpleObject" xmlns:s04="mytypes"/>
    </sequence>
  </complexType>
</schema>
```

2.5 Specifying Format Options for the Projected XML Document

There are two basic formats of XML documents, *literal* and *encoded* (SOAP-encoded). For examples of these formats, see "Introduction to Caché XML Tools" in *Using Caché XML Tools*. You specify one of these formats when you export data from or import data into an XML-enabled class.

When you add %XML.Adaptor to a class and compile it, Caché writes additional code to the generated routines. By default, this additional code supports both formats. If you need only one format, you can suppress the other format and reduce the amount of generated code. To do so, specify the *XMLFORMAT* parameter for that class. Use one of the following values (not case-sensitive):

- `"LITERAL"` — The class supports only literal format.
- `"ENCODED"` — The class supports only encoded format. (It does support both SOAP 1.1 and SOAP 1.2 versions).
- Null (the default) — The class supports both literal and encoded formats.

2.6 Controlling the Form of the Projection for Simple Properties

A simple property is a property whose type is a data type class or whose type is undeclared (any property without a specified type is assumed to be a string).

To control the form of the XML projection for a simple property, set the *XMLPROJECTION* parameter for that property. Use one of the following values (not case-sensitive):

Table 2–1: Effect of XMLPROJECTION on Simple Properties

Value of XMLPROJECTION	Effect on Non-collection Properties
`"ELEMENT"`	The property is projected as an element. This is the default for non-collection properties.
`"ATTRIBUTE"`	The property is projected as an attribute.
`"XMLATTRIBUTE"`	The property is projected as an attribute with the prefix `xml`.
`"WRAPPED"`	The property is projected as an element with a subelement; the name of the subelement is based on the data type of the property.
`"CONTENT"`	The property is projected as the primary content for this class (that is, the contents of the property are written without any enclosing element). In any class, you cannot specify this value for more than one property.
`"NONE"`	The property is not projected to XML.

The *XMLPROJECTION* parameter also accepts the values `"COLLECTION"` or `"ELEMENTREF"`, but those values are deprecated and are not documented here; for details, see the class documentation for %XML.PropertyParameters.

2.6.1 Basic XMLPROJECTION Variations

The following class defines simple properties that use all variations of *XMLPROJECTION* except for `"CONTENT"`:

```
Class xmlproj.SimpleProps Extends (%RegisteredObject, %XML.Adaptor)
{

Property Simple1 As %String (XMLPROJECTION="attribute");

Property Simple2 As %String (XMLPROJECTION="xmlattribute");

Property Simple3 As %String;

Property Simple4 As %String (XMLPROJECTION="element");

Property Simple5 As %String (XMLPROJECTION="wrapped");

Property Simple6 As %String (XMLPROJECTION="none");

}
```

The following shows an example of the XML representation of an instance of this class:

```
<SimpleProps Simple1="The quick" xml:Simple2="brown fox">
  <Simple3>jumps</Simple3>
  <Simple4>over</Simple4>
  <Simple5>
    <string>the lazy</string>
  </Simple5>
</SimpleProps>
```

2.6.2 Projecting a Property as Content

The "CONTENT" value enables you to project the class as a simple element with some text content and possibly some attributes, but no subelements.

The following shows an example class that uses this value:

```
Class xmlproj.SimpleContentProp Extends (%RegisteredObject, %XML.Adaptor)
{

Property Simple1 As %String(XMLPROJECTION = "content");

Property Simple2 As %String(XMLPROJECTION = "attribute");

Property Simple3 As %String(XMLPROJECTION = "element");

}
```

When you export an instance of such a class:

- If the property marked with "CONTENT" has a value, this value is exported as the content of the class. Any properties that are projected as attributes are also exported. Other properties are ignored. For example:

  ```
  <SimpleContentProp Simple2="other value">The quick brown fox jumps over the lazy
  dog</SimpleContentProp>
  ```

- If the property marked with "CONTENT" is null, then it is ignored, and the other properties are all exported as specified by their values for *XMLPROJECTION*. For example:

  ```
  <SimpleContentProp Simple2="other value">
    <Simple3>yet another value</Simple3>
  </SimpleContentProp>
  ```

You cannot specify *XMLPROJECTION* as "CONTENT" for more than one property in any class. Also, you can use this value only for a property that contains simple, literal value, not a collection or other type of object.

2.7 Controlling the Form of the Projection for Object-Valued Properties

For each object-valued property, the default XML projection consists of an XML element (to represent the object itself) with subelements or attributes to represent the properties of that object, as controlled by the XML projection options in that object class. For an example, see "Example XML Projection," earlier in this book.

Note: Later sections discuss the following special object-valued properties: collections, relationships, and streams.

2.7.1 Specifying the Form of the Projection for an Object-Valued Property

To control how a object property is projected, set the *XMLPROJECTION* parameter for that property, as follows:

Table 2–2: Effect of XMLPROJECTION on Object Properties

Value of XMLPROJECTION	Effect on Collection Properties
"WRAPPED"	The property is projected as an element with subelements. The element corresponds to the object class. Each subelement corresponds to a property of that class. This is the default for object properties (other than streams).
"ELEMENT"	Each property of the object class is projected as an element, without being wrapped in a parent element.
"NONE"	The property is not projected to XML.
"ATTRIBUTE", "XMLATTRIBUTE", or "CONTENT"	Compile-time error.

For example, consider the following class:

```
Class Basics.ObjectPropsDemo Extends (%RegisteredObject, %XML.Adaptor)
{

Property Object1 As SimpleObject(XMLPROJECTION = "wrapped");

Property Object2 As SimpleObject(XMLPROJECTION = "element");

}
```

The following shows an example of the XML representation of an instance of this class:

```
<ObjectPropsDemo>
  <Object1>
    <SimpleObject>
      <MyProp>abcdef</MyProp>
      <AnotherProp>qrstuv</AnotherProp>
    </SimpleObject>
  </Object1>
  <Object2>
    <MyProp>abcdef</MyProp>
    <AnotherProp>qrstuv</AnotherProp>
  </Object2>
</ObjectPropsDemo>
```

2.7.2 Specifying an XML Summary

You can easily specify which properties of a class to project to XML when that class is used as a property:

- In the class, specify the *XMLSUMMARY* class parameter as a comma-separated list of the properties to project to XML, in the exact case used in the class definition.

  This parameter has no effect unless you also specify either or both of the following parameters.

- In the same class, optionally specify the *XMLDEFAULTREFERENCE* as "SUMMARY" or "COMPLETE" (the default); these values are not case-sensitive. The option "SUMMARY" means that only the properties listed in *XMLSUMMARY* should be used in the projection when this class is used as a property. The option "COMPLETE" means that all properties that have XML projections should be used.

  These values are not case-sensitive.

- In a class that uses this class as a property, optionally specify the *XMLREFERENCE* property parameter as "SUMMARY" or "COMPLETE" (the default); these values are not case-sensitive. This overrides the *XMLDEFAULTREFERENCE* class parameter.

  You can set this property parameter only for a property whose value is an object.

For example, consider the following Address class:

```
Class xmlsummary.Address Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLSUMMARY = "City,ZipCode";

Parameter XMLDEFAULTREFERENCE = "SUMMARY";

Property Street As %String;

Property City As %String;

Property State As %String;

Property ZipCode As %String;

}
```

The following shows an example of the XML representation of an instance of this class:

```
<Address>
  <Street>47 Winding Way</Street>
  <City>Middlebrook</City>
  <State>GA</State>
  <ZipCode>50291</ZipCode>
</Address>
```

Notice that all properties are included.

Now consider another class that uses the `Address` class as a property:

```
Class xmlsummary.Person Extends (%RegisteredObject, %XML.Adaptor)
{

Property Name As %String;

Property Address as Address;

}
```

The following shows an example of the XML representation of an instance of this class:

```
<Person>
  <Name>Penelope Farnsworth</Name>
  <Address>
    <City>Middlebrook</City>
    <ZipCode>50291</ZipCode>
  </Address>
</Person>
```

Here, because the `Address` class is a property of this class, the *XMLSUMMARY* and *XMLDEFAULTREFERENCE* parameters are used, and only the class properties listed in *XMLSUMMARY* are used in the projection.

You can use the `"COMPLETE"` option to force an override. For example, the following class also uses the `Address` class as a property but specifies *XMLREFERENCE* as `"COMPLETE"`:

```
Class xmlsummary.Employee Extends (%RegisteredObject, %XML.Adaptor)
{

Property Name As %String;

Property Address As Address(XMLREFERENCE = "COMPLETE");

}
```

The following shows an example of the XML representation of an instance of this class:

```
<Employee>
  <Name>Malcom Winters</Name>
  <Address>
    <Street>770 Enders Lane</Street>
    <City>Middlebrook</City>
    <State>GA</State>
    <ZipCode>50293</ZipCode>
  </Address>
</Employee>
```

All the properties that have XML projections are included; in this case, this means all properties are included.

2.7.3 Projecting Only Object Identifiers

Instead of projecting an object-valued property as shown earlier in this section, you can project only an identifier for the object. To do so, use one of the following values for the *XMLDEFAULTREFERENCE* class parameter or the *XMLREFERENCE* property parameter:

- The `"ID"` option projects only the internal ID of the object, as stored on disk. It does not project any properties. For example, consider the following class:

  ```
  Class xmlidentifiers.Person Extends (%Persistent, %XML.Adaptor)
  {

  Property Name As %String;

  Property PrimaryCarePhysician As Person (XMLREFERENCE = "ID");

  }
  ```

  The following shows an example of the XML representation of an instance of this class:

  ```
  <Person>
    <Name>Sam Smith</Name>
    <PrimaryCarePhysician>24</PrimaryCarePhysician>
  </Person>
  ```

- The `"OID"` option projects only the OID of the object (as *package.class*,*ID*). It does not project any properties. If we use this option for the `PrimaryCarePhysician` property, the preceding `Person` object would be projected as follows:

  ```
  <Person>
    <Name>Sam Smith</Name>
    <PrimaryCarePhysician>xmlidentifiers.Person,24</PrimaryCarePhysician>
  </Person>
  ```

- The `"GUID"` option projects only the GUID (globally unique ID) of the object, if available. The GUID for an object is null unless the *GUIDENABLED* class parameter is 1. Suppose that we redefine the `Person` class as follows:

  ```
  Class xmlidentifiers.Person Extends (%Persistent, %XML.Adaptor)
  {

  Parameter GUIDENABLED=1;

  Property Name as %String;

  Property PrimaryCarePhysician As Person (XMLREFERENCE = "GUID");

  }
  ```

  In this case, the XML representation of an instance of this class could be as follows:

  ```
  <Person>
    <Name>Sam Smith</Name>
    <PrimaryCarePhysician>D0F383EB-DB31-4C11-AD56-AA14EB37B734</PrimaryCarePhysician>
  </Person>
  ```

Note: For the property parameter *XMLREFERENCE*, you can use the `"ID"`, `"OID"`, and `"GUID"` options only if the value is a persistent object. You receive compile-time errors otherwise.

Similarly, if you set the class parameter *XMLDEFAULTREFERENCE* to `"ID"`, `"OID"`, or `"GUID"`, and if the class has properties whose values are non-persistent objects, you must explicitly set the property parameter *XMLREFERENCE* to `"COMPLETE""` or `"SUMMARY"` for those properties.

2.8 Controlling the Form of the Projection for Collection Properties

To control the form of the XML projection for a collection property, set the *XMLPROJECTION* parameter for that property, as follows:

Table 2–3: Effect of XMLPROJECTION on Collection Properties

Value of XMLPROJECTION	Effect on Collection Properties
`"WRAPPED"`	The property is projected as an element with subelements; each subelement corresponds to an item of the collection. This is the default for collection properties.
`"ELEMENT"`	Each item in the collection is projected as an element, without being wrapped in the parent property.
`"NONE"`	The property is not projected to XML.
`"ATTRIBUTE"`, `"XMLATTRIBUTE"`, or `"CONTENT"`	Compile-time error.

The following sections show examples with properties that are lists or arrays of data types. For collections of objects, the projected elements can have further structure, recursively, depending on the XML projections of those objects.

2.8.1 Specifying the Form of the Projection for List Properties

The following class defines collection properties that use the `"WRAPPED"` and `"ELEMENT"` values:

```
Class xmlproj.DataTypeColls Extends (%RegisteredObject, %XML.Adaptor)
{

Property Collection1 As list Of %String;

Property Collection2 As list Of %String (XMLPROJECTION="wrapped");

Property Collection3 As list Of %String (XMLPROJECTION="element");

}
```

The following shows an example of the XML representation of an instance of this class:

```
<?xml version="1.0" encoding="UTF-8"?>
<DataTypeColls>
  <Collection1>
    <Collection1Item>list item 1</Collection1Item>
    <Collection1Item>list item 2</Collection1Item>
  </Collection1>
  <Collection2>
    <Collection2Item>list item 1</Collection2Item>
    <Collection2Item>list item 2</Collection2Item>
  </Collection2>
  <Collection3>list item 1</Collection3>
  <Collection3>list item 2</Collection3>
</DataTypeColls>
```

For the `Collection3` property, which uses `"ELEMENT"`, the XML projection disregards the list nature of the property and instead treats each list item as a separate property of the class.

2.8.2 Specifying the Form of the Projection for Array Properties

For an array, each array item has both a value and a key, and both of these pieces of information must be represented in XML. The key is always projected as an XML attribute within the element. Consider the following class:

```
Class xmlproj.DataTypeArray Extends (%RegisteredObject, %XML.Adaptor)
{

Property ArrayProp As array Of %String;

}
```

The following shows an example of the default XML representation of an instance of this class:

```
<?xml version="1.0" encoding="UTF-8"?>
<DataTypeArray>
  <ArrayProp>
    <ArrayPropItem ArrayPropKey="1">apples</ArrayPropItem>
    <ArrayPropItem ArrayPropKey="2">bananas</ArrayPropItem>
    <ArrayPropItem ArrayPropKey="3">chocolate</ArrayPropItem>
  </ArrayProp>
</DataTypeArray>
```

If you specify *XMLPROJECTION* as `"ELEMENT"`, the XML projection is as follows instead:

```
<?xml version="1.0" encoding="UTF-8"?>
<DataTypeArray>
  <ArrayProp ArrayPropKey="1">apples</ArrayProp>
  <ArrayProp ArrayPropKey="2">bananas</ArrayProp>
  <ArrayProp ArrayPropKey="3">chocolate</ArrayProp>
</DataTypeArray>
```

2.9 Controlling the Form of the Projection for Relationships

Relationships are projected to XML in the same way as other properties, depending on the nature of the collection used in them:

- For a parent-child relationship:
  - The relationship property in the parent object is a collection property, specifically a list of objects. See “Controlling the Projection for Collection Properties.”
  - The relationship property in the child object is an object-valued property.

    This relationship is not projected to XML by default.
- For a one-to-many relationship:
  - The relationship property in the single object is a collection property, specifically a list of objects.
  - The relationship property in the other object is an object-valued property.

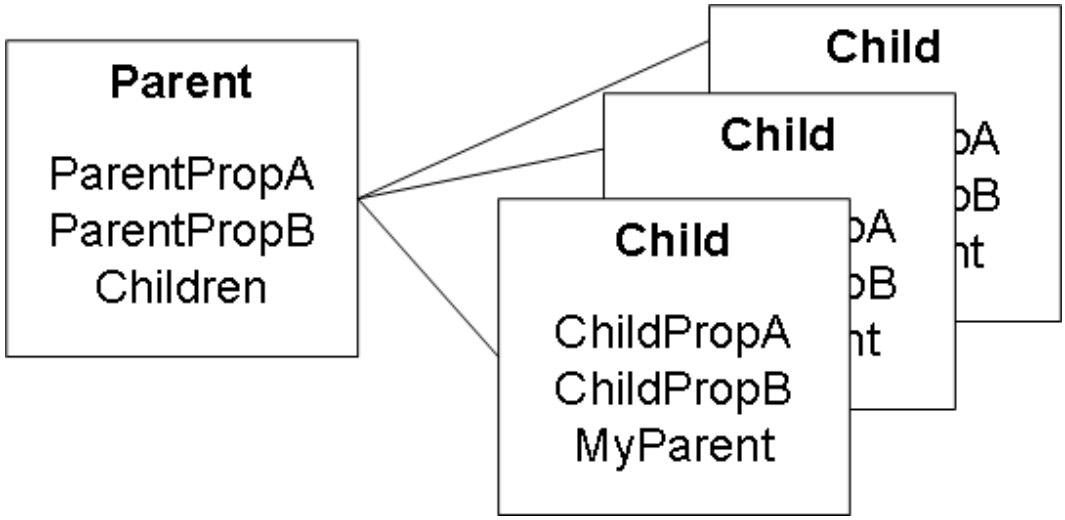
    This relationship is not projected to XML by default.

At any given time, only one side of the relationship can be projected, because otherwise there would be an infinite loop. To reverse the way in which the projection is done, use the *XMLPROJECTION* property parameter.

This principle is best explained by examples.

2.9.1 The Default Projection for Relationships

First, consider the following pair of classes:



The class `Parent` is the parent of the class `Child`.

The class `Parent` has two properties (`ParentPropA` and `ParentPropB`) in addition to the relationship property (`Children`).

Similarly, the class `Child` has two properties (`ChildPropA` and `ChildPropB`) in addition to the relationship property (`MyParent`).

When you project these classes to XML, you get the following results by default:

- The XML projection for `Parent` includes projections for three properties: `ParentPropA`, `ParentPropB`, and `Children`. The `Children` property is treated like any other collection. That is, by default it is projected as a set of nested elements.

  For example:

```
<Parent>
  <ParentPropA>12345</ParentPropA>
  <ParentPropB>67890</ParentPropB>
  <Children>
    <Child>
      <ChildPropA>abc</ChildPropA>
      <ChildPropB>def</ChildPropB>
    </Child>
    <Child>
      <ChildPropA>ghi</ChildPropA>
      <ChildPropB>jkl</ChildPropB>
    </Child>
  </Children>
</Parent>
```

- The XML projection for `Child` includes projections for two properties: `ChildPropA` and `ChildPropB`. The property `MyParent` is ignored.

  For example:

```
<Child>
  <ChildPropA>abc</ChildPropA>
  <ChildPropB>def</ChildPropB>
</Child>
```

The same is true for one-to-many relationships. Specifically, the object on the "one" side includes a projection for the relationship property. The object on the "many" side does not include a projection for the relationship property.

2.9.2 Projecting the Other Side of the Relationship Instead

By specifying the *XMLPROJECTION* parameter in the relationship properties of both classes, you can project the other side of the relationship instead. The following class is a variation of the class used in the default example.

```
Class Relationships2.Parent Extends (%Persistent, %XML.Adaptor)
{

Property ParentPropA As %String;

Property ParentPropB As %String;

Relationship Children As Child(XMLPROJECTION = "NONE") [ Cardinality = children, Inverse = MyParent ];

}
```

Similarly, the `Child` class is as follows:

```
Class Relationships2.Child Extends (%Persistent, %XML.Adaptor)
{

Property ChildPropA As %String;

Property ChildPropB As %String;

Relationship MyParent As Parent (XMLPROJECTION="element") [ Cardinality = parent, Inverse = Children
];

}
```

When you project these classes to XML, you get the following results:

- The XML projection for `Parent` ignores the `Children` property:

```
<Parent>
  <ParentPropA>12345</ParentPropA>
  <ParentPropB>67890</ParentPropB>
</Parent>
```

- The XML projection for `Child` includes projections for all its properties:

```
<Child>
  <ChildPropA>abc</ChildPropA>
  <ChildPropB>def</ChildPropB>
  <MyParent>
    <ParentPropA>12345</ParentPropA>
    <ParentPropB>67890</ParentPropB>
  </MyParent>
</Child>
```

2.10 Controlling the Form of the Projection of a Stream Property

For stream properties, the options for the *XMLPROJECTION* are as follows:

Table 2–4: Effect of XMLPROJECTION on Stream Properties

Value of XMLPROJECTION	Effect on Stream Properties
`"ELEMENT"`	The stream contents are contained in an element.

Value of XMLPROJECTION	Effect on Stream Properties
"WRAPPED"	Treated in the same way as "ELEMENT".
"CONTENT"	The stream contents are projected as described in "Projecting a Property as Content" in the chapter "Projecting Caché Objects to XML." For all other properties, the *XMLPROJECTION* parameter must be "NONE".
"NONE"	The property is not projected to XML.
"ATTRIBUTE" or "XMLATTRIBUTE"	Compile-time error.

This section shows examples of how streams are projected.

For example, consider the following class:

```
Class Basics.StreamPropDemo Extends (%Persistent, %XML.Adaptor)
{

Property BinStream As %Library.GlobalBinaryStream;

Property CharStream As %Library.GlobalCharacterStream;

}
```

The following shows an example of the XML representation of an instance of this class:

```
<StreamPropDemo>
  <BinStream>/9j/4AAQSkZJRgABAQEASABIAAD/2wBDAAUDBAQEAwUEBAQFBQUGBwwIBwcHBw8LCwkMEQ8SEhEP
ERETFhwXExQaFRERGCEYGh0dHx8fExciJCIeJBweHx7/2wBDAQUFBQcGBw4ICA4eFBEUHh4eHh4e
...
VcE/wkZ5wGJBH/joP50UVfQkqaS5dbi34EZtpJgPRlUkf1H402Fy9oIWHHlPj2K/Nn9cfhRRSGip
ZHzbmEPnEwZGGePu5/nj8qNJcpcrG4DxSuEkToDnPPtRRUyKGhPsuqlAxbypAhJ43A/y44q5HbNM
vmx3U9vuJDLG+ASCQW+pxRRSKP/Z</BinStream>
  <CharStream><![CDATA[This is a sample file.
This is line 2.
This is line 3.
This is line 4.]]></CharStream>
</StreamPropDemo>
```

2.11 Controlling the Availability of Projected Properties

You can specify whether each projected property is used by import, export, or both. To do this, you set the *XMLIO* parameter, which controls how the export and import methods of %XML.Writer and %XML.Reader classes handle a property. This parameter can take one of the following values (not case-sensitive):

- "INOUT" — This property is used by both export and import. This is the default for projected properties.
- "IN" — This property is used by import but is ignored by export.
- "OUT" — This property is used by export but causes an error on import. If an XML element corresponding to this property is found in an XML document, then import returns an error.
- "CALC" — This property is used by export but is ignored by import. If an XML element corresponding to this property is found in an XML document, then import ignores it. Typically, this is used for calculated properties (whose value is based on the value of other properties) so that you can export a document with all values and ignore the calculated values on import.

This parameter has no effect on a property that is not projected to XML.

2.12 Disabling the Projection

If a class is XML-enabled and you want to prevent the class from being projected (perhaps during testing for some reason), you can set the class parameter *XMLENABLED* to 0. The default is 1.

If you use *XMLENABLED* to prevent a class from being projected, this class cannot be used as a property by any class that is projected to XML. Setting *XMLENABLED* to 0 is the same as removing %XML.Adaptor from the superclass list.

2.13 Methods in %XML.Adaptor

The methods in %XML.Adaptor are deprecated and are mostly not documented. You should instead use the more robust classes %XML.Writer and %XML.Schema, which provide greater support for namespaces. For information, see *Using Caché XML Tools*.

3

Controlling Transformations of Values

This chapter discusses how values are transformed as you export objects to XML and import XML into objects, and it discusses your options for controlling those transformations.

- Introduction
- How to control how XML special characters are handled
- How to handle the UTC time zone indicator
- How to use the values in DISPLAYLIST
- How to control line endings for stream properties imported from XML
- How to specify a default date/time value to use when an invalid date is imported
- How to handle nonprinting characters

The XML examples in this chapter are in literal format.

The following chapter discusses null values and missing values.

Class and Property Parameters Discussed in This Chapter
• *ESCAPE* • *CONTENT* • *XMLTIMEZONE* • *DISPLAYLIST* • *VALUELIST* • *XMLDEFAULTVALUE* • *XMLLISTPARAMETER* • *XMLSTREAMMODE*

3.1 Introduction

An XML-enabled object typically includes properties defined by Caché data types. Each data type class defines the **LogicalToXSD()** and **XSDToLogical()** methods. Whenever XML output is requested for the object, the Caché XML tools

automatically call the **LogicalToXSD()** method for each property, to convert the data to the appropriate format for use in XML. Similarly, whenever an XML document is used as input, the Caché XML tools call the **XSDToLogical()** method to convert the data into the correct format for Caché.

For example, in the %Binary data type class, the **LogicalToXSD()** method converts the outbound value using the **$SYSTEM.Encryption.Base64Encode()** method. Similarly, the **XSDToLogical()** method converts the inbound value using the **$SYSTEM.Encryption.Base64Decode()** method.

A Caché class can also include stream-valued properties, but the stream classes do not define the **LogicalToXSD()** and **XSDToLogical()** methods. Instead, the XML tools treat stream classes specially when they are used as properties of an XML-enabled class. Specifically:

- Character streams are treated in the same way as strings. By default, no changes are made apart from the changes that are necessary due to the presence of XML special characters, as described in the next section.
- When Caché exports to XML, it converts binary stream properties to strings with base–64 encoding (that is, it encodes the data in that way and then exports it). When Caché imports from XML, it does the reverse.

When you use XML-enabled objects, it is sometimes necessary to consider the special cases of values that cannot be directly projected to XML or values that you want to transform for other reasons. The rest of this chapter discusses these cases.

3.2 Handling Special XML Characters

Depending on the context, Caché XML support escapes the ampersand character (&) and certain other characters, when it finds those characters within a property of type string or character stream.

Note: The *ESCAPE* property parameter controls which characters are recognized as special. This parameter is either `"XML"` (the default) or `"HTML"` (not discussed in this book). In the examples in this book, *ESCAPE* is `"XML"`.

For these special characters, you can control how the escaping is performed by setting the *CONTENT* property parameter. The details are different for literal and encoded formats, as follows:

Table 3–1: Form of Escaping for Literal and SOAP-encoded Formats

Value of CONTENT (Case-insensitive)	XML Document in Literal Format	XML Document in SOAP-encoded Format
`"STRING"` (the default)	CData	CData
`"ESCAPE"`	XML entity	XML entity
`"ESCAPE-C14N"`	XML entity*	XML entity*
`"MIXED"`	No escaping is done	CData

*For `"ESCAPE-C14N"`, the escaping is done in accordance with the XML Canonicalization standard. The main difference is that a carriage return is escaped as `&#13;`

3.2.1 Examples

Consider the following class:

```
Class ResearchXForms.CONTENT Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLNAME = "Demo";

Property String1 As %String;

Property String2 As %String(CONTENT = "STRING");

Property String3 As %String(CONTENT = "ESCAPE");

Property String4 As %String(CONTENT = "MIXED");

}
```

`String2` and `String1` are always treated in the same way, because `String2` uses the default value for `CONTENT`.

Literal XML output for this class might look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Demo>
  <String1><![CDATA[value 1 & value 2]]></String1>
  <String2><![CDATA[value 1 & value 2]]></String2>
  <String3>value 1 &amp; value 2</String3>
  <String4>value 1 & value 2</String4>
</Demo>
```

SOAP-encoded XML output would be as follows instead:

```
<?xml version="1.0" encoding="UTF-8"?>
<CONTENT 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">
  <String1><![CDATA[value 1 & value 2]]></String1>
  <String2><![CDATA[value 1 & value 2]]></String2>
  <String3>value 1 &amp; value 2</String3>
  <String4><![CDATA[value 1 & value 2]]></String4>
</CONTENT>
```

3.2.2 Alternative Way to Prevent the Escaping

There is another way to prevent the escaping of special XML characters. You can define the property as one of the special XML types: %XML.String, %XML.FileCharacterStream, or %XML.GlobalCharacterStream. For these data type classes, *CONTENT* is `"MIXED"`.

Note that your application is responsible for ensuring that the property value is valid for the scenario in which it will be used; the %XML.String and other classes do not provide this validation.

3.3 Handling the UTC Time Zone Indicator

For XML-enabled classes, you can specify whether to use the UTC time zone indicator when importing from XML documents. Similarly, you can specify whether to include the UTC time zone indicator on export.

To do so, specify the *XMLTIMEZONE* parameter. Use one of the following values:

- `"UTC"` — In this case, when you import elements with `xsd:time` or `xsd:dateTime`, the data is converted to UTC time. This is the default behavior.

  In compliance with the XML Schema specification, Caché XML support treats the time zone indicator as a pure duration and ignores any named time zones such as EDT.

- `"IGNORE"` — In this case, the UTC time zone indicator is ignored when you import elements with `xsd:time` or `xsd:dateTime`.

On export, UTC time is always used. The *XMLTIMEZONE* parameter controls the UTC zone indicator is included.

