Eclipse Implementation of JAXB Release Documentation

Eclipse Implementation of JAXB Release Documentation

Table of Contents

Overview

Table of Contents

Jakarta XML Binding provides an API and tools that automate the mapping between XML documents and Java objects.

The Jakarta XML Binding framework enables developers to perform the following operations:

- **Unmarshal** XML content into a Java representation
- **Access** and **update** the Java representation
- **Marshal** the Java representation of the XML content into XML content

Jakarta XML Binding gives Java developers an efficient and standard way of mapping between XML and Java code. Java developers using Jakarta XML Binding are more productive because they can write less code themselves and do not have to be experts in XML. Jakarta XML Binding makes it easier for developers to extend their applications with XML and Web Services technologies.

1. Documentation

Documentation for this release consists of the following:

- *Release Notes*
- Running the binding compiler (XJC): [XJC, XJC Ant Task]
- Running the schema generator (schemagen): [SchemaGen, SchemaGen Ant Task]
- *Eclipse Implementation of JAXB Users Guide*
- Javadoc API documentation (jakarta.xml.bind.*) [https://jakarta.ee/specifications/xml-binding/4.0/api-docs/]
- Sample Apps
- Jakarta XML Binding FAQs [*Frequently Asked Questions*]

2. Software Licenses

3. Sample Apps

This page summarizes basic use-cases for Java-2-Schema, Schema-2-Java, and lists all of the sample applications that ship with JAXB.

3.1. Using the Runtime Binding Framework

3.1.1. Schema-2-Java

Schema-2-Java is the process of compiling one or more schema files into generated Java classes. Here are some of the basic steps for developing an app:

1. Develop/locate your schema
2. Annotate the schema with binding customizations if necessary (or place them in an external bindings file)
3. Compile the schema with the XJC binding compiler
4. Develop your JAXB client application using the Java content classes generated by the XJC binding compiler along with the `jakarta.xml.bind` runtime framework
5. Set your `CLASSPATH` to include all of the Identifying the JAR Files
6. Compile all of your Java sources with `javac`
7. Run it!

3.1.2. Java-2-Schema

Java-2-Schema is the process of augmenting existing Java classes with the annotations defined in the `jakarta.xml.bind.annotation` package so that the JAXB runtime binding framework is capable of performing the (un)marshal operations. Here are the basic steps for developing an app:

1. Develop your data model in Java
2. Apply the `jakarta.xml.bind.annotation` annotations to control the binding process
3. Set your `CLASSPATH` to include all of the Identifying the JAR Files
4. Compile your data model with `javac`

   #### Important

   Make sure that you `CLASSPATH` includes `jaxb-xjc.jar` before running **javac**.

5. The resulting class files will contain your annotations as well other default annotations needed by the JAXB runtime binding framework
6. Develop your client application that uses the data model and develop the code that uses the JAXB runtime binding framework to persist your data model using the (un)marshal operations.
7. Compile and run your client application!

For more information about this process, see the the Java WSDP Tutorial [http://docs.oracle.com/javaee/6/tutorial/doc/gijti.html] and the extensive Sample Apps documentation.

3.1.3. Building and Running the Sample Apps with Ant

To run the sample applications, add jaxb dependencies to classpath at jaxb.home property, and run **ant** without any option into each sample directory.

A few sample applications do *not* use Ant. For those samples, refer to the included `readme.txt` files for instructions.

3.1.4. List of Sample Apps

`samples/catalog-resolver`	This example demonstrates how to use the `-catalog` compiler switch to handle references to schemas in external web sites.
`samples/character-escape`	This example shows how you can use the new JAXB RI `Marshaller` property `org.glassfish.jaxb.characterEscapeHandler` to change the default character escaping behavior.
`samples/class-resolver`	This little DI-container-by-JAXB example demonstrates how one can avoid passing in a list of classes upfront, and instead load classes lazily.
`samples/create-marshal`	This sample application demonstrates how to use the `ObjectFactory` class to create a Java content tree from scratch and marshal it to XML data. It also demonstrates how to add content to a JAXB List property.
`samples/cycle-recovery`	Eclipse Implementation of JAXB's vendor extension `CycleRecoverable` provides application a hook to handle cycles in the object graph. Advanced.
`samples/datatypeconverter`	This sample application is very similar to the inline-customize sample application (formerly SampleApp6), but illustrates an easier, but not as robust, `<jaxb:javaType>` customization.
`samples/dtd`	This sample application illustrate some of the DTD support available in the Eclipse Implementation of JAXB's extension mode. Please refer to the *Eclipse Implementation of JAXB Extensions* page for more detail.
`samples/element-substitution`	This sample application illustrates how W3C XML Schema substitution groups are supported in Eclipse Implementation of JAXB's extension mode. Please refer to the *Eclipse Implementation of JAXB Extensions* page for more detail.
`samples/external-customize`	This sample application is identical to the datatypeconverter sample application (formerly SampleApp7) except that the binding customizations are contained in an external binding file.
`samples/fix-collides`	Another binding customization example that illustrates how to resolve name conflicts. Running this sample without the binding file will result in name collisions (see `readme.txt`) . Running ant

	will use the binding customizations to resolve the name conflicts while compiling the schema.
`samples/inline-customize`	This sample application demonstrates how to customize the default binding produced by the XJC binding compiler.
`samples/j2s-crete-marshal`	This sample application demonstrates marshalling, unmarshalling and unmarshal validation with existing Java classes annotated with JAXB annotations.
`samples/j2s-xmlAccessorOrder`	This sample application demonstrates the use of mapping annotations `@XmlAccessorOrder` and `@XmlType.propOrder` in Java classes for ordering properties and fields in Java to schema bindings.
`samples/j2s-xmlAdapter`	This sample application demonstrates the use of interface `XmlAdapter` and annotation `XmlJavaTypeAdapter` for custom marshaling/unmarshaling XML content into/out of a Java type.
`samples/j2s-xmlAttribute`	This sample application demonstrates the use of annotation `@XmlAttribute` for defining Java properties and fields as XML attributes.
`samples/j2s-xmlRootElement`	This sample application demonstrates the use of annotation `@XmlRootElement` to define a class to be an XML element.
`samples/j2s-xmlSchemaType`	This sample application demonstrates the use of annotation `@XmlSchemaType` to customize the mapping of a property or field to an XML built-in type.
`samples/j2s-xmlType`	This sample application demonstrates the use of mapping annotations `@XmlAccessorOrder` and `@XmlType.propOrder` in Java classes for ordering properties and fields in Java to schema bindings.
`samples/locator-support`	This sample shows how to use the new non-standard locator support. By following the instructions in the readme.txt file, you can cause all of the generated impl classes to implement a new interface that provides more information about error locations. When a `ValidationEvent` happens on your content tree, simply retrieve the object and cast it down to `com.sun.xml.bind.extra.Locatable`.
`samples/modify-marshal`	This sample application demonstrates how to modify a java content tree and marshal it back to XML data.
`samples/namespace-prefix`	This sample application demonstrates how to use the new Eclipse Implementation of JAXB Marshaller property `org.glassfish.jaxb.namespacePrefixMapper` to customize the namespace prefixes generated during marshalling.
`samples/partial-unmarshalling`	In this example, the input document will be unmarshalled a small chunk at a time, instead of unmarshalling the whole document at once.
`samples/pull-parser`	This sample app demonstrates how a pull-parser can be used with JAXB to increase the flexibility of processing.

`samples/streaming-unmarshalling`	This example illustrates a different approach to the streaming unmarshalling, which is suitable for processing a large document.
`samples/synchronized-methods`	This sample shows how to use the new non-standard synchronized method support. By following the instructions in the `readme.txt`, you can cause all of the generated impl class methods signatures to contain the `synchronized` keyword.
`samples/type-substitution`	This sample app demonstrates type substitution using the W3C XML Schema Part 0: Primer international purchase order schema.
`samples/ubl`	This project processes a UBL (Universal Business Language) order instance and prints a report to the screen.
`samples/unmarshal-read`	This sample application demonstrates how to unmarshal an instance document into a Java content tree and access data contained within it.
`samples/unmarshal-validate`	This sample application demonstrates how to enable validation during the unmarshal operations.
`samples/updateablePartialBind`	This sample application demonstrates how to partially map a DOM tree to JAXB (using JAXP 1.3 XPath), modify JAXB mapped instance and then update modifications back to the DOM tree.
`samples/vendor-extensions`	This example demonstrates how to use `<xjc:superClass>` vendor extensions provided by Eclipse Implementation of JAXB's, as well as `<jaxb:serializable>` customization.
`samples/xml-channel`	This example demonstrates how one can use one communication channel (such as a socket) to send multiple XML messages, and how it can be combined with JAXB.
`samples/xml-stylesheet`	A common customization need for the marshalling output is about introducing extra processing instruction and/or `DOCTYPE` declaration. This example demonstrates how such modification can be done easily.

Release Notes

Table of Contents

This document contains information that should help you use this software library more effectively. See the *Frequently Asked Questions* for additional information.

The most up-to-date version of this document can be found on-line [https://eclipse-ee4j.github.io/jaxb-ri].

1. Java™ SE Requirements

This release of the Eclipse Implementation of JAXB requires Java SE 11 or higher.

2. Identifying the JAR Files

Use	Description	Jar
Runtime	Jars required to deploy a Jakarta XML Binding client	`jakarta.activation-api.jar`
		`angus-activation.jar`
		`jakarta.xml.bind-api.jar`
		`jaxb-core.jar`
		`jaxb-impl.jar`
Compiler	Jars required at your development environment (but not runtime)	`jaxb-jxc.jar`
		`jaxb-xjc.jar`

3. Identifying the JPMS module names

Jar	Module name	Maven GAV
`jakarta.activation-api.jar`	`jakarta.activation`	`jakarta.activation:jakarta.activation-api`
`angus-activation.jar`	`com.sun.activation.registries`	`org.eclipse.angus:angus-activation`

Jar	Module name	Maven GAV
jakarta.xml.bind-api.jar	jakarta.xml.bind	jakarta.xml.bind:jakarta.xml.bind-api
jaxb-core.jar	com.sun.xml.bind.core	com.sun.xml.bind:jaxb-core
jaxb-impl.jar	com.sun.xml.bind	com.sun.xml.bind:jaxb-impl
jaxb-jxc.jar	com.sun.tools.jxc	com.sun.xml.bind:jaxb-jxc
jaxb-xjc.jar	com.sun.tools.xjc	com.sun.xml.bind:jaxb-xjc

4. Locating the Normative Binding Schema

You may find information about the normative binding schema defined in the Jakarta XML Binding Specification at https://jakarta.ee/xml/ns/jaxb.

5. Changelog

The Eclipse Implementation of JAXB 4.x meets the requirements of the Jakarta XML Binding 4.x specifications.

5.1. Changes in 4.0.0 - initial release for Jakarta EE 10 [https://jakarta.ee/]

- Requires Java SE 11 or newer
- Supports usage of JAXB 2.x schema bindings customizations
- Bug fixes:
  - Fix equality on BISerializable
  - #936 [https://github.com/eclipse-ee4j/jaxb-ri/issues/936]: problem with XMLMixed in a tag annotated XmlAnyElement
  - #971 [https://github.com/eclipse-ee4j/jaxb-ri/issues/971]: annotation @XmlJavaTypeAdapters on package is ignored since JAXB v2.2.4-1
  - #1053 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1053]: Use Java 7 diamond operator
  - #1117 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1117]: xjc-generated classes may have methods with missing @param or @return
  - #1489 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1489]: DOMScanner ignores default namespace at scan method
  - #1499 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1499]: xjc - NGCCRuntimeEx.resolveRelativeURL(String namespaceURI, String relativeUri) doesn't work as it should

- #1505 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1505]: JCodeModel.parseType(String) silently ignores type params in specific scenarios
- #1590 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1590]: Marshalling an object that overrides the parent's method, the XML that gets created contains both child's and parent's tag
- #1599 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1599]: XNOR implementation in NameUtil is called "xor"
- #1624 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1624]: Order of Exceptions in generated classes is non-deterministic
- #1631 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1631]: Support setting (un)marshaller listener on binder

5.2. Changes between 3.0.1 and 3.0.2

- Bug fixes:
  - Fixed classloading in OSGI
  - #1547 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1547]: Running with -XX:-StackTraceInThrowable causes a index out of bounds exception
  - #1556 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1556]: xjc generates class reference with generics

5.3. Changes between 3.0.0 and 3.0.1

- Bug fixes:
  - #1105 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1105]: xjc mark-generated sometimes produces a wrong date value
  - #1466 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1466]: ContextFinder always load the JAXBContext from jaxb-runtime 2.3.3
  - #1475 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1475]: xjc: Option to generate old package names
  - #1502 [https://github.com/eclipse-ee4j/jaxb-ri/issues/1502]: XJC: fails to process XSD files without systemId.

5.4. Changes in 3.0.0 - initial release for Jakarta EE 9 [https://jakarta.ee/]

- Requires Java SE 8 or newer
- Adopts new API package namespace - `jakarta.xml.bind.*`
- Main implementation jar split into two parts - `jaxb-core` and (smaller) `jaxb-impl`
- Content of the new jaxb-impl moved from `com.sun.xml.bind` package to `org.glassfish.jaxb.runtime` package

- Content of the new jaxb-core moved from `com.sun.xml.bind` package to `org.glassfish.jaxb.core package`
- Changed prefix of all properties from `com.sun.xml.bind` to `org.glassfish.jaxb`
- Supports new namespace for schema customizations

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0">...</
bindings>
```

5.5. Changes in 2.3.2 - initial release for Jakarta EE 8 [https://jakarta.ee/]

- First release under Eclipse Jakarta EE Platform:
  - Uptake of moved Jakarta APIs.

Eclipse Implementation of JAXB Users Guide

Abstract

This document explains various interesting/complex/tricky aspects of Eclipse Implementation of JAXB, based on questions posted on the old JAXB users forum and answers provided there. This is an ongoing work-in-progress. Any feedback [https://accounts.eclipse.org/mailing-list/jaxb-impl-dev] appreciated.

Table of Contents

1. Compiling XML Schema

1.1. Dealing with errors

1.1.1. Schema errors

Because XML Schema is so complicated, and because there are a lot of tools out there do not implement the spec correctly, it is often the case that a schema you are trying to compile has some real errors in it. When this is the case, you'll see XJC reporting somewhat cryptic errors such as `rcase-RecurseLax.2: There is not a complete functional mapping between the particles.`

The Eclipse Implementation of JAXB uses the schema correctness checker from the underlying JAXP implementation, which is the JAXP RI in a typical setup. The JAXP RI is one of the most conformant schema validators, and therefore most likely correct. So the first course of action usually is to fix problems in the schema.

However, in some situations, you might not have an authority to make changes to the schema. If that is the case and you really need to compile the schema, you can bypass the correctness check by using the `-nv` option in XJC. When you do this, keep in mind that you are possibly feeding "garbage" in, so you may see XJC choke with some random exception.

1.1.2. Property 'fooBarZot' is already defined

One of the typical errors you'll see when compiling a complex schema is:

Example 1. Multiple property definitions error

```
parsing a schema...
[ERROR] Property "MiOrMoOrMn" is already defined.
  line 132 of
file:/C:/kohsuke/Sun/JAXB/jaxb-unit/schemas/individual/MathML2/presentation/
scripts.xsd

[ERROR] The following location is relevant to the above error
  line 138 of
file:/C:/kohsuke/Sun/JAXB/jaxb-unit/schemas/individual/MathML2/presentation/
scripts.xsd
```

This is an actual example of the offending part of a schema, taken from MathML. If you go to line 132 of `scripts.xsd`, you'll see that it has a somewhat complicated content model definition:

Example 2. Multiple property definitions in MathML

```
<xs:group name="mmultiscripts.content">
    <xs:sequence>
        <xs:group ref="Presentation-expr.class"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">      <!-- line 132
 -->
            <xs:group ref="Presentation-expr-or-none.class"/>
            <xs:group ref="Presentation-expr-or-none.class"/>
        </xs:sequence>
```

```
        <xs:sequence minOccurs="0">
            <xs:element ref="mprescripts"/>
            <xs:sequence maxOccurs="unbounded">                <!-- line 138
 -->
                <xs:group ref="Presentation-expr-or-none.class"/>
                <xs:group ref="Presentation-expr-or-none.class"/>
            </xs:sequence>
        </xs:sequence>
    </xs:sequence>
</xs:group>
```

This is a standard technique in designing a schema. When you want to say "in this element, `B` can occur arbitrary times, but `C` can occur only up to once", you write this as `B*,(C,B*)?`. This, however, confuses Eclipse Implementation of JAXB, because it tries to bind the first `B` to its own property, then `C` to its own property, then the second `B` to its own property, and so we end up having a collision again.

In this particular case, `B` isn't a single element but it's a choice of large number of elements abstracted away in `<xs:group>`s, so they are hard to see. But if you see the same content model referring to the same element/group twice in a different place, you can suspect this.

In this case, you'd probably want the whole thing to map to a single list so that you can retain the order those elements show up in the document. You can do this by putting the same `<jaxb:property>` customization on the whole "`mmultiscripts.content`" model group, like this (or you can do it externally with XPath):

Example 3. How to fix the problem?

