XML Object Binder

User Guide

Doc Version 1.6
XOB Version 3.1

http://xob.sourceforge.net/

Table of Contents

Introduction....3
  What is the point of this ?....3
  Terms....3
  What XOB does....3
  Dependencies....5
    Apache Xerces parser....5
  Jar files....5
Licence....5
Using....6
  Simple example XML and interfaces....6
  Reading XML (unmarshalling)....7
    DTD validation....7
    Schema validation....8
    Filtering....8
  Creating XML objects....9
    Namespace....10
  Writing XML (marshalling)....11
    DTD....11
    Schema....11
    Validating XML Object structure....11
  Using the interface generator (XOBGen)....12
    Example java -jar usage....13
    Example Ant task usage....14
    Special features....14
    Using the maven2 plugin....15
  Using the schema HTML doucmentation generator (XSDDoc)....16
    Example java -jar usage....16
    Example Ant task usage....17
  Writing interfaces....18
    Interface names....18
    Tags....18
      Child tags....18
        Namespace....19
      Tag namespace and naming....19
    Attributes....19
      Attribute namespace....20
    Tag value....20
  Examples....20
Tested XML parsers....20
Configuration....20
Author/contact....21

Introduction

XOB is short for XML Object Binder which lets you map interfaces to XML tags in XML documents and have them automatically implemented on the fly when unmarshalling an XML file into objects represented by your interfaces. XOB also allows you to create interface implementations through a factory interface, set values through the interfaces and then marshal into an XML file. This might sound a bit cryptic, but it will become clear if you continue to read this document.

The examples catalog contains enough examples to get you going if you hate reading documentation. Do however read the “Examples” section at the end of this document first if you want to run the examples.

What is the point of this ?

When reading an XML file using the JAXP and DOM APIs directly all XML file elements and attributes are referenced by a string (ex: document.getElement(“orderDate”);). If you misspell this string reference (which is easily done) the result is a null value returned at runtime. You can't easily see what valid references are available in each getElement() when you are coding.

When an XML element is bound to a Java object there is a specific getter for each subelement and attribute. Any misspelling is cauth by the compiler at compiletime. In most IDEs entering the name of the bound Java XML object and then a dot will give you a menu of valid choices.

When used for creating an XML file it is much harder to produce an incorrect XML file (though still possible – with effort :-). It is impossible to set an incorrect attribute. It is impossible to set an incorrect subelement. It is however possible to add an incorrect nubmer of subelements.

In general working with a Java object hierarchy is much easier than using something like the DOM API.

Terms

XML Object – A java object representing an XML element, defined by an interface, and implemented by XOB.

What XOB does

- Unmarshals any XML InputStream into java “XML objects”.
    - Doing no validation.
    - Doing DTD validation.
    - Doing Schema validation.
- Creates “XML objects”.
- Validate “XML objects”.
- Marshals “XML object” as XML to any OutputStream.
- Generates “XML Object” interfaces from XML Schema files.
- Generates HTML documentation for XML Schema files.
- Gives you control over the naming of the XML objects, which doesn't have to be identical to its

XML element.

- Allows you to filter XML data when accessing XML objects.
- Allows you to override some schema values using comments when generating XOB interfaces from an XML schema. This is useful in conjunction with filters. A returns many can for example be overridden to a returns one when that is the case when a filter is applied.

Dependencies

XOB requires an XML parser, preferably one that supports JAXP 1.1 or JAXP 1.2. It is possible to use other parsers, but then you have to implement a wrapper for the non JAXP parser that implements the xob.xml.XMLParser interface, and set that parser on the XMLObjectBinder instance.

If you use a JAXP parser you must have one that implements JAXP 1.2 to be able to use schema validation.

To use a JAXP parser, just make it available in the CLASSPATH. The default internal XMLParser implementation uses the javax.xml.parsers.* APIs.

Apache Xerces parser

The XOB distribution includes the Xerces parser from Apache (http://xml.apache.org/xerces2-j/index.html). It is used for building and for some of the examples.