Consider the following class:

```
Class ResearchXForms.UTC Extends (%Persistent, %XML.Adaptor)
{

Parameter XMLNAME = "Demo";

Property Time1 As %Time;

Property Time2 As %Time(XMLTIMEZONE = "IGNORE");

Property TimeStamp1 As %TimeStamp;

Property TimeStamp2 As %TimeStamp(XMLTIMEZONE = "IGNORE");

}
```

XML output for this class might be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Demo>
  <Time1>17:52:06Z</Time1>
  <Time2>17:52:06</Time2>
  <TimeStamp1>1976-02-18T17:52:06Z</TimeStamp1>
  <TimeStamp2>1976-02-18T17:52:06</TimeStamp2>
</Demo>
```

3.4 Projecting the Value in DISPLAYLIST

For properties of type %String (or any subclass), the XML projection can use the *DISPLAYLIST* parameter.

Simple properties can specify the *DISPLAYLIST* and *VALUELIST* parameters. The *VALUELIST* parameter specifies a list of possible values for the property; this defines an enumerated property . Often you also specify the *DISPLAYLIST* parameter, which specifies the corresponding values to display.

By default, the XML projection uses the value contained in the object, which is one of the values specified by *VALUELIST*. For properties of type %String, the *XMLLISTPARAMETER* parameter is meant to indicate which parameter contains the list of alternative values to use in the projection. Typically, you set this equal to `"DISPLAYLIST"`. For example, consider the following data type class:

```
Class xmldisplaylist.MyEnumString Extends %String
{

Parameter VALUELIST = ",a,b,c";

Parameter DISPLAYLIST = ",apples,bananas,chocolate";

Parameter XMLLISTPARAMETER = "DISPLAYLIST";

}
```

Also consider the following class, which uses the preceding data type class:

```
Class xmldisplaylist.Demo Extends (%RegisteredObject, %XML.Adaptor)
{

Property Property1 As MyEnumString;

Property Property2 As MyEnumString(DISPLAYLIST = ",red,green,blue", VALUELIST = ",r,g,b");

}
```

The following shows an example of the XML representation of an instance of this class:

```
<Demo>
  <Property1>chocolate</Property1>
  <Property2>red</Property2>
</Demo>
```

In contrast, if the data type class did not specify the *XMLLISTPARAMETER* parameter, the XML representation would be as follows:

```
<Demo>
  <Property1>c</Property1>
  <Property2>r</Property2>
</Demo>
```

3.5 Controlling the Line Endings of Imported Stream Properties

For each property that is a character stream, you can control the line endings in the stream when you import from XML. To do so, you set the *XMLSTREAMMODE* property parameter, which can have either of the following values (not case-sensitive):

- If *XMLSTREAMMODE* equals `"block"` (the default), the normalized XML data is copied unchanged to the stream. The LineTerminator property of the stream is set to $CHAR(10), which makes the import compatible with any traditional newline sequence ($CHAR(10), $CHAR(13), $CHAR(13,10)).
- If *XMLSTREAMMODE* equals `"line"`, the XML data is broken into lines separated by the character given by the LineTerminator property of the stream; see %Library.AbstractStream.

For example, suppose that we have the following data:

```
...
<Stream1>
<![CDATA[this is a line
this is another line
this is another line
]]>
</Stream1>
...
```

We import this data into an object that has a `Stream1` property of type %Stream.GlobalCharacter. By default, *XMLSTREAMMODE* is `"block"` for the property. After we import data, this property will contain the following data:

```
this is a line
              this is another line
                              this is another line
```

If we set *XMLSTREAMMODE* equal to `"line"` for this property, and reimported the data, this property would contain the following data:

```
this is a line
this is another line
this is another line
```

In this case, the line endings are determined by the LineTerminator property of the stream class. This property equals $char(13,10) for %Stream.GlobalCharacter.

3.6 Specifying a Default Date/Time Value

For the %TimeStamp and %DateTime data type classes, the *XMLDEFAULTVALUE* parameter specifies the value to use if the date fails validity check by **$zdatetimeh**. By default, in such a case, a null string is used, and this results in an error when you import data via **XMLImport()**. Specify a valid value for the class.

For example, for %TimeStamp, and %DateTime, specify a date in YYYY-MM-DD HH:MM:SS.nnnnnnnnn format, starting with any year after 1841. For example: `1841-01-1 00:00:00`

3.7 Projecting Nonprinting Characters to XML

XML does not permit nonprinting characters, specifically characters below ASCII 32 (except for carriage return, line feed, and tab).

If you need to project a property to XML and that property contains any of these nonprinting characters, that property must be of type %Binary or %xsd.base64Binary (which is equivalent). This value is automatically converted to base–64 encoding on export to XML (or automatically converted from base–64 encoding on import).

4

Handling Empty Strings and Null Values

This chapter discusses the following topics:

- Default XML projections for empty strings and null values
- How these values are exported to XML
- How these values are imported from XML

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *XMLUSEEMPTYELEMENT* • *XMLIGNORENULL* • *XMLNILNOOBJECT* • *XMLNIL*

4.1 Default Projections of Empty Strings and Null Values

The following table summarizes the default XML projections of empty strings and null values. Note that the XML projections are analogous to the SQL projections, which are also shown here for comparison.

Table 4–1: Default SQL and XML Projections of Empty Strings and Null Values

Caché Value	Default Projection to XML	Projection to SQL
`$char(0)`	Empty element or attribute	SQL empty string
`""`	No projection	SQL NULL value

The following sections describe how these projections work upon export and import, and provide details on your options for controlling these projections.

4.2 Exporting Values

The following table lists the ways in which empty strings and null values can be exported from XML-enabled objects to XML documents, for a property that is projected to XML as an element:

Table 4–2: Exporting Empty Strings and Null Values for a Property Projected to XML as an Element

Details of XML-Enabled Class	Property equals ""	Property equals `$char(0)`
Class specifies default values for the parameters described in this chapter	Exported XML document does not contain an element that corresponds to this property	Exported document contains an empty element that corresponds to this property; see the subsection
Class specifies *XMLIGNORENULL*=1	Exported document contains an empty element that corresponds to this property; see the subsection	
Class specifies *XMLNIL*=1 (and *XMLIGNORENULL* is *not* 1)	Exported document contains an empty element that corresponds to this property and this empty element specifies `xsi:nil="true"`	

The details are similar for a property projected as an attribute:

Table 4–3: Exporting Empty Strings and Null Values for a Property Projected to XML as an Attribute

Details of XML-Enabled Class	Property equals ""	Property equals `$char(0)`
Class specifies default values for the parameters described in this table	Exported XML document does not contain an attribute that corresponds to this property	Exported document contains an empty attribute that corresponds to this property
XMLIGNORENULL=1	Exported document contains an empty attribute that corresponds to this property. For example: `PropName=""`	
XMLNIL=1 (and *XMLIGNORENULL* is *not* 1)	Exported XML document does not contain an attribute that corresponds to this property (*same as the default scenario*)	

4.2.1 Controlling the Form of an Empty Element

By default, Caché writes an empty element with an opening tag and a closing tag. For example:

```
<PropName></PropName>
```

You can instead cause Caché to write a self-closing empty element (which is equivalent). For example:

```
<PropName />
```

To do so, specify the *XMLUSEEMPTYELEMENT* class parameter as 1. The default for this parameter is 0.

4.2.2 Details for XMLIGNORENULL, XMLNIL, and XMLUSEEMPTYELEMENT

XMLIGNORENULL

Used during export to XML (and when writing SOAP messages), this parameter controls whether to ignore null strings (rather than exporting them).

This parameter is a class parameter in all XML-enabled classes. *XMLIGNORENULL* can equal 0 (the default), 1, `"INPUTONLY"`, or `"RUNTIME"` (not case-sensitive).

The *XMLIGNORENULL* class parameter is inherited by subclasses.

XMLNIL

Used during export to XML (and when writing SOAP messages), this parameter controls the use of the `xsi:nil` attribute for null strings.

This parameter is a class parameter and a property parameter in all XML-enabled classes; the property parameter takes precedence. *XMLNIL* can equal 0 (the default) or 1.

The *XMLNIL* class parameter is not inherited by subclasses. The *XMLNIL* property parameter *is* inherited.

XMLUSEEMPTYELEMENT

Used during export to XML (and when writing SOAP messages), this parameter controls whether Caché writes self-closing empty tags. This parameter applies in two scenarios:

- If *XMLUSEEMPTYELEMENT* is 1 for a class, the parameter affects any string-valued properties that equal `""` and that are projected as elements. Any such property is exported as a self-closing empty element.
- If *XMLUSEEMPTYELEMENT* is 1 for a class, and none of the properties appear as elements in the XML export, then the parameter affects the form of the empty element corresponding to the class instance. This element is exported as a self-closing empty element.

If *XMLUSEEMPTYELEMENT* is 1 in a class, the system generates slightly more code for that class. Also the XML processing for that class is slightly less efficient.

4.3 Importing Values

The following table lists the ways in which Caché handles empty, null, or missing elements and attributes when it imports from XML into an XML-enabled object:

Table 4–4: Importing XML Documents with Empty, Null, or Missing Elements and Attributes

Details of XML-Enabled Class	Imported document does not contain the element or attribute	In the imported document, the element or attribute is empty	In the imported document, the element is empty and specifies `xsi:nil="true"`
Class specifies default value for *XMLNILNOOBJECT* parameter	The property is not set	The property is set equal to `$char(0)`	• If the property is a literal-valued property, the property is not set • If the property is an object-valued property, the property is set to a new instance of the referenced class; no properties are set in this instance
Class specifies *XMLNILNOOBJECT*=1			The property is not set

5

Controlling the XML Element and Attribute Names

There is a default correspondence between Caché class and property names and the names of the XML elements and attributes. This chapter describes how to override these defaults. It discusses the following topics:

- Default XML element and attribute names
- How to control the element name for a top-level object
- How to control the element or attribute name for a simple property
- How to control the element and attribute names for a list-type property
- How to control the element and attribute names for an array-type property

Note that Caché also supports the case in which an XML document contains multiple elements with the same name. See the chapter "Special Topics."

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *XMLNAME* • *XMLTYPE* • *XMLITEMNAME* • *XMLKEYNAME*

5.1 Default XML Element and Attribute Names

The default correspondence between Caché names and XML element and attribute name is as follows:

- For a class, the corresponding XML element or attribute name is the same as the short class name.
- For a property in this class, the corresponding XML element or attribute name is the same as the property name.

  (Note that the property definition determines whether it is projected as an XML element or attribute. See "Controlling the Projection for Simple Properties," earlier in this book.)

If the property name includes quotes, the quote marks are not included in the XML element or attribute name. For example, consider the following property:

```
Property "Quoted Property" As %String;
```

This property is projected as the element `<Quoted Property>` or the attribute `Quoted Property`, depending on how this property is mapped.

- If the property is a list or an array, it automatically consists of subelements, each of which is one item in that list or array. By default, the name of the subelement is the property name with `Item` appended to it.
- If a property is an array, the subelement also has an attribute to indicate the corresponding key. By default, the name of that attribute is the property name with `Key` appended to it.

5.2 Controlling the Name of the Element or Attribute for an Object Projected as a Top-Level Element

When you project a class instance as a top-level element, its XML name is determined as follows:

Table 5–1: Tag for Object Projected as Top-Level Element

XMLNAME Parameter of Class	XMLType Parameter of Class	Tag (Element or Attribute)
Specified	*Ignored*	Value of *XMLNAME*
Not specified	Specified	Value of *XMLTYPE*
	Not specified	Short class name

For information on *XMLTYPE*, see the chapter "Controlling the Projection to XML Schemas," later in this book.

For example, if you export objects of the `Sample.Address` class, each of those objects would be shown something like the following, by default:

```
<Address>
  <Street>5064 Elm Street</Street>
  <City>Jackson</City>
  <State>PA</State>
  <Zip>27621</Zip>
</Address>
```

Suppose that you specify the *XMLNAME* parameter of the `Sample.Address` class. For example:

```
Parameter XMLNAME = "HomeAddress";
```

In this case, the output would be as follows instead:

```
<HomeAddress>
  <Street>5064 Elm Street</Street>
  <City>Jackson</City>
  <State>PA</State>
  <Zip>27621</Zip>
</HomeAddress>
```

You can override these parameters when you export objects to XML, as described in *Using Caché XML Tools*.

5.3 Controlling the Tags for Simple Properties

In an XML-enabled object, each simple property is projected as an XML element or attribute, depending on how it is mapped. In either case, by default, the Caché property name is used as the XML element or attribute name. To provide a different XML name for a property, you specify the *XMLNAME* parameter of that property.

For example:

```
Property Zip As %String (XMLNAME = "PostalCode");
```

The output for the previous example would be as follows instead:

```
<HomeAddress>
  <Street>5064 Elm Street</Street>
  <City>Jackson</City>
  <State>PA</State>
  <PostalCode>27621</PostalCode>
</HomeAddress>
```

Note that if a property is in turn another Caché object class, the XML projection ignores the class name and the *XMLNAME* parameter of that class. For example, suppose that the `Person` class has a property named `Address` which is a reference to the `Address` class. The projection for a `Person` object would look something like the following:

```
<Person>
 <Name>Zevon,Juanita Q.</Name>
 <DOB>1986-08-18</DOB>
 <Address>
   <Street>5064 Elm Street</Street>
   <City>Jackson</City>
   <State>PA</State>
   <Zip>27621</Zip>
 </Address>
</Person>
```

The name of the `<Address>` element is determined by the name of the corresponding property in the `Person` object. This is because the address object is a *property* of the object that is being imported or exported (instead of being an object that is being imported or exported directly).

As with any other property, you can override this name by specifying the*XMLNAME* parameter for the property. For example:

```
Property Address As MyApp.Address (XMLNAME = "EmployeeAddress");
```

The output for the previous example would be as follows instead:

```
<Person>
 <Name>Zevon,Juanita Q.</Name>
 <DOB>1986-08-18</DOB>
 <EmployeeAddress>
   <Street>5064 Elm Street</Street>
   <City>Jackson</City>
   <State>PA</State>
   <Zip>27621</Zip>
 </EmployeeAddress>
</Person>
```

5.4 Controlling the Element and Attribute Names for List-Type Properties

Note: This section does not apply to a collection property that specifies *XMLPROJECTION* as `"ELEMENT"`. For such a property, each list item is treated as a separate property of the class. See "Controlling the Projection for Collection Properties," earlier in this book.

In an XML-enabled object, a list-type property is projected to an element with subelements, each of which is one item in that list. Suppose that a Caché object property named `ColorOptions` equals a list of three strings: `"Red"`, `"Green"`, `"Blue"`. By default, this property corresponds to the following XML fragment:

```
<ColorOptions>
  <ColorOptionsItem>Red</ColorOptionsItem>
  <ColorOptionsItem>Green</ColorOptionsItem>
  <ColorOptionsItem>Blue</ColorOptionsItem>
</ColorOptions>
```

This shows the `ColorOptionsItem` subelement, which corresponds to an item in the list. The name for this subelement is determined as follows:

Table 5–2: Tag for List Items

XMLITEMNAME Parameter of Property	XMLNAME Parameter of Property	Tag (Element or Attribute)
Specified	*Ignored*	Value of *XMLITEMNAME*
Not specified	Specified	Value of *XMLNAME* with `Item` concatenated to the end
	Not specified	If the list item corresponds to a data type property, the tag is the property name with `Item` concatenated to the end. If the list item corresponds to an object class, the tag is the short class name.

The same logic applies to the items of an array. The keys of an array are treated separately; see the next topic.

5.5 Controlling the Element and Attribute Names for Array-Type Properties

Note: This section does not apply to a collection property that specifies *XMLPROJECTION* as `"ELEMENT"`. For such a property, each array item is treated as a separate property of the class. See the section "Controlling the Projection for Collection Properties," earlier in this book.

In an XML-enabled object, an array-type property is projected to an element with subelements, each of which is one item in that array, in the same basic way that a list property is projected; see the previous section.

Each subelement has an additional attribute that indicates the key associated with the item. You can control the name of this attribute.

Consider the following example property:

```
Property Tools As %ArrayOfDataTypes;
```

For example, suppose that (for some object instance) this property consists of an array as follows:

- The value `Hammer` is stored with the key `845`.
- The value `Monkey wrench` is stored with the key `1009`.
- The value `Screwdriver` is stored with the key `3762`.

By default, this property corresponds to the following XML fragment:

```
<Tools>
  <ToolsItem ToolsKey="845">Hammer</ToolsItem>
  <ToolsItem ToolsKey="1009">Monkey Wrench</ToolsItem>
  <ToolsItem ToolsKey="3762">Screwdriver</ToolsItem>
</Tools>
```

This shows the `ToolsKey` attribute, which corresponds to the key of an array. The name of this attribute is determined as follows:

Table 5–3: Tag for List Item

XMLKEYNAME Parameter of Property	XMLNAME Parameter of Property	Name of Attribute That Contains the Key
Specified	*Ignored*	Value of *XMLKEYNAME*
Not specified	Specified	Value of *XMLNAME* with `Key` concatenated to the end
	Not specified	Property name with `Key` concatenated to the end

Notice that the *XMLITEMNAME* property parameter does not affect the attribute name; this parameter is discussed in the previous section.

For example, suppose you do not set *XMLKEYNAME* and you set *XMLNAME* equal to `MyXMLName`, as follows:

```
Property Tools As %ArrayOfDataTypes(XMLNAME = "MyXMLName");
```

Then the same property would correspond to the following XML fragment:

```
<MyXMLName>
  <MyXMLNameItem MyXMLNameKey="845">Hammer</MyXMLNameItem>
  <MyXMLNameItem MyXMLNameKey="1009">Monkey Wrench</MyXMLNameItem>
  <MyXMLNameItem MyXMLNameKey="3762">Screwdriver</MyXMLNameItem>
</MyXMLName>
```

6

Specifying Namespaces for Elements and Attributes

XML elements and attributes can belong to different namespaces, and the XML Schema specification provides for multiple ways of controlling and representing namespace assignment. The %XML.Adaptor class provides the corresponding support for your XML documents.

This section discusses the following topics:

- Overview
- How to specify the namespace for top-level objects
- How to specify the namespace for properties projected as elements
- How to specify the namespaces for properties projected as attributes
- How to specify custom prefixes for namespaces
- Recommendations

Also see "Specifying the Namespaces for Types," later in this book.

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *NAMESPACE* • *ELEMENTQUALIFIED* • *ATTRIBUTEQUALIFIED* • *XMLREF* • *REFNAMESPACE* • *XSDTYPE* • *XMLPREFIX*

6.1 Overview

This section provides a refresher on XML namespaces and an overview of how Caché objects are assigned to XML namespaces.

6.1.1 Namespace Refresher

The general assumption in this book is that the reader is familiar with XML. It may be worthwhile, however, to review how to determine the namespace, if any, to which an element or attribute is assigned in an XML document.

First, unless the XML document includes a default namespace or a namespace prefix for every element and attribute that it includes, it is necessary to see the corresponding XML schema. Apart from any imported elements or attributes, any element or attribute is one of the following:

- *Qualified*, which means that the element or attribute is in the target namespace of the schema.
- *Unqualified*, which has different meaning for elements and attributes. An unqualified element is in no namespace. An unqualified attribute is in the default namespace, if any, of its containing element.

For each element and attribute that it defines, a schema indicates whether that item is qualified or unqualified. The schema does this by a combination of the following pieces:

- The `<schema>` element can specify the `elementFormDefault` and `attributeFormDefault` attributes. This controls the default namespace assignment of any elements and attributes in the schema. The possible values are `"qualified"` and `"unqualified"`.

  These attributes are optional. The default for both of them is `"unqualified"`. That is, by default if a element or attribute is used without a prefix, it is in no namespace.
- Second, the definition of an element or attribute can specify the `form` attribute, which indicates how that item is assigned to a namespace. The possible values are `"qualified"` and `"unqualified"`.

Consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ClassA xmlns="mynamespace" xmlns:s01="mynamespace" s01:String1="abcdef">
  <s01:ClassB xmlns="">
    <String3>qrstuv</String3>
    <String4>wxyz</String4>
  </s01:ClassB>
  <String2>ghijkl</String2>
</ClassA>
```

For simplicity, we assume that the corresponding schema document uses the default values of `elementFormDefault` and `attributeFormDefault`, and does not specify the `form` attribute for any items it defines. Then the items in this document are in namespaces as follows:

- The `<ClassA>` element is in `mynamespace`, because of two items:
  - That is the namespace given by the default namespace declaration for this element and its immediate children (`xmlns="mynamespace"`).
  - The `<ClassA>` element does not have a namespace prefix that would indicate some other namespace.
- The `String1` attribute is in `mynamespace`, because this attribute uses the `s01` prefix, and the `xmlns:s01` namespace declaration indicates that the `s01` refers to the `mynamespace` namespace.

  Because the schema uses the default for `attributeFormDefault` (`"unqualified"`), the `String1` attribute would be in `mynamespace` even if it did not use a namespace prefix.

- The <ClassB> element is in mynamespace, because this attribute uses the s01 prefix.
- The <String3> and <String4> elements are not in any namespace, because of two items:
  - The namespace declaration for the parent element indicates that the default namespace here is null (xmlns="").
  - These elements do not have a namespace prefix that would indicate some other namespace.
- The element <String2> is the namespace mynamespace, because that is the default namespace specified in its parent element.

6.1.2 XML Namespaces and Classes

In Caché XML support, you specify namespaces on a class-by-class basis. You use the *NAMESPACE* class parameter to specify the namespace for instances of that class, as well as its immediate child objects. Also, you use the *ELEMENTQUALIFIED* and *ATTRIBUTEQUALIFIED* parameters to specify whether properties of its object-valued properties are global (and belong to the same namespace as the parent) or local.

Note: You can also specify *ELEMENTQUALIFIED* as a property parameter, if needed for unusual scenarios, not discussed in this book.

6.1.3 Namespaces and Context

Particularly with namespaces, it is important to remember that an XML-enabled object is handled differently depending on the context. For example, if you export an Address object at the top level, it is a global element. If you export a Person object that includes a reference to an Address object, then Address is a local element (as are all other properties of Person). Global and local elements are assigned to namespaces differently.

6.2 Specifying the Namespaces for Objects Treated as Global Elements

If you import or export an XML-enabled object at the top level, that object becomes a global element and is assigned to a namespace as follows:

- If the *NAMESPACE* parameter of the class is specified, the element is assigned to that namespace.
- If the *NAMESPACE* parameter of the class is not specified, the element does not belong to any namespace. You can, however, specify a namespace during export. See "Writing XML Output from Caché Objects" in *Using Caché XML Tools*.

For example, consider the following class definition:

```
Class MyApp.Person Extends (%Persistent, %XML.Adaptor)
{
Parameter NAMESPACE = "http://www.person.org";

Property Name As %Name [ Required ];

Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h") [ Required ];
}
```

If you export or import an object of this class, the projection might look as follows:

```
<Person xmlns="http://www.person.org">
  <Name>Isaacs,Rob G.</Name>
  <DOB>1981-01-29</DOB>
</Person>
```

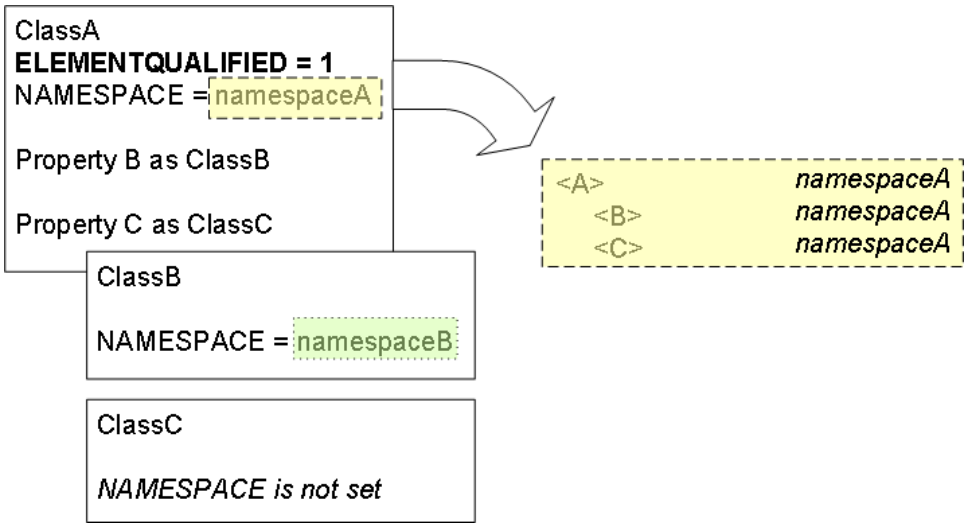
6.3 Specifying the Namespaces for Properties Projected as Elements

This section describes how to specify the namespace for a property that is projected as an element.

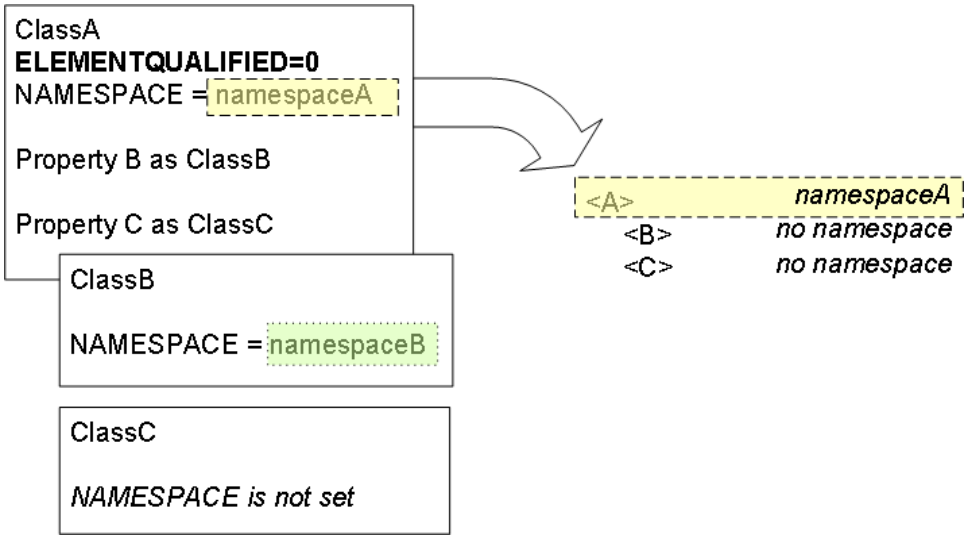
6.3.1 Case 1: Property Is Treated as Local Element

If you import or export an XML-enabled object at the top level, any property that is projected as an element becomes a local element by default. There are two possible namespace assignments for these local elements:

- If the *ELEMENTQUALIFIED* class parameter is 1 for the parent class, the local elements are qualified, and they are explicitly included in namespace of their parent element.



- If the *ELEMENTQUALIFIED* class parameter is 0 for the parent class, the local elements are unqualified, and they do not belong to any namespace. (You can, however, specify a namespace during export. See "Writing XML Output from Caché Objects" in *Using Caché XML Tools*.)



Notice that in both cases, the namespace in the child class is ignored.

Note: The default for *ELEMENTQUALIFIED* depends on whether the input or output is in literal format or encoded format. Literal format is the default and the most common.

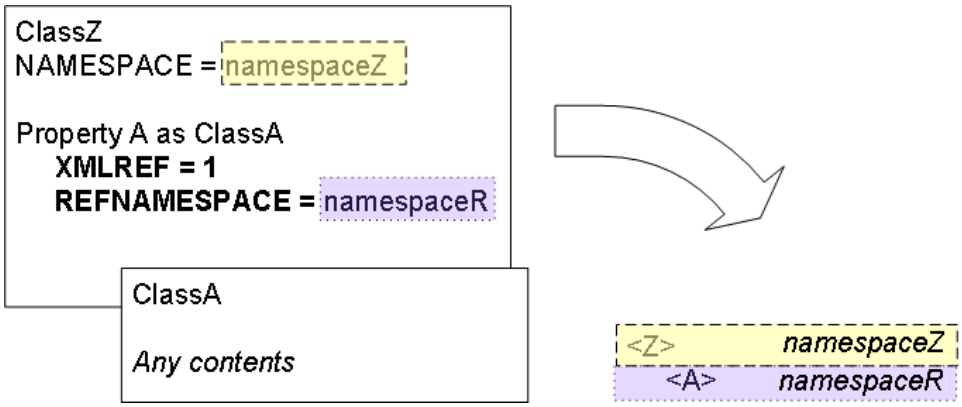
For literal format, *ELEMENTQUALIFIED* defaults to 1. For encoded format, *ELEMENTQUALIFIED* defaults to 0.

For information on these formats, see "Specifying Format Options for the XML Document," earlier in this book.

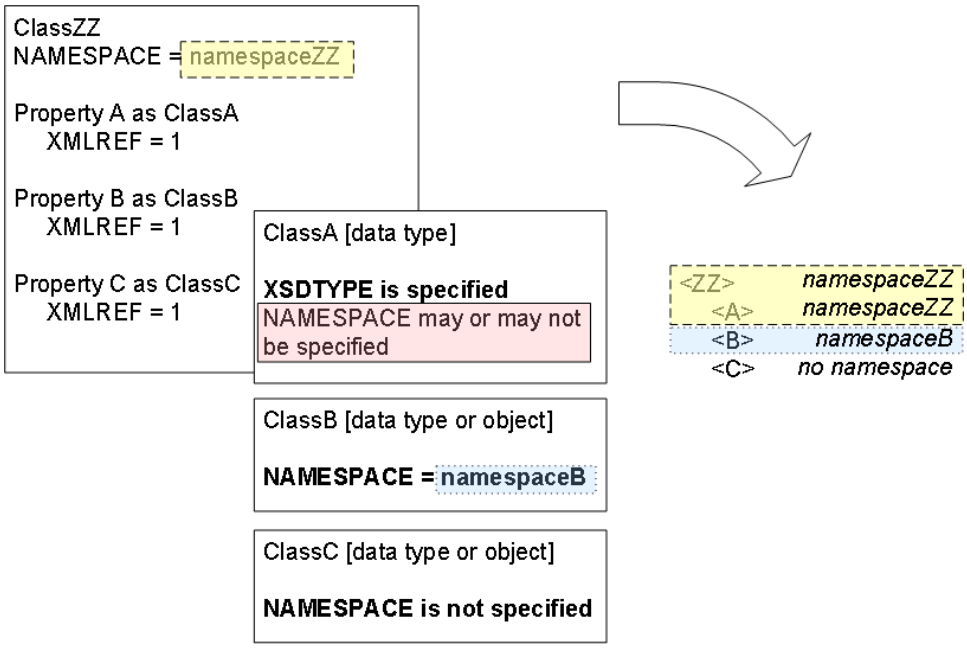
6.3.2 Case 2: Property Is Treated as Global Element

You can make a property into a global element and assign it to a namespace. To do so, you set the *XMLREF* property parameter to 1. The following describes how the corresponding element is assigned to a namespace:

1. If the *REFNAMESPACE* property parameter is specified, the element is in that namespace.



2. Otherwise the element is assigned to a namespace as follows:
   a. If the property is a type *and* specifies the *XSDTYPE* class parameter, the element is in the namespace of the parent class.
   b. Otherwise, if the property class defines the *NAMESPACE* class parameter, the element is in the namespace of the property class.
   c. If the property class does not define either of these class parameters, the element is in no namespace.