```
<xs:groupname="mmultiscripts.content">
<xs:annotation>
    <xs:appinfo>
        <jaxb:propertyname="content"/>
    </xs:appinfo>
</xs:annotation>
<xs:sequence>
<xs:groupref="Presentation-expr.class"/>
```

Another way to fix this problem is to use the simpler and better binding mode in XJC, which is a Eclipse Implementation of JAXB vendor extension.

1.1.3. Two declarations cause a collision in the ObjectFactory class

When schemas contain similar looking element/type names, they can result in "Two declarations cause a collision in the ObjectFactory class" errors. To be more precise, for each of all types and many elements (exactly what elements get a factory and what doesn't is bit tricky to explain), XJC produces one method on the `ObjectFactory` class in the same package. The `ObjectFactory` class is created for each package that XJC generates some files into. The name of the method is derived from XML element/type names, and the error is reported if two elements/types try to generate the same method name.

There are two approaches to fix this problem. If the collision is coming from two different schemas with different target namespaces, then you can easily avoid the collision by compiling them into different Java packages. To do this, use `<schemabindings>` customization on two schemas and specify the package name.

Another way to fix this problem is to use `<factoryMethod>` customization on two conflicting elements/types to specify different factory method names. This can be used in all cases, but if you have a large number of conflicts, you'll have to specify this customization one by one.

Notice that `<class>` customization doesn't affect the `ObjectFactory` method name by itself.

1.1.4. Customization errors

1.1.4.1. XPath evaluation of ... results in empty target node

External Jakarta XML Binding customizations are specified by using XPath (or using SCD.) This works by writing an XPath expression that matches a particular element in the schema document. For example, given the following schema and binding file:

Example 4. Schema and external binding file

`test.xsd`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexTypename="foo"/>
</xs:schema>
```

`test.xjb`.

```
<bindings version="3.0" xmlns="https://jakarta.ee/xml/ns/jaxb"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <bindings schemaLocation="test.xsd">
        <bindings node="//xs:complexType[@name='foo']">
            <classname="Bar"/>
        </bindings>
    </bindings>
</bindings>
```

will be interpreted as if the class customization is attached to the complex type '`foo`'.

For this to work, the XPath expression needs to match one and only one element in the schema document. When the XPath expression is incorrect and it didn't match anything, you get this "XPath evaluation of ... results in empty target node" problem.

Common causes of this problem include typos, incorrect namespace URI declarations, and misunderstanding of XPath.

1.2. Fixing broken references in schema

Sometimes a schema may refer to another schema document without indicating where the schema file can be found, like this:

Example 5. Schema reference without location

```
<xs:import namespace="http://www.w3.org/1999/xlink" />
```

In other cases, a schema may refer to another schema on the network, which often slows down your compilation process and makes it unreliable. Yet in some other cases, a schema may reference another schema in relative path, and that may not match your directory structure.

XJC bundles a catalog resolver [http://xml.apache.org/commons/components/resolver/resolver-article.html] so that you can work around these situations without changing the schema documents. The main idea behind the catalog is "redirection" --- when XJC is about to fetch resources, it will consult the catalog resolver to see if it can find the resource elsewhere (which is usually your local resources.)

1.2.1. Catalog format

The catalog resolver supports many different formats, but the easiest one is a line based `*.cat` format. Other than comments and empty lines, the file mainly consists of two kinds of declarations, `SYSTEM`, and `PUBLIC`.

Example 6. `sample-catalog.cat`

```
--
  sample catalog file.

  double hyphens are used to begin and end a comment section.
--

SYSTEM "http://www.w3.org/2001/xml.xsd" "xml.xsd"

PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN" "s4s/XMLSchema.dtd"
```

1.2.2. Resolve by system ID

The SYSTEM entry has the format of "SYSTEM *REFERENCE ACTUAL-LOCATION*", which defines a simple redirection. Every time XJC loads any resource (be it schemas, DTDs, any entities referenced within), it will first resolve relative paths to absolute paths, then looks for a matching *REFERENCE* line. If it is found, the specified actual location is read instead. Otherwise XJC will attempt to resolve the absolutepath.

ACTUAL-LOCATION above accepts relative paths, and those are resolved against the catalog file itself (so in the above example, `xml.xsd` is assumed to be in the same directory with `sample-catalog.cat`.

What you need to be careful is the fact that the *REFERENCE* portion must be absolute, and when XJC finds a reference in schema, it will first convert that to the absolute path before checking the catalog. So what this means is that if your schema is written like this:

Example 7. Schema reference by relative path

```
<xs:import namespace="http://www.w3.org/1999/xlink"
 schemaLocation="xlink.xsd" />
```

Then your catalog entry would have to look like this:

Example 8. xlink.cat

```
-- this doesn't work because xlink.xsd will be turned into absolute path --
SYSTEM "xlink.xsd" "http://www.w3.org/2001/xlink.xsd"

-- this will work, assuming that the above schema is in /path/to/my/test.xsd
 --
SYSTEM "/path/to/my/xlink.xsd" "http://www.w3.org/2001/xlink.xsd"
```

1.2.3. Resolve by public ID / namespace URI

Another kind of entry has the format of "PUBLIC *PUBLICID ACTUAL-LOCATION*" or "PUBLIC *NAMESPACEURI ACTUAL-LOCATION*".

The "`PUBLICID`" version is used to resolve DTDs and entities in DTDs. But this type of entry is also used to resolve `<xs:import>` statements. XJC will match the value of the namespace attribute and see if there's any matching entry. So given a schema like this:

Example 9. Schema import

```
<xs:import namespace="http://www.w3.org/1999/xlink"
 schemaLocation="xlink.xsd" />
<xs:import namespace="http://www.w3.org/1998/Math/MathML" />
```

The following catalog entries will match them.

Example 10. by-publicid.cat

```
PUBLIC "http://www.w3.org/1999/xlink" "http://www.w3.org/2001/xlink.xsd"
PUBLIC "http://www.w3.org/1998/Math/MathML" "/path/to/my/mathml.xsd"
```

As you can see, XJC will check the PUBLIC entries regardless of whether `<xs:import>` has the schemaLocation attribute or not. As with the case with the SYSTEM entry, the ACTUAL-LOCATION part can be relative to the location of the catalog file.

1.2.4. Specifying the catalog file

Once you write a catalog file, you'd need to specify that when you invoke XJC.

CLI To do this from the CLI, use the `-catalog` option. See **xjc -help** for more details.

Ant Use the catalog attribute on the `<xjc>` task. See XJC ant task documentation for more details.

Maven For the Maven plugin, use the `<catalog>` element in the configuration:

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <configuration>
        <!-- relative to the POM file -->
        <catalog>mycatalog.cat</catalog>
    </copnfiguration>
</plugin>
```

1.2.5. Debugging catalog file

If you are trying to write a catalog file and banging your head against a wall because it's not working, you should enable the verbose option of the catalog resolver. How you do this depends on what interface you use:

CLI Specify **export XJC_OPTS="-Dxml.catalog.verbosity=999"** then run XJC.

Ant/Maven Add `-Dxml.catalog.verbosity=999` as a command line option to Ant/Maven.

If you are otherwise invoking XJC programmatically, you can set the above system property before invoking XJC.

1.3. Mapping of `<xs:any />`

XJC binds `<xs:any />` in the following ways:

1.3.1. `processContents="skip"`

`<xs:any />` with `processContents=skip` means any well-formed XML elements can be placed. Therefore, XJC binds this to DOM `Element` interface.

Example 11. Any/Skip schema

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:any processContents="skip" maxOccurs="unbounded" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example 12. Any/Skip binding

```
import org.w3c.dom.Element;

@XmlRootElement
class Person {
  public String getName();
  public void setName(String);

  @XmlAnyElement
  public List<Element> getAny();
}
```

1.3.2. `processContents="strict"`

`<xs:any />` with `processContents=strict` (or `<xs:any />` without any processContents attribute, since it defaults to "strict") means any XML elements placed here must have corresponding schema definitions. This mode is not what people typically expect as "`wildcard`", but this is the default. The following shows this binding. (`lax=true` is unintuitive, but it's not an error in this document):

Example 13. Any/Strict schema

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:any maxOccurs="unbounded" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example 14. Any/Strict binding

```
@XmlRootElement
class Person {
  public String getName();
  public void setName(String);

  @XmlAnyElement(lax=true)
  public List<Object> getAny();
}
```

Jakarta XML Binding binds any such element to an `Object`, and during unmarshalling, all elements encountered are unmarshalled into corresponding Jakarta XML Binding objects (including `JAXBElement`s if necessary) and placed in this field. If it encounters elements that cannot be unmarshalled, DOM elements are produced instead.

At runtime, you can place either DOM elements or some Jakarta XML Binding objects that map to elements. A typical mistake is to put a `String` that contains XML fragment, but this won't work; you'd have to first read that into a DOM.

1.3.3. `processContents="lax"`

`<xs:any />` with `processContents=lax` means any XML elements can be placed here, but if their element names match those defined in the schema, they have to be valid. XJC actually handles this exactly like processContents='strict', since the strict binding allows unknown elements anyway.

1.4. Mapping of `<xs:element />` to JAXBElement

Sometimes XJC binds an element declaration to `JAXBElement`. Sometimes XJC binds an element declaration to a Java class. What makes this difference?

1.5. How modularization of schema interacts with XJC

Over time schema authors have developed several techniques to modularize large schemas. Some of those techniques have some noteworthy interactions with XJC.

1.5.1. Chameleon schema

Chameleon schema" [http://www.xfront.com/ZeroOneOrManyNamespaces.html#mixed] (read more [http://www.google.com/search?q=chameleon+schema], in particular this [http://www.kohsuke.org/xmlschema/XMLSchemaDOsAndDONTs.html#avoid_chameleon]) is a technique used to define multiple almost-identical sets of definitions into multiple namespaces from a single schema document.

For example, with this technique, you can write just one "foo" complex type and define it into namespace X and Y. In this case, one tends to hope that XJC will only give you one `Foo` class for this, but unfortunately because it's actually defined in two namespaces, Jakarta XML Binding needs two Java classes to distinguish `X:foo` and `Y:foo`, so you'll get multiple copies.

If you find this to be problematic, there are a few ways to work around the problem.

1. If you are in control of the schema, see if you can rewrite the schema to avoid using this technique. In some cases, the schema doesn't actually exploit the additional power of this technique, so this translation can be done without affecting XML instance documents. In some other cases, the chameleon schema can be argued as a bad schema design, as it duplicates definitions in many places.
2. If you are not in control of the schema, see if you can rewrite the schema nevertheless. This will only work if your transformation doesn't affect XML instance documents.
3. Perhaps there can be a plugin that eases the pain of this, such as by defining common interfaces among copies.

1.6. Adding behaviors

Adding behaviors to the generated code is one area that still needs improvement. Your feedback is appreciated.

Suppose if Eclipse Implementation of JAXB generated the following classes.

Example 15. Simple Eclipse Implementation of JAXB Generated Code

```
package org.acme.foo;
```

```
@XmlRootElement
class Person {
  private String name;

  public String getName() { return name; }
  public void setName(String) { this.name=name; }
}

@XmlRegistry
class ObjectFactory {
  Person createPerson() { ... }
}
```

To add a behavior, first write a class that extends from `Person`. You also need to extend ObjectFactory to return this new class. Notice that neither classes have any Jakarta XML Binding annotation, and I put them in a separate package. This is because we'd like `PersonEx` class to be used in place of `Person`, and we don't want `PersonEx` to be bound to its own XML type.

Example 16. Extended Person class

```
package org.acme.foo.impl;

class PersonEx extends Person {
  @Override
  public void setName(String name) {
    if(name.length()<3) throw new IllegalArgumentException();
    super.setName(name);
  }
}

@XmlRegistry
class ObjectFactoryEx extends ObjectFactory {
  @Override
  Person createPerson() {
    return new PersonEx();
  }
}
```

At runtime, you can create `JAXBContext` normally, like this.

Example 17. Creating JAXBContext

```
JAXBContext context = JAXBContext.newInstance(ObjectFactory.class);
// or JAXBContext.newInstance("org.acme.foo");
```

`PersonEx` can be marshalled out just like `Person`:

Example 18. Marshalling

```
Person p = new PersonEx();
context.createMarshaller().marshal(p,System.out);
// this will produce <person />
```

To unmarshal XML documents into `PersonEx`, you'll need to configure the unmarshaller to use your `ObjectFactoryEx` as the factory, like this:

Example 19. Unmarshalling

```
Unmarshaller u = context.createUnmarshaller();
```

```
u.setProperty("org.glassfish.jaxb.core.ObjectFactory",new ObjectFactoryEx());
PersonEx p = (PersonEx)u.unmarshal(new StringReader("<person />"));
```

If you have multiple packages and thus multiple `ObjectFactory`s, you can pass in an array of them (`new Object[]{new OFEx1(),new OFEx2(),...}`.)

1.6.1. Inserting your class in the middle

If you have a type hierarchy and would like to insert your class in the middle, you can use the combination of `XmlTransient` and `@implClass` of `<class>` customization. See the following example:

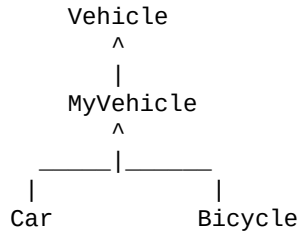
Example 20. Hierarchy of types and `<jaxb:class implClass>`

```
<xs:schema ...>
  <xs:complexType name="vehicle">
    <xs:annotation><xs:appinfo>
      <jaxb:class implClass="MyVehicle" />
    </xs:appinfo></xs:annotation>
  </xs:complexType>

  <xs:complexType name="car">
    <xs:complexContent>
      <xs:extension base="vehicle" />
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="bicycle">
    <xs:complexContent>
      <xs:extension base="vehicle" />
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Example 21. This creates a class hierarchy like the following (among the generated Java code):



You'll then manually write `MyVehicle` class that extends from `Vehicle`. Annotate this class with `XmlTransient` to achieve the desired effect.

1.7. Avoid strong databinding

Under some limited circumstances, a weaker databinding is preferable for various reasons. Jakarta XML Binding does offer a few ways for you to achieve this.

1.7.1. Avoid mapping to enum

The following customization will stop binding a simple type to a type-safe enum. This can be convenient when number of constants is too large to be an useful enum (by default, the Jakarta XML Binding spec won't generate enum with more than 256 constants, but even 100 might be too large for you.)

Example 22. Avoid mapping one simple type

```
<xs:simpleType name="foo">
  <xs:annotation><xs:appinfo>
    <jaxb:typesafeEnumClass map="false" />
  </xs:appinfo></xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="x" />
    <xs:enumeration value="y" />
    <xs:enumeration value="z" />
  </xs:restriction>
</xs:simpleType>
```

To disable such type-safe enum binding altogether for the entire schema, use a global binding setting like this (this is actually telling XJC not to generate enums if a simple type has more than 0 constants --- the net effect is no enum generation):

Example 23. Avoid generating enums at all

```
<xs:schema ...>
  <xs:annotation><xs:appinfo>
    <jaxb:globalBindings typesafeEnumMaxMembers="0" />
  </xs:appinfo></xs:annotation>
  ...
</xs:schema>
```

1.7.2. Mapping to DOM

The `<jaxb:dom>`customization allows you to map a certain part of the schema into a DOM tree. This customization can be attached to the following schema components:

- Wildcards (`<xs:any>`)
- Type definitions (`<xs:complexType>` and `<xs:simpleType>`)
- Model groups (`<xs:choice>`,`<xs:all>`,`<xs:sequence>`)
- Model group declarations (`<xs:group>`)
- Particles
- Element declarations (`<xs:element>`)

In the following example, a wildcard is mapped to a DOM node. Each element that matches to the wildcard will be turned into a DOM tree.