Jar files

The XOB binary distribution contains the following 4 jar files:

xob.jar	This is the main XOB implementation. This jar file is required runtime if you are going to use XOB! This is ALL that is needed runtime! Just drop this jar into .../lib/ext or somewhere in your project and add it to the classpath.
xobxsd.jar	This jar file contains common XML schema code used by both xobgen.jar and xsddoc.jar.
xobgen.jar	This generates XOB interfaces from an XML schema file. This jar is executable with “java -jar xobgen.jar”. It also contains an Ant task for use in Ant build scripts.
xsddoc.jar	This generates HTML documentation for XML schemas. This jar is executable with “java -jar xsddoc.jar”. It also contains an Ant task for use in Ant build scripts.
xob-m2-plugin.jar	Since version 3.1 XOB is now built with maven2. This jar is a maven2 plugin for running xobgen in a simple way. The xobgen Ant task can also be used with maven2 via the antrun plugin, but having a real maven plugin is both simpler and more flexible.

Licence

From version 3.1 this code is released under the Apache Software License 2.0, which is included as a text file in the binary distribution and also available at http://www.apache.org/licenses/LICENSE-2.0.

Using

Using XOB is really simple! To use XOB you begin by creating interfaces that maps to the tags in the XML document you are going to read/write/create. See the “Writing interfaces” section below for a description on rules for the interfaces. Alternatively if your have an XML Schema or create one you can use xobgen.jar to generate the interfaces. See below for more information on using the generator.

Simple example XML and interfaces

Take the following XML file:

```
<purchaseOrder shipped="false">
    <orderDate>2003-04-22</orderDate>
    <productId>832684</productId>
    <productId>346734</productId>
    <customerId>2674346</customerId>
    <comment>Delayed delivery</comment>
</purchaseOrder>
```

Then you create the following interfaces:

```
public interface PurchaseOrder extends XMLObject {
    public boolean getShipped();
    public void setShipped(boolean shipped);
    public OrderDate getOrderDate();
    public void setOrderDate(OrderDate orderDate);
    public Iterator getProductIds();
    public void addProductId(ProductId id);
    public void addProductIdGrouped(ProductId id);
    public CustomerId getCustomerId();
    public void setCustomerId(CustomerId id);
    public Comment getComment();
    public void setComment(Comment comment);
}

public interface OrderDate extends XMLObject {
    public xob.xml.datatypes.XSDDate getOrderDateValue();
    public void setOrderDateValue(xob.xml.datatypes.XSDDate orderDateValue);
}

public interface ProductId extends XMLObject {
    public int getProductIdValue();
    public void setProductIdValue(int value);
}

public interface CustomerId extends XMLObject {
    public int getCustomerIdValue();
    public void setCustomerIdValue(int value);
}

public interface Comment extends XMLObject {
    public String getCommentValue();
    public void setCommentValue(String value);
}
```

Reading XML (unmarshalling)

To read the above example XML file using the example interfaces you would do:

```
import java.io.FileInputStream;
import xob.XOBParseException;
import xob.Factory;
import xob.XMLObjectBinder;
import xob.XMLUnmarshaller;

public class PurchaseOrderReaderExample {

    public static void main(String[] args) throws Exception {
        XMLObjectBinder binder =
            Factory.createXMLObjectBinder(PurchaseOrder.class);
        binder.getXMLParser().setValidating(false); // We dont have a schema or DTD!

        try {
            XMLUnmarshaller unmarshaller = binder.createUnmarshaller();

            PurchaseOrder po = (PurchaseOrder)unmarshaller.unmarshal(
                new FileInputStream(args[0])
            );

            // The following will not throw any exception! We place it in the try
            // block just so that it only executes after a successful unmarshal.
            System.out.println("Has been shipped:" + po.getShipped());
            System.out.println("Order date: " + po.getOrderDate().getOrderDateValue());
            for (Iterator it = po.getProductIds(); it.hasNext();) {
                ProductId prodId = (ProductId)it.next();
                System.out.println("ProductId:" + prodId.getProductIdValue());
            }
            System.out.println("CustomerId:" + po.getCustomerId().getCustomerIdValue());
            System.out.println("Comment:" + po.getComment().getCommentValue());
        }
        catch (XOBParseException xpe) {
            System.out.println("Failed to parse " + args[0] + " (" +
                xpe.getMessage() + ")");
        }
    }
}
```

DTD validation

To do DTD validation you manipulate the XMLUnmarshaller object before the unmarshal() call:

```
unmarshaller.setValidating(true);
unmarshaller.setSchemaValidating(false);
```

Or the binder before creating the unmarshaller:

```
binder.getXMLParser().setValidating(true);
binder.getXMLParser().setSchemaValidating(false);
```

Making the XMLUnmarshaller validating, and setting schema validation to false, implies DTD validation. After the unmarshal() call you can also do:

```
xob.xml.DTDRef dtdRef = unmarshaller.getDTDRef()
```

To get the parsed DTD reference. If you are going to rewrite the XML file again you need to save this DTDRef and set it on the XMLMarshaller before calling the marshal() method. DTD references are not retained if you simply unmarshal and then marshal again.

