XBinder

XML Schema Compiler
Version 2.0
Java/C#
Reference Manual

Objective Systems, Inc. — August 2009

Author's Contact Information

Comments, suggestions, and inquiries regarding XBinder may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

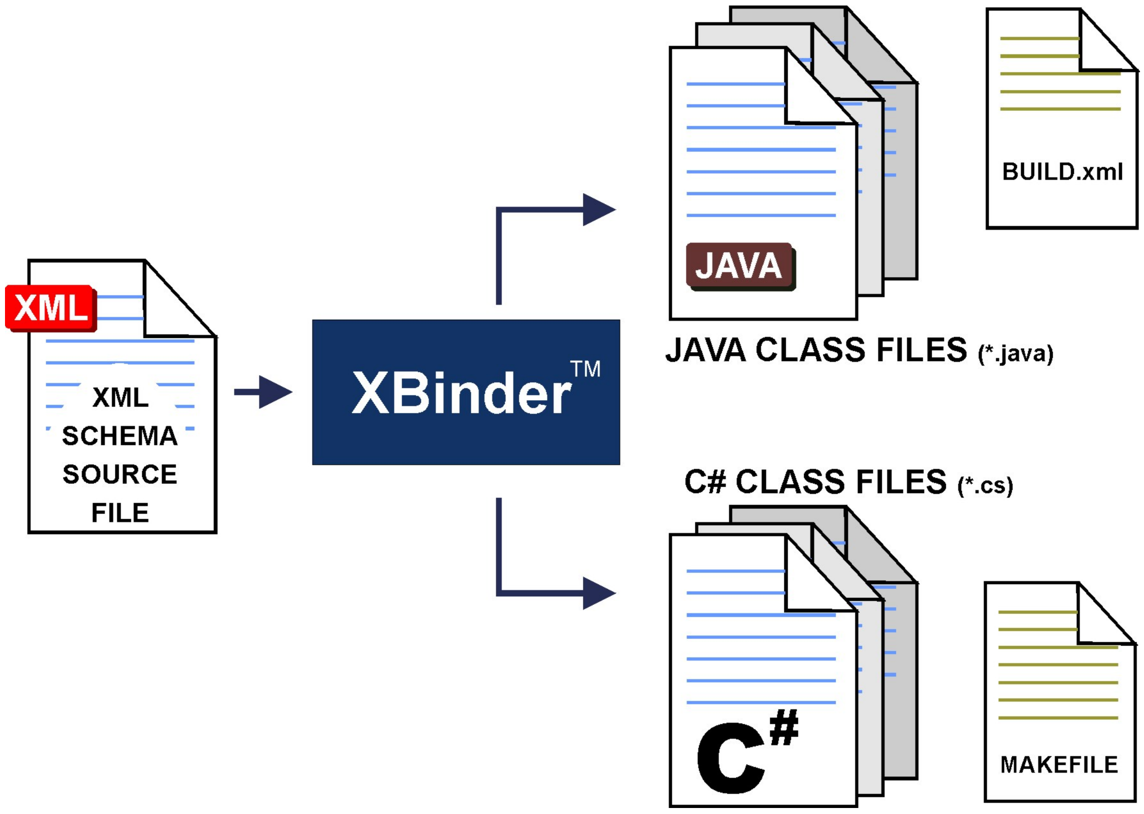
Chapter 1. XBinder Overview

The XBinder code generation tool translates an XML Schema Definition (XSD) source file into computer language C# or Java source files. These source files contain an application programming interface (API) that allows programmatic data to be encoded to XML format and decoded to programmatic variables. Each variable is of a type that corresponds to a type, element, or attribute defined within the XML schema document.

Each XSD source file results in the generation of C#/Java classes that represent each of the XSD types and global elements contained within the XSD source file. These classes contain encode, decode, and utility functions. A print utility function may be generated to print the object tree.

These generated classes, along with the XBinder run-time, provide a complete package for working with XML encoded data.

The diagram below shows the flow that accomplishes the translation from an XML schema definition to Java or C# source code:



Chapter 2. Using XBinder

Running XBinder from the Command-line

The XBinder distribution contains a command-line compiler executable as well as a graphical user interface (GUI) wizard that can aid in the specification of compiler options. This section describes how to run the command-line version; the operation of the GUI is described in the online help files built into the wizard.

To test if XBinder was successfully installed, enter xbinder with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
xbinder
```

You should observe the following display (or something similar):

```
XBinder Compiler, Version 2.0.x
Copyright (c) 2002-2009 Objective Systems, Inc. All Rights Reserved.

Usage: xbinder <filename> options

    <filename>         XML schema or WSDL source file name(s).
                         Multiple filenames may be specified.
                         * and ? wildcards are allowed.

Language options (choose only one):
    -c                 Generate C code
    -c++ or -cpp       Generate C++ code
    -c# or -csharp     Generate C# code
    -java              Generate Java code

Basic options:
    -xml               Generate XML encode/decode functions
    -config <file>     Specify schema bindings file.
    -o <directory>     Output file directory
    -I <directory>     Import file directory
    -all               Compile all dependent files
    -warnings          Output compiler warning messages
    -compat <version>  Generate code compatible with previous
                         compiler version. <version> format is
                         x.x (for example, 1.0)

Options to reduce amount of generated code:
    -lax               Generate code that does lax error checking
    -noderiv           Do not generate special derived type code

Options to alter generated code:
    -namespace         Specify a namespace prefix for all generated items
    -nodatestamp       Do not put date stamp in header
    -nomixed           Do not generate string structure for mixed content
    -elemCasing <value> Set element name case to lower/upper
    -typeCasing <value> Set type name case to lower/upper

Options for the generation of additional code:
    -genPrint or
    -print              Generate print functions

C/C++ basic options:
    -dom                Generate DOM encode/decode functions (C only)
    -sax                Generate SAX-based decoders (default is pull-parser)
    -modularize         Handle included schemas as separate modules
    -nodecode           Do not generate decode functions
    -noencode           Do not generate encode functions
    -trace              Add trace diag msgs to generated code

C/C++ options to reduce amount of generated code:
    -compact            Generate compact code
    -nocopy             Do not generate copy methods (C++ only)
    -noheader           Do not add code to encode XML header (<? xml ...)
    -noxmlns            Do not generate namespace attributes in all structs

C/C++ options to alter generated code:
    -c14n               Generate C14N format encode functions
    -cppns <ns>         Add given C++ namespace to generated code (C++ only)
    -fragments          Generate code to encode XML fragments
                          (start element, contents, end element)
    -static             Generate code that uses static memory
                          (when possible, C only)
    -initlists          Generate code that initializes lists to default
                          (when possible)
    -noNamedBits        Do not generate named bits for EnumList,
                          use regular list instead.
    -numDateTime        Use numeric structures for all date/time types
    -pdu <element>      Designate element to be a PDU
    -project <prj_name> Set project name
    -soap, -soap12      Generate code to format/parse SOAP v1.2 messages
    -soap11             Generate code to format/parse SOAP v1.1 messages
    -strict             Generate code that does strict syntax checking
    -useNSPfx           Use XSD namespace prefixes in C/C++ code
    -useflteq           Use float equality functions to ignore rounding
                          errors in floating-point comparisons
    -usestl             Use C++ Standard Template Library (STL)
                          (currently, only STL string type is used)
    -derivModel <model> Set the derivation model to extended/inteface.

C/C++ options for the generation of additional code:
    -genMake            Generate makefile
    -genMakeLib [<libname>]  Generate code in makefile to put all objects
                             into a static library
    -genMakeDLL [<dllname>]  Generate code in makefile to build shared object
    -makeopts <dynamiclib | staticlib | multithreaded>
         Use compilation options for dynamic, static, and multithreaded
         libraries.
    -genWriter          Generate writer test program
    -genReader          Generate reader test program
    -genFree            Generate memory free functions (C only)
    -genKeyTest         Generate code to test identity constraints
                          (xsd:key, xsd:keyref, xsd:unique)
    -genRWTest          Generate read/write test program
    -genStubs          Generate SOAP client stub functions from WSDL
    -genSkel           Generate SOAP skeleton server program from WSDL
    -genClient         Generate web service client test program from WSDL
    -genTest [<xmlfile>]  Generate test code.  If XML instance provided,
                          this will be used; otherwise, random data
    -genValid          Generate validation functions
    -usePDU <element>  Use PDU for writer/reader test program
    -w32               Use with '-genMake' to generate Windows NMAKE file
                         (default = GNU)

C/C++ compression options:
    -exicompress       Add code to generated reader/writer to use
                         Efficient XML Interchange (EXI) format
    -zip               Add code to generated reader/writer to use
                         standard gzip compression (requires zlib install)

C# extra options:
    -csnsname <name>   Name for C# namespace
    -csnspfx <prefix>  C# namespace names will be this prefix, followed by
                       schema name
    -genMake           Generate makefile
    -genWriter         Generate writer test program
    -genReader         Generate reader test program
    -usePDU <element>  Use PDU for writer/reader test program