Note: The *XMLREF* property parameter replaces the *XMLELEMENTREF* property parameter. The *XMLELEMENTREF* parameter, however, will be supported indefinitely.

6.4 Specifying the Namespaces for Properties Projected as Attributes

This section describes how to specify the namespace for a property that is projected as an attribute. The *ATTRIBUTEQUALIFIED* parameter specifies whether attributes are qualified by a namespace prefix; the possible values are as follows:

- 0 (the default), which means that no namespace prefix is included.
- 1, which means that a namespace prefix is included.

The *XMLREF* and *REFNAMESPACE* property parameters are also supported for properties that you project as attributes.

For a property projected as an attribute, if you set the *XMLREF* property parameter to 1, the corresponding attribute is assigned to a namespace as follows:

1. If the *REFNAMESPACE* property parameter is specified, the attribute is in that namespace.
2. Otherwise:
    a. If the property is a type *and* specifies the *XSDTYPE* class parameter, the attribute is in the namespace of the parent class.
    b. Otherwise, if the property class defines the *NAMESPACE* class parameter, the attribute is in the namespace of the property class.
    c. If the property class does not define either of these class parameters, the attribute is in no namespace.

6.5 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 example, suppose your class definition is as follows:

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

Parameter XMLPREFIX = "p";

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

Parameter XMLNAME = "Person";

Property Name As %Name;

}
```

For this class, XML output looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person xmlns="http://www.person.com" xmlns:p="http://www.person.com">
  <Name>Umansky,Jocelyn O.</Name>
</Person>
```

For information on exporting to XML, see *Using Caché XML Tools*.

6.6 Recommendations

To simplify development, debugging, and troubleshooting, InterSystems recommends the following practices:

- If you need to specify namespaces, specify *NAMESPACE* in *all* your XML-enabled classes. Otherwise, the defaulting rules become complex.
- If you need to control whether local elements are qualified, also specify the *ELEMENTQUALIFIED* parameter in *all* your XML-enabled classes.

7

Controlling the Projection to XML Schemas

This chapter discusses how to control the projection of Caché classes to XML schemas. It discusses the following topics:

- How to view the schema for an XML-enabled class
- How data types are projected to XML types
- How streams are projected to XML types
- How collections are projected to XML types
- How other XML-enabled classes are projected to XML types and how to control that projection
- How to specify namespaces for XML types
- How to suppress the namespace prefix in the type QName

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *CONTENT* • *DISPLAYLIST* • *VALUELIST* • *ESCAPE* • *MAXLEN* • *MINLEN* • *MINVAL* • *XMLFractionDigits* • *XMLTotalDigits* • *XMLLISTPARAMETER* • *XSDTYPE* • *XMLTYPE* • *SUPPRESSTYPEPREFIX*

7.1 Viewing the Schema for an XML-Enabled Class

When you enable a Caché class for XML, you are implicitly creating an XML schema for that class, which you can view. To see the schema for a given XML-enabled class, you have two options:

- You can use %XML.Schema and %XML.Writer to generate complete schema documents. For details, see "Generating XML Schemas from Classes" in *Using XML in Caché*.
- You can use the **XMLSchema()** class method of your XML-enabled class, which writes the XML schema for this class to the current device. This method does not write the XML declaration and ignores namespaces and thus has limited use. This method can be helpful, however, if you are interested only in the XML types.

This chapter primarily uses the **XMLSchema()** class method, because using it requires only a single line of code.

7.1.1 Example

For example, consider the following class definitions:

```
Class GXML.Person Extends (%Persistent, %Populate, %XML.Adaptor)
{
Property Name As %Name;
Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h");
Property GroupID As %String (XMLPROJECTION="ATTRIBUTE");
Property OtherID As %String(XMLPROJECTION = "NONE");
Property Address As GXML.Address;
Property Doctors As list Of GXML.Doctor;
}
```

The `GXML.Address` class is as follows:

```
Class GXML.Address Extends (%Persistent, %Populate, %XML.Adaptor)
{
Property Street As %String;
Property City As %String;
Property State As %String(MAXLEN = 2, PATTERN = "2u");
Property Zip As %String(MAXLEN = 10, PATTERN = "5n.1(1""-""4n)");
}
```

And the `GXML.Doctor` class is as follows:

```
Class GXML.Doctor Extends (%Persistent, %Populate, %XML.Adaptor)
{
Property Name As %Name;
}
```

To see the schema for the `GXML.Person` class, enter the following command in the Terminal:

```
 do ##class(GXML.Person).XMLSchema()
```

You then see the following:

```
<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:element name="Doctors" type="ArrayOfDoctorDoctor" 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>
<s:complexType name="ArrayOfDoctorDoctor">
    <s:sequence>
```

```
        <s:element name="Doctor" type="Doctor"
minOccurs="0" maxOccurs="unbounded" nillable="true" />
    </s:sequence>
</s:complexType>
<s:complexType name="Doctor">
    <s:sequence>
        <s:element name="Name" type="s:string" minOccurs="0" />
    </s:sequence>
</s:complexType>
```

Notice the following:

- The schemas for the `<Person>`, `<Address>`, and `<Doctor>` types are based directly on the corresponding class definitions.
- The schema consists of only the properties that are projected.
- The schema recognizes whether each property is projected as an element or as an attribute. For example, `GroupID` is an attribute and `Name` is an element.
- Other parameters of the properties can affect the schema.
- In this example, the class properties are of type string, which is one of the basic XSD types (see http://www.w3.org/TR/xmlschema-2/).

7.2 Projection of Literal Properties to XML Schemas

This section discusses how literal (non-collection) properties are projected to XML types, as well as options that affect the XML schema. It discusses the following:

- Default XSD types for data type classes
- Compiler keywords that affect the schema
- Parameters that affect the schema

7.2.1 Default XSD Types for Caché Data Type Classes

If a class or a class property is based on one of the common Caché data type classes, the XML type is set automatically, according to the following table. Classes in the %xsd package map directly to the XML types, as shown in the table.

Table 7–1: XML Types for Caché Data Types in the %Library and %xsd Packages

Caché Class in the %xsd Package	Caché Class in the %Library Package	XSD Type Used in Projections to XML
%xsd.anyURI		anyURI
%xsd.base64Binary	%Binary %Status	base64Binary
%xsd.boolean	%Boolean	boolean
%xsd.byte	%TinyInt	byte
%xsd.date	%Date %FilemanDate	date

Caché Class in the %xsd Package	Caché Class in the %Library Package	XSD Type Used in Projections to XML
%xsd.dateTime	%StringTimeStamp %TimeStamp %FilemanTimeStamp	dateTime
%xsd.decimal	%Currency %Decimal %Numeric	decimal
%xsd.double	%Double %Float	double
%xsd.float		float
%xsd.hexBinary		hexBinary
%xsd.int		int
%xsd.integer		integer
%xsd.long	%BigInt %Integer	long
%xsd.negativeInteger		negativeInteger
%xsd.nonNegativeInteger		nonNegativeInteger
%xsd.nonPositiveInteger		nonPositiveInteger
%xsd.positiveInteger		positiveInteger
%xsd.short	%SmallInt	short
%xsd.string	%Name %String %Text %List	string
%xsd.time	%Time	time
%xsd.unsignedByte		unsignedByte
%xsd.unsignedInt		unsignedInt
%xsd.unsignedLong		unsignedLong
%xsd.unsignedShort		unsignedShort

For information on the XML data types, see http://www.w3.org/TR/xmlschema-2/.

For example, consider the following class:

```
Class Schema.DataTypesDemo Extends (%RegisteredObject, %XML.Adaptor)
{
```

```
Parameter XMLTYPENAMESPACE="mytypes";

Property binaryprop As %xsd.base64Binary;

Property booleanprop As %Boolean;

Property dateprop As %Date;

Property datetimeprop As %TimeStamp;

Property decimalprop As %Numeric;

Property integerprop As %Integer;

Property stringprop As %String;

Property timeprop As %Time;

}
```

The schema for this class 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"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="DataTypesDemo">
    <sequence>
      <element minOccurs="0" name="binaryprop" type="s:base64Binary"/>
      <element minOccurs="0" name="booleanprop" type="s:boolean"/>
      <element minOccurs="0" name="dateprop" type="s:date"/>
      <element minOccurs="0" name="datetimeprop" type="s:dateTime"/>
      <element minOccurs="0" name="decimalprop" type="s:decimal"/>
      <element minOccurs="0" name="integerprop" type="s:long"/>
      <element minOccurs="0" name="stringprop" type="s:string"/>
      <element minOccurs="0" name="timeprop" type="s:time"/>
    </sequence>
  </complexType>
</schema>
```

7.2.2 Compiler Keywords That Affect the Schema

The Required keyword affects the XML schema, by removing the `minOccurs="0"` attribute. For example, consider the following class:

```
Class Schema.PropKeywords Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE="mytypes";

Property Property1 As %String;

Property Property2 As %String [ Required ];

}
```

If we generate a schema for the namespace used here, we see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="test">
  <complexType name="PropKeywords">
    <sequence>
      <element minOccurs="0" name="Property1" type="s:string"/>
      <element name="Property2" type="s:string"/>
    </sequence>
  </complexType>
</schema>
```

Note that the default for `minOccurs` is 1; that is, `Property2` is required.

Note: For compatibility reasons, %XML.Reader does not check for required properties by default, but you can cause it to do so; see "Checking for Required Elements and Attributes" in *Using Caché XML Tools*. Also by default, a Caché web service does not check for required properties, but you can cause it to do so; see "Checking for Required Elements and Attributes" in *Creating Web Services and Web Clients in Caché*.

No other property keywords affect the schema for data type classes.

7.2.3 Parameters That Affect XML Schemas

The Caché data type classes use many parameters. (For a table that lists the parameters supported in each data type class, see "Data Types" in *Using Caché Objects*.) In most cases, you can also specify these as property parameters.

The parameters that affect XML schemas are as follows:

CONTENT

Influences how the property values are escaped; see "Handling Special XML Characters," earlier in this book.

The `"MIXED"` value causes a change in the schema, as compared to the other possible values. Consider the following class:

```
Class Schema.CONTENT Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property Property1 As %String;

Property Property2 As %String(CONTENT = "STRING");

Property Property3 As %String(CONTENT = "ESCAPE");

Property Property4 As %String(CONTENT = "MIXED");

}
```

If we generate a schema for the namespace used here, we see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="CONTENT">
    <sequence>
      <element minOccurs="0" name="Property1" type="s:string"/>
      <element minOccurs="0" name="Property2" type="s:string"/>
      <element minOccurs="0" name="Property3" type="s:string"/>
      <element name="Property4">
        <complexType mixed="true">
          <choice maxOccurs="unbounded" minOccurs="0">
            <any processContents="lax"/>
          </choice>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

Notice that the three of these properties have the same type information, because XML treats them in the same way. Caché, however, treats the properties differently as described in "Handling Special XML Characters," later in this book.

If you use the object as input or output for a web method, and SoapBodyUse is `encoded` for that method, then Caché treats mixed content like string content, the default. That is, if you specify *CONTENT* as `"MIXED"`, that value is ignored.

DISPLAYLIST

Affects the schema if *VALUELIST* is also specified and if *XMLLISTPARAMETER* equal to `"DISPLAYLIST"`. See the discussions for those two parameters.

MAXLEN

Controls the `maxLength` attribute, which is a *facet* or *restriction*. Facets define acceptable values for XML types. The following example shows several of them. Consider the following class:

```
Class Schema.BasicFacets Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property Property1 As %Integer (MINVAL=10, MAXVAL=1000);

Property Property2 As %String (MINLEN=20, MAXLEN=100);

}
```

If we generate a schema for the namespace used here, we see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:s="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="BasicFacets">
    <sequence>
      <element minOccurs="0" name="Property1">
        <simpleType>
          <restriction base="s:long">
            <maxInclusive value="1000"/>
            <minInclusive value="10"/>
          </restriction>
        </simpleType>
      </element>
      <element minOccurs="0" name="Property2">
        <simpleType>
          <restriction base="s:string">
            <maxLength value="100"/>
            <minLength value="20"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
</schema>
```

When the SOAP Wizard or the XML Schema Wizard finds a `maxLength` restriction in a schema, it sets the *MAXLEN* property parameter as appropriate in the generated class.

MAXVAL

Controls the `maxInclusive` attribute. See the example in *MAXLEN*.

MINLEN

Controls the `minLength` attribute. See the example in *MAXLEN*.

When the SOAP Wizard or the XML Schema Wizard finds a `minLength` restriction in a schema, it sets the *MINLEN* property parameter as appropriate in the generated class.

MINVAL

Controls the `minInclusive` attribute. See the example in *MAXLEN*.

VALUELLIST

Adds an `<enumeration>` restriction to the type. Consider the following class:

```
Class Schema.VALUELIST Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property Property1 As %String;

Property Property2 As %String (VALUELIST = ",r,g,b");

}
```

The following shows the schema for this class:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="VALUELIST">
    <sequence>
      <element minOccurs="0" name="Property1" type="s:string"/>
      <element minOccurs="0" name="Property2">
        <simpleType>
          <restriction base="s:string">
            <enumeration value="r"/>
            <enumeration value="g"/>
            <enumeration value="b"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
</schema>
```

XMLFractionDigits

Applicable to a %Numeric property. This parameter corresponds to the <fractionDigits> facet, as shown in the following fragment:

```
<element minOccurs="0" name="Property2">
  <simpleType>
    <restriction base="s:decimal">
      <fractionDigits value="2"/>
      <totalDigits value="5"/>
    </restriction>
  </simpleType>
</element>
```

XMLTotalDigits

Applicable to a %Numeric property or an %Integer property. This parameter corresponds to the <totalDigits> facet, as shown in the following fragment:

```
<element minOccurs="0" name="Property2">
  <simpleType>
    <restriction base="s:decimal">
      <fractionDigits value="2"/>
      <totalDigits value="5"/>
    </restriction>
  </simpleType>
</element>
```

XMLLISTPARAMETER

Applicable to a %String property that specifies the *VALUELIST* parameter. Specifies the name of the parameter that contains the list of values to project to XML, instead of the values contained in the object. In most cases, you also specify the standard *DISPLAYLIST* parameter, and you set *XMLLISTPARAMETER* equal to `"DISPLAYLIST"`.