Example 24. Dom Customization example

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
               jaxb:version="3.0">

        <xs:element>
           <xs:complexType>
              <xs:sequence>
                 <xs:any maxOccurs="unbounded" processContents="skip">
                    <xs:annotation><xs:appinfo>
```

```
                    <jaxb:dom/>
                  </xs:appinfo></xs:annotation>
               </xs:any>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   .
   .
   .
   </xs:schema>
```

This extension can be used to access wildcard content or can be used to process a part of a document by using other technologies that require "raw" XML. By default, Jakarta XML Binding generates a `get-Content()` method for accessing wildcard content, but it only supports "lax" handling which means that unknown content is discarded. You may find more information in 7.12 chapter of Jakarta XML Binding 2 specification [https://jakarta.ee/specifications/xml-binding/].

1.8. Working with generated code in memory

1.8.1. Cloning

The generated beans (and in particular the `JAXBElement` class) do not support the clone operation. There was a suggestion by another user that beanlib [http://beanlib.sourceforge.net/] has been used successfully to clone Jakarta XML Binding objects.

2. Customization of Schema Compilation

2.1. Customizing Java packages

The Jakarta XML Binding specification provides a `<jaxb:schemaBindings>` customization so that you can control which namespace goes to which package. See the example below:

Example 25. package customization

```
<jaxb:schemaBindings>
  <jaxb:package name="org.acme.foo"/>
</jaxb:schemaBindings>
```

You can do this as an internal customization (in which case you put this in `<xs:annotation><xs:appinfo>` under place it right under the `<xs:schema>` element), or do this as an external customization, like this:

Example 26. External package customization

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0">
  <bindings schemaLocation="../path/to/my.xsd">
    <schemaBindings>
      <package name="org.acme.foo"/>
    </schemaBindings>
  </bindings>
</bindings>
```

Note that this customization is per namespace. That is, even if your schema is split into multiple schema documents, you cannot put them into different packages if they are all in the same namespace.

2.1.1. Tip: get rid of the org.w3._2001.xmlschema package

Under some rare circumstances, XJC will generate some Java classes into a package called `org.w3._2001.xmlschema`. This happens when XJC decides that it needs some Java artifacts for the XML Schema built-in namespace of `http://www.w3.org/2001/XMLSchema`.

Since this package name is most often problematic, you can rename this by simply saving the following text in an .xsd file and submitting it to XJC along with the other schemas you have:

Example 27. Schemalet to get rid of org.w3._2001.xmlschema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.w3.org/2001/XMLSchema"
  xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
  jaxb:version="3.0">
  <annotation><appinfo>
    <jaxb:schemaBindings>
      <jaxb:package name="org.acme.foo"/>
    </jaxb:schemaBindings>
  </appinfo></annotation>
</schema>
```

This is bit tricky, but the idea is that since you can define a schema for one namespace in multiple schema documents, this makes XJC think that this schema is a part of the built-in "XML Schema for XML Schema".

2.2. Using SCD for customizations

When using an external customization file, the Jakarta XML Binding spec requires that you use XPath as a means to specify what your customization is attached to. For example, if you want to change the class name generated from a complex type, you'd write something like:

Example 28. External customization example

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <bindings schemaLocation="../path/to/my.xsd" node="/xs:schema/
xs:complexType[@name='foo']">
    <class name="FooType"/>
  </bindings>
</bindings>
```

While the above process works, the problem with this is that the XPath+ `schemaLocation` combo tends to be verbose and error prone. It's verbose, because often a trivial target schema component like this "global complex type foo" takes up a lot of characters. The xs namespace declaration also takes up some space, although in this case we managed to avoid declaring the "tns" namespace (that represents the namespace that the schema defines.)

It's also error prone, because it relies on the way schema documents are laid out, because the schemaLocation attribute needs to point to the right schema document file. When a schema is split into multiple files for modularity (happens especially often with large schemas), then you'd have to find which schema file it is. Even though you can use relative paths, this hard-coding of path information makes it hard to pass around the binding file to other people.

JAXB RI 2.1 and onward offers a better way to do this as a vendor extension.

The key technology to solve this problem is a "`schema component designator`" [http://www.w3.org/TR/xmlschema-ref/] (SCD.) This is a path language just like XPath, but whereas XPath is designed to refer to XML infoset items like elements and attributes, SCD is designed to refer to schema components like element declarations or complex types.

With SCD, the above binding can be written more concisely as follows:

Example 29. External customization by SCD

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0"
 xmlns:tns="http://my.namespace/">
  <bindings scd="/type::tns:foo">
    <class name="FooType"/>
  </bindings>
</bindings>
```

`/type::tns:foo` can be written more concisely as `/~tns:foo`, too. If you are interested in more about the syntax of SCDs, read the example part of the spec [http://www.w3.org/TR/xmlschema-ref/#section-path-examples], and maybe EBNF [http://www.w3.org/TR/xmlschema-ref/#section-path-ebnf]. If you know XPath, I think you'll find this fairly easy to learn.

Another benefit of an SCD is that tools will have easier time generating SCDs than XPath, as XPaths are often vulnerable to small changes in the schema document, while SCDs are much more robust. The downside of using SCD is as of JAXB 2.1, this feature is a vendor extension and not defined in the spec.

2.3. Using different datatypes

Eclipse Implementation of JAXB has a built-in table that determines what Java classes are used to represent what XML Schema built-in types, but this can be customized.

One of the common use cases for customization is to replace the `XMLGregorianCalendar` with the friendlier `Calendar` or `Date`. `XMLGregorianCalendar` is designed to be 100% compatible with XML Schema's date/time system, such as providing infinite precision in sub-seconds and years, but often the ease of use of those familiar Java classes win over the precise compatibility.

One very easy way to do this is to simply use your IDE (or even "`sed`") to replace all the references to `XMLGregorianCalendar` by `Calendar`. This is of course not a very attractive option if your build process runs XJC as a part of it.

Alternatively, the following customization file can be used to do this. When using external customization file, the Jakarta XML Binding spec requires you to use XPath as a means to specify what your customization is attached to. For example, if you want to change the class name generated from a complex type, you'd use the following customization:

Example 30. Customization to use Calendar for `xs:date`

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <globalBindings>
    <javaType name="java.util.Calendar" xmlType="xs:date"
      parseMethod="jakarta.xml.bind.DatatypeConverter.parseDate"
      printMethod="jakarta.xml.bind.DatatypeConverter.printDate"
    />
  </globalBindings>
</bindings>
```

Save this in a file and specify this to Eclipse Implementation of JAXB with the "-b" option.

To use the `Date` class, you'll need to do a bit more work. First, put the following class into your source tree:

Example 31. Adapter for Date

```
package org.acme.foo;
public class DateAdapter {
  public static Date parseDate(String s) {
    return DatatypeConverter.parseDate(s).getTime();
  }
  public static String printDate(Date dt) {
    Calendar cal = new GregorianCalendar();
    cal.setTime(dt);
    return DatatypeConverter.printDate(cal);
  }
}
```

... then your binding file will be the following:

Example 32. Customization to use Date for `xs:date`

```
<bindings xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <globalBindings>
    <javaType name="java.util.Date" xmlType="xs:date"
      parseMethod="org.acme.foo.DateAadpter.parseDate"
      printMethod="org.acme.foo.DateAdapter.printDate"
    />
  </globalBindings>
</bindings>
```

3. Annotating Your Classes

3.1. Mapping your favorite class

3.1.1. ResultSet

Jakarta XML Binding (or any other databinding engine, for that matter) is for binding strongly-typed POJO-like objects to XML, such as `AddressBook` class, `PurchaseOrder` class, and so on, where you have fields and methods that shape a class.

There are other kinds of classes that are more close to reflection. Those classes don't have methods like `getAddress`, and instead you'd do `get("Address")`. JDBC ResultSet is one of those classes. It's one class that represents million different data structures, be it a customer table or a product table. Generally speaking, these classes does not allow Jakarta XML Binding to statically determine what the XML representation should look like. Instead, you almost always need to look at an instance to determine the shape of XML.

These classes are not really suitable for binding in Jakarta XML Binding. If this is the only object that you'd want to write out, then you'd be better off using XMLStreamWriter or some such XML infoset writing API. There are a few online articles [http://www.google.com/search?q=ResultSet+XML] that cover this topic. Also, many modern database offers a native ability to export a query into XML, which is supposed to work a lot faster than you'd do in Java (and saves your time of writing code.)

If you are using ResultSet as a part of your object tree that you want to marshal to Jakarta XML Binding, then you can use `XmlJavaTypeAdapter`.

3.1.2. HashMap

Jakarta XML Binding spec defines a special handling for `Map` when it's used as a propety of a bean. For example, the following bean would produce XMLs like the following:

Example 33. Bean with Map

```
@XmlRootElement
class Foo {
  public HashMap<String,Integer> map;
}
```

Example 34. XML representation

```
<foo>
  <map>
    <entry>
      <key>a</key>
      <value>1</value>
    </entry>
    <entry>
      <key>b</key>
      <value>2</value>
    </entry>
  </map>
</foo>
```

Unfortunately, as of 2.1, this processing is only defined for bean properties and not when you marshal `HashMap` as a top-level object (such as a value in `JAXBElement`.) In such case, `HashMap` will be treated as a Java bean, and when you look at `HashMap` as a bean it defines no getter/setter property pair, so the following code would produce the following XML:

Example 35. Bean with Map

```
m = new HashMap();
m.put("abc",1);
marshaller.marshal(new JAXBElement(new
 QName("root"),HashMap.class,m),System.out);
```

Example 36. XML representation

```
<root />
```

This issue has been recorded as #223 [https://github.com/eclipse-ee4j/jaxb-ri/issues/223] and the fix needs to happen in later versions of the Jakarta XML Binding spec.

In the mean time, such top-level objects have to be first adapted to a bean that Jakarta XML Binding can process. This has added benefit of being able to control XML representation better. The following code illustrates how to do this:

Example 37. Adapting HashMap

```
public class MyHashMapType {
    public List<MyHashMapEntryType> entry = new
 ArrayList<MyHashMapEntryType>();
    public MyHashMapType(Map<String,Integer> map) {
```

```
        for( Map.Entry<String,Integer> e : map.entrySet() )
            entry.add(new MyHashMapEntryType(e));
    }
    public MyHashMapType() {}
}

public class MyHashMapEntryType {
    @XmlAttribute // @XmlElement and @XmlValue are also fine
    public String key;

    @XmlAttribute // @XmlElement and @XmlValue are also fine
    public int value;

    public MyHashMapEntryType() {}
    public MyHashMapEntryType(Map.Entry<String,Integer> e) {
       key = e.getKey();
       value = e.getValue();
    }
}

marshaller.marshal(new JAXBElement(new QName("root"),MyHashMapType.class,new
 MyHashMapType(m)),System.out);
```

If you have a lot of difference kinds of `Map`, you can instead use `Object` as the key and the value type. In that way, you'll be able to use maps with different type parameters, at the expense of seeing `xsi:type` attribute on the instance document.

3.2. Mapping interfaces

Because of the difference between the XML type system induced by W3C XML Schema and the Java type system, Jakarta XML Binding cannot bind interfaces out of the box, but there are a few things you can do.

3.2.1. Use `@XmlRootElement`

When your interface is implemented by a large number of sub-classes, consider using `XmlRootElement` annotation like this:

Example 38. XmlRootElement for open-ended interfaces

```
@XmlRootElement
class Zoo {
  @XmlAnyElement
  public List<Animal> animals;
}

interface Animal {
  void sleep();
  void eat();
  ...
}

@XmlRootElement
class Dog implements Animal { ... }

@XmlRootElement
class Lion implements Animal { ... }
```

This will produce XML documents like this:

Example 39. XML for XmlRootElement

```
<zoo>
    <lion> ... </lion>
    <dog> ... </dog>
</zoo>
```

The key characteristics of this approach is:

1. Implementations are open-ended; anyone can implement those interfaces, even by different people from different modules, provided they are all given to the `JAXBContext.newInstance` method. There's no need to list all the implementation classes in anywhere.

2. Each implementation of the interface needs to have an unique element name.

3. Every reference to interface needs to have the `XmlElementRef` annotation. The `type=Object.class` portion tells Jakarta XML Binding that the greatest common base type of all implementations would be `java.lang.Object`.

`@XmlElementWrapper` is often useful with this, as it allows you need to group them. Such as:

Example 40. XmlRootElement for open-ended interfaces

```
@XmlRootElement
class Zoo {
  @XmlElementWrapper
  @XmlAnyElement
  public List<Animal> onExhibit;
  @XmlElementWrapper
  @XmlAnyElement
  public List<Animal> resting;
}
```

Example 41. Effect of XmlElementWrapper

```
<zoo>
    <onExhibit>
        <lion> ... </lion>
        <dog> ... </dog>
    </onExhibit>
    <resting>
        <lion> ... </lion>
        <dog> ... </dog>
    </resting>
</zoo>
```

3.2.2. Use `@XmlJavaTypeAdapter`

When you use interfaces just to hide your implementation classes from exposure, and when there's 1-to-1 (or close to 1-on-1) relationship between a class and an interface, `XmlJavaTypeAdapter` can be used like below.

Example 42. XmlJavaTypeAdapter for interfaces

```
@XmlJavaTypeAdapter(FooImpl.Adapter.class)
interface IFoo {
  ...
}
```

```
class FooImpl implements IFoo {
  @XmlAttribute
  private String name;
  @XmlElement
  private int x;

  ...

  static class Adapter extends XmlAdapter<FooImpl,IFoo> {
    IFoo unmarshal(FooImpl v) { return v; }
    FooImpl marshal(IFoo v) { return (FooImpl)v; }
  }
}

class Somewhere {
  public IFoo lhs;
  public IFoo rhs;
}
```

Example 43. XML of XmlJavaTypeAdapter

```
<somewhere>
  <lhs name="...">
    <x>5</x>
  </lhs>
  <rhs name="...">
    <x>5</x>
  </rhs>
</somewhere>
```

The key characteristics of this approach is:

1. Interface and implementation will be tightly coupled through an adapter, although changing an adapter code will allow you to support multiple implementations.

2. There's no need of any annotation in where interfaces are used.

A variation of this technique is when you have a few implementations for interface, not just one.

Example 44. XmlJavaTypeAdapter for interfaces with multiple implementations

```
@XmlJavaTypeAdapter(AbstractFooImpl.Adapter.class)
interface IFoo {
  ...
}
abstract class AbstractFooImpl implements IFoo {
  ...

  static class Adapter extends XmlAdapter<AbstractFooImpl,IFoo> {
    IFoo unmarshal(AbstractFooImpl v) { return v; }
    AbstractFooImpl marshal(IFoo v) { return (AbstractFooImpl)v; }
  }
}

class SomeFooImpl extends AbstractFooImpl {
  @XmlAttribute String name;
  ...
}
```

```
class AnotherFooImpl extends AbstractFooImpl {
  @XmlAttribute int id;
  ...
}

class Somewhere {
  public IFoo lhs;
  public IFoo rhs;
}
```

Example 45. XML of XmlJavaTypeAdapter with multiple implementations

```
<somewhere>
  <lhs xsi:type="someFooImpl" name="...">
  </lhs>
  <rhs xsi:type="anotherFooImpl" id="3" />
</somewhere>
```

Note that `SomeFooImpl` and `AnotherFooImpl` must be submitted to `JAXBContext.newInstance` one way or the other.