Java extra options:
    -pkgname <name>    Name for Java package
    -pkgpfx <prefix>   Java package names will be this prefix, followed by
                       schema name
    -genMake           Generate Apache Ant buildfile (build.xml)
    -genWriter         Generate writer test program
    -genReader         Generate reader test program
    -usePDU <element>  Use PDU for writer/reader test program
```

To use the compiler, at a minimum, a single XSD source file must be provided along with at least one set of encoding rules and a target output language. The current version of XBinder supports the generation of C (-c option), C++ (-cpp option), C# (-c# or -csharp) and Java (-java) source code. For C# and Java, XBinder supports the generation of code to encode/decode XML (-xml).

The source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters ‘*’ and ‘?’ are also allowed in source filenames (for example, the command ‘xbinder *.xsd -c -xml’ will compile all XSD files in the current working directory).

The source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters ‘*’ and ‘?’ are also allowed in source filenames (for example, the command ‘xbinder *.xsd -c -xml’ will compile all XSD files in the current working directory).

The following table lists all of the command line options (those relevant to C# or Java) in alphabetical order and what they are used for:

Option	Argument	Description
-all	None	Generate code for all dependent files in a given compilation. This includes the main XSD files specified on the command line as well as all imported and included schema files.
-c# -csharp	None	Generate C# code
-compat	<version>	Generate code that is compatible with an older version of the XBinder compiler. The format of the version number is "n.n" (for example, 1.0). As of XBinder 2.0, this has no meaning for C# or Java code generation, but it may be applicable in the future.
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. This is similar to the ‘binding schema’ used with some other XML data binding applications
-csnsname	<name>	Provides a C# namespace name. All classes will be generated into this namespace.
-csnspfx	<prefix>	Provides a prefix for C# namespace names. Each class will be generated into a namespace named <prefix>.<schema>, where <schema> is based on the name of the XSD file from which the class was generated.
-elemCasing	lower or upper	This option is used to change the case of the first letter in element names in the generated code from what is specified in the XSD file. This option is typically used when the XSD file contains type, element, and/or attribute names that are the same. It provides a way to disambiguate names in the generated code. Typically, element names are set to lower case. lower is the default for C# and Java See -typeCasing for changing the case of type names.
-genPrint	None	Generate print utility functions. Print functions are debug functions that allow the contents of generated type variables to be written to any given output stream.
-genMake	None	For C#, XBinder will generate a makefile. For Java, XBinder will generate an Apache Ant buildfile.
-genReader	None	Generate a sample reader program
-genWriter	None	Generate a sample writer program
-I	<directory>	This option is used to specify a directory that will be searched for XSD <import> and <include> items. Multiple –I qualifiers can be used to specify multiple directories to search.
-lax	None	This option causes decode/validation functions to be generated that contain less schema-validation error checking code. This can be useful for working with XML documents that are not completely valid with regards to the schema.
-modularize	None	This option is used when XSD <include> directives are included in a schema to tell the processor to put the generated code in separate output files based on the definitions in the included files. The default behavior if this is not used is to include all of the code in the main file that is including the definitions. This can only be used if the included

Option	Argument	Description
		files can be successfully compiled on their own (i.e. are not dependent on definitions from the parent module).
-namespace	<prefix>	Add the given prefix to all class names. This makes it possible to disambiguate items with the same names that are in different schemas. Note that the -csnspfx (for C#) or -pkgpfx (for Java) can be used to derive C# namespaces or Java package names from the name of the XSD file.
-nodatestamp	None	Do not insert a date stamp in the header of each generated file. This can be useful when using a source control system to prevent identical source files from appearing different.
-noderiv	None	Suppress generation of extra code for run-time derived type handling. This code makes it possible to decode complexContent types using xsi:type declarations.
-nomixed	None	Do not generate a special structure to hold mixed content items. The generated code will more closely match the schema layout. However, mixed content will not be supported.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-pkgname	<name>	Provides a Java package name. All classes will be generated into this package.
-pkgpfx	<prefix>	Provides a prefix for package names. Each class will be generated into a package named <prefix>.<schema>, where <schema> is based on the name of the XSD file from which the class was generated.
-print	None	see -genPrint
-typeCasing	lower or upper	This option is used to change the case of the first letter in generated class names from what is specified in the XSD file. This option is typically used when the XSD file contains type, element, and/or attribute names that are the same. It provides a way to disambiguate the names in the generated code. The default for C# and Java is upper. See -elemCasing for changing the case of element names.
-usePDU	name of element	When generating sample code (example -genReader -genWriter), this option tells XBinder which global element should be handled by the sample code.

Building and Running Generated Code

Java

There are three main components that you need:

1. Java compiler and runtime. JSE 5+ is required.
2. XBinder Java Runtime (xbrt.jar). This needs to be in your classpath when compiling, and at runtime.
3. StAX API and StAX implementation. The API must be available during compilation; both the API and the implementation must be available at runtime. Note that JSE 6+ includes the StAX API and a StAX implementation.

Obtaining StAX API

If you are using JSE 5, you will need to obtain the StAX API. It is available at http://dist.codehaus.org/stax/jars/ stax-api-1.0.1.jar. As of JSE6, the StAX API is part of the JSE platform; you do not need to obtain it separately.

Be sure to include stax-api-1.0.1.jar in your classpath at compilation and run-time.

Obtaining StAX Implementation (Woodstox)

If you are using JSE 5, you *must* obtain an implementation of StAX. We recommend Woodstox 3.9.2 or higher. Even if you are using JSE 6, you *may* want to use Woodstox, as this is the implementation we are currently using internally.

You can obtain Woodstox at http://woodstox.codehaus.org/.

To use Woodstox, simply include the downloaded JAR in your classpath (this works even if you are using JSE 6).

C#

There are two main components that you need:

1. Version 2.0 or higher of the Microsoft .NET Framework software development kit. You will need the full software development kit, not just the redistributable package. The .NET Framework software development kit includes the C# compiler (csc.exe) and a set of classes that are used at runtime during decode and encode operations. The Xbinder C# functionality was tested with version 2.0 of the .NET Framework.

2. XBinder C# Runtime (xbrt.dll and xbrt.snk). The DLL needs to be referenced when compiling and in a location where it can be found at runtime. If you copy the DLL to another location before compiling against it, you will also need to copy the .snk file. At runtime one way to ensure that the file is found is to copy xbrt.dll into the same folder as the .exe that is referencing it. Another way is to add xbrt.dll to the global assembly cache via the gacutil.exe program. Other mechanisms might also be available via application configuration files; consult the appropriate Microsoft documentation for the development tool you're using.

Obtaining the Microsoft .NET Framework

You can obtain the Microsoft .NET Framework via the download center on the Microsoft web site (www.microsoft.com). Depending on the version you select, the words “software development kit” may or may not be specified with the selection on the web site. But the words “redistributable package” are consistently used for kits that contain just runtime components. You will want the software development kit, not the redistributable package.

If Microsoft Visual Studio 2005 or higher is installed on the machine where you will be using the Xbinder C# compiler, then you already have at least version 2.0 of the Microsoft .NET Framework. The Microsoft .NET Framework is typically installed into the \WINDOWS\Microsoft.NET\Framework folder; you can probably check this location to see if you already have it. You can also search for csc.exe to see if you already have the Microsoft C# compiler.

Getting Started

PurchaseOrder Sample

You will find C# and Java PurchaseOrder sample programs in csharp/samples/PurchaseOrder and java/samples/ PurchaseOrder, respectively.

The sample consists of the following:

- Sample XML schema (po.xsd)
- Sample Writer program (Writer.cs/Writer.java)
- Sample Reader program (Reader.cs/Reader.java)

When you build the sample, XBinder is used to compile the sample schema and the C#/Java compiler is used to compile the writer and reader programs. When XBinder compiles the schema, it generates classes that serve both as data structures, modeling the schema data, and as encoder-decoders, encoding and decoding the data structures to/from XML. The reader and writer programs depend on these generated classes.

To build the C# sample program, execute the makefile in the PurchaseOrder folder via the “nmake” command.

To build the Java sample program, execute the build.bat or build.sh script

Writer

The Writer program creates an XML file. Its code populates the data structures generated by XBinder, and encodes this data to XML using the generated encode method. The Reader program decodes an XML file, using the generated decode method, into the data structures generated by XBinder. It then prints this data to standard output. By default, the writer encodes to message.xml, and the reader decodes from message.xml. To run the writer for Java, use the writer.bat or writer.sh script; for C# run writer.exe.

```
PurchaseOrder_CC root = new PurchaseOrder_CC();
```

This code creates the document codec, which contains convenience methods for encoding/decoding an entire XML document. You can think of it as representing the document as a whole and containing a single element (purchaseOrder).

```
PurchaseOrderType purchaseOrder = new PurchaseOrderType();
root.setPurchaseOrder(purchaseOrder);
```

This code creates an object of the purchaseOrder element's type (PurchaseOrderType), and supplies it to the root as being the data for the purchaseOrder element.

Java

```
XBXmlEncoder encoder;
encoder = new XBXmlEncoder(bos, charset);
root.encodeDocument(encoder);
encoder.close();
```

C#:

```
XBXmlEncoder encoder;
encoder = new XBXmlEncoder(fout, charset);
root.encodeDocument(encoder);
encoder.Close();
```

This code performs the encoding. After all of the data structures are populated, we create encoder1, supplying it with an OutputStream and a character set. We then pass the encoder to the PurchaseOrder_CC encodeDocument method, which encodes the data into a purchaseOrder element. Lastly, we close the encoder. [1]

[1] We use "encoder" to refer to two different things, but the context should make it clear what is being referenced. Both PurchaseOrder_CC and XBXmlEncoder are encoders, but they work at different levels of abstraction. PurchaseOrder_CC (and the other classes generated by XBinder) encode/decode specific XML structures, while XBXmlEncoder provides generic XML encoding services. Here, we are obviously referring to the XBXmlEncoder object.

Controlling XML Namespaces for encoding

The PurchaseOrder sample doesn't involve XML namespaces. However, we can still look at the generated code to see the facilities that XBinder provides for controlling XML namespaces.

Class _po includes the following declaration:

```
public static XBXmlNamespace[] namespaceContext = {};
```

Class _po is a "schema class" (generated for each schema you compile). The `namespaceContext` field is used to hold namespace declarations that you may want to use, derived from the XSD you compiled. This sample doesn't use XML namespaces, so the array is empty. For schemas that do use namespaces, you can manipulate this array, and use the following code to tell the encoder to encode those namespaces with the document root:

```
encoder.setNamespaces(_po.namespaceContext);
```

XBinder will try to guess whether the schema's target namespace should be encoded as the default namespace, or using a namespace prefix. This choice will be reflected in the contents of the namespaceContext array. Whether this choice is used or not depends on whether you change the contents of the array and/or actually invoke `setNamespaces` prior to invoking `encodeDocument`.

Another way to control namespaces, assuming your root element is of a complex type, is to use a method provided in class XBComplexType. All classes generated for complex types ultimately derive from `XBComplexType`. (In our sample, `PurchaseOrderType` is a complex type and so it extends `XBComplexType`) The method is `XBComplexType.addNamespace`:

```
public void addNamespace(String nsUri, String prefix)
```

Reader

The Reader program decodes an XML file using the generated classes, which provide both the data structure to decode into, as well as the logic for decoding. It then prints the decoded information to the standard output.

The reader program first creates a source to read from:

C#:

```
XmlTextReader reader = null;
try
{
   FileStream fin = new FileStream(inputFile, FileMode.Open, FileAccess.Read);
   reader = new XmlTextReader(fin);
```

Java:

```
XMLStreamReader[2] reader = null;
try {
   XMLInputFactory inputFactory = XMLInputFactory.newInstance();
   reader = inputFactory.createXMLStreamReader( inputFile,
             new BufferedInputStream(
                   new FileInputStream(inputFile) ) );
```

Next, as with the Writer, an instance of PurchaseOrder_CC is created. Then, decodeDocument is invoked:

```
PurchaseOrder_CC root = new PurchaseOrder_CC();
```

[2]XMLStreamReader is defined as part of the StAX API. As of JSE 6, this API is a part of the JSE. It was defined under JSR-173.