The *XMLLISTPARAMETER* parameter controls the `value` attribute used in the `<enumeration>` restriction.

You cannot specify this as a property parameter.

XMLPATTERN

Controls the `pattern` restriction. Consider the following class:

```
Class Schema.Pattern Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE = "mytypes";

Property Property1 As %String;

Property Property2 As %String(XMLPATTERN = "[A-Z]");

}
```

The schema for this class 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"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="Pattern">
    <sequence>
      <element minOccurs="0" name="Property1" type="s:string"/>
      <element minOccurs="0" name="Property2">
        <simpleType>
          <restriction base="s:string">
            <pattern value="[A-Z]"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
</schema>
```

If multiple patterns appear in a simple type, then Caché combines the patterns according to https://www.w3.org/TR/xmlschema-2 (see section 4.3.4.3, “Constraints on XML Representation of pattern”). The patterns are combined as separate branches in the same pattern (separated by a vertical bar) in the *XMLPATTERN* parameter.

XSDTYPE

Declares the XSD type used when projecting to XML. This parameter is set appropriately in all Caché data type classes. The Caché XML tools use this parameter when generating schemas. This parameter does not directly affect the input and output transformations, although it should be consistent with them.

7.3 Projection of Stream Classes to XML Types

If a class or a property is based on a Caché stream, it is projected to an XML type as shown in the following table:

Table 7–2: XML Types for Caché Streams

Caché Stream Type	XSD Type Used in Projections to XML
%Library.GlobalCharacterStream, %Library.FileCharacterStream, %Stream.FileCharacter, and %Stream.GlobalCharacter	string
%Library.GlobalBinaryStream, %Library.FileBinaryStream, %Stream.FileBinary, and %Stream.GlobalBinary	base64Binary

For example, consider the following class:

```
Class Schema.StreamPropDemo Extends (%Persistent, %XML.Adaptor)
{

Parameter XMLTYPENAMESPACE="mytypes";

Property BinStream As %Library.GlobalBinaryStream;

Property CharStream As %Library.GlobalCharacterStream;

}
```

The schema for this class 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"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="StreamPropDemo">
    <sequence>
      <element minOccurs="0" name="BinStream" type="s:base64Binary"/>
      <element minOccurs="0" name="CharStream" type="s:string"/>
    </sequence>
  </complexType>
</schema>
```

7.4 Projection of Collection Properties to XML Schemas

This section describes how collection properties are projected to XML schemas, for XML-enabled classes. This section discusses the following:

- Projection of collection properties
- Options for using collection classes

7.4.1 Projection of Collection Properties to XML Schemas

For most kinds of properties, the class definition contains enough information to specify the complete XML projection — both to project objects as XML documents and to define a complete XML schema for validation purposes. For collection properties, however, Caché supports some forms of definitions that do not provide enough information for a complete XML schema. If you are using the XML projections in a context where the schema is needed (such as in web services and clients), it is necessary to have a complete XML schema; otherwise validation against the schema will fail. If you are not validating against a schema, this consideration does not apply. The following table lists the scenarios:

Table 7–3: Forms of Collection Properties and Their XML Projection Details

Form of Property Definition	XML Is Usable?	XML Schema Is Usable?
`Property PropName As List of classname` or `Property PropName As Array of classname`	Yes	Yes
`Property PropName As %ListOfDataTypes` or `Property PropName As %ArrayOfDataTypes`	Yes	Yes (but the default type for the collection item is string, which might not be appropriate)
`Property PropName As %ListOfObjects` or `Property PropName As %ArrayOfObjects`	Yes	No (the schema does not specify the type for the collection item)

The following subsections show the XML schemas for these scenarios.

7.4.1.1 List of Classname

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as `List of Classname`. For example, consider the following property definition:

```
Property
PropName As
list
Of
%Integer(XMLITEMNAME
= "MyXmlItemName");
```

If this property is in an XML-enabled class named Test.DemoList1, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="DemoList1">
    <sequence>
     <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNameLong" xmlns:s01="mytypes"/>

    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNameLong">
    <sequence>
     <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true" type="s:long"/>

    </sequence>
  </complexType>
...
</schema>
```

The following rules govern the names of the types:

- For the *PropName* property, the corresponding type is named `ArrayOf`*XMLItemNameType*, where:
  - *XMLItemName* is the name of items in the collection as described as in "Controlling the Element and Attribute Names for List-Type Properties." For a data type property, the default item name is the property name with `Item` appended to the end. (For a object property, the default item name is the short class name.)
  - *Type* is the XML type to which the property class is projected.

  ```
  <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNameLong" xmlns:s01="mytypes"/>
  ```

  Note: If *XMLItemName* is identical to *Type*, then for the *PropName* property, the corresponding type is named `ArrayOf`*XMLItemName*. That is, the redundant array item is removed from the type name. To cause the type name to include the redundant name, specify the AllowRedundantArrayName property (of your instance of %XML.Schema) as 1. Similarly, in a web service class, to include the redundant array item name in the type in the WSDL, specify the *ALLOWREDUNDANTARRAYNAME* parameter (of the web service class) as 1.

- The type `ArrayOf`*XMLItemNameType* is defined as a <sequence> of another type, named *XMLItemName*:

  ```
    <complexType name="ArrayOfMyXmlItemNameLong">
      <sequence>
        <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
  type="s:long"/>
      </sequence>
    </complexType>
  ```

- The element *XMLItemName* is based on the XSD type corresponding to the data type class:

  ```
  <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true" type="s:long"/>
  ```

The same rules apply when *Classname* refers to an object class. For example, consider the following property definition:

```
Property
PropName As
list
Of
SimpleObject(XMLITEMNAME
= "MyXmlItemName");
```

Where Simple.Object contains two properties, MyProp and AnotherProp. If this property is in an XML-enabled class named Test.DemoObjList, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="DemoObjList">
    <sequence>
      <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNameSimpleObject"
xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNameSimpleObject">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:SimpleObject" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="SimpleObject">
    <sequence>
      <element minOccurs="0" name="MyProp" type="s:string"/>
      <element minOccurs="0" name="AnotherProp" type="s:string"/>
    </sequence>
  </complexType>
...
</schema>
```

7.4.1.2 Array of Classname

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as `Array of Classname`. For example, consider the following property definition:

```
Property
PropName As
array
Of
%Integer(XMLITEMNAME
= "MyXmlItemName", XMLKEYNAME
= "MyXmlKeyName");
```

If this property is in an XML-enabled class named Test.DemoArray1, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="DemoArray1">
    <sequence>
      <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNamePairOfMyXmlKeyNameLong"
xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNamePairOfMyXmlKeyNameLong">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:PairOfMyXmlKeyNameLong" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="PairOfMyXmlKeyNameLong">
    <simpleContent>
      <extension base="s:long">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </simpleContent>
  </complexType>
...
</schema>
```

The following rules govern the names of the types:

- For the *PropName* property, the corresponding type is named `ArrayOf`*XMLItemName*`PairOf`*XMLKeyNameType*, where:

- *XMLItemName* is the name of items in the collection as described as in "Controlling the Element and Attribute Names for Array-Type Properties." For a data type property, the default item name is the property name with `Item` appended to the end. (For a object property, the default item name is the short class name.)
- *XMLKeyName* is the name of the key for the collection as described in "Controlling the Element and Attribute Names for Array-Type Properties." The default is the property name with `Key` concatenated to the end
- *Type* is the XML type to which the property class is projected.

```
<element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNamePairOfMyXmlKeyNameLong"
xmlns:s01="mytypes"/>
```

Note: If *XMLKeyName* is identical to *Type*, then for the *PropName* property, the corresponding type is named `ArrayOf`*XMLItemName*`PairOf`*XMLKeyName*. That is, the redundant array item is removed from the type name. To cause the type name to include the redundant name, specify the AllowRedundantArrayName property (of your instance of %XML.Schema) as 1. Similarly, in a web service class, to include the redundant array item name in the type in the WSDL, specify the *ALLOWREDUNDANTARRAYNAME* parameter (of the web service class) as 1.

- The type `ArrayOf`*XMLItemName*`PairOf`*XMLKeyNameType* is defined as a <sequence> of another type, named `PairOf`*XMLKeyNameType*:

```
 <complexType name="ArrayOfMyXmlItemNamePairOfMyXmlKeyNameLong">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:PairOfMyXmlKeyNameLong" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
```

- The type `PairOf`*XMLKeyNameType* is an extension of the given XSD type. This extension adds an attribute named *XMLKeyName*:

```
  <complexType name="PairOfMyXmlKeyNameLong">
    <simpleContent>
      <extension base="s:long">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </simpleContent>
  </complexType>
```

The same rules apply when *Classname* refers to an object class. For example, consider the following property definition:

```
Property
PropName As
%ArrayOfObjects(XMLITEMNAME
= "MyXmlItemName", XMLKEYNAME
= "MyXmlKeyName");
```

Where Simple.Object contains two properties, MyProp and AnotherProp. If this property is in an XML-enabled class named Test.DemoObjArray, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="DemoObjArray">
    <sequence>
     <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNamePairOfMyXmlKeyNameSimpleObject"
 xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNamePairOfMyXmlKeyNameSimpleObject">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:PairOfMyXmlKeyNameSimpleObject" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="PairOfMyXmlKeyNameSimpleObject">
    <complexContent>
      <extension base="s01:SimpleObject" xmlns:s01="mytypes">
```

```
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="SimpleObject">
    <sequence>
      <element minOccurs="0" name="MyProp" type="s:string"/>
      <element minOccurs="0" name="AnotherProp" type="s:string"/>
    </sequence>
  </complexType>
</schema>
```

7.4.1.3 %ListOfDataTypes

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as %ListOfDataTypes. For example, consider the following property definition:

```
Property
PropName As
%ListOfDataTypes(XMLITEMNAME
= "MyXmlItemName");
```

If this property is in an XML-enabled class named Test.DemoList, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="DemoList">
    <sequence>
     <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNameString" xmlns:s01="mytypes"/>

    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNameString">
    <sequence>
     <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true" type="s:string"/>

    </sequence>
  </complexType>
</schema>
```

For the rules for the names of the types, see "List of Classname," earlier in this section. Note that the collection item (`PropNameItem` in this example) is based on the XSD string type:

```
<element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true" type="s:string"/>
```

That is, the collection item is assumed to be a string. Also see "Options for Using Collection Classes," later in this chapter.

7.4.1.4 %ArrayOfDataTypes

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as %ArrayOfDataTypes. For example, consider the following property definition:

```
Property
PropName As
%ArrayOfDataTypes(XMLITEMNAME
= "MyXmlItemName", XMLKEYNAME
= "MyXmlKeyName");
```

If this property is in an XML-enabled class named Test.DemoArray, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="DemoArray">
    <sequence>
      <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNamePairOfMyXmlKeyNameString"
xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNamePairOfMyXmlKeyNameString">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:PairOfMyXmlKeyNameString" xmlns:s01="mytypes"/>
    </sequence>
```

```
  </complexType>
  <complexType name="PairOfMyXmlKeyNameString">
    <simpleContent>
      <extension base="s:string">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </simpleContent>
  </complexType>
...
</schema>
```

For the rules for the names of the types, see "Array of Classname," earlier in this section. Note that the collection item (`PairOfMyXmlKeyNameString` in this example) is based on the XSD string type:

```
  <complexType name="PairOfMyXmlKeyNameString">
    <simpleContent>
      <extension base="s:string">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </simpleContent>
  </complexType>
```

That is, the collection item is assumed to be a string. Also see "Options for Using Collection Classes," later in this chapter.

7.4.1.5 %ListOfObjects

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as %ListOfObjects. For example, consider the following property definition:

```
Property
PropName As
list
Of
%Integer(XMLITEMNAME
= "MyXmlItemName");
```

If this property is in an XML-enabled class named Test.DemoObjList1, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="mytypes">
  <complexType name="DemoObjList1">
    <sequence>
      <element minOccurs="0" name="PropName" type="s01:ArrayOfMyXmlItemNameRegisteredObject"
xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNameRegisteredObject">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:RegisteredObject" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
...
</schema>
```

For the rules for the names of the types, see "List of Classname," earlier in this section. Note that the collection item type is `RegisteredObject`, which is not defined:

```
<element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:RegisteredObject" xmlns:s01="mytypes"/>
```

As a result, this schema is unusable. See "Options for Using Collection Classes," later in this chapter.

7.4.1.6 %ArrayOfObjects

This section shows the part of an XML schema that is generated from an XML-enabled class, when that class includes a property that is defined as %ArrayOfObjects. For example, consider the following property definition:

```
Property
PropName As
%ArrayOfObjects(XMLITEMNAME
= "MyXmlItemName", XMLKEYNAME
= "MyXmlKeyName");
```

If this property is in an XML-enabled class named Test.DemoObjArray1, the XML schema for this class contains the following:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:s="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="mytypes">
  <complexType name="DemoObjArray1">
    <sequence>
      <element minOccurs="0" name="PropName"
type="s01:ArrayOfMyXmlItemNamePairOfMyXmlKeyNameRegisteredObject" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="ArrayOfMyXmlItemNamePairOfMyXmlKeyNameRegisteredObject">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="MyXmlItemName" nillable="true"
type="s01:PairOfMyXmlKeyNameRegisteredObject" xmlns:s01="mytypes"/>
    </sequence>
  </complexType>
  <complexType name="PairOfMyXmlKeyNameRegisteredObject">
    <complexContent>
      <extension base="s01:RegisteredObject" xmlns:s01="mytypes">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </complexContent>
  </complexType>
...
</schema>
```

For the rules for the names of the types, see "List of Classname," earlier in this section. Note that the collection item type is based on `RegisteredObject`, which is not defined:

```
  <complexType name="PairOfMyXmlKeyNameRegisteredObject">
    <complexContent>
      <extension base="s01:RegisteredObject" xmlns:s01="mytypes">
        <attribute name="MyXmlKeyName" type="s:string" use="required"/>
      </extension>
    </complexContent>
  </complexType>
```

As a result, this schema is unusable. See "Options for Using Collection Classes," later in this chapter.