To take this example a bit further, you can use `Object` instead of `AbstractFooImpl`. The following example illustarates this:

Example 46. XmlJavaTypeAdapter for interfaces with multiple implementations

```
@XmlJavaTypeAdapter(AnyTypeAdapter.class)
interface IFoo {
  ...
}
public class AnyTypeAdapter extends XmlAdapter<Object,Object> {
  Object unmarshal(Object v) { return v; }
  Object marshal(Object v) { return v; }
}

class SomeFooImpl implements IFoo {
  @XmlAttribute String name;
  ...
}

class Somewhere {
  public IFoo lhs;
  public IFoo rhs;
}
```

Example 47. Corresponding schema

```
<xs:complexType name="somewhere">
  <xs:sequence>
    <xs:element name="lhs" type="xs:anyType" minOccurs="0"/>
    <xs:element name="rhs" type="xs:anyType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

As you can see, the schema will generated to accept `xs:anyType` which is more relaxed than what the Java code actually demands. The instance will be the same as the above example. Starting from JAXB RI 2.1, we bundle the `AnyTypeAdapter` class in the runtime that defines this adapter. So you won't have to write this adapter in your code.

3.2.3. Use `@XmlElement`

If the use of interface is very little and there's 1-to-1 (or close to) relationship between interfaces and implementations, then you might find `XmlElement` to be the least amount of work.

Example 48. XmlElement for interfaces

```
interface IFoo {
  ...
}
class FooImpl implements IFoo {
  ...
}

class Somewhere {
  @XmlElement(type=FooImpl.class)
  public IFoo lhs;
}
```

Example 49. XML of XmlElement

```
<somewhere>
  <lhs> ... </lhs>
</somewhere>
```

This effectively tells Jakarta XML Binding runtime that "even though the field is `IFoo`, it's really just `FooImpl`.

In this approach, a reference to an interface has to have knowledge of the actual implementation class. So while this requires the least amount of typing, it probably wouldn't work very well if this crosses module boundaries.

Like the `XmlJavaTypeAdapter` approach, this can be used even when there are multiple implementations, provided that they share the common ancestor.

The extreme of this case is to specify `@XmlElement(type=Object.class)`.

3.2.4. Hand-write schema

Occasionally the above approaches cause the generated schema to become somewhat ugly, even though it does make the Jakarta XML Binding runtime work correctly. In such case you can choose not to use the generated schema and instead manually modify/author schemas tht better match your needs.

3.2.5. Do schema-to-java

With sufficient knowlege, one can also use `<jaxb:class ref="..."/>` annotation so that you can cause XJC to use the classes you already wrote. See this thread for an example. TODO: more details and perhaps an example.

3.2.6. DOESN'T WORK: Have Jakarta XML Binding generate interaces and swap different implementations

Some users attempted to use the "`generateValueClass`" customization and see if they can completely replace the generated implementations with other implementations. Unfortunately, this does not work.

Even with the interface/implementation mode, Jakarta XML Binding runtime still requires that the implementation classes have all the Jakarta XML Binding annotations. So just implementing interfaces is not sufficient. (This mode is mainly added to simplify the migration from JAXB 1.0 to Jakarta XML Binding, and that's a part of the reason why things are done this way.)

3.3. Evolving annotated classes

Here is the basic problem of evolution. You got your CoolApp v1, which contains class Foo that has some Jakarta XML Binding annotations. Now you are working towawrd CoolApp v2, and you want to make some changes to Foo. But you want to do so in such a way that v1 and v2 can still talk to each other.

The evolution compatibility has two different aspects. One is the *schema compatibility*, which is about the relationship between the v1 schema and the v2 schema. The other is about *runtime compatibility*, which is about reading/writing documents between two versions.

3.3.1. Runtime compatibility

There are two directions in the runtime compatibility. One is whether v1 can still read what v2 write (*forward compatible*), and the other is whether v2 can read what v1 wrote (*backward compatible*).

3.3.2. "`Semi-compatible`"

Jakarta XML Binding can read XML documents that don't exactly match what's expected. This is the default behavior of the Jakarta XML Binding unmarshaller, yet you can change it to a more draconian behavior (TODO: pointer to the unmarshalling section.)

When we are talking about evolving classes, it's often convenient to leave it in the default behavior, as that would allow Jakarta XML Binding to nicely ignore elements/attributes newly added in v2. So we call it *backward semi-compatible* if v2 can read what v1 wrote in this default unmarshalling mode, and similarly *forward semi-compatible* if v1 can read what v2 wrote in this default unmarshalling mode.

Technically, these are weaker than true backward/forward compatibility (since you can't do a draconian error detection), yet in practice it works just fine.

3.3.3. Adding/removing/changing non-annotated things

You can add, remove, or change any non-annotated fields, methods, inner/nested types, constructors, interfaces. Those changes are both backward and forward compatible, as they don't cause any change to the XML representation.

Adding super class is backward compatible and forward semi-compatible. Similarly, removing super class is forward compatible and backward semi-compatible.

3.3.4. Adding/removing/changing properties

Adding new annotated fields or methods is backward compatible and forward semi-compatible. Similarly, removing them is forward compatible and backward semi-compatible.

Changing a property is bit more tricky.

1. If you change the property name from X to Y, that would be the equivalent of deleting X and adding Y, so it would be backward and forward semi-compatible. What Jakarta XML Binding really cares is properties' XML names and not Java names, so by using the `name` parameter of `XmlElement`,

`XmlAttribute` et al, you can change Java property names without affecting XML, or change XML without affecting Java properties. These are backward and forward semi-compatible. See below:

2. **Example 50. Changing Java without affecting XML**

```
// BEFORE
public class Foo {
    public String abc;
}
// AFTER: Java name changed, but XML remains the same
public class Foo {
    @XmlElement(name="abc")
    public String def;
}
```

Example 51. Changing XML without affecting Java

```
// BEFORE
public class Foo {
    public String abc;
}
// AFTER: no Java change, but XML will look different
public class Foo {
    @XmlElement(name="def")
    public String abc;
}
```

3. If you change a property type, generally speaking it will be not compatible at all. For example, you can't change from `java.util.Calendar` to `int` and expect it to work. To make it a somewhat compatible change, the old type and the new type has to be related. For example, `String` can represent all `int` values, so changing `int` to `String` would be a backward compatible and forward semi-compatible change. `XmlJavaTypeAdapter` allows you to make changes to Java without affecting XML (or vice versa.)

3.3.5. Changing class names

`XmlType` and `XmlRootElement` allows you to change a class name without affecting XML.

Example 52. Changing class name without affecting XML (1)

```
// BEFORE
@XmlRootElement
public class Foo { ... }

// AFTER: no XML change
@XmlRootElement(name="foo")
@XmlType(name="foo")
public class Bar { ... }
```

Example 53. Changing class name without affecting XML (2)

```
// BEFORE
public class Foo { ... }

// AFTER: no XML change
@XmlType(name="foo")
public class Bar { ... }
```

3.3.6. Schema Compatibility

TODO.

3.4. XML layout and in-memory data layout

Your program sometimes needs to have a different in-memory data structure from its XML representation. Jakarta XML Binding has a few different ways to achieve this.

3.4.1. XmlJavaTypeAdapter

`XmlJavaTypeAdapter` allows you to de-couple the in-memory representation and the XML representation by introducing an intermediate representation. The basic model is as follows:

```
In-memory objects  <===>  Intermediate objects   <===>
XML
                  adapter                          XMLBinding
```

Your adapter code will be responsible for converting in-memory objects to/from intermediate objects. Intermediate objects are then bound to XML by following the standard Jakarta XML Binding rules. See `XmlAdapter` for a general description of how adapters works.

Adapters extend from the `XmlAdapter` class and provide two methods "unmarshal" and "marshal" that converts values in both directions, and then the `XmlJavaTypeAdapter` annotation is used to tell Jakarta XML Binding where and what adapters kick in.

(TODO: more info about XmlJavaTypeAdapter needed)

1. adapting a class
2. adapting a property
3. adapting an external class
4. adapting a collection and its effect
5. adapting and using interfaces

3.4.2. Using XmlJavaTypeAdapter for element/attribute values

One of the common use cases of `XmlJavaTypeAdapter` is to map a "value object" to a string in XML. The following example illustrates how to do this, by using `java.awt.Color` as an example.

Example 54. Mapping Color to #RRGGBB

```
@XmlRootElement
class Box {
  @XmlJavaTypeAdapter(ColorAdapter.class)
  @XmlElement
  Color fill;
}

class ColorAdapter extends XmlAdapter<String,Color> {
  public Color unmarshal(String s) {
    return Color.decode(s);
```

```
  }
  public String marshal(Color c) {
    return "#"+Integer.toHexString(c.getRGB());
  }
}
```

This maps to the following XML representation:

Example 55. Box instance

```
<box>
  <fill>#112233</fill>
</box>
```

Since `XmlJavaTypeAdapter` is on a field, this adapter only kicks in for this particular field. If you have many `Color` fields and would like them all to use the same adapter, you can move the annotation to a package:

Example 56. package-info.java

```
@XmlJavaTypeAdapter(type=Color.class,value=ColorAdapter.class)
package foo;
```

Example 57. Box.java

```
@XmlRootElement
class Box {
  @XmlElement Color fill;
  @XmlElement Color border;
}
```

This causes all the fields in the classes in the `foo` package to use the same specified adapter.

Also see the `DatatypeConverter` class that defines a series of basic conversion routines that you may find useful.

3.4.3. Pair property

Another useful technique is to define two properties, one for Jakarta XML Binding and the other for your application. See the following example:

Example 58. Pair property sample

```
@XmlRootElement
class Person {
  private int age;

  // This public property is for users
  @XmlTransient
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }

  // This property is for Jakarta XML Binding
```

```
  @XmlAttribute(name="age")
  private String getAge_() {
    if(age==-1)  return "dead";
    else         return String.valueOf(age);
  }
  private void setAge_(String v) throws NumberFormatException {
    if(v.equals("dead"))   this.age=-1;
    else                   this.age=Integer.parseInt(age);
}
```

The main "`age`" property is public, but marked as `XmlTransient`, so it's exposed in your program, but Jakarta XML Binding will not map this to XML. There's another private "`age_`" property. Since this is marked with `XmlAttribute`, this is what Jakarta XML Binding is going to use to map to the attribute. The getter and setter methods on this property will handle the conversion between the in-memory representation and the XML representation.

3.5. Mapping cyclic references to XML

Object models designed in Java often have cycles, which prevent straight-forward conversion to XML by Jakarta XML Binding. In fact, when you try to marshal an object tree that contains a cycle, the Jakarta XML Binding marshaller reports an error, pointing out the objects that formed the cycle. This is because Jakarta XML Binding by itself cannot figure out how to cut cycles into a tree.

Thus it is your responsibility to annotate classes and use other means to "tell" Jakarta XML Binding how to handle a cycle. This chapter talks about various techniques to do this.

3.5.1. Parent pointers

One of the very common forms of cycle is a parent pointer. The following example illustrates a typical parent pointer, and how this can be turned into "natural" XML:

Example 59. Classes with parent pointer

```
@XmlRootElement
class Department {
  @XmlAttribute
  String name;
  @XmlElement(name="employee")
  List<Employee> employees;
}

class Employee {
  @XmlTransient
  Department department;  // parent pointer
  @XmlAttribute
  String name;

  public void afterUnmarshal(Unmarshaller u, Object parent) {
    this.department = (Department)parent;
  }
}
```

This will produce the following XML:

Example 60. XML view of department

```
<department name="accounting">
```

```
  <employee name="Joe Chin" />
  <employee name="Adam Smith" />
  ...
</department>
```

And reading this document back into Java objects will produce the expected tree with all the proper parent pointers set up correctly.

The first technique here is the use of `XmlTransient` on the parent pointer. This tells Jakarta XML Binding that you don't need this parent pointer to be represented explicitly in XML, because the fact that `employee` is always contained inside `department` implies the parent/child relationship. This causes the marshaller to produce the expected XML. However, when you unmarshal it, since this field is not bound, the `Employee.department` field will be left null.

That's where the second technique comes in, which is the use of the `afterUnmarshal` callback. This method is invoked by the Jakarta XML Binding implementation on each instance when the unmarshalling of a `Employee` object completes. Furthermore, the second parameter to the method is the parent object, which in this case is a `Department` object. So in this example, this sets up the parent pointer correctly.

This callback can be also used to perform other post-unmarshalling set up work.

3.5.2. Many-to-many relationship

TBD

3.5.3. `@XmlID` and `@XmlIDREF`

When a reference to another object is annotated with `XmlIDREF`, its corresponding XML it will be referenced by `xs:IDREF`, instead of containment. See below for an example:

Example of `@XmlID` and `@XmlIDREF`

```
@XmlRootElement
class Root {
  List<Foo> foos;
  List<Bar> bars;
}
class Foo {
  // you don't have to make it an attribute, but that's more common
  @XmlAttribute @XmlIDREF Bar bar;
}
class Bar {
  // you don't have to make it an attribute, but that's more common
  @XmlAttribute @XmlID String id;
}
```

Example 61. Schema for above

```
<xs:complexType name="foo">
  <xs:sequence/>
  <xs:attribute name="bar" type="xs:IDREF"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="bar">
  <xs:sequence/>
  <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>
```

Example 62. A sample instance

```
<root>
  <foo bar="x"/>
  <foo bar="y"/>
  <bar id="x"/>
  <bar id="y"/>
</root>
```

There are a few things to consider when you do this. First, the object to be referenced must have an ID that is unique within the whole document. You'd also need to ensure that the referenced objects are *contained* somewhere else (like in the `Root` class in this case), or else `Bar` objects will never be marshalled. This technique can be used to remove the cyclic references, but it's only possible when your object model has an easy cut point.

3.5.4. Use the CycleRecoverable interface

Starting 2.1 EA2, the Eclipse Implementation of JAXB exposes `CycleRecoverable` interface. Applications can choose to implement this interface in some of its objects. When a cyclic reference is detected during marshalling, and if the object that formed a cycle implements this interface, then the method on this interface is called to allow an application to nominate its replacement to be written to XML. In this way, the application can recover from a cycle gracefully.

This technique allows you to cope with a situation where you cannot easily determine upfront as to where a cycle might happen. On the other hand, this feature is a Eclipse Implementation of JAXB feature. Another downside of this is that unless you nominate your replacement carefully, you can make the marshalling output invalid with respect to the schema, and thus you might hit another problem when you try to read it back later.

4. Unmarshalling

4.1. `@XmlRootElement` and unmarshalling

Classes with `XmlRootElement` can be unmarshalled from XML elements simply by invoking the unmarshal method that takes one parameter. This is the simplest mode of unmarshalling.

Unmarshalling with `@XmlRootElement`

```
@XmlRootElement
class Foo {
  @XmlAttribute
  String name;
  @XmlElement
  String content;
}

Unmarshaller u = ...;
Foo foo = (Foo)u.unmarshal(new File("foo.xml"));
```

Example 63. foo.xml

```
<foo name="something">
  <content>abc</content>
</foo>
```

However, sometimes you may need to unmarshal an instance of a type that does not have an `XmlRootElement`. For example, you might dynamically find out at the runtime that a certain element has a certain type. For example, the following document illustrates an XML instance where the content of `<someOtherTagName>` element is represented by the `Foo` class.

Example 64. foo2.xml

```
<someOtherTagName name="something">
  <content>abc</content>
</someOtherTagName>
```

To unmarshal this into a `Foo` class, use the version of the `unmarshal` method that takes the 'expectedType' argument, as follows:

Example 65. Unmarshalling into a known type

```
Unmarshaller u = ...;
JAXBElement<Foo> root = u.unmarshal(new StreamSource(new
 File("foo.xml")),Foo.class);
Foo foo = root.getValue();
```

To reduce the number of the `unmarshal` methods, this two-argument version is not defined for every single-argument version. So as in this example, you might need to perform additional wrapping of the input parameter.

This instructs Jakarta XML Binding that the caller is expecting to unmarshal `Foo` instance. Jakarta XML Binding returns a `JAXBElement` of `Foo`, and this `JAXBElement` captures the tag name of the root element.

4.2. Unmarshalling is not working! Help!

There are a few common causes for this problem. These causes often exhibit similar symptoms:

1. Instance documents are invalid
2. `JAXBContext` is not created correctly.

4.2.1. Make sure your instance document is valid

First, use an independent schema validator to check if your document is really valid with respect to the schema you compiled. When the root element of a document is invalid, then the unmarshaller will issue "unexpected element" errors. When a portion of a document is invalid, Eclipse Implementation of JAXB skips that portion, so the end result is that the unmarshalling returns normally, yet you notice that a part of the content tree is missing. This is often the desirable behavior, but it sometimes ends up masking a problem.

Also, try to install `ValidationEventHandler` on the unmarshaller. When a portion of a document is skipped, the unmarshaller notifies a `ValidationEventHandler`, so it allows you to see what's going on.

Example 66. Installing ValidationEventHandler

```
Unmarshaller u = ...;
// this implementation is a part of the API and convenient for trouble-
shooting,
// as it prints out errors to System.out
```

```
u.setEventHandler(new
 jakarta.xml.bind.helpers.DefaultValidationEventHandler());

u.unmarshal(new File("foo.xml"));
```

Also consider installing a `Schema` object to the unmarshaller, so that the unmarshaller performs a schema validation while unmarshalling. Earlier I suggested that you try an independent schema validator, but for various reasons (not all tools are reliable, you might have made an error and used a different schema/instance), using validating unmarshalling is a better way to guarantee the validity of your instance document being unmarshalled. Please follow the JAXP tutorial [http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXPIntro.html#wp65584] for more about how to construct a `Schema` object from your schema.

If you are unmarshalling from XML parser APIs (such as DOM, SAX, StAX), then also make sure that the parser/DOM is configured with the namespace enabled.

4.2.2. Check if your JAXBContext is correct

(TODO: This also applies to the marshaller. Think about moving it.)

The other possibility is that `JAXBContext` is not set up correctly. `JAXBContext` "knows" a set of classes, and if it doesn't know a class that it's supposed to know, then the unmarshaller may fail to perform as you expected.

To verify that you created `JAXBContext` correctly, call `JAXBContext.toString()`. It will output the list of classes it knows. If a class is not in this list, the unmarshaller will never return an instance of that class. Make you see all the classes you expect to be returned from the unmarshaller in the list. When dealing with a large schema that spans across a large number of classes and packages, this is one possible cause of a problem.

If you noticed that a class is missing, explicitly specify that to `JAXBContext.newInstance`. If you are binding classes that are generated from XJC, then the easiest way to include all the classes is to specify the generated `ObjectFactory` class(es).

4.3. Element default values and unmarshalling

Because of the "strange" way that element default values in XML Schema work, people often get confused about their behavior. This section describes how this works.

When a class has an element property with the default value, and if the document you are reading is missing the element, then the unmarshaller does *not* fill the field with the default value. Instead, the unmarshaller fills in the field when the element is present but the content is missing. See below:

Example 67. XML instance 1

```
<foo />
```

Example 68. XML instance 2

```
<foo>
  <a/>  <!-- or <a></a> -->
</foo>
```

Example 69. XML instance 3

```
<foo>
```

```
  <a>abc</a>
</foo>
```

Example 70. Element defaults and XML

```
@XmlRootElement
class Foo {
  @XmlElement(defaultValue="value") public String a=null;
}

Foo foo = unmarshaller.unmarshal("instance1.xml");
System.out.println(foo.a);   // null

Foo foo = unmarshaller.unmarshal("instance2.xml");
System.out.println(foo.a);   // "value". The default kicked in.