```
root.decodeDocument(reader);
```

Finally, the decoded data is printed to an output stream (pw or System.Console.Out):

Java:

```
root.print(pw, "", 0);
```

C#:

```
root.print(System.Console.Out, "", 0);
```

The print method invoked above is generated when the -print option is given to XBinder (as is done in the sample build script). The information is printed in terms of generated fields, rather than XML elements.

Using your own XML Schema

To use you own XML Schema, you will basically do the same things you saw in the PurchaseOrder sample. Keep in mind the following:

- For every global element, an *_CC class is created. This will provide you with encodeDocument, decodeDocument methods.
- Using the _CC classes is not required. They are fairly simple and you could use them as a basis for creating something else that meets your particular needs.
- The XBXmlEncoder class encodes to an underlying stream. You control where the data goes by configuring the stream you provide.
- Under Java, decoding requires StAX. See the section on Building and Running Generated Code for information about StAX.

Chapter 3. Generated Java and C# Source Code

Schema Classes

For each schema being compiled, a schema class is generated. It will be named _<schema>, where <schema> comes from the .xsd file name.

Here is an abbreviated example:

Java:

```
public class _myschema {
   public static XBXmlNamespace[] namespaceContext = {
      new XBXmlNamespace("", "http://obj-sys.com/example")};

   //declare constants for the schema's ns and preferred prefix
   public static final String NS_URI = "http://obj-sys.com/example";
   public static final String NS_PREFIX = "osys";

   public static XBDoubleFormat defaultDoubleFmt = new XBDoubleFormat();
   public static XBDoubleFormat globalFloatNumFmt = new XBDoubleFormat(...);
   public static XBDoubleFormat numFmt0 = new XBDoubleFormat(...);

   public _myschema() {
   }
}
```

C#:

```
public class _myschema {
   public static XBXmlNamespace[] namespaceContext = {
      new XBXmlNamespace("", "http://obj-sys.com/example")};

   //declare constants for the schema's ns and preferred prefix
   public static readonly String NS_URI = "http://obj-sys.com/example";
   public static readonly String NS_PREFIX = "osys";

   public static XBDoubleFormat defaultDoubleFmt = new XBDoubleFormat();
   public static XBDoubleFormat globalFloatNumFmt = new XBDoubleFormat(...);
   public static XBDoubleFormat numFmt0 = new XBDoubleFormat(...);

   public _myschema() {
   }
}
```

This schema class demonstrates the following features:

1. namespaceContext: represents prefix-namespace mappings you might want to initialize the encoder with. These are derived from the compiled schema.
2. NS_URI, NS_PREFIX: referenced in generated code when encoding elements from this schema

3. *Fmt* fields: referenced in generated code for formatting numeric values during encoding. These are the result of specifying format options in the configuration file.

Type Classes

A type class will be generated for most types defined in your schema. The exception to this is an anonymous simple type, in which case the necessary code is generated "in-line" at the point of use. There are two categories of type classes, based on the two categories of XSD types: simple and complex

Simple Type Classes

Simple types represent a single value, and XBinder represents their data using predefined C# or Java types (e.g. int, String; see XSD Simple Type Mappings). The simple type classes provide static encode/decode methods and are used only for encoding/decoding; they do not contain the simple type value. You should not have to interact with the simple type classes.

Enumerations

A simple type with enumeration facets is an exception to the above. In this case, for Java an enum class is generated, and the enum constants are used to represent the data. For C# a class resembling a Java enum class is generated that can wrap a possible value for the enum.

Complex Type Classes

Complex types combine attributes with a content model. XBinder generates a class for each complex type. Both attributes and the content model are represented using properties. So, unlike simple type classes, instances of complex type classes actually represent the corresponding data.

For additional information, see the section *XSD Complex Type Mappings*.

Group Classes

In XSD, a group is a collection of particles (elements, element wildcards, and other groups). A group may be nested inside another group. Also, named groups may be defined. In both cases, XBinder generates a group class. A group class is basically a complex type class, except that there are no attributes to create properties for.

Document Classes

For each global element, a document control class is generated. These are named <elem name>_CC.

The document class contains a value that represents the global element, along with encodeDocument and decodeDocument methods.

Here is an abbreviated example:

Java:

```
public class MyDoubleElement_CC implements XBDocumentCodec
{
   private XBContext _xbContext = new XBContext();
   ...
   public double getValue() {...}
```

```
      public void setValue(double value) {...}

      public void decodeDocument(XMLStreamReader reader) {...}

      public void encodeDocument(XBXmlEncoder encoder) {...}

      ...
   }
```

C#:

```
   public class MyDoubleElement_CC : XBDocumentCodec
   {
      private XBContext _xbContext = new XBContext();
      ...
      public double getValue() {...}

      public void setValue(double value) {...}

      public void decodeDocument(XMLStreamReader reader) {...}

      public void encodeDocument(XBXmlEncoder encoder) {...}

      ...
   }
```

To use the namespace prefixes generated for your schema, do the following before calling encodeDocument (or, create and pass your own array):

```
   encoder.setNamespaces(_myschema.namespaceContext);
```

Features to note are:

1. get/set methods for the element's value. In this example, the element was of simple type double.

2. The _xbContext field is used during encoding/decoding; you don't need to create an XBContext object when you use the decodeDocument or encodeDocument methods.

C# Namespaces & Java Packages

There are two options for specifying the namespace/package that classes will be generated into:

1. 1.-csnsname (C#) or -pkgname (Java): This option lets you specify a single namespace/package that all classes will be generated into. This option is a good alternative when your schemas do not have overlapping type names.

2. 1.-csnspfx (C#) or -pkgpfx (Java): This option causes the classes for each schema to be generated into a separate namespace/package, each of which will be a child of the namespace/package specified with the option. For example, given "`-csnspfx com.test`", classes from `myschema.xsd` will be in `namespace com.test.myschema`

TIP: Choose your namespace/package name carefully. Avoid using a name whose first piece will be used as a name in your schema. For example, using test as a package name when there are elements (and thus generated fields) named test will likely cause C#/Java compilation errors (the field name will hide the namespace/package name).

Chapter 4. XSD Simple Type Mappings

The following table describes the mappings for the XSD built-in simple types.

XSD Built-In Type	C# Type	Java Type
anyURI	string	string
base64Binary	com.objsys.xbinder.runtime.XBByteArray	com.objsys.xbinder.runtime.XBByteArray
boolean	bool	boolean
byte	sbyte	byte
date	string	javax.xml.datatype.XMLGregorianCalendar
dateTime	string	javax.xml.datatype.XMLGregorianCalendar
decimal	decimal	java.math.BigDecimal
double	double	double
duration	com.objsys.xbinder.runtime.XBDuration	javax.xml.datatype.Duration
ENTITIES	string[]	string[]
ENTITY	string	string
float	float	float
gDay	string	javax.xml.datatype.XMLGregorianCalendar
gMonth	string	javax.xml.datatype.XMLGregorianCalendar
gMonthDay	string	javax.xml.datatype.XMLGregorianCalendar
gYear	string	javax.xml.datatype.XMLGregorianCalendar
gYearMonth	string	javax.xml.datatype.XMLGregorianCalendar
hexBinary	string	com.objsys.xbinder.runtime.XBByteArray
ID	string	string
IDREF	string	string
IDREFS	string[]	String[]
integer	int	int
int	int	int
language	string	string
long	long	long
Name	string	string
NCName	string	string
negativeInteger	int	int
NMTOKEN	string	string
NMTOKENS	string[]	string[]
nonNegativeInteger	uint	int
nonPositiveInteger	int	int
normalizedString	string	string
positiveInteger	uint	int
short	short	short

XSD Built-In Type	C# Type	Java Type
string	string	string
time	string	javax.xml.datatype.XMLGregorianCalendar
token	string	String
unsignedByte	byte	short
unsignedShort	ushort	int
unsignedInt	uint	long
unsignedLong	ulong	java.math.BigInteger

Atomic, User-Defined types

Atomic, user-defined types are specified in XSD using <xsd:simpleType> with <xsd:restriction>. The purpose is to further restrict some facet of the base type.

These XSD types are frequently defined anonymously. In that case, XBinder will generate validation code where the anonymous type is being used. If a named type is defined, XBinder will generate a corresponding class with static encode/decode methods. Validation logic will be generated into the encode/decode methods. In both cases, values of these types are represented according to the type mappings given in the table above (with the exception of enumerations; see *Enumeration*, below).

Example XSD: define an integer-based type that ranges from 27 to 65:

```
<xsd:simpleType name="TwentySevenSixtyFive">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="27"/>
      <xsd:maxInclusive value="65"/>
   </xsd:restriction>
</xsd:simpleType>
```

C#:

```
public class TwentySevenSixtyFive
{
   public static readonly byte _MIN = 27;
   public static readonly byte _MAX = 65;

   //constructor
   private TwentySevenSixtyFive() {}


   public static String encode(byte value, XBContext xbContext) {...}

   public static byte decode(String text, XBContext xbContext) {...}
}
```

Java:

```
public class TwentySevenSixtyFive
{
   public static final byte _MIN = 27;
   public static final byte _MAX = 65;
```

```
      //constructor
      private TwentySevenSixtyFive() {}


      public static String encode(byte value, XBContext xbContext) {...}

      public static byte decode(String text, XBContext xbContext) {...}
   }
```

The generate encode and decode methods validate the min/max facets, as well as converting between a value representation (byte) and lexical representation (String). As mentioned above, the encode/decode methods in this case are static; the class simply provides coding services and does not represent values of the type it supports.

Note that XBinder chose to represent values of this type using byte. Given the value range, XBinder determined this is the most appropriate representation. This selection can be overridden using a configuration file. As a quick example, to use a short instead of a byte in the above example, you would use the following configuration file:

```
   <bindings version="1.0">
      <schemaBindings schemaLocation="example.xsd">
         <nodeBindings node=
          "/xsd:schema/xsd:simpleType[@name=&quot;TwentySevenSixtyFive&quot;]">
            <javatype>int16</javatype>
            <cstype>int16</cstype>
         </nodeBindings>
      </schemaBindings>
   </bindings>
```

- This example uses the node attribute on the nodeBindings element; this is the most selective way to specify a node.
- Note that the configuration file is itself an XML file. In the node attribute, quotation marks must be escaped using &quot;

For more information on using configuration files, see the section "Configuration File".

Lists

A list type is represented using an array. During decoding, the array will be created to be the exact size needed for the number of items in the list. During encoding, the entire array will be encoded into the list, which means that you must correctly size the array when populating it.

Example XSD:

```
   <xsd:simpleType name="IntList" >
      <xsd:list itemType="xsd:int" />
   </xsd:simpleType>
```

C# and Java:

```
   {
      //constructor
      private IntList() {}

      public static String encode(int[] value, XBContext xbContext) {...}
      public static int[] decode(String text, XBContext xbContext) {...}
   }
```

Just as with the atomic types, a list type is represented using an array; the generated class only provides static coding methods.

Unions

Handling for a union depends on the member types. If the member types all map to the same Java type, then that type will be used. Otherwise, XBinder will use String. If any memberType is a list type, XBinder will use an array of the appropriate type (as just described).

Example XSD:

```
   <xsd:simpleType name="positiveFloat">
      <xsd:restriction base="xsd:float">
         <xsd:minExclusive value="0"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="NegIntPosFloat">
      <xsd:union memberTypes="xsd:negativeInteger positiveFloat"/>
   </xsd:simpleType>
```

C#:

```
   public class NegIntPosFloat
   {
      public static readonly int _member1MAX = -1;

      //constructor
      private NegIntPosFloat() {}

      public static String encode(string value, XBContext xbContext) {...}

      public static string decode(String text, XBContext xbContext) {...}
   }
```

Java:

```
   public class NegIntPosFloat
   {
      public static final int _member1MAX = -1;

      //constructor
      private NegIntPosFloat() {}

      public static String encode(String value, XBContext xbContext) {...}

      public static String decode(String text, XBContext xbContext) {...}
   }
```

The example defines a type that combines an integer-based type and a float-based type. XBinder would represent each of the memberTypes using a different C#/Java type, so it uses String as the representation.

Enumeration

If a type is restricted using an enumeration, an enum class is generated.

Example XSD:

```
   <xsd:element name="IntegerElement">
      <xsd:simpleType>
         <xsd:restriction base="xsd:integer">
            <xsd:enumeration value="-2" />
            <xsd:enumeration value="2" />
            <xsd:enumeration value="4" />
            <xsd:enumeration value="6" />
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
```

Beginning of resulting **Java** class:

```
public enum IntegerElement_ELEM
{
   x_2("-2"),
   x2("2"),
   x4("4"),
   x6("6");
   ...
```

Beginning of resulting **C#** class:

```
public class IntegerElement_ELEM
{
   public static IntegerElement_ELEM x_2 = new IntegerElement_ELEM("-2");
   public static IntegerElement_ELEM x2 = new IntegerElement_ELEM("2");
   public static IntegerElement_ELEM x4 = new IntegerElement_ELEM("4");
   public static IntegerElement_ELEM x6 = new IntegerElement_ELEM("6");
   ...
```

Common code:

```
      //fields for lexical and actual value
      private String lexicalValue;
      private int actualValue;

      //constructor
      private IntegerElement_ELEM(String lexicalValue) {...}

      public int valueOf() {...}

      public static IntegerElement_ELEM validate(int value) {...}

      public String toString() {...}

      public static String encode(IntegerElement_ELEM value, XBContext xbContext)
      {...}
```

```
        public static IntegerElement_ELEM decode(String text, XBContext xbContext)
        {...}
    }
```

Features to note are:

1. Each instance of the enum class contains both its lexical and actual value. This means no formatting is required when encoding, and no parsing is necessary when asking for the actual value.

2. The valueOf method returns the actual value (e.g. the int value represented by the enum).

3. The validate method returns the enum constant that corresponds to the given actual value.

4. The toString method returns the lexical representation.

5. The encode/decode methods are invoked by other generated code, wherever the restricted type is used.

The naming for the enum constants follows the following rules:

1. We begin with the enumeration value as the name.

2. If an enumeration identifier consists of whitespace only (for example, enumeration value=""), the special name BLANK is used.

3. Other special names are used for other single punctuation mark identifiers (for example, '+' = PLUS).

4. If, after applying these rules, the name still has a non-alphabetic start character, the character 'x' is prepended. (Thus, x2 for "2" in the example above).

5. All invalid C#/Java identifier characters are replaced with underscores (_) within the name. (Thus, x_2 for "-2" in the example above).

Chapter 5. XSD Complex Type Mappings

XSD complex types involve attributes, attribute wildcards (xsd:anyAttribute), elements, groups of elements, element wildcards (xsd:any), and character data.

For each complex type, a complex type class is generated. All of the above XSD features represent data, and all of this data is modeled in the complex type class using properties.

Properties

There are three kinds of properties generated: atomic properties, indexed properties, and list properties.

Atomic properties

An atomic property has a either a primitive type or a non-array, non-list, reference type.

Suppose that field is an optional property of primitive type (int, in this example). The following methods are generated:

Java:

```
public int getField() {...}

//only defined because field is optional
public boolean isSetField() {...}

public void setField(int value) {...}

//unsets field
public void setField() {...}
```

C#:

```
public int getField() {...}

//only defined because field is optional
public bool isSetField() {...}

public void setField(int value) {...}

//unsets field   public void setField() {...}
```

Note that a required, primitive-typed atomic property is always considered set, because the underlying field always has some value.

Now, suppose that *field*is an optional property of some reference type, *MyComplexType*. The following methods are generated:

Java:

```
public MyComplexType getField() {...}
```

```
//always defined because default value for reference type is null
public boolean isSetField() {...}

//pass null to unsetfield
public void setField(MyComplexType value) {...}
```

C#:

```
public MyComplexType getField() {...}

//always defined because default value for reference type is null
public bool isSetField() {...}

//pass null to unset field
public void setField(MyComplexType value) {...}
```

Indexed properties

An indexed property is represented using an array. If the property represents a list type, then the entire array is used. Otherwise, only a portion of the array is used, and the length of the property is managed internally. This is transparent to you.

Suppose that *field*is an indexed property of type T. The methods generated are:

Java:

```
//returns a copy of the array, a new array having length getFieldLength()
public T[] getField() {...}

public T getField(int index) {...}

public int getFieldLength() {...}

public boolean isSetField() {...}

//assigns field, making a copy of the given array
public void setField(T[] value) {...}

public void setField(int index, T value) {...}

//assigns field to a new array of the given length
public void setField(int length) {...}
```

C#:

```
//returns a copy of the array, a new array having length getFieldLength()
public T[] getField() {...}

public T getField(int index) {...}
public int getFieldLength() {...}

public bool isSetField() {...}

//assigns field, making a copy of the given array
public void setField(T[] value) {...}

public void setField(int index, T value) {...}

//assigns field to a new array of the given length
public void setField(int length) {...}
```

List properties

A list property is represented using a List or IList.

Suppose that field is a list property of type T (if T is a primitive type, then let T be the corresponding Java wrapper class or C# underlying struct). The following methods are generated:

Java:

```
public List<T> getField() {...}

public boolean isSetField() {...}

public void unsetField()[1] {...}
```

C#:

```
public IList<T> getField() {...}

public bool isSetField() {...}

public void unsetField()[1] {...}
```

XBinder's design for indexed and list properties takes its lead from JAXB's design. All mutation of the array and list can be under the control of the generated code, which might allow for earlier validation in the future.

Attributes

Attributes of simple, atomic types are modeled as atomic properties. Attributes of simple, list types (including unions having a memberType that is a list) are modeled as indexed properties.

During encoding, you must have set any required attributes. Attributes which are optional may be set or not. Attributes which are optional and have a fixed or default value must be set if you want the fixed or default value to actually be encoded. If an attribute has a fixed value, and it is set, it must be set to the fixed value.

Attribute wildcards are modeled as list properties of type `com.objsys.xbinder.runtime.XBAttributeBase`.

[1]Since List properties do not have a setField method, it seemed odd to define either a 0- or 1-argument setField method, just to perform the function on "unsetting" the property. Thus, we have the unsetField method.

Content Models

XSD supports a few content models:

1. empty: no child elements or child character data allowed
2. simple: No child elements are allowed; the character data must conform to a given simple type
3. non-mixed, group: The group describes the elements that are allowed; no character data may occur outside the child elements.
4. mixed, group: The group describes the elements that are allowed; character data may be mixed in, occurring outside of child elements.

Generated Class Hierarchy

At the root of the class hierarchy for all complex type classes is class `com.objsys.xbinder.runtime.XBComplexType`. This class contains a list of namespace-prefix mappings. This list represents the namespace declarations that will be/were encoded/decoded.

More details on the generated class hierarchy are discussed under the topics of extension and restriction.

Empty Content

A complex type with an empty content model may have attributes, but no other content. In this case, there will only be properties generated from attributes.

XSD:

```
<xsd:complexType name="Empty">
   <xsd:attribute name="attrint" type="xsd:integer"/>
   <!-- other attributes -->
</xsd:complexType>
```

Generated code:

Java:

```
public class Empty extends XBComplexType
{
   ...
   //attribute methods

   public int getAttrint() {...}

   public boolean isSetAttrint() {...}

   public void setAttrint(int value) {...}

   public void setAttrint() {...}

   [property methods for other attributes here]
```

```
      ...
   }
```

C#:

```
   public class Empty : XBComplexType
   {
      ...
      //attribute methods

      public int getAttrint() {...}

      public bool isSetAttrint() {...}

      public void setAttrint(int value) {...}

      public void setAttrint() {...}

      [property methods for other attributes here]
      ...
   }
```

Simple Content

This model forces the character data to conform to a simple type. Thus, a single value represents the content. XBinder models this by defining a property called "value". The value property will be defined just the same as would be done for an attribute of the same simple type.

A complex type with simple content may be defined in one of three ways:

1. extend a simple type

2. extend another complex type with simple content

3. restrict another complex type with simple content [2]

XSD:

```
   <xsd:complexType name="TypeName">
      <xsd:simpleContent>
         <xsd:extension base="BaseType">
             <!-- attributes declared here -->
         </xsd:extension>
      </xsd:simpleContent>
   </xsd:complexType>
```

Resulting code if *BaseType*is a simple type, mapped to a primitive type PT:

Java:

```
public class TypeName extends XBComplexType
   {
      ...
```

```
        //attribute methods
        ...
        //content methods

        public PT getValue() {...}

        public void setValue(PT value) {...}
        ...
    }
```

C#:

```
    {
        ...
        //attribute methods
        ...
        //content methods

        public PT getValue() {...}

        public void setValue(PT value) {...}
        ...
    }
```

Groups

In XSD, a group is a container for particles. Each particle may be an element, an element wildcard, or another group.

If a complex type has a group for its content model, then the properties that are generated for the complex type are derived from the particles in the group.

Representation of a Single Element

A single element is represented using some type *T*. If the element's type is a simple type, then *T*is determined as described in the section XSD Simple Type Mappings. If the element's type is a complex type, then *T*is a generated type, as described in this section, XSD Complex Type Mappings.

Representation of Element Wildcard

A single replacement for an element wildcard is represented using a String. The contents of the String will be raw XML (including character escaping).

The properties for element wildcards are named "any", "any_1", "any_2" etc..

Example **XSD**:

```
    <xsd:complexType name="MyType">
        <xsd:sequence maxOccurs="unbounded">
            <xsd:any namespace="##any"/>
            <xsd:any namespace="##other"/>
        </xsd:sequence>
    </xsd:complexType>
```

C# and Java:

```
{
   public String getAny() {...}

   public boolean isSetAny() {...}

   public void setAny(String value) {...}

   public String getAny_1() {...}

   public boolean isSetAny_1() {...}

   public void setAny_1(String value) {...}

   ...
}
```

In this example, a group class (MyType_1) is generated to represent the repeating sequence group. The methods shown are the property methods for the two element wildcards.