7.4.2 Options for Using Collection Classes

Within an XML-enabled class, tor each property of type %ListOfDataTypes or %ArrayOfDataTypes, the collection item is assumed to be a string; this assumption may or may not be suitable for your needs. Similarly, for each property of type %ListOfObjects or %ArrayOfObjects, the collection item type is `RegisteredObject`, and Caché does not include an XML projection for the type `RegisteredObject`, so the XML schema is not usable. (See the previous subsections for examples.)

In these scenarios, you can do either of the following:

- Modify the property definition to have the form `List of Classname` or `Array of Classname`, where *Classname* is a suitable class. If *Classname* is an object class, XML-enable the class.
- Create a custom subclass of the collection class (%ListOfDataTypes, %ArrayOfDataTypes, %ListOfObjects, %ArrayOfObjects). In the subclass, specify the *ELEMENTTYPE* class parameter. For example:

  ```
  Class MyApp.MyIntegerCollection Extends %ListOfDataTypes
  {
  Parameter ELEMENTTYPE="%Library.Integer";

  }
  ```

  For *ELEMENTTYPE*, specify the complete package and class name of the class used in the collection. If you subclass %ListOfDataTypes or %ArrayOfDataTypes, specify a data type class. Then the type for the collection element is controlled by the *XSDTYPE* parameter for that class.

  If you subclass %ListOfObjects or %ArrayOfObjects, specify an XML-enabled class. For example

```
Class MyApp.MyObjectCollection Extends %ListOfObjects
{
Parameter ELEMENTTYPE="MyApp.SimpleObject";

}
```

Then use your custom collection class in your property definition. For example:

```
Property MyProp as MyApp.MyIntegerCollection;
```

7.5 Projection of Other XML-Enabled Classes to XML Types

For an XML-enabled class or a property that is based on an XML-enabled class, the XML type is determined as follows: If the class has a value for the *XMLTYPE* parameter, that is used as the type name. Otherwise, the short class name is taken as the XML type name.

For example, consider the following class definitions:

```
Class GXML.PersonWithAddress Extends (%Persistent, %XML.Adaptor)
{
Parameter XMLTYPE = "PersonType";

Property Name As %Name;

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

Property HomeAddress As GXML.Address;

}
```

For an instance of this class, the XML type is `PersonType`, which is taken from the *XMLTYPE* parameter.

Suppose that the `GXML.Address` class does not include the *XMLTYPE* parameter. In this case, for the `<HomeAddress>` element, the XML type is `Address`, which is the short class name.

7.6 Specifying the Namespaces for Types

The previous sections discussed how to assign elements to namespaces. This section discusses the namespaces of the types.

XML types are assigned to namespaces as follows:

1. If the corresponding class definition defines the *XSDTYPE* class parameter, the type is in the following W3 namespace:

   ```
   http://www.w3.org/2001/XMLSchema
   ```

   You specify *XSDTYPE* only within data type classes.

   Note: A data type class does not inherit the *XSDTYPE* class parameter. That is, if you subclass an existing data type class, you must specify this parameter, if the class should be mapped to one of the XSD types.

2. If the class definition does not define *XSDTYPE* but does define *NAMESPACE*, the type is in the namespace specified by *NAMESPACE*.
3. Otherwise the type is not in any namespace.

You can, however, specify a namespace when you generate a schema. See "Generating XML Schemas from Classes" in *Using XML in Caché*.

To see the namespaces to which the types are assigned, you must use %XML.Schema and %XML.Writer. For details, see "Generating XML Schemas from Classes" in *Using XML in Caché*.

7.7 Suppressing the Namespace Prefix for the Type QName

As described in *Using Caché XML Tools*, when you generate output with %XML.Writer, you can include the XML type attribute; to do so, you specify the writer's OutputTypeAttribute property as 1.

By default, the type attribute is written as a QName (qualified name), which indicates both the name of the type as well as the namespace to which the type belongs. For example:

```
<TeamA xmlns:s01="http://mynamespace" xsi:type="s01:TeamA">
```

You can define the corresponding Caché class definition so that the namespace prefix is suppressed. For example:

```
<TeamB xsi:type="TeamB">
```

For example, consider the following class definition:

```
Class STP.TeamA Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter NAMESPACE = "http://mynamespace";

Property Member1 as %String;

Property Member2 as %String;

}
```

The class STP.TeamB, which is not shown, has the same definition but also specifies *SUPPRESSTYPEPREFIX* as 1.

Both classes are used as properties in a third class:

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

Parameter NAMESPACE = "http://mynamespace";

Property TeamA As STP.TeamA;

Property TeamB As STP.TeamB;

}
```

When we generate output for an instance of STP.Container (and we enable output of the XML type attribute), we see something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Container xmlns="http://mynamespace"
           xmlns:s="http://www.w3.org/2001/XMLSchema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <TeamA xmlns:s01="http://mynamespace" xsi:type="s01:TeamA">
    <Member1 xsi:type="s:string">Jack O'Neill</Member1>
    <Member2 xsi:type="s:string">Samantha Carter</Member2>
  </TeamA>
  <TeamB xsi:type="TeamB">
    <Member1 xsi:type="s:string">Jasper O'Nelson</Member1>
    <Member2 xsi:type="s:string">Sandra Chartres</Member2>
  </TeamB>
</Container>
```

Notice that the `<TeamA>` element includes the `xsi:type` attribute, which equals `"s01:TeamA"`. The namespace declaration in this element indicates that the `s01` prefix refers to the namespace `http://mynamespace`.

The `<TeamB>` element, however, does not include a prefix within the `xsi:type` attribute.

Note: The *SUPPRESSTYPEPREFIX* does not affect the namespace to which the XML type belongs. It just suppresses the writing of the type prefix.

8

Advanced Options for XML Schemas

This chapter discusses advanced options for creating XML schemas. It discusses the following topics:

- How XML types are automatically created for subclasses and how you can use them
- How to use subtypes to create a choice list for a type
- How to use subtypes to create a substitution group for a type
- How superclasses of a projected class are represented as XML types
- How Caché projects a class with multiple XML-enabled superclasses

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *XMLTYPECONSTRAINT* • *XMLINCLUDEINLIST* • *XMLINHERITANCE*

8.1 Automatic Creation of Types for Subclasses

When you define the XML projection for a class, all its subclasses are automatically mapped to separate types, all of which use the superclass as the base type. This means that wherever the supertype is used, you could instead use one of the subtypes. You can also use the subtypes to define either choice lists or substitution groups in the XML schema.

Note that you can define the XML projection for an abstract class; the class appears as the base type in any derived class schema even though, being abstract, it cannot be instantiated.

Consider an example. We start with a simple `Person` class:

```
Class UsingSubclasses.Person Extends (%Persistent, %XML.Adaptor)
{
Property Name As %String [Required];
Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h") [Required];
}
```

Suppose that we have two classes that are based directly on the `Person` class. First is the `Patient` class:

```
Class UsingSubclasses.Patient Extends UsingSubclasses.Person
{
Property PatientID As %String [Required];
}
```

Next is the `Employee` class:

```
Class UsingSubclasses.Employee Extends UsingSubclasses.Person
{
Property EmployeeID As %String [Required];
}
```

Finally, consider a class that uses `Person` as a property:

```
Class UsingSubclasses.Example1 Extends (%Persistent, %XML.Adaptor)
{
Property Person As UsingSubclasses.Person;
}
```

When you generate the schema for the `Example1` class, the result is as follows:

```
<s:complexType name="Example1">
    <s:sequence>
        <s:element name="Person" type="Person" minOccurs="0" />
    </s:sequence>
</s:complexType>
<s:complexType name="Employee">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="EmployeeID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Person">
    <s:sequence>
        <s:element name="Name" type="s:string" />
        <s:element name="DOB" type="s:date" />
    </s:sequence>
</s:complexType>
<s:complexType name="Patient">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="PatientID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
```

Notice the following:

- The type for `Example1` is `Person`.
- The `Employee` type and the `Patient` type are both defined; this is the default. (For reference, this corresponds to setting the *XMLTYPECONSTRAINT* property parameter equal to `"EXPLICIT"`.)
- According to the XML Schema specification, the preceding schema means that wherever a `<Person>` element is included, you could include either an `<Employee>` element or a `<Patient>` element.

8.2 Creating a Choice List of Subtypes

According to the XML Schema specification, a complex type can consist of a choice list of types, particularly related types. Suppose that instead of a `<Person>` element, we want the schema to permit a `<Person>`, `<Patient>`, or `<Employee>` element. To define such a schema, we would set the *XMLTYPECONSTRAINT* property parameter equal to `"CHOICE"` for the `Person` property, as follows:

```
Class UsingSubclasses.Example2 Extends (%Persistent, %XML.Adaptor)
{
Property Person As UsingSubclasses.Person(XMLTYPECONSTRAINT = "CHOICE");
}
```

By default, the choice list consists of all subclasses of the `Person` class. The schema for `Example2` is as follows:

```
<s:complexType name="Example2">
    <s:sequence>
        <s:choice minOccurs="0">
            <s:element name="Employee" type="Employee" />
            <s:element name="Patient" type="Patient" />
            <s:element name="Person" type="Person" />
        </s:choice>
    </s:sequence>
</s:complexType>
<s:complexType name="Employee">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="EmployeeID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Person">
    <s:sequence>
        <s:element name="Name" type="s:string" />
        <s:element name="DOB" type="s:date" />
    </s:sequence>
</s:complexType>
<s:complexType name="Patient">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="PatientID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
```

In contrast to the previous example, the type for `Example2` is a choice list that consists of `Person`, `Patient`, or `Employee`. The latter three types are defined in the same way as in the previous example.

8.2.1 Restricting a Subclass from the Choice List

You might not want all subclasses to be included in the choice list. You can restrict the list of subclasses in two different ways.

- You can use the *XMLINCLUDEINGROUP* class parameter to mark a given subclass so that it is not included in the choice list.
- You can set the *XMLCHOICELIST* property parameter equal to a comma-separated list of the subclasses in the choice list.
- This parameter takes precedence; that is, if a subclass is listed in *XMLCHOICELIST*, it is included in the choice list, even if the subclass is marked as *XMLINCLUDEINGROUP* = 0.

The following sections show examples.

8.2.2 Example for Choice List With Explicit List

Suppose that we set the *XMLCHOICELIST* property parameter equal to a comma-separated list of the subclasses that we want to include in the choice list; for example:

```
Class UsingSubclasses.Example2A Extends (%Persistent, %XML.Adaptor)
{
Property Person As UsingSubclasses.Person
(XMLCHOICELIST = "UsingSubclasses.Patient, UsingSubclasses.Employee",
XMLTYPECONSTRAINT = "CHOICE");
}
```

The schema for this class would be as follows:

```
<s:complexType name="Example2A">
    <s:sequence>
        <s:choice minOccurs="0">
            <s:element name="Patient" type="Patient" />
            <s:element name="Employee" type="Employee" />
        </s:choice>
    </s:sequence>
</s:complexType>
<s:complexType name="Patient">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="PatientID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Person">
    <s:sequence>
        <s:element name="Name" type="s:string" />
        <s:element name="DOB" type="s:date" />
    </s:sequence>
</s:complexType>
<s:complexType name="Employee">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="EmployeeID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
```

Notice that all four classes are represented (`Example2A`, `Patient`, `Person`, and `Employee`), but the choice list does not include the `Person` class. Also notice that the `Person`, `Patient`, and `Employee` types are defined in the same way as in the default case.

8.2.3 Example for Choice List with XMLINCLUDEINGROUP=0

Suppose that we add another subclass of the class `Person` and that we restrict it by setting *XMLINCLUDEINGROUP* to 0:

```
Class UsingSubclasses.Other Extends UsingSubclasses.Person
{
Parameter XMLINCLUDEINGROUP = 0;
}
```

In this case, this class is not included in the choice list and it is not included in the schema.

8.3 Creating a Substitution Group of Subtypes

The XML Schema specification also permits you to define a substitution group, which can be an alternative way to create choices. The syntax is somewhat different. Instead of making an explicit, central list of the types, you annotate the possible substitutes, as follows:

```
<s:complexType name="Example3">
    <s:sequence>
        <s:element ref="Person" minOccurs="0" />
    </s:sequence>
</s:complexType>
<s:element name="Person" type="Person"/>
<s:element name="Employee" type="Employee" substitutionGroup="Person"/>
<s:complexType name="Employee">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="EmployeeID" type="s:string" />
```

```
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Person">
    <s:sequence>
        <s:element name="Name" type="s:string" />
        <s:element name="DOB" type="s:date" />
    </s:sequence>
</s:complexType>
<s:element name="Patient" type="Patient" substitutionGroup="Person"/>
<s:complexType name="Patient">
    <s:complexContent>
        <s:extension base="Person">
            <s:sequence>
                <s:element name="PatientID" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
```

To create this schema, we use the following class:

```
Class UsingSubclasses.Example3 Extends (%Persistent, %XML.Adaptor)
{
Property Person As UsingSubclasses.Person
(XMLTYPECONSTRAINT = "SUBSTITUTIONGROUP");
}
```

8.3.1 Restricting a Subclass from the Substitution Group

For a given property, if you set the *XMLTYPECONSTRAINT* property parameter equal to `"SUBSTITUTIONGROUP"`, the group automatically consists of all subclasses of the type of the property, as shown in the previous example. You can use the *XMLINCLUDEINGROUP* parameter to mark a given subclass so that it is not included in the substitution group. For example, suppose we add another subclass of the class `Person`:

```
Class UsingSubclasses.Other Extends UsingSubclasses.Person
{

Parameter XMLINCLUDEINGROUP = 0;

Property OtherID As %String [ Required ];
}
```

In this case, this class is not included in the substitution group. And because you have explicitly marked this class in this way, it is not included in the schema at all.

8.4 How Superclasses Are Represented as Types

If you need the XML schema to show a certain hierarchy of types, you need to understand how the projection interprets your Caché class hierarchy.

Your class hierarchy represents a meaningful organization of data, among other things. This hierarchy is mirrored, as much as possible, in the corresponding XML type definitions.

For example, suppose that you have the following classes:

- A class named `Base`, which defines three public properties (`Property1`, `Property2`, and `Property3`).
- A class named `Addition1`, which extends `Base`and which defines an additional public property (`Addition1`).
- A class named `Addition2`, which extends `Addition1` and which defines an additional public property (`Addition2`).

What should the schema for Addition2 contain? It must represent all five properties. Also, because these classes are all user-defined, the schema for Addition2 should show the details of the class hierarchy; in contrast, if the Base extends a class from the Caché class library, which in turn extends other classes from that library, those details are less interesting.