Foo foo = unmarshaller.unmarshal("instance3.xml");
System.out.println(foo.a);   // "abc". Read from the instance.
```

This is consistent with the XML Schema spec, where it essentially states that the element defaults do not kick in when the element is absent, so unfortunately we can't change this behavior.

Depending on your expectation, using a field initializer may achieve what you are looking for. See below:

Example 71. Possible changes by using field initializer

```
@XmlRootElement
class Foo {
  @XmlElement public String a="value";
}

Foo foo = unmarshaller.unmarshal("instance1.xml");
System.out.println(foo.a);   // "value", because Jakarta XML Binding didn't
 overwrite the value

Foo foo = unmarshaller.unmarshal("instance2.xml");
System.out.println(foo.a);   // "", because <a> element had 0-length string
 in it

Foo foo = unmarshaller.unmarshal("instance3.xml");
System.out.println(foo.a);   // "abc". Read from the instance.
```

Alternatively, attribute default values work in a way that agrees with the typical expectation, so consider using that. Also, see Element default values and marshalling.

4.4. Dealing with large documents

Jakarta XML Binding API is designed to make it easy to read the whole XML document into a single tree of Jakarta XML Binding objects. This is the typical use case, but in some situations this is not desirable. Perhaps:

1. A document is huge and therefore the whole may not fit the memory.

2. A document is a live stream of XML (such as XMPP [http://www.xmpp.org/]) and therefore you can't wait for the EOF.

3. You only need to databind the portion of a document and would like to process the rest in other XML APIs.

This section discusses several advanced techniques to deal with these situations.

4.4.1. Processing a document by chunk

When a document is large, it's usually because there's repetitive parts in it. Perhaps it's a purchase order with a large list of line items, or perhaps it's an XML log file with large number of log entries.

This kind of XML is suitable for chunk-processing; the main idea is to use the StAX API, run a loop, and unmarshal individual chunks separately. Your program acts on a single chunk, and then throws it away. In this way, you'll be only keeping at most one chunk in memory, which allows you to process large documents.

See the streaming-unmarshalling example and the partial-unmarshalling example in the Eclipse Implementation of JAXB distribution for more about how to do this. The streaming-unmarshalling example has an advantage that it can handle chunks at arbitrary nest level, yet it requires you to deal with the push model --- Jakarta XML Binding unmarshaller will "`push`" new chunk to you and you'll need to process them right there.

In contrast, the partial-unmarshalling example works in a pull model (which usually makes the processing easier), but this approach has some limitations in databinding portions other than the repeated part.

4.4.2. Processing a live stream of XML

The techniques discussed above can be used to handle this case as well, since they let you unmarshal chunks one by one. See the xml-channel example in the Eclipse Implementation of JAXB distribution for more about how to do this.

4.4.3. Creating virtual infosets

For further advanced cases, one could always run a streaming infoset conversion outside Jakarta XML Binding API and basically curve just the portion of the infoset you want to data-bind, and feed it as a complete infoset into Jakarta XML Binding API. Jakarta XML Binding API accepts XML infoset in many different forms (DOM, SAX, StAX), so there's a fair amount of flexibility in choosing the right trade off between the development effort in doing this and the runtime performance.

For more about this, refer to the respective XML infoset API.

5. Marshalling

5.1. Changing prefixes

By default, a Jakarta XML Binding marshaller uses random namespace prefixes (such as `ns1`, `ns2`, ...) when it needs to declare new namespace URIs. While this is perfectly valid XML wrt the schema, for human readability, you might want to change them to something that makes more sense.

The Eclipse Implementation of JAXB defines NamespacePrefixMapper to allow you to do this. See the `namespace-prefix` sample in the distribution for more details.

5.2. Element default values and marshalling

Because of a "strange" way element default values in XML Schema work, people often get confused about its behavior. This section describes how this works.

When a class has an element property with the default value, and if a value is null, then the marshaller will not produce the corresponding element in XML:

Example 72. Element defaults and XML

```
@XmlRootElement
class Foo {
  @XmlElement(defaultValue="value") public String a=null;
}

marshaller.marshal(new Foo(),System.out);
```

Example 73. Marshalling output from above

```
<foo />
```

This is consistent with the XML Schema spec, where it essentially states that the element defaults do not kick in when the element is absent. Attribute default values do not have this problem, so if you can change the schema, changing it to an attribute is usually a better idea. Alternatively, depending on your expectation, setting the field to a default value in Java may achieve what you are looking for.

Example 74. Possible changes

```
@XmlRootElement
class Foo {
  @XmlElement public String a="value";
}
@XmlRootElement
class Bar {
  @XmlAttribute public String a;
}

marshaller.marshal(new Foo(),System.out);
marshaller.marshal(new Bar(),System.out);
```

Example 75. Marshalling output from above

```
<foo>
    <a>value</a>
</foo>

<bar/>
```

Also, see Element default values and unmarshalling.

5.3. Different ways of marshalling

5.3.1. Different output media

The most basic notion of the marshalling is to take a Jakarta XML Binding-bound object that has `@XmlRootElement`, and write it out as a whole XML document. So perhaps you have a class like this:

Example 76. Jakarta XML Binding POJO

```
class Point {
  @XmlElement
```

```
  public int x;
  @XmlElement
  public int y;
  Point(...) { ... }
}
```

Then you can do:

Example 77. Plain marshalling

```
marshaller.marshal( new Point(1,3), System.out );
marshaller.marshal( new Point(1,3), new File("out.xml") );
```

.. and so on. There're seven `Marshaller.marshal` methods that takes different output media as the second parameter. If you are writing to a file, a socket, or memory, then you should use the version that takes `OutputStream`. Unless you change the target encoding to something else (default is UTF-8), there's a special marshaller codepath for `OutputStream`, which makes it run really fast. You also don't have to use `BufferedOutputStream`, since the Eclipse Implementation of JAXB does the adequate buffering.

You can also write to `Writer`, but in this case you'll be responsible for encoding characters, so in general you need to be careful. If you want to marshal XML into an encoding other than UTF-8, it's best to use the `JAXB_ENCODING` property and then write to `OutputStream`, as it escapes characters to things like `&#x1824;` correctly.

The next medium we support is W3C DOM. This is bit unintuitive, but you'll do it like this:

Example 78. Marshal to DOM

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
Document doc = dbf.newDocumentBuilder().newDocument();

marshaller.marshal( new Point(1,3), doc );
```

And after the method invocation you get a complete DOM tree that represents the marshalled document.

The other versions of the marshal methods are there to write XML documents in terms of other XML APIs, such as SAX and StAX. The version that takes `ContentHandler` is useful when you need a custom formatting needs (like you want each attribute to be in new line, etc), but otherwise they are not very interesting if you are writing a whole document.

5.3.2. Marshalling into a subtree

Another common use of Jakarta XML Binding is where you are writing a bigger document, and you use Jakarta XML Binding to generate part(s) of it. The Eclipse Implementation of XML Web Services is the prime example. It produces a SOAP message, and Jakarta XML Binding is only used to produce the body. When you are doing this, you first set `JAXB_FRAGMENT` property on the marshaller. This changes the behaviors of the marshaller so that it works better in this situation.

If you are writing to an `OutputStream` or `Writer` and generally sending it to someone else, you can do something like this:

Example 79. Marshalling into a subtree

```
System.out.println("<envelope>");
```

```
marshaller.marshal( object, System.out );
System.out.println("</envelope>");
```

Like I mentioned, this is probably the fastest, even though `println` isn't very pretty. `JAXB_FRAGMENT` prevents the marshaller from producing an XML declaration, so the above works just fine. The downside of this approach is that if the ancestor elements declare the namespaces, Jakarta XML Binding won't be able to take advantage of them.

You can also marshal an object as a subtree of an existing DOM tree. To do this, you pass the `Element` object as the second parameter, and the marshaller will marshal an object as a child of this node.

StAX is also very convenient for doing this sort of things. You can create `XMLStreamWriter`, write some stuff, and then pass that to the marshaller. `JAXB_FRAGMENT` prevents the marshaller from producing `startDocument` and `endDocument` token. When doing this sub-tree marshaling to DOM and StAX, Jakarta XML Binding can take advantage of available in-scope namespace bindings.

Finally, you can marshal an object as a subtree into `ContentHandler`, but it requires a fair amount of SAX programming experience, and it goes beyond the scope of this entry.

5.3.3. Marshalling a non-element

Another common use case is where you have an object that doesn't have `@XmlRootElement` on it. Jakarta XML Binding allows you to marshal it like this:

Example 80. Marshalling a non-element

```
marshaller.marshal( new JAXBElement(
  new QName("","rootTag"),Point.class,new Point(...)));
```

This puts the `<rootTag>` element as the root element, followed by the contents of the object, then `</rootTag>`. You can actually use it with a class that has `@XmlRootElement`, and that simply renames the root element name.

At the first glance the second `Point.class` parameter may look redundant, but it's actually necessary to determine if the marshaller will produce (infamous) `@xsi`:type. In this example, both the class and the instance are `Point`, so you won't see `@xsi`:type. But if they are different, you'll see it.

This can be also used to marshal a simple object, like `String` or an integer.

Marshalling a non-element with `@xsi`:type

```
marshaller.marshal( new JAXBElement(
  new QName("","rootTag"),String.class,"foo bar"));
```

But unfortunately it **cannot** be used to marshal objects like `List` or `Map`, as they aren't handled as the first-class citizen in the Jakarta XML Binding world.

5.3.4. Connecting to other XML APIs

Because of the `Source` and `Result` support, Jakarta XML Binding objects can be easily marshalled into other XML APIs that are not mentioned here. For example, dom4j [http://www.dom4j.org/] has `DocumentResult` that extends `Result`, so you can do:

Example 81. Marshalling to dom4j

```
DocumentResult dr = new DocumentResult();
```

```
marshaller.marshal( object, dr );
o = dr.getDocument();
```

Similar mechanism is available for JDOM and XOM. This conversion is much more efficient than first marshalling to `ByteArrayOutputStream` and then read it back into these DOMs. The same mechanism can be used to marshal to FastInfoset or send the marshaled document to an XSLT engine (`TransformerHandler`.)

The other interesting connector is `JAXBSource`, which wraps a marshaller and allows a Jakarta XML Binding object to be used as a "source" of XML. Many XML APIs take `Source` as an input, and now Jakarta XML Binding object can be passed to them directly.

For example, you can marshal a Jakarta XML Binding object and unmarshal it into another JAXBContext like this:

Example 82. Loading into a different JAXBContext

```
JAXBContext context1 = ... ;
JAXBContext context2 = ... ;

context1.createUnmarshaller().unmarshal( new JAXBSource(context2,object) );
```

This amounts to looking at the same XML by using different schema, and again this is much more efficient than going through `ByteArrayOutputStream`.

5.4. Interaction between marshalling and DOM

Sometimes you may notice that Jakarta XML Binding is producing XML with seemingly unnecessary namespace declarations. In this section, we'll discuss the possible causes and how to resolve this.

5.4.1. Caused by DOM mapping

The #1 cause of extra namespace declarations is due to the DOM mapping. This mainly happens because of a schema construct that forces XJC to generate a property with DOM. This includes the use of wildcard `<xs:any/>` (see more about this Mapping of `<xs:any />`), as well as `xs:anyType` (which can also happen by omission, such as `<xs:element name="foo"/>`, which is interpreted as `<xs:element name="foo" type="xs:anyType" />`.

During unmarshalling, when a subtree of the input XML is converted into XML, Jakarta XML Binding copies all the in-scope namespace bindings active at that time to the root of the DOM element. So for example, given the following Java class and XML, the DOM tree that the `child` field will get will look like the following:

Example 83. Bean with wildcard

```
@XmlRootElement
class Foo {
  @XmlAnyElement
  public Element child;
}
```

Example 84. Instance with subtree matching wildcard

```
<foo xmlns:a="a" xmlns:b="b" xmlns:c="c">
  <subtree xmlns:c="cc">
```

```
    <data>a:xyz</data>
  </subtree>
</foo>
```

Example 85. DOM tree to be stored in Foo.child

```
<subtree xmlns:a="a" xmlns:b="b" xmlns:c="cc">
    <data>a:xyz</data>
  </subtree>
```

Note that the two namespace declarations are copied over, but `c` is not because it's overridden. Also not that Jakarta XML Binding is not touching the whitespace in document. This copying of namespace declarations is necessary to preserve the infoset in the input document. For example, if the `<data>` is a QName, its meaning would change if Jakarta XML Binding unmarshaller doesn't copy it.

Now, imagine what happens when you marshal this back to XML. Despite the fact that in this example neither `b` nor `c` prefixes are in use, Jakarta XML Binding cannot delete them, because it doesn't know if those attributes are significant to the application or not. Therefore, this could end up producing XML with "extra namespace declarations" like:

Example 86. DOM tree to be stored in Foo.child

```
<foo>
  <subtree xmlns:a="a" xmlns:b="b" xmlns:c="cc">
    <data>a:xyz</data>
  </subtree>
</foo>
```

Resolving this problem is not possible in the general case, but sometimes one of the following strategy works:

1. Sometimes schema author incorrectly assumes that `<xs:element name="foo"/>` means `<xs:element name="foo" type="xs:string"/>`, because attribute declarations work somewhat like this. In such a case, adding explicit `type` attribute avoids the use of DOM, so things will work as expected.

2. The wildcard processing mode "`strict`" would force a typed binding, and thereby eliminate any DOM mapping.

3. You might be able to manulally go into the DOM tree and remove unnecessary namespace declarations, if your application knows what are necessary and what are not.

6. Schema Generation

6.1. Invoking schemagen programatically

Schemagen tools by default come in as CLI, ant task, and Maven plugin. These interfaces allow you to invoke schemagen functionality from your program.

6.1.1. At runtime

If the classes you'd like to generate schema from are already available as `java.lang.Class` objects (meaning they are already loaded and resolved in the current JVM), then the easiest way to generate a schema is to use the Jakarta XML Binding API:

Example 87. Generate schema at runtime

```
File baseDir = new File(".");

class MySchemaOutputResolver extends SchemaOutputResolver {
    public Result createOutput( String namespaceUri, String
 suggestedFileName ) throws IOException {
        return new StreamResult(new File(baseDir,suggestedFileName));
    }
}

JAXBContext context = JAXBContext.newInstance(Foo.class, Bar.class, ...);
context.generateSchema(new MySchemaOutputResolver());
```

6.1.2. CLI interface

The CLI interface (`public static int com.sun.tools.jxc.SchemaGenerator.run(String[])`) is the easiest API to access. You can pass in all the schemagen command-line arguments as a string array, and get the exit code as an int value. Messages are sent to `System.err` and `System.out`.

6.1.3. Ant interface

Ant task can be invoked very easily from a non-Ant program. The schemagen ant task is defined in the SchemaGenTask class,

6.1.4. Native Java API

The above two interfaces are built on top of externally committed contracts, so they'll evolve only in a compatibile way. The downside is that the amount of control you can exercise over them would be limited.

So yet another approach to invoke schemagen is to use Eclipse Implementation of JAXB's internal interfaces. But be warned that those interfaces are subject to change in the future versions, despite our best effort to preserve them. This is the API that the Eclipse Implementation of XML Web Services uses to generate schema inside WSDL when they generate WSDL, so does some other web services toolkits that work with the Eclipse Implementation of JAXB.

Most of those interfaces are defined and well-documented in the com.sun.tools.xjc.api package. You can see how the schemagen tools are eventually calling into this API at the implementaion of SchemaGenerator class.