```
  MyType_1 myType = new MyType1();

myType.setAny("<room><closet>space&gt;20</closet></room>");
```

The value must begin with an element start tag, and XBinder will validate that the namespace of that element is valid according to the <xsd:any> namespace attribute.

If the XML includes namespace prefixes, use the addNamespace method on the complex type class to ensure that the namespace will be encoded with the desired prefix. See Controlling XML Namespaces for encoding.

Representation of a Group

The group may be a reference to a named group, or it may be an unnamed, nested group. Either way, XBinder will generate a class for it (see Group Classes).

A named group results in a class of similar name, with a suffix of "_Group" appended (this avoids name clashes with classes created for types).

A nested group is named based on the type under which it is ultimately nested. The name is of the form *ComplexType_S*, where S is a sequence number based on the location of the group relative in the overall construct.

Example **XSD:**

```
   <xsd:group name="MyGroup">
      <xsd:sequence>
         <xsd:element name="elem1" type="xsd:string"/>
         <xsd:element name="elem2" type="xsd:string"/>
         <xsd:sequence maxOccurs="3">
            <xsd:element name="elem3" type="xsd:string"/>
            <xsd:element name="elem4" type="xsd:string"/>
         </xsd:sequence>
       </xsd:sequence>
```

```
</xsd:group>
```

This XSD will result in a group class named MyGroup_Group (representing class MyGroup), and in a group class named MyGroup_Group_3 (representing the nested sequence group). The suffix is 3 because the nested sequence is the third item within the enclosing sequence. Had the nested sequence been placed directly after elem1 instead of elem2, then the suffix would have been 2.

Generating Properties for Particles

Let *T*be the type to represent a single occurrence of a particle, as just described above for elements, element wildcards, and groups.

If the particle does not repeat, we will have an atomic property of type *T*.

If the particle does repeat, we will have either an indexed property or a list property, of type *T*(that is, List/IList< *T*> or *T*[]). If the particle has maxOccurs > 100 or maxOccurs = unbounded, a list property is used; otherwise, an indexed property is used.

*T*may itself be a single-dimensional array type, so that the particle may be a two dimensional array. This will occur, for example, if you have a repeating element of a list type.

Sequence, All, and Choice Groups

XSD defines three kinds of groups that particles may belong to: sequence, all, and choice. Each of the particles in a group are modeled as just described in *Generating Properties for Particles*. However, the kind of group in which particles appear also affects the code that is generated.

An < `xsd:sequence`> group fixes the order in which the particles must appear. This is handled by the encode/decode methods. Similarly, an <xsd:all> group requires each of the particles to appear, but they may appear in any order. This is again handled by the encode/decode methods.

An < `xsd:choice`> group specifies a number of particles, any one of which may appear. In this case, XBinder generates an extra property that indicates which of the choices has been made.

Example choice group:

```
<xsd:complexType name="MyChoiceType">
   <xsd:choice>
      <xsd:element name="choice1" type="..."/>
      ...
      <xsd:element name="choiceN" type="..."/>
   </xsd:choice>
</xsd:complexType>
```

Abbreviated generated code:

Java:

```
public class MyChoiceType extends XBComplexType
{
   ...
   public enum Choice  {
      choice1, choice2, ..., choiceN}
   ...
```

```
      //content methods

      public int getChoice1() {...}

      public boolean isSetChoice1() {...}

      public void setChoice1(int value) {...}

      public void setChoice1() {...}
      ...
      public Choice getWhichField() {...}
   }
```

C#:

```
   public class MyChoiceType : XBComplexType
   {
      ...
      public enum Choice  {
         None,
         choice1,
         choice2, ...,
         choiceN
      }

      ...

      //content methods

      public int getChoice1() {...}

      public bool isSetChoice1() {...}

      public void setChoice1(int value) {...}

      public void setChoice1() {...}
      ...
      public Choice getWhichField() {...}
   }
```

Note the following features:

1. The usual property methods are generated for each of the particles in the choice group. However, since each particle is essentially optional, there will be an isSet method for every particle.
2. You may determine which field is set either by using one of the isSet methods, or by using the getWhichField method.
3. Setting the property for a particle automatically selects that particle as the chosen one.

Representation of Mixed Content

A "mixed" content model is represented as a single String of raw XML (ie, as it would appear in the XML file). The `AnyType`class is an example of what is generated for mixed content models (see the section called “anyType”).

When decoding mixed content, the events received by XBinder from the XML parser must be encoded back into XML, so that a string of XML can be made available to you, the programmer. As a result, in some cases, you will not see the exact, original XML. For example (admittedly a strange scenario), the original XML may have encoded the 'A' character using a character reference of "&#x41;" but since the character 'A' can be encoded as itself (unlike, for example, the '&' character), you will see it as an 'A' character.

When encoding mixed content, XBinder writes out the mixed content string exactly as-is. You must handle any character escaping yourself. For example, if you wish to encode a literal '<' character in some text, you must encode it as "&lt;". If you are embedding namespace prefixes, you will want to be sure the prefix you use is mapped to the correct namespace. To declare these namespaces on the parent element, use method `XBComplexType.addNamespace(nsUri, prefix)`. Otherwise, you may embed a namespace declaration in your mixed content string.

Mixed content handling can be turned off using the -nomixed command line option. In that case, XBinder compiles your schema as though it were first modified to have all mixed content models changed to non-mixed. Therefore, any instances you decode must be valid according to this imaginary, modified version of your schema, or else a validation error results.

Complex Content

The XSD *ComplexContent*element <*xsd:complexContent*> is used to create a modified version of a base type through extension or restriction mechanisms. It is similar in concept to creating derived types in Java or C++.

Type Substitution and the Type Hierarchy

The use of complexContent with *extension*or *restriction*creates a type hierarchy. XML Schema allows for type substitutions, by using the XML schema instance type attribute (xsi:type).

There are two ways XBinder will represent the XSD type hierarchy in the target language. One approach uses class extension. The second approach uses interfaces. With the extension approach, every subtype of *BaseType*will be modeled as a subclass of class *BaseType*. This means that any occurrences of type *BaseType*can be represented using a reference of class *BaseType*. This is what you would likely have expected.

With the interface approach, every subtype of *BaseType*, call it *BaseSubType*, will be modeled using class *BaseSubType*, but *BaseSubType*may or may not be a subclass of *BaseType*. Instead, *BaseSubType*will implement (directly or indirectly) an interface named *BaseType_derivations*. Any occurrences of type *BaseType*will be represented using a reference of interface *BaseType_derivations*.

The interface approach is used when some type *BaseRestr*either directly or indirectly restricts *BaseType*. In that case, it is inconvenient to have class *BaseRestr*be a subclass of *BaseType*(because it may have a very different content model.) Yet, in order to support type substitution, we still need to somehow represent the type hierarchy in the target language. This is accomplished using special "derivation interfaces", such as *BaseType_derivations*.

When the -noderiv option is chosen, the "derivation interfaces" will not be generated or implemented, and type substitution is not supported. If type *R*is a restriction of type *B*, their corresponding classes will have no relationship to each other.

Thus, there are two forms for a class that is a base type for some other type. The first does not involve any interfaces:

```
//-noderiv was used or there are no subtypes that are restrictions ofBaseType
public class BaseType
{
   ...
   /**
```

```
       * Factory method
       */
      public staticBaseType createObject(QName type) {...}
      ...
   }
```

In the second form, an interface is implemented:

Java:

```
   // -noderiv was not used and there is a subtype that is a restriction
   // of BaseType
   public class BaseType implements BaseType_derivations
   {
      ...
      /**
       * Factory method
       */
      public static BaseType_derivations createObject(QName type) {...}
      ...
   }
```

C#:

```
   // -noderiv was not used and there is a subtype that is a restriction
   // of BaseType
   public class BaseType : BaseType_derivations
   {
      ...
      /**
       * Factory method
       */
      public static BaseType_derivations createObject(QName type) {...}
      ...
   }
```

The factory method is used during decoding to create the correct class of object, based on the xsi:type attribute that was present.

There are likewise two forms for a derivations interface. If *BaseType*is at the root of the type hierarchy, it's derivations interface will be as follows:

Java:

```
   public interface BaseType_derivations
   {
      public void decode(XMLStreamReader] reader, XBContext xbContext,
         boolean isNilled, boolean hasDefault) ;

      public void encode(XBXmlEncoder encoder, XBContext xbContext,
         QName elemDeclType, boolean ignoreContent) ;
   }
```

C#:

```
public interface BaseType_derivations
{
   void decode(XmlTextReader reader, XBContext xbContext,
      bool isNilled, bool hasDefault) ;

   void encode(XBXmlEncoder encoder, XBContext xbContext,
      XBQualifiedName elemDeclType, bool ignoreContent) ;
}
```

Additionally, if *BaseType2*is a direct extension/restriction of *BaseType*, and some other type restricts *BaseType2*(so that a derivations interface must also be generated for *BaseType2*), then we have:

Java:

```
public interface BaseType2_derivations extends BaseType_derivations
{
}
```

C#:

```
public interface BaseType2_derivations : BaseType_derivations
{
}
```

If you have an reference to one of the _derivations interfaces, you can use the is operator (C#) or the instanceof operator (Java) to determine what the actual type is.

Extension

When a type is extended with *extension*, the new type may add attributes or append a group of elements. XBinder models this by generating a class that extends the class generated for the original type.

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:complexContent>
      <xsd:extension base="BaseType">
         <xsd:group[3]>
            <xsd:element name="elem1" type="Type1"/>
            <xsd:element name="elem2" type="Type2"/>
            ...
            <xsd:element name="elemN" type="TypeN"/>
         </xsd:group>
         <xsd:attribute name="attr1" type="Type1"/>
         <xsd:attribute name="attr2" type="Type2"/>
         ...
         <xsd:attribute name="attrN" type="TypeN"/>
      </xsd:extension>
```

[3]This may be "sequence" or "choice". A group reference may also be used, but it should refer to a sequence or choice group.

```
      </xsd:complexContent>
   </xsd:complexType>
```

Generated Java:

```
   public class TypeName extends BaseType
   {
      ...
      [methods for properties for attributes attr1..attrN]
      [methods for properties for elements elem1..elemN]
      ...
   }
```

Generated C#:

```
   public class TypeName : BaseType
   {
      ...
      [methods for properties for attributes attr1..attrN]
      [methods for properties for elements elem1..elemN]
      ...
   }
```

Restriction

It is possible to restrict elements and attributes in an existing content model group by using the *restriction*element. For either elements or attributes, it is possible to exclude optional items from the derived content model. It is also possible to restrict wildcards (*any*or *anyAttribute*) to contain values of a given type. It is also possible to further restrict facets such as *minOccurs*and *maxOccurs*to specify a narrower range than was defined in the base type.