Schema validation

By now I'm sure you can guess that for schema validation you do:

```
unmarshaller.setValidation(true);
unmarshaller.setSchemaValidation(true);
```

Or

```
binder.getXMLParser().setValidation(true);
binder.getXMLParser().setSchemaValidation(true);
```

before creating the unmarshaller.

And after the unmarshal() call you can do:

```
xob.xml.SchemaRef schemaRef = unmarshaller.getSchemaRef()
```

to get the schema reference. There are 2 ways to access and set schema references. One is by using the SchemaRef object. The other is by simply defining getters/setters for the schema attributes in your top level interface.

Filtering

It is possbile to filter the XML data when getting it using the XOB interfaces. Filters are applied to a binder before unmarshalling, the whole file are however parsed and loaded so remarshalling will not loose any data.

To use filtering you must first create a filterset. This is done using the binder instance:

```
XMLDataFilterSet filterSet = binder.createXMLDataFilterSet();
```

Next you set an attribute name that specifies the name of a filter to apply to subtags of a tag:

```
filterSet.setFilterNameAttribute(“filterName”);
```

In this example, if a tag in the input has a filterName=”myfilter” attribute a filter named “myfilter” will be applied to subtags of the tag if it is defined (continue reading!).

Next you use the filterset to create a filter:

```
XMLDataFilter myFilter = filterSet.makeFilter(“myfilter”);
```

Then you configure the filter by specifying an attribute to filter and a value the attribute must have to be accepted:

```
myFilter.setFilterAttribute(“access”);
myFilter.setFilterValue(“public”);
```

Lastly you set the filterSet on the unmarshaller:

```
unmarshaller.setXMLDataFilterSet(filterSet);
```

Then the XML file read might look like this:

```
...
<... filterName=”myfilter”>
    <... access=”public”/>
    <... access=”public”/>
    <... access=”private”/>
    <... access=”public”/>
</...>
...
```

The getter of the XOB interface representing the tag containing filterName=”myfilter” for getting the subtags will only return those that contain access=”public”.

Creating XML objects

If you first unmarshal an XML file then make changes through the interfaces and then marshal again, the interface implementations have been provided by the unmarshal. But if you want to create a new XML file from scratch, how do you get an implementation of your interfaces so that you can set information on them and then marshal? The answer is: a factory interface!

To create objects implementing your interfaces you have to create a factory interface for creating them. xobgen.jar (explained below) also generates a factory interface unless you specify the readonly flag. Here is an example factory for the above example:

```
public interface PurchaseOrderFactory {
    PurchaseOrder createPurchaseOrder() throws Exception;
    OrderDate createOrderDate() throws Exception;
    CustomerId createCustomerId() throws Exception;
    ProductId createProductId() throws Exception;
    Comment createComment() throws Exception;
}
```

As you can see, all methods start with “create” and then the name of the interface they return an implementation of. The return type is of course the same interface. Make all create methods throw Exception. There are mostly reflection exceptions that can be thrown by the methods, but if they fail, they fail. It is not really important exactly what failed when catching these exceptions. You can always dump the exception to find out what really happened if you want, but from a program logic perspective just throwing and catching Exception is good enough in this case.

This interface class is then passed as a second argument to the createXMLObjectBinder() factory method. Ex:

```
XMLObjectBinder binder =
    Factory.createXMLObjectBinder(PurchaseOrder.class,
                                  PurchaseOrderFactory.class);
```

You get an implementation of the factory interface by calling the getFactoryInstance() method on the binder instance.