Correspondingly, the XML schema for Addition2 looks as follows by default:

```
<s:complexType name="Addition2">
    <s:complexContent>
        <s:extension base="Addition1">
            <s:sequence>
                <s:element name="Addition2" type="s:decimal" minOccurs="0" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Addition1">
    <s:complexContent>
        <s:extension base="Base">
            <s:sequence>
                <s:element name="Addition1" type="s:string" minOccurs="0" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="Base">
    <s:sequence>
        <s:element name="Property1" type="s:string" minOccurs="0" />
        <s:element name="Property2" type="s:decimal" minOccurs="0" />
        <s:element name="Property3" type="s:date" minOccurs="0" />
    </s:sequence>
</s:complexType>
```

Because XML type definitions do not support multiple inheritance, the Caché XML support makes certain simplifying assumptions. For a class that extends multiple superclasses, the type for the class is assumed to be the *first listed* superclass. An example is shown below. Consider the following three class definitions. AddressPart1 contains one property:

```
Class GXML.Writer.ShowMultiple.AddressPart1 Extends %XML.Adaptor
{
Property Street As %String [ Required ];
}
```

The AddressPart2 class contains another property:

```
Class GXML.Writer.ShowMultiple.AddressPart2 Extends %XML.Adaptor
{
Property City As %String [ Required ];
}
```

Finally, Address inherits from both of these classes (with AddressPart1 as the first superclass) and adds more properties:

```
Class GXML.Writer.ShowMultiple.Address Extends
(GXML.Writers.ShowMultiple.AddressPart1,
GXML.Writers.ShowMultiple.AddressPart2)
{
Property State As %String(MAXLEN = 2, PATTERN = "2u") [ Required ];
Property Zip As %String(MAXLEN = 10, PATTERN = "5n.1(1""-""4n)") [ Required ];
}
```

The XML schema for Address is as follows:

```
<s:complexType name="Address">
    <s:complexContent>
        <s:extension base="AddressPart1">
            <s:sequence>
                <s:element name="City" type="s:string" />
                <s:element name="State" type="s:string" />
                <s:element name="Zip" type="s:string" />
            </s:sequence>
        </s:extension>
    </s:complexContent>
</s:complexType>
<s:complexType name="AddressPart1">
    <s:sequence>
        <s:element name="Street" type="s:string" />
    </s:sequence>
</s:complexType>
```

Notice the following:

- The first listed superclass, `AddressPart1`, is represented by the corresponding XML type, which contains all the expected details.
- Apart from the property contained by the `AddressPart1` type, all remaining properties are assigned to the `Address` type. This is the only possible representation of these properties, once the `AddressPart1` class has been mapped.
- Both `AddressPart1` and `AddressPart2` are subclasses of %XML.Adaptor, yet no structure of %XML.Adaptor class is exposed. The emphasis is on the custom classes, which is appropriate.

The following rules govern how the superclasses are handled when you view the schema for a given class:

- If a superclass inherits from %XML.Adaptor, it is represented by an XML type, which represents all the projected properties of this class. The short class name is taken as the XML type for the property. If the class specifies a value for the*XMLTYPE* parameter, that value is used as the type name instead.
- If a superclass does not inherit from %XML.Adaptor, it is not represented by an XML type. If it has any properties, they are assigned to the inheriting class (the class whose schema you are viewing).
- If the given class inherits from multiple superclasses, an XML type is created for the first superclass (if applicable; see the preceding rules). All properties that do not belong to the first superclass are assigned to the inheriting class, as in the previous example.

8.5 Classes Based on Multiple XML-Enabled Superclasses

In some cases, a given class might be based on multiple XML-enabled superclasses. In such cases, the corresponding XML schema considers the order in which these classes are listed. For example, consider the following class, which inherits from two XML-enabled superclasses:

```
Class Test.Default Extends (Test.Superclass1, Test.Superclass2)
{
///additional class members ...
}
```

The XML schema for *this* class lists the XML types derived from the left-most class `Test.Superclass1` before the XML types derived from `Test.Superclass2`. That same order occurs when you generate XML output for objects of this class.

If you instead want the XML schema (and the output) to be determined from right to left, specify the *XMLINHERITANCE* parameter as `"right"`. For example:

```
Class Test.Default Extends (Test.Superclass1, Test.Superclass2)
{
Parameter XMLINHERITANCE = "right";

///additional class members ...
}
```

9

Special Topics

This chapter describes the following additional special topics:

- How to control the closing of elements
- How to handle an XML document with multiple elements that have the same name
- How to control unswizzling after export
- How to project Caché internal IDs
- How to control namespace prefixes on export
- How to handle unexpected elements or attributes on import

The XML examples in this chapter are in literal format.

Class and Property Parameters Discussed in This Chapter
• *XMLNAME* • *XMLSEQUENCE* • *XMLUNSWIZZLE* • *XMLPREFIX* • *XMLIGNOREINVALIDTAG* • *XMLIGNOREINVALIDATTRIBUTE*

9.1 Controlling the Closing of Elements

In XML, an element that contains only attributes can be represented in either of the following ways:

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

Caché recognizes these forms as equivalent. When you export objects with %XML.Writer, you can control the closing form, but not by modifying the XML projection itself. See "Controlling the Closing of Elements" in the chapter "Writing XML Output from Caché Objects" in *Using Caché XML Tools*.

9.2 Handling a Document with Multiple Tags with the Same Name

A given element in XML can contain multiple elements that have the same name; these elements are distinguished from each other by their order. For example, the following is a legitimate XML document:

```
<?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>
      <Address>
         <Street>110 High Street</Street>
         <City>Zanesville</City>
         <State>OR</State>
         <Zip>80719</Zip>
      </Address>
   </Person>
</Root>
```

It is slightly tricky to map such a document to a Caché class, because each class property must have a unique name.

To map such a document to a Caché class, do the following:

- Set the *XMLNAME* property parameter as needed to map different class properties to the same XML name.
- Set the *XMLSEQUENCE* class parameter equal to 1. As a precaution, this ensures that the mapping respects the order of the properties as listed in the class definition.
- Make sure that the properties are listed in the class definition in the same order as in the XML document.

For example, consider the following class definition:

```
Class GXML.TestSequence.Person Extends (%Persistent, %XML.Adaptor)
{

Property Name As %Name [ Required ];
Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h") [ Required ];
Property HomeAddress As GXML.TestSequence.Address(XMLNAME = "Address");
Property WorkAddress As GXML.TestSequence.Address(XMLNAME = "Address");

/// If the XMLSEQUENCE = 1, then the order of the XML elements must match the
/// order of the class properties.  This allows us to deal with XML where the
/// same field appears multiple times and is distinguished by the order.
Parameter XMLSEQUENCE = 1;

}
```

This class definition maps to the XML document shown previously.

Note: If *XMLSEQUENCE* is 1, the *XMLIGNOREINVALIDTAG* parameter is ignored.

9.3 Controlling Unswizzling After Export

When you use Caché XML tools to 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, Caché

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.

9.4 Projecting Caché IDs for Export

When you project a Caché object at the top level (rather than as a property of another object), its internal ID, OID, and globally unique ID are not available as object properties, and these IDs are thus are not projected. However, in some cases, you might want to use an object ID as the unique identifier. Then, for example, you can match an incoming (changed) object to the corresponding stored object, before updating the stored object.

Caché XML support provides several helper classes that you can use to project Caché object identifiers to XML documents: %XML.Id (for the internal ID), %XML.Oid (for the OID), and %XML.GUID (for the globally unique ID).

To use these classes, add a special property to your XML-enabled class whose purpose is to contain the ID that you will export. The property must be of type %XML.Id, %XML.Oid, or %XML.GUID. Make sure that this property is projected, and mark it as transient so that it is not included in the SQL projection of the class.

When you are exporting to XML, you bring an object of your XML-enabled class into memory. When the object is in memory, the special property you added retrieves the requested ID from Caché internal storage and contains that value (so that you can export it).

For example, consider the following class:

```
Class MyApp4.Obj.Person4 Extends (%Persistent,%Populate,%XML.Adaptor)
{

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

Property Name As %Name;

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

}
```

In this class, the special property is `IdForExport`. This property is specifically projected with the XML element name of `CacheID`.

Example output for this class is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <Person>
      <CacheID>1</CacheID>
      <Name>Marks,Jules F.</Name>
      <DOB>1989-04-02</DOB>
   </Person>
   <Person>
      <CacheID>2</CacheID>
      <Name>Palmer,Angelo O.</Name>
      <DOB>1937-11-15</DOB>
   </Person>
...
```

9.5 Controlling the Namespace Prefix on Export

When you generate XML output for an object, the system generates namespace prefixes as needed, but you can specify the prefixes if needed. To do so, set the following parameter in the class definitions for the XML-enabled objects:

XMLPREFIX

Specifies the prefix to associate with the namespace for this class.

For details, see "Writing XML Output from Caché Objects," in *Using Caché XML Tools*.

9.6 Handling Unexpected Elements and Attributes on Import

Because the source XML documents might contain unexpected elements and attributes, your XML-enabled classes provide two parameters to specify how to react when you import such a document. For example, consider the following class definition:

```
Class GXML.TestImportParms.Person Extends (%Persistent,%XML.Adaptor)
{

Property Name As %Name [ Required ];
Property DOB As %Date(FORMAT = 5, MAXVAL = "+$h") [ Required ];
}
```

Also consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <Person employeeID="450">
      <Name>Dillard, Daniel</Name>
      <DOB>1962-09-18</DOB>
      <UserID>fr0078</UserID>
      <Address>
         <Street>810 Main Street</Street>
         <City>Reston</City>
         <State>NJ</State>
         <Zip>02641</Zip>
      </Address>
   </Person>
</Root>
```

The `employeeID` attribute and the `<Address>` element do not correspond to properties in the class and are therefore unexpected.

To specify how to handle unexpected attributes and elements, use the following parameters of your XML-enabled classes:

XMLIGNOREINVALIDATTRIBUTE

Controls how to handle attributes that are unexpected. If this parameter is 1 (the default), such attributes are ignored. If it is 0, they are treated as errors, and import fails.

XMLIGNOREINVALIDTAG

Controls how to handle elements that are unexpected. If this parameter is 1, such elements are ignored. If it is 0 (the default), they are treated as errors, and import fails.

These parameters affect only import.

Note: The `xmlns` attribute, array key `name` attribute, and schema instance (`xsi`) attribute are always ignored.

Also, if *XMLSEQUENCE* is 1, the *XMLIGNOREINVALIDTAG* parameter is ignored. See the section "Handling a Document with Multiple Tags with the Same Name," earlier in this chapter.

A

Summary of XML Projection Parameters

This appendix summarizes the XML projection options in Caché. Unless otherwise indicated, the class parameters are available in your XML-enabled classes, and the property parameters are available for properties of those classes.

Topic	Parameters
Enabling the XML projection. See "Projecting a Caché Object to XML."	*XMLENABLED* class parameter
Mapping of properties to elements or attributes. See "Projecting Caché Objects to XML."	• *XMLPROJECTION* property parameter (`"NONE"`, `"ATTRIBUTE"`, `"XMLATTRIBUTE"`, `"CONTENT"`, `"ELEMENT"`, or `"WRAPPED"`) • *XMLSUMMARY* class parameter • *XMLDEFAULTREFERENCE* class parameter (`"SUMMARY"`, `"COMPLETE"`, `"ID"`, `"OID"`, or `"GUID"`) • *XMLREFERENCE* property parameter (`"SUMMARY"`, `"COMPLETE"`, `"ID"`, `"OID"`, or `"GUID"`)
XML element names and attribute names. See "Controlling the XML Element and Attribute Names."	• *XMLNAME* class parameter • *XMLNAME* property parameter • *XMLITEMNAME* property parameter • *XMLKEYNAME* property parameter Defaults are based on XML type names.
XML types. See "Controlling the Projection to XML Schemas."	• *XMLTYPE* class parameter • *XMLTYPE* property parameter • *XSDTYPE* class parameter

Topic	Parameters
Namespaces. See "Specifying Namespaces for Elements and Attributes."	• *NAMESPACE* class parameter • *ELEMENTQUALIFIED* class parameter (0 or 1) You can override this upon export. • *ELEMENTQUALIFIED* property parameter (0 or 1) You can override this upon export. • *ATTRIBUTEQUALIFIED* class parameter (0 or 1) You can override this upon export. • *XMLREF* property parameter (0 or 1) • *REFNAMESPACE* property parameter • *XMLPREFIX* class parameter
Empty strings and nulls. See "Handling Empty Strings and Null Values."	• *XMLUSEEMPTYELEMENT* class parameter (0 or 1) • *XMLIGNORENULL* class parameter (0, 1, `"INPUTONLY"`, or `"RUNTIME"` • *XMLNIL* class parameter (0 or 1) *XMLNIL* property parameter (0 or 1) You can override *XMLNIL* upon export or import if *XMLIGNORENULL* is `"RUNTIME"` • *XMLNILNOOBJECT* class parameter (0 or 1) • *XMLNILNOOBJECT* property parameter (0 or 1)
Escaping XML special characters. See "Handling Special XML Characters."	• *CONTENT* parameter (`"STRING"`, `"ESCAPE"`, `"ESCAPE-C14N"`, or `"MIXED"`) • *ESCAPE* parameter (`"XML"` or `"HTML"`)
Time zones. See "Handling the UTC Time Zone Indicator."	*XMLTIMEZONE* property parameter (`"UTC"` or `"IGNORE"`)
XML type details, including restrictions. See "Controlling the Projection to XML Schemas" and "Advanced Options for XML Schemas"	• *XMLTYPECONSTRAINT* property parameter (`"EXPLICIT"`, `"CHOICE"`, or `"SUBSTITUTIONGROUP"`) • *XMLINCLUDEINGROUP* class parameter (0 or 1) • *XMLCHOICELIST* property parameter • *XMLINHERITANCE* class parameter (`"left"` or `"right"`) • Many Caché data type property parameters
Using property for input, output, or both. See "Controlling the Availability of Projected Properties."	*XMLIO* property parameter (`"INOUT"`, `"IN"`, `"OUT"`, or `"CALC"`)

Topic	Parameters
Controlling available XML document formats. See "Specifying Format Options for the XML Document."	*XMLFORMAT* class parameter (`"LITERAL"`, `"ENCODED"`, or null for both formats)
Multiple elements with the same name. See "Handling a Document with Multiple Elements with the Same Name."	*XMLSEQUENCE* class parameter (0 or 1)
Stream properties. See "Controlling the Line Endings of Stream Properties."	*XMLSTREAMMODE* property parameter (`"BLOCK"` or `"LINE"`)
Unexpected elements and attributes. See "Handling Unexpected Elements and Attributes on Import."	• *XMLIGNOREINVALIDTAG* class parameter (0 or 1) • *XMLIGNOREINVALIDATTRIBUTE* class parameter (0 or 1)
Namespace prefixes. See "Controlling the Namespace Prefix on Export."	*XMLPREFIX* class parameter
Specifying the `pattern` restriction in the schema.	*XMLPATTERN* property parameter

There is an additional parameter: the *XMLELEMENTREF* property parameter, which is deprecated (replaced by *XMLREF*), but will be supported indefinitely.