6.2. Generating Schema that you want

This section discusses how you can change the generated XML schema. For changes that also affect the infoset (such as changing elements to attributes, namespaces, etc.), refer to a different section "XML layout and in-memory data layout".

6.2.1. Adding facets to datatypes

As of Eclipse Implementation of JAXB 4.0.0, currently no support for this, although there has been several discussions in the users alias.

The Eclipse Implementation of JAXB project is currently lacking resources to attack this problem, and therefore looking for volunteers to work on this project. The basic idea would be to define enough annota-

tions to cover the basic constraint facets (such as length, enumerations, pattern, etc.) The schema generator would have to be then extended to honor those annotations and generate schemas accordingly.

Some users pointed out relevance of this to Jakarta Bean Validation [https://jakarta.ee/specifications/bean-validation/]. If you are interested in picking up this task, let us know!

7. Deployment

7.1. Using Eclipse Implementation of JAXB with Maven

7.1.1. Maven coordinates for Eclipse Implementation of JAXB artifacts

- **jakarta.xml.bind:jakarta.xml.bind-api:** API classes for Jakarta XML Binding. Required to compile against Jakarta XML Binding.
- **org.glassfish.jaxb:jaxb-core:** Contains sources required by XJC, JXC and Runtime modules.
- **org.glassfish.jaxb:jaxb-runtime:** Contains the main runtime used for serialization and deserialization java objects to/from xml.
- **org.glassfish.jaxb:jaxb-xjc:** Tool to generate Jakarta XML Binding java sources from XML representation.
- **org.glassfish.jaxb:jaxb-jxc:** Tool to generate XML schema from Jakarta XML Binding java sources.

7.1.2. JAXB RI bundles

- **com.sun.xml.bind:jaxb-core:** Contains sources required by XJC, JXC and Runtime modules with dependencies.
- **com.sun.xml.bind:jaxb-impl:** Eclipse Implementation of JAXB runtime jar.
- **com.sun.xml.bind:jaxb-xjc:** Class generation tool jar.
- **com.sun.xml.bind:jaxb-jxc:** Schema generation tool jar.

In contrast to `org.glassfish.jaxb` artifacts, these jars have all dependency classes included inside.

7.1.3. Binary distribution

- **com.sun.xml.bind:jaxb-ri:** Zip distribution containing tooling scripts and all dependency jars in one archive.

7.1.4. Jakarta XML Binding API and Runtime

Minimum requirement to compile is `jakarta.xml.bind-api.jar`. If a client application is running on an environment where Jakarta XML Binding runtime is provided, `jakarta.xml.bind-api.jar` is all that is needed.

Example 88. API only

```
<!-- API -->
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
    <version>4.0.0</version>
</dependency>
```

If client application needs to include the runtime, e.g. running standalone on Java SE `jaxb-impl` should be also included.

Example 89. API + Runtime

```
<!-- API -->
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
    <version>4.0.0</version>
</dependency>

<!-- Runtime -->
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>4.0.0</version>
</dependency>
```

7.2. Using Eclipse Implementation of JAXB on JPMS

Java SE 11 features JSR 376 Java Platform Module System. Starting from version 2.3.2 Eclipse Implementation of JAXB supports JPMS and can be loaded and used from module path. There are only a few things to be aware of.

7.2.1. Eclipse Implementation of JAXB classes openness

Eclipse Implementation of JAXB does reflectively access private members of the class, so client application if loaded from module path needs to "open" packages containing jaxb classes to Jakarta XML Binding. There are alternative Jakarta XML Binding implementations, having different module names, Jakarta XML Binding requires pojo classes to be open only to API module.

Example 90. JPMS module descriptor opening Jakarta XML Binding pojo classes to Jakarta XML Binding API

```
//JPMS module descriptor
module com.example.jaxbclasses {

    //Jakarta XML Binding module name
    requires jakarta.xml.bind;

    //open pojo package to make accessing private members possible for
 Jakarta XML Binding.
    opens com.example.jaxbclasses.pojos to jakarta.xml.bind;
}
```

Jakarta XML Binding API will delegate openness to implementation module after resolving it with service discovery mechanism.

Example 91. Eclipse Implementation of JAXB on JPMS Command line examples

```
#Both client and Eclipse Implementation of JAXB on module path:
$ java -m com.example.jaxbclasses/com.example.jaxb.Main --module-path
 jaxbclient.jar:jakarta.xml.bind-api.jar:jakarta.activation-api.jar:jaxb-
core.jar:jaxb-impl.jar

#Both client and Eclipse Implementation of JAXB on classpath:
$ java com.example.jaxb.Main -cp jaxbclient.jar:jakarta.xml.bind-
api.jar:jakarta.activation-api.jar:jaxb-core.jar:jaxb-impl.jar

#Client on classpath, Eclipse Implementation of JAXB on module path:
$ java com.example.jaxb.Main -cp jaxbclient.jar --module-path
 jakarta.xml.bind-api.jar:jakarta.activation-api.jar:jaxb-core.jar:jaxb-
impl.jar --add-modules jakarta.xml.bind
```

Jakarta XML Binding API will delegate openness to implementation module after resolving it with service discovery mechanism.

8. Other Miscellaneous Topics

8.1. Performance and thread-safety

The JAXBContext class is thread safe, but the Marshaller, Unmarshaller, and Validator classes are not thread safe.

For example, suppose you have a multi-thread server application that processes incoming XML documents by Jakarta XML Binding. In this case, for the best performance you should have just one instance of JAXBContext in your whole application like this:

Example 92. Singleton JAXBContext

```
class MyServlet extends HttpServlet {
    static final JAXBContext context = initContext();

    private static JAXBContext initContext() {
        return JAXBContext.newInstance(Foo.class,Bar.class);
    }
}
```

And each time you need to unmarshal/marshal/validate a document. Just create a new Unmarshaller/Marshaller/Validator from this context, like this:

Example 93. Thread local Unmarshaller

```
    public void doGet( HttpServletRequest req, HttpServletResponse ) {
        Unmarshaller u = context.createUnmarshaller();
        u.unmarshal(...);
    }
```

This is the simplest safe way to use the Eclipse Implementation of JAXB from multi-threaded applications.

If you really care about the performance, and/or your application is going to read a lot of small documents, then creating Unmarshaller could be relatively an expensive operation. In that case, consider pooling Un-

marshaller objects. Different threads may reuse one Unmarshaller instance, as long as you don't use one instance from two threads at the same time.

8.2. Compiling DTD

The Eclipse Implementation of JAXB is shipped with an "experimental" DTD support, which let's you compile XML DTDs. It is marked "experimental" not because the feature is unstable nor unreliable, but rather because it's not a part of the JAXB specification and therefore the level of commitment to compatibility is lower.

Example 94. To compile a DTD, run the XJC binding compiler as follows:

```
$ xjc.sh -dtd test.dtd
```

All the other command-line options of the XJC binding compiler can be applied. Similarly, the XJC ant task supports DTD. The generated code will be no different from what is generated from W3C XML Schema. You'll use the same JAXB API to access the generated code, and it is portable in the sense that it will run on any JAXB 2.0 implementation.

DTD long predates XML namespace, although people since then developed various techniques to use XML namespaces in conjunction with DTD. Because of this, XJC is currently unable to reverse-engineer the use of XML namespace from DTD. If you compile DTDs that use those techniques, you'd either manuallly modify the generated code, or you can try a tool like Trang [http://www.thaiopensource.com/relaxng/trang.html] that can convert DTD into XML Schema in ways that better preserves XML namespaces.

8.2.1. Customizations

The customization syntax for DTD is roughly based on the ver.0.21 working draft of the JAXB specification, which is available at xml.coverpages.org [http://xml.coverpages.org/jaxb0530spec.pdf]. The deviations from this document are:

- The `whitespace` attribute of the `conversion` element takes " `preserve`", " `replace`", and " `collapse`" instead of " `preserve`", " `normalize`", and " `collapse`" as specified in the document.
- The `interface` customization just generates marker interfaces with no method.

8.2.2. Compiling DTD from Maven

Example 95. The following POM snippest describes how to invoke XJC to compile DTD from a Maven project:

```
<plugin>
  <groupId>org.jvnet.jaxb2.maven2</groupId>
  <artifactId>maven-jaxb2-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        <!--  if you want to put DTD somewhere else
        <schemaDirectory>src/main/jaxb</schemaDirectory>
        -->
        <extension>true</extension>
```

```
        <schemaLanguage>DTD</schemaLanguage>
        <schemaIncludes>
          <schemaInclude>*.dtd</schemaInclude>
        </schemaIncludes>
        <bindingIncludes>
          <bindingInclude>*.jaxb</bindingInclude>
        </bindingIncludes>
        <args>
          <arg>-Xinject-listener-code</arg>
        </args>
      </configuration>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.jvnet.jaxb2-commons</groupId>
      <artifactId>property-listener-injector</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</plugin>
```

Example 96. The dependencies section inside the plugin element can be used to specify additional XJC plugins. If you'd like to use more recent version of the Eclipse Implementation of JAXB, you can specify a dependency to XJC here to do so, like this:

```
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-xjc</artifactId>
  <version>4.0.0</version>
</dependency>
```

8.3. Designing a client/server protocol in XML

Occasionally, people try to define a custom protocol that allows multiple XML requests/responses to be sent over a single transport channel. This section discusses the non-trivial interaction between XML and sockets, and how you can design a protocol correctly.

XML1.0 requires a conforming parser to read the entire data till end of the stream (because a parser needs to handle documents like `<root/><!-- post root comment -->`). As a result, a naive attempt to keep one `OutputStream` open and marshal objects multiple times fails.

Example 97. One easy way to work around this limitation is to design your protocol so that the data on the wire will look like the following:

```
<conversation>
  <!-- message 1 -->
  <message>
    ...
  </message>

  <!-- message 2 -->
  <message>
    ...
  </message>
```

```
  ...
</conversation>
```

The `<conversation>` start tag is sent immediately after the socket is opened. This works as a container to send multiple "messages", and this is also an excellent opportunity to do the hand-shaking (e.g., `protocol-version='1.0'` attribute.) Once the `<conversation>` tag is written, multiple messages can be marshalled as a tree into the channel, possibly with a large time lag in between. You can use the Jakarta XML Binding marshaller to produce such message. When the sender wants to disconnect the channel, it can do so by sending the `</conversation>` end tag, followed by the socket disconnection.

Of course, you can choose any tag names freely, and each message can have different tag names.

The receiver would use the StAX API and use `XMLStreamReader` to read this stream. You'd have to use this to process the first `<conversation>` start tag. After that, every time you call a Jakarta XML Binding unmarshaller, you'll get the next message.

For the concrete code, see the `xml-channel` example in the Eclipse Implementation of JAXB distribution.

Tools

Table of Contents

1. XJC

1.1. xjc Overview

Eclipse Implementation of JAXB also provides an Ant task to run the binding complier - see the instructions for XJC Ant Task.

1.2. Launching xjc

The binding compiler can be launched using the appropriate `xjc` shell script in the `bin` directory for your platform.

- **Solaris/Linux**

```
% /path/to/jaxb/bin/xjc.sh -help
```

- **Windows**

```
> c:\path\to\jaxb\bin\xjc.bat -help
```

1.2.1. Execute the `jaxb-xjc.jar` JAR File

If all else fails, you should be able to execute the `jaxb-xjc.jar` file:

- **Solaris/Linux**

```
% java -jar $JAXB_HOME/lib/jaxb-xjc.jar -help
```

- **Windows**

```
> java -jar %JAXB_HOME%\lib\jaxb-xjc.jar -help
```

This is equivalent of running **xjc.sh** or **xjc.bat**, and it allows you to set the JVM parameters.

1.3. xjc Syntax

`xjc` [OPTION]... <schema file/URL/dir/jar> [`-b` <binding>...]

```
Usage: xjc [-options ...] <schema file/URL/dir/jar> ... [-b <bindinfo>] ...
If dir is specified, all schema files in it will be compiled.
If jar is specified, /META-INF/sun-jaxb.episode binding file will be
 compiled.
Options:
  -nv               :  do not perform strict validation of the input
 schema(s)
  -extension        :  allow vendor extensions - do not strictly follow the
                       Compatibility Rules and App E.2 from the JAXB Spec
  -b <file/dir>     :  specify external bindings files (each <file> must
 have its own -b)
                       If a directory is given, **/*.xjb is searched
  -d <dir>          :  generated files will go into this directory
  -p <pkg>          :  specifies the target package
  -httpproxy <proxy> :  set HTTP/HTTPS proxy. Format is
 [user[:password]@]proxyHost:proxyPort
  -httpproxyfile <f> :  Works like -httpproxy but takes the argument in a
 file to protect password
  -classpath <arg>   :  specify where to find user class files
  -catalog <file>    :  specify catalog files to resolve external entity
 references
                       support TR9401, XCatalog, and OASIS XML Catalog
 format.
  -readOnly         :  generated files will be in read-only mode
  -npa              :  suppress generation of package level annotations (**/
package-info.java)
  -no-header        :  suppress generation of a file header with timestamp
  -target (2.0|2.1)  :  behave like XJC 2.0 or 2.1 and generate code that
 doesn't use any 2.2 features.
  -encoding <encoding> :  specify character encoding for generated source
 files
  -enableIntrospection :  enable correct generation of Boolean getters/
setters to enable Bean Introspection apis
  -disableXmlSecurity  :  disables XML security features when parsing XML
 documents
  -contentForWildcard  :  generates content property for types with multiple
 xs:any derived elements
  -xmlschema        :  treat input as W3C XML Schema (default)
  -relaxng          :  treat input as RELAX NG (experimental,unsupported)
```

```
  -relaxng-compact   :  treat input as RELAX NG compact syntax
 (experimental,unsupported)
  -dtd               :  treat input as XML DTD (experimental,unsupported)
  -wsdl              :  treat input as WSDL and compile schemas inside it
 (experimental,unsupported)
  -verbose           :  be extra verbose
  -quiet             :  suppress compiler output
  -help              :  display this help message
  -version           :  display version information
  -fullversion       :  display full version information

Extensions:
  -Xinject-code      :  inject specified Java code fragments into the
 generated code
  -Xlocator          :  enable source location support for generated code
  -Xsync-methods     :  generate accessor methods with the 'synchronized'
 keyword
  -mark-generated    :  mark the generated code as
 @javax.annotation.Generated
  -episode           :  generate the episode file for separate compilation
  -Xpropertyaccessors :  Use XmlAccessType PROPERTY instead of FIELD for
 generated classes
```

1.3.1. Summary of Command Line Options

-nv

By default, the XJC binding compiler performs strict validation of the source schema before processing it. Use this option to disable strict schema validation. This does not mean that the binding compiler will not perform any validation, it simply means that it will perform less-strict validation.

-extension

By default, the XJC binding compiler strictly enforces the rules outlined in the Compatibility chapter of the Jakarta XML Binding Specification. In some cases, you may be allowed to use them in the "-extension" mode enabled by this switch. In the default (strict) mode, you are also limited to using only the binding customizations defined in the specification. By using the "-extension" switch, you will be allowed to use the Overview.

-b <file>

Specify one or more external binding files to process. (Each binding file must have it's own `-b` switch.) The syntax of the external binding files is extremely flexible. You may have a single binding file that contains customizations for multiple schemas or you can break the customizations into multiple bindings files:

```
xjc schema1.xsd schema2.xsd schema3.xsd -b
 bindings123.xjb
xjc schema1.xsd schema2.xsd schema3.xsd -b
 bindings1.xjb -b bindings2.xjb -b bindings3.xjb
```

In addition, the ordering of the schema files and binding files on the command line does not matter.

-d <dir>

By default, the XJC binding compiler will generate the Java content classes in the current directory. Use this option to specify an alternate output directory. The output directory must already exist, the XJC binding compiler will not create it for you.