For C# and Java, the class generated for a restriction is the same as would have been generated for an equivalent XSD type defined without restriction. Any attribute or elements that are inherited from the base type are generated into the restricted type's class.

Assuming -noderiv was not used, the generated class will implement the derivations interface for the restricted type's supertype.

The general mapping is as follows:

XSD:

```
   <xsd:complexType name="TypeName">
      <xsd:complexContent>
         <xsd:restriction base="BaseType">
            ...
         </xsd:restriction>
      </xsd:complexContent>
   </xsd:complexType>
```

Generated Java:

```
   public class TypeName extends XBComplexTypeimplements BaseType_derivations
   {...}
```

Generated C#:

```
   public class TypeName : XBComplexType, BaseType_derivations
   {...}
```

anyType

XSD type *anyType*is a complex type with mixed content, which is the ultimate base type for all complex types. XBinder represents *anyType*by generating a class called *AnyType*:

Java:

```
   public class AnyType extends XBComplexType
   {
      ...
      //attribute methods for attribute wildcard

      public List<XBAttributeBase> getAnyAttr() {...}

      public boolean isSetAnyAttr() {...}

      public void unsetAnyAttr() {...}

      //content methods for mixed content model
      public String getMixedContent() {...}

      public boolean isSetMixedContent() {...}

      public void setMixedContent(String value) {...}
      ...
   }
```

C#:

```
   public class AnyType : XBComplexType
   {
      ...
      //attribute methods for attribute wildcard

      public IList<XBAttributeBase> getAnyAttr() {...}

      public bool isSetAnyAttr() {...}

      public void unsetAnyAttr() {...}

      //content methods for mixed content model
      public String getMixedContent() {...}

      public bool isSetMixedContent() {...}
      public void setMixedContent(String value) {...}
      ...
   }
```

Class *AnyType*is used wherever *anyType*is referenced: if an element is declared to be of type *anyType*, class *AnyType*is used; if a complex type is defined as an extension of *anyType*, the generated class will extend class *AnyType*. Class *AnyType*is only generated when it is needed.

Regarding restriction on *anyType*, note that:

```
   <xsd:complexType name="MyType">
      <xsd:complexContent>
         <xsd:restriction base="xsd:anyType">
           XSD content group definition...
         </xsd:restriction>
      </xsd:complexContent>
   <xsd:complexType>
```

is equivalent to:

```
   <xsd:complexType name="MyType">
      XSD content group definition...
   </xsd:complexType>
```

Therefore, a restriction on anyType is handled as a normal complex type definition, as discussed elsewhere.

Named Groups

The XSD *group*element < *xsd:group*> is used to create a reusable content model group. A group declaration is translated into a class. The class name will include a _Group suffix to avoid collisions with classes created for types of the same name. The group class is used wherever the group is referenced.

XSD:

```
   <xsd:group name="GroupName">
      XSD content group definition ...
   </xsd:group>
```

Java/C# code:

```
   public class GroupName_Group {
      [properties for particles from group]
      ...
   }
```

Element Substitution Groups

Substitution groups are very similar to choice types. They allow a given base element (referred to as the substitution group head) to be replaced with a different element. The replacement element is designated as being part of the group through the use of the XSD substitutionGroup attribute.

For example, the following element declarations declare a group in which the head element (Publication) would be replaced with either the Book element or Magazine element:

```
<xsd:element name="Publication" abstract="true"
type="PublicationType"/>

<xsd:element name="Book" substitutionGroup="Publication"
type="BookType"/>

<xsd:element name="Magazine" substitutionGroup="Publication"
type="MagazineType"/>
```

In these declarations, the types *BookType*and *MagazineType*must be derived from the substitution group head type (in this case, *PublicationType*). This now allows Book or Magazine to be used anywhere Publication was declared to be used (in fact, the elements in this case must be Book or Magazine because Publication was declared to be abstract and therefore cannot appear in an XML instance).

XBinder generates a special group class to hold each of the substitution group alternative elements. This class is generated as for an <xsd:choice> whose particles consist of each of the elements in the substitution group. The format of the name for the special group class is "_<element>SG", where <element> would be replaced with the name of the substitution group head element. In the example above, the generated type name would be "_PublicationSG".

Chapter 6. Configuration File

The default bindings of source schema components to C#/Java types as presented above may not meet the requirements of all applications. In such cases, the default bindings can be customized by using a configuration file. This is sometimes refered to as a *binding schema* in similar products. A configuration file contains binding declarations which are specified by a *binding language*, the syntax and semantics of which are defined in this section.

Binding Language

The binding language is an XML based language that defines constructs referred to as *binding declarations*. A binding declaration can be used to customize the default binding between an XML schema component and its C#/Java representation.

The schema for binding declarations is defined in the namespace `http://www.obj-sys.com/XBConfig`.

Binding Declaration

The configuration file enables customized binding without requiring modification of the source schema. The schema component to which the binding declaration applies must be identified explicitly. Minimally, a configuration file is of the following form:

```
<bindings version=”1.0">
   <schemaBindings namespace | schemaLocation = "xsd:anyURI">
      <nodeBindings name | node = "xsd:string">*
          <node bindings declaration>
      <nodeBindings>
   </schemaBindings>
</bindings>
```

The `schemaBindings` node has the attribute `namespace` or `schemaLocation` to refer to a schema. The `namespace` attribute is used to specify a schema using its target namespace. The `schemaLocation` attribute specifices a schema using its physical file location.

The `nodeBindings` node has attributes `name` and `node` to construct a reference to one or more nodes within the schema. The `name` attribute specifies a node using its QName. The `node` attribute uses an XPath expression to specify a set of nodes.

A summary of these attribute values follows:

`namespace`: A reference to a schema’s target namespace.

`schemaLocation:` A URI reference to an XML schema document.

`name`: The qualified name (QName) of a node within the schema.

`node`: An XPath 1.0[1] expression that identifies the node(s) within a schema with which to associate binding declarations.

An example of a configuration file can be found in the section “Configuration File Example”.

Version Attribute

The normative binding schema specifies a global `version` attribute. This is used to identify the version of the binding declarations. For example, a future version of this specification may use the version attribute to specify backward

[1] XML Path Language (XPath) Version 1.0 [http://www.w3.org/TR/xpath]

compatibility. For this version of the specification, the `version` must always be `1.0`. If any other version is specified, the configuration file will be skipped.

The `version` attribute must be specified in the root `<bindings>` element in the configuration file:

```
    <bindings version="1.0" ... />
```

Configuration File Language Overview

A binding declaration customizes the default binding of a schema element to a C#/Java representation. The binding declaration defines one or more customization values, each of which customizes a part of C#/Java representation.

Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. The scope of a customization value is the set of schema elements to which it applies.

The defined scopes are as follows:

- **global scope:** A customization value defined in `<bindings>` has *global scope*. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
- **schema scope:** A customization value defined in `<schemaBindings>` has *schema scope*. A schema scope covers all the schema elements in the target namespace of a schema.
- **node scope:** A customization value defined in `<nodeBindings>` has *node scope*. A node scope covers all schema elements that reference the type definition, the global declaration, or the local declaration.

A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- A schema element in schema scope inherits a customization value defined in global scope.
- A schema element in node scope inherits a customization value defined in schema or global scope.

Likewise, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- A value in schema scope overrides a value inherited from global scope.
- # value in node scope overrides a value inherited from schema scope or global scope.

Global Bindings: `<bindings>`

The customization values in the `<bindings>` declaration have global scope. These affect all elements within all schemas defined in the compilation project.

Usage

```
   <bindings version=”1.0”>
          [<prefix>xs:token</prefix>]
          [<schemaBindings>. . .</schemaBindings>]
          [<nameXmlTransform>. . .</nameXmlTransform>]
          [<doubleFormat/>]
          [<decimalFormat/>]
          [<floatFormat/>]
          [<typemap>. . .</typemap>]
          [<reservedWords>. . .</reservedWords>]
          ...
   </bindings>
```

The following attributes are defined for the <bindings> node:

version	See the section "Version Attribute" above for details.

The following customization elements may be defined within the global scope:

prefix	This is used to specify a prefix that is prepended to all XML names including type names and global element names to form C#/Java type and variables names. It should be a legal C#/Java identifier.
schemaBindings	This is used to identify individual schemas for schema scope binding declarations (see the section called "`<schemaBindings>` Declaration"). It can be specified multiple times, but only once per schema.
nameXmlTransform	This is used to perform more accurate XML names transformation than *prefix* allows. See the section called "Advanced XML Names Transformation" for further details.
doubleFormat	This specifies a default (global) format for encoding values of "double" type. See the section called "XML Numeric Values Format Specification".
decimalFormat	This specifies a default (global) format for encoding of values of "decimal" type. See the section called "XML Numeric Values Format Specification".
floatFormat	This specifies a default (global) format for encoding of values of "float" type. See the section called "XML Numeric Values Format Specification".
typemap	This specifies a default (global) mapping of a specific XSD type to a C#/Java type (see the section called " `<typemap>` *Declaration* "). It can be specified multiple times. In each typemap declaration, a space-separated list of XSD types can be mapped to one C#/Java type.
reservedWords	This element is used to add additional reserved words to the reserved words list. These are words that are defined in the output target language (for example, C# or Java). XBinder will alter these words when they are defined in a schema file so there is not a name clash in the output file. By default, all reserved words defined by the target language are included in this table. The reserved word list is specified as a space-separated list of words.

`<schemaBindings>` Declaration

The customization values in `<schemaBindings>` binding declarations have schema scope. These apply to all elements within the referenced XML schema document.

Usage

```
    <schemaBindings namespace | schemaLocation="xs:anyURI">
           [<prefix>xs:token</prefix>]
           [<sourceFile>xs:anyURI</sourceFile>]
           [<nameXmlTransform>. . .</nameXmlTransform>]
           [<doubleFormat/>]
           [<decimalFormat/>]
```

```
          [<floatFormat/>]
          [<nodeBindings>. . .<nodeBindings>]
          [<typemap>. . .</typemap>]
          ...
    </schemaBindings>
```

The following attributes are defined for `<schemaBindings>`:

namespace:	A URI reference to a schema's target namespace. The processor will look at the target namespace in all of the schemas currently being compiled for a match with the given namespace.
schemaLocation:	URL as it is used in `<xsd:import>` or `<xsd:include>` statements. In the latter case, `<sourceFile>` should be provided, to map the schema URL to an actual schema file. XBinder does not have the capability to automatically reference schemas remotely; therefore, any imported or included schemas must have been downloaded in advance and be present on the user's computer.