Example:

```
PurchaseOrderFactory pof = (PurchaseOrderFactory)binder.getFactoryInstance();

PurchaseOrder purchaseOrder = pof.createPurchaseOrder();
purshaseOrder.setShipped(false);

OrderDate orderDate = pof.createOrderDate();
orderDate.setOrderDateValue(new xob.xml.datatypes.XSDDateTime("2003-05-02T12:20:05"));
purshaseOrder.setOrderDate(orderDate);

ProductId prodId = pof.createProductId();
prodId.setProductIdValue(9213465);
purshaseOrder.addProductId(prodId);

...
```

Namespace

If the objects you are creating needs to be prefixed with a namespace prefix you need to use a third version of the createXMLObjectBinder() method. Say for example that your top element have something like this:

```
xmlns:nse="http://my.site.com/mynamespace/NameSpaceExample"
```

and the schema contains the following attribute:

```
elementFormDefault="qualified"
```

Then if you create your binder like this:

```
XMLObjectBinder binder =
    Factory.createXMLObjectBinder(PurchaseOrder.class,
                                  PurchaseOrderFactory.class,
                                  "nse");
```

Then any object created by the factory returned by "binder.getFactoryInstance()" will have the "nse" namespace prefix. You also need to bind the prefix with the namespace. There are 2 ways to do that. If you manually created your interfaces you could add the following 3 methods to the top level interfaces:

```
public void setXmlns_Nse(String nseNamespace);
public void setXmlns_Xsi(String xsiNamespace);
public void setXsi_SchemaLocation(String schemaLocation);
```

You can then do:

```
myobj.setXmlns_Nse("http://my.site.com/mynamespace/NameSpaceExample");
myobj.setXmlns_Xsi("http://www.w3.org/2001/XMLSchema-instance");
myobj.setXsi_SchemaLoation("http://my.site.com/mynamespace/NameSpaceExample myxsd.xsd");
```

A Simpler alternative is to create a SchemaRef object and set it on the marshaller. You have to use this alternative if you have generated your interfaces with xobgen.jar. See Writing XML/Schema below for an example.

Writing XML (marshalling)

To write XML objects you simply create an XMLMarshaller from the binder and then call the marshal() method.

Example:

```
XMLMarshaller marshaller = binder.createMarshaller();
marshaller.marshal(po, System.out);
```

This example writes po, which is a PurchaseOrder instance, with all of its children to the System.out stream as XML.

DTD

To add a DTD reference to the written XML, create a DTDRef object using the factory methods in XMLObjectBinder or get it from an XMLUnmarshaller after unmarshalling and set it on the XMLMarshaller instance before calling the marshal() method.

Example:

```
XMLMarshaller marshaller = binder.createMarshaller();
DTDRef dtdRef = binder.createDTDRef(mySystemId, myPublicId);
marshaller.setDTDRef(dtdRef);
marshaller.marshal(po, System.out);
```

Schema

If you unmarshal, change and then marshal again, the schema references will be retained. If you are creating a new XML document from scratch you can either define setters for the schema attributes in your top level interface or use the createSchemaRef() factory method in XMLObjectBinder to create a new SchemaRef and then set it on the XMLMarshaller instance before calling the unmarshal() method.

Example:

```
XMLMarshaller marshaller = binder.createMarshaller();
SchemaRef schemaRef = binder.createSchemaRef("http://my.site.com/mynamespcace/NameSpaceExample",
                                             "nse",
                                             "myxsd.xsd");
marshaller.setSchemaRef(schemaRef);
marshaller.marshal(po, System.out);
```

Validating XML Object structure

You might want to validate your XML object structure before you write it. This can be done with the XMLValidator. Just like the XMLUnmarshaller and XMLMarshaller it is created by the XMLObjectBinder instance.

Example:

```
PurchaseOrder po ...
...
XMLValidator validator = binder.createValidator();
validator.setSchemaRef(schemaRef);
try {
    validator.validate(po);
}
catch (XOBValidateException xve) {
    System.out.println("Validation failed:");
```

```
        System.out.println(xve.getMessage()); // Will list all errors.
    }
```

The example does setSchemaRef() on the validator. You need to set either a SchemaRef or a DTDRef depending on if you validate against a schema or dtd.

Using the interface generator (XOBGen)

If you have an XML Schema describing your xml file then the quickest and simplest is to use the interface generator. It resides in xobgen.jar and can be run with "java -jar xobgen.jar ..." or using the xob.xobgen.XOBGenAntTask Ant task which also resides in xobgen.jar.