-encoding <encoding>	Set the encoding name for generated sources, such as EUC-JP or UTF-8. If `-encoding` is not specified, the platform default encoding is used.
-p <pkg>	Specifying a target package via this command-line option overrides any binding customization for package name and the default package name algorithm defined in the specification.
-httpproxy <proxy>	Specify the HTTP/HTTPS proxy. The format is [user[:password]@]proxyHost[:proxyPort]. The old `-host` and `-port` are still supported by the RI for backwards compatibility, but they have been deprecated.
-httpproxyfile <f>	Same as the `-httpproxy <proxy>` option, but it takes the <proxy> parameter in a file, so that you can protect the password (passing a password in the argument list is not safe.)
-classpath <arg>	Specify where to find client application class files used by the `<jxb:javaType>` and `<xjc:superClass>` customizations.
-catalog <file>	Specify catalog files to resolve external entity references. Supports TR9401, XCatalog, and OASIS XML Catalog format. Please read the XML Entity and URI Resolvers [http://xml.apache.org/commons/components/resolver/resolver-article.html] document or the `catalog-resolver` sample application.
-readOnly	By default, the XJC binding compiler does not write-protect the Java source files it generates. Use this option to force the XJC binding compiler to mark the generated Java sources read-only.
-npa	Supress the generation of package level annotations into **/package-info.java. Using this switch causes the generated code to internalize those annotations into the other generated classes.
-no-header	Supress the generation of a file header comment that includes some note and timestamp. Using this makes the generated code more `diff`-friendly.
-target (2.0|2.1)	Avoid generating code that relies on any JAXB 2.1|2.2 features. This will allow the generated code to run with JAXB 2.0 runtime (such as JavaSE 6.)
-xmlschema	treat input schemas as W3C XML Schema (default). If you do not specify this switch, your input schemas will be treated as W3C XML Schema.
-relaxng	Treat input schemas as RELAX NG (experimental, unsupported). Support for RELAX NG schemas is provided as a Overview.
-relaxng-compact	Treat input schemas as RELAX NG compact syntax(experimental, unsupported). Support for RELAX NG schemas is provided as a Overview.
-dtd	Treat input schemas as XML DTD (experimental, unsupported). Support for RELAX NG schemas is provided as a Overview.

-wsdl	Treat input as WSDL and compile schemas inside it (experimental,unsupported).
-quiet	Suppress compiler output, such as progress information and warnings..
-verbose	Be extra verbose, such as printing informational messages or displaying stack traces upon some errors..
-help	Display a brief summary of the compiler switches.
-version	Display the compiler version information.
-fullversion	Display the compiler full version information.
<schema file/URL/dir>	Specify one or more schema files to compile. If you specify a directory, then xjc will scan it for all schema files and compile them.
-Xlocator	This feature causes the generated code to expose SAX Locator information about the source XML in the Java bean instances after unmarshalling.
-Xsync-methods	This feature causes all of the generated method signatures to include the synchronized keyword.
-mark-generated	This feature causes all of the generated code to have `@Generated` [http://docs.oracle.com/javaee/5/api/javax/annotation/Generated.html] annotation.
-episode <FILE>	Generate an episode file from this compilation, so that other schemas that rely on this schema can be compiled later and rely on classes that are generated from this compilation. The generated episode file is really just a Jakarta XML Binding customization file (but with vendor extensions.)
-Xinject-code	
-Xpropertyaccessors>	Annotate the `@XmlAccessorType` of generated classes with XmlAccessType PROPERTY instead of FIELD

1.3.2. Summary of Deprecated and Removed Command Line Options

-host & -port	These options have been deprecated and replaced with the **-http-proxy** option. For backwards compatibility, we will continue to support these options, but they will no longer be documented and may be removed from future releases.
-use-runtime	Since the Jakarta XML Binding specification has defined a portable runtime, it is no longer necessary for the Eclipse Implementation of JAXB to generate **/impl/runtime packages. Therefore, this switch is obsolete and has been removed.

1.4. Compiler Restrictions

In general, it is safest to compile all related schemas as a single unit with the same binding compiler switches.

Please keep the following list of restrictions in mind when running xjc. Most of these issues only apply when compiling multiple schemas with multiple invocations of xjc.

- To compile multiple schemas at the same time, keep the following precedence rules for the target Java package name in mind:

  1. The `-p` command line option takes the highest precedence.
  2. `<jaxb:package>` customization
  3. If `targetNamespace` is declared, apply `targetNamespace` -> Java package name algorithm defined in the specification.
  4. If no `targetNamespace` is declared, use a hardcoded package named "generated".

- It is not legal to have more than one < `jaxb:schemaBindings`> per namespace, so it is impossible to have two schemas in the same target namespace compiled into different Java packages.
- All schemas being compiled into the same Java package must be submitted to the XJC binding compiler at the same time - they cannot be compiled independently and work as expected.
- Element substitution groups spread across multiple schema files must be compiled at the same time.

1.5. Generated Resource Files

XJC produces a set of packages containing Java source files and also `jaxb.properties` files, depending on the binding options you used for compilation. When generated, `jaxb.properties` files must be kept with the compiled source code and made available on the runtime classpath of your client applications:

2. XJC Ant Task

2.1. xjc Task Overview

The `jaxb-xjc.jar` file contains the `XJCTask.class` file, which allows the XJC binding compiler to be invoked from the Ant [http://ant.apache.org/] build tool. To use `XJCTask`, include the following statement in your `build.xml` file:

```
<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
    <classpath>
        <fileset dir="path/to/jaxb/lib" includes="*.jar"/>
    </classpath>
</taskdef>
```

This maps `XJCTask` to an Ant task named `xjc`. For detailed examples of using this task, refer to any of the `build.xml` files used by the Sample Apps.

2.2. xjc Task Attributes

2.2.1. Environment Variables

- ANT_OPTS [http://wiki.apache.org/ant/TheElementsOfAntStyle] - command-line arguments that should be passed to the JVM. For example, you can define system properties or set the maximum Java heap size here.

2.2.2. Parameter Attributes

`xjc` supports the following parameter attributes.

Attribute	Description	Required
`schema`	A schema file to be compiled. A file name (can be relative to the build script base directory), or an URL.	This or nested < `schema`> elements are required.
`binding`	An external binding file that will be applied to the schema file.	No
`package`	If specified, generated code will be placed under this Java package. This option is equivalent to the "-p" command-line switch.	No
`destdir`	Generated code will be written under this directory. If you specify `destdir="abc/def"` and `package="org.acme"`, then files are generated to `abc/def/org/acme`.	Yes
`disableXmlSecurity`	Disable XML security features when parsing XML documents. `false` by default.	No
`encoding`	Set the encoding name for generated sources, such as EUC-JP or UTF-8. If it is not specified, the platform default encoding is used.	No
`readonly`	Generate Java source files in the read-only mode if `true` is specified. `false` by default.	No
`header`	Generate a header in each generated file indicating that this file is generated by such and such version of Eclipse Implementation of JAXB when. `true` by default.	No
`extension`	If set to `true`, the XJC binding compiler will run in the extension mode. Otherwise, it will run in the strict conformance mode. Equivalent of the " `-extension`" command line switch. The default is `false`.	No
`catalog`	Specify the catalog file to resolve external entity references. Support TR9401, XCatalog, and OASIS XML Catalog format. See the catalog-resolver sample for details.	No

Attribute	Description	Required
`removeOldOutput`	Used in pair with nested `<produces>` elements. When this attribute is specified as "`yes`", the files pointed to by the `<produces>` elements will be all deleted before the XJC binding compiler recompiles the source files. See the up-to-date check section for details.	No
`target`	Specifies the runtime environment in which the generated code is supposed to run. Expects 2.0 or 2.1 values. This allows more up-to-date versions of XJC to be used for developing applications that run on earlier JAXB versions.	No, defaults to "2.2"
`language`	Specifies the schema language to compile. Supported values are "WSDL", "XMLSCHEMA", and "WSDL." Case insensitive.	No, defaults to "XMLSCHEMA"

2.2.3. Nested Elements

`xjc` supports the following nested element parameters.

2.2.3.1. `schema`

To compile more than one schema at the same time, use a nested `<schema>` element, which has the same syntax as `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

2.2.3.2. `binding`

To specify more than one external binding file at the same time, use a nested `<binding>` element, which has the same syntax as `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

2.2.3.3. `classpath`

To specify locations of the user-defined classes necessary during the compilation (such as an user-defined type that is used through a `<javaType>` customization), use nested `<classpath>` elements. For the syntax, see "path-like structure" [http://ant.apache.org//manual/using.html#path] .

2.2.3.4. `arg`

Additional command line arguments passed to the XJC. For details about the syntax, see the relevant section [http://ant.apache.org/manual/using.html#arg] in the Ant manual. This nested element can be used to specify various options not natively supported in the **xjc** Ant task. For example, currently there is no native support for the following **xjc** command-line options:

- `-nv`
- `-use-runtime`
- `-schema`

- `-dtd`
- `-relaxng`
- `-Xlocator`
- `-Xsync-methods`

To use any of these features from the `<xjc>` Ant task, you must specify the appropriate nested `<arg>` elements.

2.2.3.5. depends

Files specified with this nested element will be taken into account when the XJC task does the up-to-date check. See the up-to-date check section for details. For the syntax, see `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

2.2.3.6. produces

Files specified with this nested element will be taken into account when the XJC task does the up-to-date check. See the up-to-date check section for details. For the syntax, see `<fileset>` [http://ant.apache.org/manual/Types/fileset.html].

2.2.3.7. xmlcatalog

The xmlcatalog [http://ant.apache.org/manual/Types/xmlcatalog.html] element is used to resolve entities when parsing schema documents.

2.3. Generated Resource Files

Please see the Generated Resource Files for more detail.

2.4. Up-to-date Check

By default, the XJC binding compiler always compiles the inputs. However, with a little additional setting, it can compare timestamps of the input files and output files and skip compilation if the files are up-to-date.

Ideally, the program should be able to find out all the inputs and outputs and compare their timestamps, but this is difficult and time-consuming. So you have to tell the task input files and output files manually by using nested `<depends>` and `<produces>` elements. Basically, the XJC binding compiler compares the timestamps specified by the `<depends>` elements against those of the `<produces>` set. If any one of the "depends" file has a more recent timestamp than some of the files in the "produces" set, it will compile the inputs. Otherwise it will skip the compilation.

This will allow you to say, for example "if any of the `.xsd` files in this directory are newer than the `.java` files in that directory, recompile the schema".

Files specified as the schema files and binding files are automatically added to the "depends" set as well, but if those schemas are including/importing other schemas, you have to use a nested `<depends>` elements. No files are added to the `<produces>` set, so you have to add all of them manually.

A change in a schema or an external binding file often results in a Java file that stops being generated. To avoid such an "orphan" file, it is often desirable to isolate all the generated code into a particular package and delete it before compiling a schema. This can be done by using the `removeOldOutput` attribute. This option allows you to remove all the files that match the "produces" filesets before a compilation. *Be careful when you use this option so that you don't delete important files.*

2.5. Schema Language Support

This release of the Eclipse Implementation of JAXB includes experimental support for RELAX NG, DTD, and WSDL. To compile anything other than W3C XML Schema from the **xjc** Ant task, you must use the nested < `arg`> element to specify the appropriate command line switch, such as `-dtd`, `-relaxng`, or `-wsdl`. Otherwise, your input schemas will be treated as W3C XML Schema and the binding compiler will fail.

2.6. xjc Examples

Compile `myschema.xsd` and place the generated files under src/org/acme/foo:

```
<xjc schema="src/myschema.xsd" destdir="src" package="org.acme.foo"/>
```

Compile all XML Schema files in the `src` directory and place the generated files under the appropriate packages in the `src` directory:

```
<xjc destdir="src">
    <schema dir="src" includes="*.xsd"/>
</xjc>
```

Compile all XML Schema files in the `src` directory together with binding files in the same directory and places the generated files under the appropriate packages in the `src` directory. This example assumes that binding files contain package customizations. This example doesn't search subdirectories of the `src` directory to look for schema files.

```
<xjc destdir="src">
    <schema dir="src" includes="*.xsd"/>
    <binding dir="src" includes="*.xjb"/>
</xjc>
```

Compile `abc.xsd` with an up-to-date check. Compilation only happens when `abc.xsd` is newer than any of the files in the `src/org/acme/foo` directory (and its `impl` subdirectory). Files in these two directories will be wiped away before a compilation, so *don't add your own code in those directories*. Note that the additional `mkdir` task is necessary because Ant's fileset requires the directory specified by the `dir` attribute to exist.

```
<mkdir dir="src/org/acme/foo"/>
<xjc destdir="src" schema="abc.xsd" removeOldOutput="yes"
     package="org.acme.foo">
    <produces dir="src/org/acme/foo" includes="* impl/*"/>
</xjc>
```

More complicated example of up-to-date check. In this example, we assume that you have a large set of schema documents that reference each other, with DTDs that describe the schema documents. An explicit <depends> is necessary so that when you update one of the DTDs, XJC will recompile your schema. But <depends> don't have to re-specify all the schema files, because you've already done that via <schema>.

```
<mkdir dir="src/org/acme/foo"/>
<xjc destdir="src" removeOldOutput="yes"
     package="org.acme.foo">
    <schema dir="schema" includes="*.xsd"/>
    <depends dir="schema" includes="*.dtd"/>
    <produces dir="build/generated-src/org/acme/foo"
              includes="**/*"/>
</xjc>
```

Compile all XML Schema files in the `src` directory and subdirectories, excluding files named `debug.xsd`, and place the generated files under the appropriate packages in the `src` directory. This example also specifies the `-nv` option, which disables the strict schema correctness checking:

```
<xjc destdir="src">
    <schema dir="src" includes="**/*.xsd"
            excludes="**/debug.xsd"/>
    <arg value="-nv"/>
</xjc>
```

If you depend on a proxy server to resolve the location of imported or included schemas (as you might if you're behind a firewall), you need to make the hostname and port number accessible to the JVM hosting `ant`. Do this by setting the environment variable `ANT_OPTS` to a string containing the appropriate `java` options. For example, from DOS:

```
> set ANT_OPTS=-Dhttp.proxyHost=webcache.east
> set ANT_OPTS=%ANT_OPTS% -Dhttp.proxyPort=8080
> ant
```

3. SchemaGen

3.1. schemagen Overview

The current schema generator can process either Java source files or class files.

We also provide an Ant task to run the schema generator - see the instructions for SchemaGen Ant Task.

3.2. Launching schemagen

The schema generator can be launched using the appropriate `schemagen` shell script in the `bin` directory for your platform.

If your java sources/classes reference other classes, they must be accessable on your system CLASSPATH environment variable, or they need to be given to the tool by using the `-classpath`/ `-cp` options. Otherwise you will see errors when generating your schema.

- **Solaris/Linux**

  ```
  % path/to/jaxb/bin/schemagen.sh Foo.java Bar.java ...
  Note: Writing schema1.xsd
  ```

- **Windows**

  ```
  > path\to\jaxb\bin\schemagen.bat Foo.java Bar.java ...
  Note: Writing schema1.xsd
  ```

3.3. schemagen Syntax

`schemagen` [OPTION]... <java files>

```
Usage: schemagen [-options ...] <java files>

Options:
    -d <path>             : specify where to place processor and javac
 generated class files
```

```
    -cp <path>            : specify where to find user specified files
    -classpath <path>     : specify where to find user specified files
    -encoding <encoding>  : specify encoding to be used for annotation
 processing/javac invocation
    -episode <file>       : generate episode file for separate compilation
    -disableXmlSecurity   : disables XML security features when parsing XML
 documents
    -version              : display version information
    -fullversion          : display full version information
    -help                 : display this usage message
```

3.3.1. Summary of Command Line Options

-d <dir> — By default, the schema generator will generate the content in the current directory. Use this option to specify an alternate output directory. The output directory must already exist, the schema generator will not create it for you.

-encoding <encoding> — Set the encoding name for generated sources, such as EUC-JP or UTF-8. If `-encoding` is not specified, the platform default encoding is used.

-classpath <arg> — Specify where to find client application class files.

-episode — Generates the "episode file", which is really just a Jakarta XML Binding customization file (but with vendor extensions). When people develop additional schemas that depend on what this schemagen invocation produces, they can use this episode file to have their generated code refer to your classes.

-help — Display a brief summary of the generator switches.

-version — Display the compiler version information.

-fullversion — Display the compiler full version information.

3.4. Generated Resource Files

The current schema generator simply creates a schema file for each namespace referenced in your Java classes. There is no way to control the name of the generated schema files at this time. Use SchemaGen Ant Task for that purpose.

4. SchemaGen Ant Task

4.1. schemagen Task Overview

The `jaxb-jxc.jar` file contains the `SchemaGenTask.class` file, which allows the schema generator to be invoked from the Ant [http://ant.apache.org] build tool. To use `SchemaGenTask`, include the following statement in your `build.xml` file:

```
<taskdef name="schemagen"
         classname="com.sun.tools.jxc.SchemaGenTask">
    <classpath>
        <fileset dir="path/to/jaxb/lib" includes="*.jar"/>
    </classpath>
```

```
</taskdef>
```

This maps `SchemaGenTask` to an Ant task named `schemagen`. For detailed examples of using this task, refer to the `build.xml` files used by the java to schema Sample Apps.

4.2. schemagen Task Attributes

4.2.1. Environment Variables

- ANT_OPTS [http://wiki.apache.org/ant/TheElementsOfAntStyle] - command-line arguments that should be passed to the JVM. For example, you can define system properties or set the maximum Java heap size here.