The following customization values are defined in schema scope:

prefix:	This is used to specify a prefix that is prepended to all XML names, including type names and global element names, to form C#Java type and variable names. It should be a legal C#/Java identifier.
sourceFile:	The actual schema file path. XBinder does not have the capability to automatically reference schemas remotely; therefore, any imported or included schemas must have been downloaded in advance and be present on the user's computer. This element is used to map a schema URL to a file on the local system.
nameXmlTransform:	This is used to perform more accurate XML names transformation than *prefix* allows. See the section called "Advanced XML Names Transformation" for further details.
doubleFormat:	This specifies a schema-level format for encoding of values of "double" type. See the section called "XML Numeric Values Format Specification".
decimalFormat:	This specifies a schema-level format for encoding of values of "decimal" type. See the section called "XML Numeric Values Format Specification".
floatFormat:	This specifies a schema-level format for encoding of values of "float" type. See the section called "XML Numeric Values Format Specification".
nodeBindings:	Node scope binding declarations (see the section called "`<nodeBindings>` Declaration"). This element can be specified multiple times, but only once per definition.
typemap:	This specifies a schema-level mapping of a specific XSD type to a C#/Java type (see the section called "

	`<typemap>` *Declaration* ”). It can be specified multiple times. In each typemap declaration, a space-separated list of XSD types can be mapped to one C#/Java type.

`<nodeBindings>` Declaration

The customization values in the `<nodeBindings>` binding declaration have node scope. These refer to individual type or element definitions within a schema. It is also possible to reference local elements within complex types for customization.

Usage

```
<nodeBindings name | node=”xs:string”>
       [<prefix>xs:token</prefix>]
       [<nameXmlTransform>. . .</nameXmlTransform>]
       [<array [maxSize=”xs:nonNegativeInteger”/>]]
       [<isBigInteger/>]
       [<isDynamic/>]
       [<ctype> string | numeric </ctype>]
       [<noPatternTest/>]
       [<numericFormat>. . .</numericFormat>]
       [<nodeBindings>. . .<nodeBindings>]
       ...
</nodeBindings>
```

The following attributes are defined for `<nodeBindings>` node:

name:	This attribute selects a node for configuration processing based on its QName.
node:	An XPath 1.0 expression that identifies the schema node within the referenced schema with which to associate binding declarations

The following customization values are defined in node scope:

nameXmlTransform	This is used to perform more accurate XML names transformation than *prefix* allows. See Section “Advanced XML Names Transformation” for further details.
prefix	This is used to specify a prefix that is prepended to all XML names including type names and global element names to form C#/Java type and variables names. It should be a legal C#/Java identifier.
array	This specifies that an array should be used instead of a linked list for repeated elements. The *maxSize* attribute specifies the maximum size of the array. The default value if not specified is 100.
isBigInteger	By default, XBinder represents xsd:integer as an int. This option specifies that System.decimal or java.lang.BigInteger should be used instead. This qualifier can be applied to either an integer or complex type. In the latter case, all integer elements within the complex type are flagged as big integers.

cstype javatype	This is used to control the C#/Java type generated by the XBinder compiler. These elements also occur in the context of a `typemap` and they are discussed in the section called “ `<typemap>` *Declaration* ”.
noPatternTest	If type uses a pattern facet, this may be used to turn off the pattern match test.
numericFormat	This specifies a node-level format for encoding of numeric values. Affects values of `xsd:double`, `xsd:decimal`, and `xsd:float` types. See the section called “XML Numeric Values Format Specification”.
nodeBindings	Nested `<nodeBindings>` declarations to allow more accurate references to enclosed elements, such as local elements inside groups (*sequence*, *all*, *choice*, *group*, etc).

`<typemap>` *Declaration*

The customization values in `<typemap>` binding declarations are used to map a specific XSD type, or a space-separated list of XSD types, to a C#/Java type. This can be done at global or schema level. This mapping configuration can be used to preserve the format of floating point numbers after decoding and reencoding.

Usage

```
<typemap>
       [<xsdtype>. . .</xsdtype>]
       [<cstype>. . .</cstype>]
       [<javatype>. . .</javatype>]
       ...
</typemap>
```

<xsdtype> is used to specify the XSD Type being mapped, and <cstype> or <javatype> are used to specify the C# or Java type. For example, to map xsd:decimal, xsd:double and xsd:float types to string:

```
<typemap>
       <xsdtype>decimal double float</xsdtype>
       <cstype>string</cstype>
       <javatype>string<javatype>
</typemap>
```

It is possible to specify multiple mappings.

The valid values for `cstype` and `javatype` are: byte, int16, uint16, int32, uint32, string. The values are not target language names, but have an obvious correspondence to target language types. The unsigned types (u*) are not legal values for `javatype`.

Advanced XML Names Transformation

The advanced XML names transformation allows a prefix or suffix to be added to type or element names.

Usage

```
<nameXmlTransform>
      [<typeName [prefix=”xs:token”] [suffix=”xs:token”]/>]
      [<elementName [prefix=”xs:token”] [suffix=”xs:token”]/>]
```

```
</nameXmlTransform>
```

It is possible to specify separate prefixes and suffixes for type names and element names. If `<typeName>` is used then values of the optional attributes “prefix” and “suffix” will be applied to all custom types in the scope of this transformation. If <elementName> is used then prefix and suffix will be applied to element names.

XML Numeric Values Format Specification

This section addresses customization of the encoding format of numeric values (XSD *double*, *decimal*, or *float* types). It is sometimes necessary to have numbers formatted in a certain way. For example, a decimal value of 12 may need to be formatted as "+0012.00". It is possible using these qualifiers to specify the exact required format of such values. This can be done at any scope - global, schema or node. To customize the format of all decimal, double or float values at the global or schema level, use the `<decimalFormat>`, `<doubleFormat>` and `<floatFormat>` configuration elements respectively. For the node scope, use `<numericFormat>` element.

Usage

```
<decimalFormat | doubleFormat | floatFormat | numericFormat
       [totalDigits=”xs:byte”]
       [fractionDigits=”xs:byte”]
       [fractionMinDigits=”xs:byte”]
       [integerMaxDigits=”xs:byte”]
       [integerMinDigits=”xs:byte”]
       [expSymbol=”xs:token”]
       [expMinValue=”xs:short”]
       [expMaxValue=”xs:short”]
       [expDigits=”xs:byte”]
       [signPresent=”xs:boolean”]
       [pointPresent=”xs:boolean”]
       [expSignPresent=”xs:boolean”]
       [expPresent=”xs:boolean”]
       >
```

The application of these attributes varies according to the XSD type. The formatting of float/double values and the formatting of decimal values are discussed in detail in the next two sections.

Warning

The application of this configuration information to C# and Java differs from C and C++.

Formatting xsd:double/float Values

Overview

Formatted numeric strings may have one of two forms: decimal notation (no exponent) or scientific notation (“E” notation). Scientific notation uses the form:

```
[sign] mantissa E [sign] exponent
```

where mantissa is written in decimal notation, and exponent is an integer. The choice of decimal or scientific notation depends on the choice of exponent and on the `expPresent` parameter. By default, if the selected exponent is zero, the exponent is not shown (resulting in decimal notation). However, `expPresent` can be used to force display of the exponent, even when it is zero.

Overview of Options

totalDigits integerMaxDigits fractionDigits	These each limit the number of digits, either overall or in the integer or fraction part. They do not count leading zeros in the integer part, nor trailing zeros in the fraction part. totalDigits, if specified must be at least 1. integerMaxDigits and fractionDigits must be zero or more. Note that the default for totalDigits is based on the datatype (double vs. float).
integerMinDigits	This specifies the minimum number of digits to display in the integer part. If necessary, leading zeros are added. Using 0 means that the integer part, if 0, will not be displayed. Using 1 forces the display of a leading zero when the integer part is 0. The default value is 1. Note that integerMinDigits > integerMaxDigits is allowed (this will force some leading zeros).
fractionMinDigits	This specifies the minimum number of digits to display in the fraction part. If necessary, trailing zeros are added. Note that fractionMinDigits > fractionDigits is allowed (this will force some trailing zeros). This option may produce strange looking results, as it may make a number appear to be more precise when digits have been rounded away. For example, we may round 12.345 to 12.35 due to a totalDigits setting of 4, but format this as “12.3500” due to a fractionMinDigits setting of 4.
expMinValue expMaxValue	These specify exponential notation and control the range of exponents that may be used. The location of the decimal point in the mantissa is adjusted accordingly. When used, integerMaxDigits is ignored.
expSymbol	Specify use of 'E' or 'e' for the exponent symbol. Specify '0' to prevent falling back on exponential notation for large magnitude values (see details in the section on Standard Decimal Notation, below).
signPresent expSignPresent	Specify `1` or `true` to force the display of signs.
expPresent	Specify `1` or `true` to force the display of a zero exponent. Note that this will force display of a zero exponent even if decimal notation as been selected.
expDigits	The minimum number of digits to use in the exponent. If expDigits >= 1 is specified, this option forces exponential notation. If specified as 0, this option forces decimal notation.
pointPresent	Specify `1` or `true` to force the display of the decimal point. Note that specifying fractionMinDigits > 0 will also force the display of the decimal point.

Exponential Notation: Method 1

The first way to specify exponential notation is by specifying at least one of the expMinValue or expMaxValue options.

If possible, the mantissa will be normalized such that $1 \le \text{mantissa} \le 10$ ("normalized scientific notation"). If that requires an out-of-range exponent, then expMinValue/expMaxValue will be used for the exponent.

When either of these options (expMinValue or expMaxValue) are specified, the integerMaxDigits setting is ignored.

We attempt to honor the totalDigits setting by rounding off fractional digits of the mantissa. This is not possible, when expMaxValue is specified, for values such that $|\text{value}| > 10^{\text{totalDigits + expMaxValue}}$. In that case, we will round integer digits to zero. For example, with totalDigits=2, expMaxValue=3, and value = 234567, the formatted result would be "230E3".

The fractionDigits setting is always honored. Both fractionDigits and totalDigits may cause rounding off of fractional digits in the mantissa.