The generator and the Ant task have the following arguments:

Option	***Argument***	***Description***
readonly	true/false	Only generates getters and no factory if true.
outputdir	dir	Where the interfaces should be generated. If the package argument is also specified then the resulting package path is added to this, so this becomes the package root dir.
package	package	The java package the generated interfaces should belong to.
verbose	true/false	If true then the files being generated are listed on standard out.
xsd	Schema file	The XML schema file to generate from.

map	elementName objectName	This maps the specified element name to the specified object name. If this is specified for an element name the generator will not try to convert the element name to an object name itself but will use the specified object name instead. This option can be specified any number of times.
		If run with java -jar there should be a space between the element name and the object name. If run with the Ant task this is specified with
		`<map elementName="name" objectName="name"/>`
		subtags.
		Any mapped elements will have a:
		`public static final String XOM_objectName = "elementName";`
		generated in the interface. This is how XOB finds the real element name for the object.

Option	*Argument*	*Description*
automap	true/false	If true A map is automatically created for all elements not mapped specifically with map. Automap maps to the exact name specified by the schema. There is a good reason for this due to how XOB works: For any non mapped element XOB uses the name of the getter/setter to determine the element name. A getter/setter has the first character after the word get/set capitalized! Therefore XOB makes a possibly incorrect assumption that the element name does not start with a capital letter. When reading XML it could try both and see what it finds, but when writing it has no way of knowing, making some XML files readable but not writable, if XOB were to do that. I did not like that solution. This problem however completely goes away when you have a map! In the map case XOB knows the element name exactly and does not have to construct it. I'm actually considering making this default to true in the future.

When run with “java -jar” the options should be prefixed with “--” and true/false options take no argument, just specifying the options makes them true (which means that they default to false!).

When run through the Ant task the options are attributes and true/false options must be specified as either “true” or “false”. The Ant task also supports the <fileset> subtask as an alternative to xsd=”file.xsd”. This also allows for generating for more than one input file in one run which the java -jar version does not.

If <xs:annotation><xs:documentation>bla</xs:documentation></xs:annotation> tags are found when parsing the schema they will be used as javadoc comments for the generated interfaces. Any “[“ will be translated to “<”, any “]” will be translated to “>”, any “[[“ will be translated to “<” and any “]]” will be translated to “>”. This makes it rather simple to put html tags in the documentation without using CDATA.

Example java -jar usage

java -jar .../xob/jars/xobgen.jar –-verbose –-readonly –-package “my.pkg” --outputdir “src” –-xsd MySchema.xsd

Example Ant task usage

```
<target name="def-xobgen" depends="init" description="Depend on this if you are going to use xobgen.">
    <taskdef  name="xobgen" classname="xob.xobgen.XOBGenAntTask[Real]">
        <classpath>
            <pathelement location="${xob-dir}/jars/xob.jar"/>
            <pathelement location="${xob-dir}/jars/xobxsd.jar"/>
            <pathelement location="${xob-dir}/jars/xobgen.jar"/>
            <!-- Using the xerces jars from the xob dist. -->
            <!-- These are not needed if you run jdk 1.4! -->
            <pathelement location="${xob-dir}/extjars/xercesImpl.jar}"/>
            <pathelement location="${xob-dir}/extjars/xmlParserAPIs.jar"/>
            <pathelement location="${xob-dir]/extjars/xml-apis.jar"/>
        </classpath>
    </taskdef>
</target>

<target name=”generateIfs” depends=”init,def-xobgen”>
    <xobgen verbose="true" outputdir="../src/runtime" package=”my.pkg” xsd="PurchaseOrder.xsd"/>

    <!-- This is probably not very useful, but possible. -->
    <xobgen verbose=”true” outputdir=”../src/prod” package=”my.pkg.xml”>
        <fileset dir=”../src/prod”>
            <include name=”**/*.xsd”/>
        </fileset>
    </xobgen>
</target>
```

Please note that you should use classname=”xob.xobgen.XOBGenAntTask” if you are using Ant 1.5.4 or below. This Ant task wraps XOBGenAntTaskReal which is loaded using an XOB internal class loader to go around a bug in Ants class loader for these versions. If you are using Ant 1.6.0 or above you should use classname=”xob.xobgen.XOBGenAntTaskReal” directly since the Ant classloader bug has been corrected, and the XOB internal class loader has a problem with these versions of Ant.

Special features

You can affect the multiplicity of an element through a comment in the XML schema. Example:

```
    <xsd:element name="officeCity">
        <xsd:complexType>
            <xsd:sequence>
                <!-- XOB:Generator-override maxOccurs="1" -->
                <xsd:element ref="city" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="filterName" use="optional" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>
```

In this case the XML schema says that there are one to many city elements under the officeCity element. The “<!--XOB:Generator-override maxOccurs=”1” -->” comment however tells the generator to see this as only one city element being available under officeCity. This will generate a getter that returns a City object instead of an Iterator.

Now you are probably wondering “Why?”. This might seen quite an odd thing to do, but this works well in conjunction with filters as the example above implies. If you apply a filter that will always only return one sub-element then you can make the generated API reflect this.

Using the maven2 plugin

The maven2 plugin for xobgen works a bit different than the Ant task. To start with you need to add the following plugin declaration to your pom.xml:

```
<plugins>
    <plugin>
        <groupId>net.sf.xob</groupId>
        <artifactId>xob-m2-plugin</artifactId>
        <executions>
            <execution>
                <id>generate-xob-apis</id>
                <configuration>
                    <scanDirs>src/main/resources</scanDirs>
                    <outputRelativeDir>src/main/java</outputRelativeDir>
                    <schemaExt>.xsd</schemaExt>
                    <readOnly>true</readOnly>
                    <!--map>elementName=objectName,...</map-->
                    <verbose>true</verbose>
                </configuration>
                <goals>
                    <goal>genapi</goal>
                </goals>
                <phase>generate-sources</phase>
            </execution>
        </executions>
    </plugin>
</plugins>
```

This configures the plugin to run during the generate-sources phase. The “scanDirs” configuration can actually take a comma separated list of paths.

Now you might think that there is something missing, like what schemas to generate from and what package to generate to for each schema. This is where “different” comes in. The plugin scans the filessystem threes starting at the specified “scanDirs” roots. Any file with the “schemaExt” extension will be opened and the first 10 lines scanned for a “@xobgen <package>” specification, preferably in a comment if you want the schema to be parsable. The <package> part is the package of the generated API. When the plugin sees this is will add the file to the list of schemas to generate APIs for. When the scanning is done the found schemas will have java APIs generated for them at “outputRelativeDir” plus path of specified package.

Using the schema HTML doucmentation generator (XSDDoc)

Since the interface generator resulted in an xsd parser that also extracts documentation I realized that it would be nice and rather simple to also make a HTML documentation generator for a schema. The doc generator resides in xsddoc.jar and can be run with “java -jar xsddoc.jar ...” or with the xob.xsddoc.XSDDocAntTask Ant task also residing in xsddoc.jar.

The generator and the Ant task have the following options:

Option	*Argument*	*Description*
alt	true/false	Normally the generated documentation consists of 4 sections: complexTypes, groups, attributeGroups, and elements. Attributes and simple types are expanded into those objects that references them. Specifying true for alt generates an alternative output with only elements and everything expanded into the elements. This view can be nicer since for each element you can very easily see all subelements and attributes allowed. The normal view more reflects the schema structure.
verbose	true/false	If true files are listed as they are being doc generated.
docdir	dir	The directory where the HTML docs will be written.

When run with “java -jar” all options are prefixed with “--”. True/false type options have no argument, just specifying them makes them true. The input files to genereate docs for are specified after all options. You can specify more than one file.

When run through the Ant task the options are attributes. True/false type options must be specified as either “true” or “false”. All input files are specified using a <fileset> subtask.

There are 2 extra files written to the docdir catalog: xsddocIndex.html and XSDIndex.html. The xsddocIndex.html is a frames document that shows XSDIndex.html in a small frame to the left and the first generated document in the doc frame to the right. XSDIndex.html contains links to all documents generated in the same run. Clicking on them will display them in the right frame.

For any <xs:annotation><xs:documentation>bla</xs:documentation></xs:annotation> any “[“ will be translated to “<”, any “]” will be translated to “>”, any “[[“ will be translated to “<” and any “]]” will be translated to “>”. This makes it rather simple to put html tags in the documentation without using CDATA. For the <xs:schema> tag docs you can use @version and @author tags also, like javadoc. They can actually be used in any documentation block, but they make most sense for the schema documentation.