Note that using expMinValue may cause some smaller values to round to zero. In particular, values such that $|\text{value}| < 5 \cdot 10^{\text{expMinValue - fractionDigits - 1}}$ will round to zero (substitute totalDigits for fractionDigits, if fractionDigits was not specified). For example, suppose expMinValue = -2 and consider the value 4.23E-6. This value is precisely .000423E-2, which is formatted as "0E-2", given totalDigits (or fractionDigits) = 3. Likewise, 4.567 rounds to "0E3" with expMinValue = 3 and totalDigits = 2 (4.567 = .004567E3).

To summarize, using expMinValue and expMaxValue allows you to fix the exponent range, but a side effect is that it may present fewer significant digits than with normalized scientific notation. However, note that these rules imply that all values where $10^{\text{expMinValue}} \le |\text{value}| \le 10^{\text{expMaxValue + 1}}$ will be presented in normalized scientific notation.

Tip

If you specify expMaxValue = 0 and expPresent=false, if $1 \le |\text{value}| \le 10^{\text{totalDigits}}$, the value will appear in decimal notation (the exponent will be 0 and hidden), while smaller values will use normalized scientific notation. Larger values, however, will appear in decimal notation with trailing zeros in the integer part (e.g. 8900000).

Exponential Notation: Method 2

The second way to specify exponential notation is through the use of the expDigits option. If you specify expDigits ≥ 1, then exponential notation will be used. The integerMaxDigits option controls the location of the decimal point and thus the choice of exponent. You must not specify either the expMinValue or the expMaxValue for this method.

By default (i.e., when no integerMaxDigits value is specified), the exponent will be normalized such that $1 \le |\text{mantissa}| < 10$. However, if you specify integerMaxDigits ≥ 0, this will affect the location of the decimal point, and thus the choice of exponent.

If you specify integerMaxDigits = 0, the integer part will be 0. Whether this digit is displayed or not depends on the integerMinDigits value (≥ 1 means a leading zero).

If you specify integerMaxDigits = 1, the default behavior results (normalized exponent).

If you specify integerMaxDigits > 1, we have some freedom on where to place the decimal point. We follow these rules:

- Use at least 1 integer digit.
- Use at least as many integer digits as totalDigits – fractionDigits. This avoids losing significant digits due to rounding.

- If integerMinDigits > 0 is specified and using this many integer digits does not require a negative exponent, use at least this many integer digits. If this does require using a negative exponent, and some lesser number of integer digits does not, use at least the lesser number of digits. This avoids unnecessary padding with leading zeros, but also avoids moving the decimal point beyond its “natural” location (where it would appear in decimal notation).
- Use no more than the number of integer digits prescribed by the preceding rules, and of course, no more than integerMaxDigits.

Specifying integerMinDigits > integerMaxDigits forces padding with at least 1 leading zero and is allowed. Using totalDigits or fractionDigits will cause rounding after the appropriate number of digits. Some examples:

totalDigits	**fractionDigits**	**integerMaxDigits**	**integerMinDigits**	**value**	**result**
7	7	1	3	456.789	“004.56789E2”
7	7	2	3	456.789	“004.56789E2”
7	7	2	2	456.789	“045.6789E1”
7	4	4	0	9876.54321	“987.6543E1”

Standard Decimal Notation

If you specify expDigits = 0 and do not specify either expMinValue or expMaxValue, decimal notation will be used. In this case, values should be constrained as follows:

- $|\text{value}| < 10^{\text{totalDigits}}$
- $|\text{value}| < 10^{\text{integerMaxDigits}}$ (if integerMaxDigits is specified)

If either of these constraints is violated, rather than produce an error (as would be the case for xsd:decimal types), we have fall-back behavior, depending on the expSymbol option:

- If expSymbol is set to '0', then we will format the value in decimal notation, using as many integer digits as allowed, and then as many trailing zeros as needed to indicate the correct magnitude of the number.
- Otherwise, we will format the value using exponential notation, with as many integer digits as allowed (getting as close to the decimal representation as possible) and an exponent that conveys the magnitude. In this case, the extra digits fall into the fraction part, which is rounded away.

Fractional digits will be rounded, based on totalDigits and/or fractionDigits. Values, such that $|\text{value}| < 5 \cdot 10^{-\text{fractionDigits} - 1}$, will round to zero.

Mixed Notation

If you do not specify expMinValue, expMaxValue, or expDigits, then mixed notation is used. Mixed notation basically means that we use exponential notation with a normalized exponent, except that for some values, we use an exponent of zero (which is only displayed if expPresent is true).

Use an exponent of zero iff: $1 \le |\text{value}| \le 10^{10}$. Otherwise, use the normalized exponent.

Consider carefully the use of integerMaxDigits or fractionDigits with mixed notation, as the notation affects rounding. For example, 0.1235 is rounded to “0.124” when using 3 fraction digits in decimal notation, but is presented without rounding when using exponential notation with 3 fraction digits: “1.235E-1”. With mixed notation, rounding will not occur in the same decimal location for all numbers, neither will it occur after a given number of significant digits for all numbers. Trailing integer zeros will be used if either of the following conditions occur:

- $|\text{value}| \ge 10^{\text{totalDigits}}$

- |value| ≥ $10^{\text{integerMaxDigits}}$ (if integerMaxDigits is specified)

Formatting xsd:decimal Values

The formatting of xsd:decimal values differs from xsd:double values, and is much simpler. This because the decimal type may not use exponents.

Overview of Options

totalDigits	This limits the number of digits in the number. Leading zeros in the integer part and trailing zeros in the fraction part do not count (as with the XML Schema totalDigits facet).
fractionDigits	This limits the number of fraction digits in the number. Trailing zeros in the fraction part do not count (as with the XML Schema fractionDigits facet)
integerMaxDigits	Not used with xsd:decimal.
integerMinDigits fractionMinDigits	These specify the minimum number of digits to display in the integer and fraction parts. For the integer part, leading zeros may be added. For the fraction part, trailing zeros may be added.
signPresent	Formatting option to force display of positive signs.
pointPresent	Force the display of the decimal point. Note that specifying fractionMinDigits > 0 also will force the display of the decimal point.

Rounding

Fractional digits are rounded off to meet the given totalDigits and fractionDigits settings. In some cases, rounding causes carryover and adds an additional integer digit, which may then violate the totalDigits setting. See the discussion of validation below.

value	**totalDigits setting**	**fractionDigits setting**	**result**
456	2	any	violates totalDigits
999.999	3	any	violates totalDigits (after rounding, 4 integer digits are needed)
999.99	4	any	“1000”

Formatting Ouput

After rounding, leading and trailing zeros are added, as needed, to meet the integerMinDigits and fractionMinDigits parameters. Note that integerMinDigits and fractionMinDigits are not limited by totalDigits or fractionDigits (since these add leading/trailing zeros).

If the integer portion is zero, integerMinDigits controls whether the zero is displayed. If integerMinDigits = 0 was specified, it is not displayed. In all other cases, the zero will be displayed.

Note that fractionMinDigits, in particular, may produce strange looking results, as it may make a number appear to be more precise when digits have been rounded away. For example, we may round 12345.67 to 12346 due to a totalDigits setting of 5, but format this as “12346.00” due to a fractionMinDigits setting of 2.

Relationship to totalDigits/fractionDigits Facets

If the value has a totalDigits or fractionDigits facet, the facet will be used instead of any value specified in the configuration file.

Relationship to Validation

The only possible validation violation is for a totalDigits setting to be violated by a value such that $|value| \geq 10^{totalDigits}$. For all other values, rounding is used to avoid validation violations.

If the totalDigits setting being applied comes from a totalDigits facet and -lax is not in use, an error will occur. In all other cases, the violation of the totalDigits setting will be ignored. However, the totalDigits setting will still serve to limit the number of fractional digits – there will be 0 fractional digits (since the integer digits will consume all of the available digits, and more).

Default Behavior

The default behavior (when no configuration options or facets are present) is to format the value using as many digits as needed for an exact representation.

Example

integerMinDigits	fractionMinDigits	signPresent	value	result
4	2	true	12	"+0012.00"

Configuration File Example

The following is an example of a configuration file for a framework consisting of two schemas:

```
<bindings version="1.0">
   <schemaBindings
      schemaLocation=
         "http://www.example.org/whizbang/schema/sales.xsd">
      <sourceFile>C:\XBinder\xsd\whizbang\sales.xsd</sourceFile>

      <!-- Names for this schema prefixed with SALES_ -->
      <prefix>SALES_</prefix>

      <!-- Any element named myGlobalElem is given a prefix and forced
           to an unsigned 16-bit integer representation.
      -->
      <nodeBindings node="//xsd:element[@name='myGlobalElem']">
         <prefix>GE_</prefix>
         <cstype>uint16</ctype>
      </nodeBindings>
   </schemaBindings>

   <schemaBindings schemaLocation="product.xsd">

      <nodeBindings node="//xsd:element[@name='Product']">
         <prefix>prod</prefix>
      </nodeBindings>
      <!-- Any type or element named itemRecord is given a prefix -->
      <nodeBindings name="itemRecord">
         <prefix>ZZZ_</prefix>
      </nodeBindings>

      <!-- Force all elements p, q, r defined anywhere under type ProductType
           to be represented as System.Decimal/java.math.BigInteger
      -->
      <nodeBindings node="//xsd:complexType[@name='ProductType']">
         <nodeBindings node=".//xsd:element[@name='p']">
            <isBigInteger/>
         </nodeBindings>
         <nodeBindings node=".//xsd:element[@name='q']">
            <isBigInteger/>
         </nodeBindings>
         <nodeBindings node=".//xsd:element[@name='g']">
            <isBigInteger/>
         </nodeBindings>
      </nodeBindings>
   </schemaBindings>
</bindings>
```

Chapter 7. Miscellany

Canonical XML

XBinder can help you create documents that follow the W3C Canonical XML Recommendation. XBinder for C and C++ have a -c14n command line option for this purpose. For Java and C#, however, you programmatically turn on Canonical XML support by invoking the method setCanonicalXML() on your XBXmlEncoder object.

Turning on Canonical XML support has the following effects:

- Namespace declarations will appear before attributes.
- Namespaces declarations and attributes will be sorted.
- All start tags will have a matching end tag.
- Character escaping will be done according to Canonical XML (this is actually the standard behavior).
- The indendation and line terminators that are normally added to the encoding to improve readability will be omitted.
- The XML prolog will be omitted ("<?xml version="1.0" ...?>").

Additionally, you must take some care to comply with the Canonical XML recommendation. You must use UTF-8 character encoding (the zero argument constructor for XBXmlEncoder does this). You must not add superfluous namespaces into your complex-type objects. Finally, for mixed content and xml:any types (where you provide raw XML that passes directly into the output), you must ensure that the raw XML you provide complies with the Canonical XML recommendation.