Example java -jar usage

```
java -jar xob/jars/xsddoc.jar –-verbose –-alt –-docdir “schemaDocs” myschema.xsd myotherschema.xsd
```

Example Ant task usage

```
<target name="def-xsddoc" depends="init" description="Depend on this if you are going to use xsddoc.">
    <taskdef  name="xsddoc" classname="xob.xsddoc.XSDDocAntTask[Real]">
        <classpath>
            <pathelement location="${xob-dir}/jars/xob.jar"/>
            <pathelement location="${xob-dir}/jars/xobxsd.jar"/>
            <pathelement location="${xob-dir}/jars/xsddoc.jar"/>
            <!-- Using the xerces jars from the xob dist. -->
            <!-- These are not needed if you run jdk 1.4! -->
            <pathelement location="${xob-dir}/extjars/xercesImpl.jar}"/>
            <pathelement location="${xob-dir}/extjars/xmlParserAPIs.jar"/>
            <pathelement location="${xob-dir]/extjars/xml-apis.jar"/>
        </classpath>
    </taskdef>
</target>

<target name=”generateDocs” depends=”init,def-xsddoc”>
    <xsddoc verbose=”true” docdir=”../schemaDocs” alt=”true”>
        <fileset dir=”../resources/”>
            <include name=”**/*.xsd”/>
        </fileset>
    </xsddoc>
</target>
```

Please note that you should use classname=”xob.xsddoc.XSDDocAntTask” if you are using Ant 1.5.4 or below. This Ant task wraps XSDDocAntTaskReal which is loaded using an XOB internal class loader to go around a bug in Ants class loader for these versions. If you are using Ant 1.6.0 or above you should use classname=”xob.xsddoc.XSDDocAntTaskReal” directly since the Ant classloader bug has been corrected, and the XOB internal class loader has a problem with these versions of Ant.

Writing interfaces

If you are going to write your own interfaces instead of generating them, which you will have to do if you don't have a schema or want to make one, this section explains how.

Interface names

The name of the interfaces should match the name of the element they represent, but with the first charater capitalized. Any '.' in an elment name must be replaced with "_dot
" (underscore-"dot"-underscore). Any '-' in an element name must be replaced with "_dash
" (underscore-"dash"-underscore). If you dont like these type of annoying names then call your interface anything you want and then define the following constant in it:

```
public static final String XOM_interfaceName = "elementName";
```

where *interfaceName* is whatever you choose to call your inteface, and *elementName* is the real name of the element this interface represents.

Tags

An interface represents an XML tag. It should as minimum contain getters for the child tags and attributes wanted. If you want to use these interfaces for createing an XML file then you should also add setters to the interfaces.

All interfaces **must** extend the xob.XMLObject interface!

Child tags

The getter for a child tag that returns zero or one child tag object should look like this:

```
public childtype getchildtype();
```

where *childtype* is exactly the name of the interface representing the child tag.

The getter for a child tag that returns zero or more child tag objects should look like this:

```
public Iterator getchildtypes();
```

where *childtype* is exactly the name of the interface representing the child tag, and is the interface implementation returned by the Iterator. Please note the 's' at the end of the name!

The setter for a child tag object that sets one child should look like this:

```
public void setchildtype(childtype child);
```

where *childtype* is the name of the interface representing the child tag being set.

The setter for a child tag object that can set one or more children should look like this:

```
public void addchildtype(childtype child);
public void addchildtypeGrouped(childtype child);
```

where *childtype* is the name of the interface representing the child tag being added. The second version whose name end with "Grouped" means that the child will be added at the end of its group. That is, if there are several children of the same type they are considered a group and are kept together. This is important if <xsd:sequence> is used to define children in a schema.

Namespace

If a child tag belongs to a namespace then the *childtype* in the getter and setter names in the examples above can be replaced with:

```
namespaceprefix_childtype
```

Example:

```
public childtype getnamespaceprefix_childtype();
public void setnamespaceprefix_childtype(childtype child);
```

Both the *namespaceprefix* and *childtype* have the first character capitalized!

There is an alternative to prefixing the getter and setter names with the namespace prefix. You can declare a constant called QUALIFIED_NAMESPACE in your interfaces, containing the namespace url. Then you are also required to use the third version of the createXMLObjectBinder() mehtod that also takes a namespace prefix if you are going to create a new XML object from scratch. Any object created with the factory instance returned by the binder will then have that prefix without it having to be part of the setter and getter name.

If you define the constant in your interfaces and unmarshall then you will only get objects that are prefixed with the prefix defined for the namespace that the constant defines. The unmarshall will also extract the namespace prefix, so if you unmarshall, add/change, and then marshal, any objects created by the factory after unmarshal will have the same namespace prefix as the unmarshalled xml file. You don't have to care what the prefix is in that case! See xob/examples/validating/schema/readchangewritens for an example.

Interfaces generated by xobgen makes use of this alternative to handling namespaces. If the schema has the elementFormDefault=”qualified” attribute then QUALIFIED_NAMESPACE constants will be generated for all interfaces.

Tag namespace and naming

Please note that the interface names does not contain namespace names in them!

Attributes

The getter for an attribute should look like this:

```
public primitivetype getAttributename();
```

where *Attributename* is the name of the attribute with the first character capitalized and the rest identical to the attribute name. Please note that if the real attribute name starts with a capital then any getter for it will fail! This is a very uncommon case, and I have no workaround for the moment.

The setter for an attribute should look like this:

```
public void setAttributename(primitivetype value);
```

where *Attributename* is the name of the attribute with the first character capitalized and the rest identical to the attribute name.

primitivetype is one of int, long, float, double, boolean, String, XSDDate, XSDTime, XSDDateTime, Date, XSDDay, XSDMonth, XSDMonthDay, XSDYear, and XSDYearMonth.

The XSD* types reside in the xob.xml.datatypes package. Date is a java.util.Date, which can be used as an alternative to XSDDate, but not to XSDTime or XSDDateTime! The XSDDate,

XSDDateTime, and XSDTime extends java.util.Date and have constructors that takes java.util.Date objects so they can easily be converted back and forth to java dates. The other XSD* objects are only value and type holders. Interpretation of their contents is up to you!

Please note that an XSDDate toString() value conforms to the XMLSchema date format while java.util.Date does not!

Attribute namespace

If an attribute belongs to a namespace then *attributename* should be replaced with:

```
namespace_attributename
```

There is no alternative to this, and the xobgen generator does not handle namespaces for attributes! (Until someone can explain to me how a namespace qualified attribute is defined in a schema I cannot fix this. The schema documentation I have does not explain this. --Tommy)

Tag value

The getter of the content value of a tag should look like this:

```
public primitivetype getinterfacenameValue();
```

where *interfacename* is the name of the interface the getter belongs to. This naming is to avoid collisions with a getter for a "value" attribute, which might not be that uncommon!

The setter of the content value of a tag should look like this:

```
public void setinterfacenameValue(primitivetype value);
```

where *interfacename* is the name of the interface the getter belongs to.

Examples

The xob/examples catalog contains variants of the purchase order xml file example that covers all features of xob. They double as test code. The non validating and DTD validating examples require any JAXP 1.1 parser in the classpath prior to running the ant script. If you are using JDK 1.4 you have the crimson parser already available.

For the schema validating examples you need a JAXP 1.2 parser. The ant scripts for those examples add the bundled Xerces jar files to the classpath. It finds them in xob/extjars.

Tested XML parsers

XOB have been tested with Xerces 2.4.0 and the crimson parser included with jdk 1.4 from Sun.

Configuration

In xob/lib there is an TypeMap.xml file. It maps XML Schema types to XOB types. It lists the valid XOB types to map to. It should contain mappings for all XML Schema datatypes listed by the XML Schema tutorial at www.w3schools.org.

XOB 2.1 and earlier versions had the type map hardcoded. If there is any XML Schema datatype missing, you can just add it to the map, but please also inform me.

Please note that the xob/lib/TypeMap.xsd file is only included as a convenience if you want to

validate the TypeMap.xml file when editing it. By default the TypeMap.xsd does not reference the TypeMap.xsd for validation since that requires JAXP 1.2 or later. XOB does not try to validate it when it reads it even if you add the schema reference to it!

The type mapping only affects the generator (xobgen.jar)! xob.jar which is needed runtime does not reference xob/lib/TypeMap.xml in any way!

Author/contact

Name:	Tommy Svensson
Email:	tommy@tommysvensson.net