4.2.2. Parameter Attributes

`schemagen` supports most of the attributes defined by the javac task [http://ant.apache.org/manual/Tasks/javac.html], plus the following parameter attributes.

Attribute	Description	Required
`destdir`	Base directory to place the generated schema files	No
`classpath`	Works just like the nested <classpath> element	No
`episode`	If specified, generate an episode file in the specified name. For more about the episode file, see **-episode**.	No

4.2.3. Nested Elements

`xjc` supports all the nested elements defined by the javac task [http://ant.apache.org/manual/Tasks/javac.html], the following nested element parameters.

4.2.3.1. `schema`

Control the file name of the generated schema. This element takes a mandatory `namespace` attribute and a mandaotry `file` attribute. When this element is present, the schema document generated for the specified namespace will be placed in the specified file name.

The file name is interpreted as relative to the destdir attribute. In the absence of the destdir attribute, file names are relative to the project base directory. This element can be specified multiple times.

4.2.3.2. `classpath`

A path-like structure [http://ant.apache.org/manual/using.html#path] that represents the classpath. If your Java sources/classes depend on other libraries, they need to be available in the classpath.

4.3. schemagen Examples

Generate schema files from source files in the `src` dir and place them in the `build/schemas` directory.

```
<schemagen srcdir="src" destdir="build/schemas">
```

Compile a portion of the source tree.

```
<schemagen destdir="build/schemas">
    <src path="src"/>
    <exclude name="Main.java"/>
</schemagen>
```

Set schema file names.

```
<schemagen srcdir="src" destdir="build/schemas">
    <schema namespace="http://myschema.acme.org/common"
            file="myschema-common.xsd"/>
    <schema namespace="http://myschema.acme.org/onion"
            file="myschema-onion.xsd"/>
</schemagen>
```

5. 3rd Party Tools

5.1. Maven Plugins

To generate Jakarta XML Binding classes from an XML schema in Maven project, there are multiple community plugins available:

- The most advanced and feature-full Maven plugin for XML Schema compilation: highsource maven-jaxb2-plugin [https://github.com/highsource/maven-jaxb2-plugin]

Example 98. Using highsource maven-jaxb2-plugin

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jvnet.jaxb2.maven2</groupId>
            <artifactId>maven-jaxb2-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the maven-jaxb2-plugin user guide [https://github.com/highsource/maven-jaxb2-plugin/wiki/User-Guide] for configuration details.

- A Maven plugin originating from MojoHaus which has been updated for Jakarta XML Binding 3+ by Evolved Binary: MojoHaus jaxb-maven-plugin [https://github.com/evolvedbinary/mojohaus-jaxb-maven-plugin]

Example 99. Using MojoHaus jaxb-maven-plugin

```
<build>
```

```
    <plugins>
        <plugin>
            <groupId>com.evolvedbinary.maven.mojohaus</groupId>
            <artifactId>jaxb-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>schemagen</id>
                    <goals>
                        <goal>schemagen</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the MojoHaus jaxb-maven-plugin documentation [https://evolvedbinary.github.io/mojohaus-jaxb-maven-plugin/] for configuration details.

- A Maven plugin originating from java.net which has been updated for Jakarta XML Binding 3+ by Evolved Binary: jvnet jaxb-maven-plugin [https://github.com/evolvedbinary/jvnet-jaxb-maven-plugin]

Example 100. Using jvnet jaxb-maven-plugin

```
<build>
    <plugins>
        <plugin>
            <groupId>com.evolvedbinary.maven.jvnet</groupId>
            <artifactId>jaxb-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>generate</id>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

See the jvnet jaxb-maven-plugin documentation [https://github.com/evolvedbinary/jvnet-jaxb-maven-plugin] for configuration details.

Alternatively to community plugins, there are tooling artifacts jaxb-xjc and jaxb-jxc, which can be used for java from XML schema generation and vice versa.

Example 101. Using Eclipse Implementation of JAXB tooling artifacts

```
<!-- Tooling dependencies -->
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-xjc</artifactId>
    <version>4.0.0</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
```

```
    <artifactId>jaxb-jxc</artifactId>
    <version>4.0.0</version>
</dependency>

<!-- Invoke tooling API (Java 11) -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
        <!-- Generate java sources from schema -->
        <execution>
            <id>xjc</id>
            <goals>
                <goal>exec</goal>
            </goals>
            <configuration>
                <executable>java</executable>
                <arguments>
                    <argument>--module-path</argument>
                    <modulepath/>
                    <argument>-m</argument>
                    <argument>com.sun.tools.xjc</argument>
                    <argument>-p</argument>
                    <argument>com.example</argument>
                    <argument>-d</argument>
                    <argument>${project.build.directory}/generated-sources</
argument>
                    <argument>${project.build.directory}/classes/schema.xsd</
argument>
                </arguments>
            </configuration>
        </execution>

        <!-- Generate XML Schema from sources -->
        <execution>
            <id>jxc</id>
            <goals>
                <goal>exec</goal>
            </goals>
            <configuration>
                <executable>java</executable>
                <arguments>
                    <argument>--module-path</argument>
                    <modulepath/>
                    <argument>-m</argument>
                    <argument>com.sun.tools.jxc</argument>
                    <argument>-d</argument>
                    <argument>${project.build.directory}/generated-sources</
argument>
                    <argument>${project.build.directory}/classes/com/example/
Author.java</argument>
                    <argument>${project.build.directory}/classes/com/example/
Book.java</argument>
                </arguments>
                <longModulepath>false</longModulepath>
            </configuration>
        </execution>
    </executions>
</plugin>
```

See also xml schema compiler usage.

`schemagen` and `xjc` command line scripts are available only in the zip distribution.

5.2. XJC Plugins

Various people in the community have developed plugins for XJC that you can use today. These plugins allow you to enhance/alter the code generation of XJC in many different ways. See for example JAXB2 Basics Pligins [https://github.com/highsource/jaxb2-basics/wiki].

5.3. RDBMS Persistence

Lexi has developed the HyperJAXB3 project [https://github.com/highsource/hyperjaxb3/wiki] for RDBMS persistence support for the Eclipse Implementation of JAXB.

Eclipse Implementation of JAXB Extensions

Table of Contents

1. Overview

This page contains information about vendor-specific features provided by the Eclipse Implementation of JAXB.

Runtime Properties	This document describes Eclipse Implementation of JAXB specific properties that affect the way that the Jakarta XML Binding runtime library behaves.
XJC Customizations	This document describes additional binding customizations that can be used to control the generated source code.
DTD	This document describes the Eclipse Implementation of JAXB's experimental support for W3C XML Schema features not currently described in the Jakarta XML Binding Specification as well as support for other schema languages (RELAX NG and DTD).

2. Runtime Properties

2.1. Marshaller Properties

The Eclipse Implementation of JAXB provides additional Marshaller properties that are not defined by the Jakarta XML Binding specification. These properties allow you to better control the marshalling process, but they only work with the Eclipse Implementation of JAXB; they may not work with other Jakarta XML Binding providers.

2.1.1. Index of Marshaller Properties

- Namespace Prefix Mapping
- Indentation
- Character Escaping Control

- XML Declaration Control
- Jaxb Annotation Control

2.1.2. Namespace Prefix Mapping

Property name:	`org.glassfish.jaxb.namespacePrefixMapper`
Type:	`org.glassfish.jaxb.runtime.marshaller.Namespace`
Default value:	null

The Eclipse Implementation of JAXB provides a mechanism for users to control declarations of namespace URIs and what prefixes they will be bound to. This is the general procedure:

1. The application developer provides an implementation of `org.glassfish.jaxb.runtime.marshaller.NamespacePrefixMapper`.
2. This class is then set on the marshaller via the RI specific property `org.glassfish.jaxb.namespacePrefixMapper`.
3. Each time the marshaller sees a URI, it performs a callback on the mapper: "What prefix do you want for this namespace URI?"
4. If the mapper returns something, the marshaller will try to use it.

The `org.glassfish.jaxb.runtime.marshaller.NamespacePrefixMapper` class has the following method that you need to implement:

```
/**
 * Implemented by the user application to determine URI -> prefix
 * mapping.
 *
 * This is considered as an interface, though it's implemented
 * as an abstract class to make it easy to add new methods in
 * a future.
 *
 * @author
 *     Kohsuke Kawaguchi (kohsuke.kawaguchi@sun.com)
 */
public abstract class NamespacePrefixMapper {

    private static final String[] EMPTY_STRING = new String[0];

    /**
     * Returns a preferred prefix for the given namespace URI.
     *
     * This method is intended to be overrided by a derived class.
     *
     * <p>
     * As noted in the return value portion of the javadoc, there
     * are several cases where the preference cannot be honored.
     * Specifically, as of JAXB RI 2.0 and onward:
     *
     * <ol>
     * <li>
     * If the prefix returned is already in use as one of the in-scope
     * namespace bindings. This is partly necessary for correctness
     * (so that we don't unexpectedly change the meaning of QNames
```

```
     * bound to {@link String}), partly to simplify the marshaller.
     * <li>
     * If the prefix returned is "" yet the current {@link JAXBContext}
     * includes classes that use the empty namespace URI. This allows
     * the JAXB RI to reserve the "" prefix for the empty namespace URI,
     * which is the only possible prefix for the URI.
     * This restriction is also to simplify the marshaller.
     * </ol>
     *
     * @param namespaceUri
     *      The namespace URI for which the prefix needs to be found.
     *      Never be null. "" is used to denote the default namespace.
     * @param suggestion
     *      When the content tree has a suggestion for the prefix
     *      to the given namespaceUri, that suggestion is passed as a
     *      parameter. Typicall this value comes from the QName.getPrefix
     *      to show the preference of the content tree. This parameter
     *      may be null, and this parameter may represent an already
     *      occupied prefix.
     * @param requirePrefix
     *      If this method is expected to return non-empty prefix.
     *      When this flag is true, it means that the given namespace URI
     *      cannot be set as the default namespace.
     *
     * @return
     *      null if there's no prefered prefix for the namespace URI.
     *      In this case, the system will generate a prefix for you.
     *
     *      Otherwise the system will try to use the returned prefix,
     *      but generally there's no guarantee if the prefix will be
     *      actually used or not.
     *
     *      return "" to map this namespace URI to the default namespace.
     *      Again, there's no guarantee that this preference will be
     *      honored.
     *
     *      If this method returns "" when requirePrefix=true, the return
     *      value will be ignored and the system will generate one.
     *
     * @since JAXB 1.0.1
     */
    public abstract String getPreferredPrefix(String namespaceUri, String
suggestion, boolean requirePrefix);

    /**
     * Returns a list of namespace URIs that should be declared
     * at the root element.
     *
     * <p>
     * By default, the JAXB RI 1.0.x produces namespace declarations only
when
     * they are necessary, only at where they are used. Because of this
     * lack of look-ahead, sometimes the marshaller produces a lot of
     * namespace declarations that look redundant to human eyes. For example,
     * <pre><xmp>
     * <?xml version="1.0"?>
     * <root>
     *   <ns1:child xmlns:ns1="urn:foo"> ... </ns1:child>
     *   <ns2:child xmlns:ns2="urn:foo"> ... </ns2:child>
     *   <ns3:child xmlns:ns3="urn:foo"> ... </ns3:child>
```

```
     *   ...
     * </root>
     * </xmp></pre>
     *
     * <p>
     * The JAXB RI 2.x mostly doesn't exhibit this behavior any more,
     * as it declares all statically known namespace URIs (those URIs
     * that are used as element/attribute names in JAXB annotations),
     * but it may still declare additional namespaces in the middle of
     * a document, for example when (i) a QName as an attribute/element value
     * requires a new namespace URI, or (ii) DOM nodes as a portion of an
object
     * tree requires a new namespace URI.
     *
     * <p>
     * If you know in advance that you are going to use a certain set of
     * namespace URIs, you can override this method and have the marshaller
     * declare those namespace URIs at the root element.
     *
     * <p>
     * For example, by returning <code>new String[]{"urn:foo"}</code>,
     * the marshaller will produce:
     * <pre><xmp>
     * <?xml version="1.0"?>
     * <root xmlns:ns1="urn:foo">
     *   <ns1:child> ... </ns1:child>
     *   <ns1:child> ... </ns1:child>
     *   <ns1:child> ... </ns1:child>
     *   ...
     * </root>
     * </xmp></pre>
     * <p>
     * To control prefixes assigned to those namespace URIs, use the
     * {@link #getPreferredPrefix(String, String, boolean)} method.
     *
     * @return
     *      A list of namespace URIs as an array of {@link String}s.
     *      This method can return a length-zero array but not null.
     *      None of the array component can be null. To represent
     *      the empty namespace, use the empty string <code>""</code>.
     *
     * @since
     *      JAXB RI 1.0.2
     */
    public String[] getPreDeclaredNamespaceUris() {
        return EMPTY_STRING;
    }

    /**
     * Similar to {@link #getPreDeclaredNamespaceUris()} but allows the
     * (prefix,nsUri) pairs to be returned.
     *
     * <p>
     * With {@link #getPreDeclaredNamespaceUris()}, applications who wish to
control
     * the prefixes as well as the namespaces needed to implement both
     * {@link #getPreDeclaredNamespaceUris()} and {@link
#getPreferredPrefix(String, String, boolean)}.
     *
     * <p>
```

```
    * This version eliminates the needs by returning an array of pairs.
    *
    * @return
    *      always return a non-null (but possibly empty) array. The array
stores
    *      data like (prefix1,nsUri1,prefix2,nsUri2,...) Use an empty string
to represent
    *      the empty namespace URI and the default prefix. Null is not
allowed as a value
    *      in the array.
    *
    * @since
    *      JAXB RI 2.0 beta
    */
   public String[] getPreDeclaredNamespaceUris2() {
       return EMPTY_STRING;
   }

   /**
    * Returns a list of (prefix,namespace URI) pairs that represents
    * namespace bindings available on ancestor elements (that need not be
repeated
    * by the JAXB RI.)
    *
    * <p>
    * Sometimes JAXB is used to marshal an XML document, which will be
    * used as a subtree of a bigger document. When this happens, it's nice
    * for a JAXB marshaller to be able to use in-scope namespace bindings
    * of the larger document and avoid declaring redundant namespace URIs.
    *
    * <p>
    * This is automatically done when you are marshalling to {@link
XMLStreamWriter},
    * {@link XMLEventWriter}, {@link DOMResult}, or {@link Node}, because
    * those output format allows us to inspect what's currently available
    * as in-scope namespace binding. However, with other output format,
    * such as {@link OutputStream}, the JAXB RI cannot do this
automatically.
    * That's when this method comes into play.
    *
    * <p>
    * Namespace bindings returned by this method will be used by the JAXB
RI,
    * but will not be re-declared. They are assumed to be available when you
insert
    * this subtree into a bigger document.
    *
    * <p>
    * It is <b>NOT</b> OK to return  the same binding, or give
    * the receiver a conflicting binding information.
    * It's a responsibility of the caller to make sure that this doesn't
happen
    * even if the ancestor elements look like:
    * <pre><xmp>
    *   <foo:abc xmlns:foo="abc">
    *     <foo:abc xmlns:foo="def">
    *       <foo:abc xmlns:foo="abc">
    *         ... JAXB marshalling into here.
    *       </foo:abc>
    *     </foo:abc>
```

```
     *   </foo:abc>
     * </xmp></pre>
     *
     * @return
     *      always return a non-null (but possibly empty) array. The array
 stores
     *      data like (prefix1,nsUri1,prefix2,nsUri2,...) Use an empty string
 to represent
     *      the empty namespace URI and the default prefix. Null is not
 allowed as a value
     *      in the array.
     *
     * @since JAXB RI 2.0 beta
     */
    public String[] getContextualNamespaceDecls() {
        return EMPTY_STRING;
    }
}
```

See the Sample Apps sample application for a detailed example.

2.1.3. Indentation

Property name:	`org.glassfish.jaxb.indentString`
Type:	`java.lang.String`
Default value:	" " (four whitespaces)

This property controls the string used for the indentation of XML. An element of depth *k* will be indented by printing this string *k* times. Note that the "`jaxb.formatted.output`" property needs to be set to "true" for the formatting/indentation of the output to occur. See the API documentation for `jakarta.xml.bind.Marshaller` [api/jakarta/xml/bind/Marshaller.html] interface for details of this property.

2.1.4. Character Escaping Control

Property name:	`org.glassfish.jaxb.characterEscapeHandler`
Type:	`org.glassfish.jaxb.core.marshaller.CharacterEsc`
Default value:	null

By default, the marshaller implementation of the Eclipse Implementation of JAXB tries to escape characters so they can be safely represented in the output encoding (by using Unicode numeric character references of the form &#dddd;)

Unfortunately, due to various technical reasons, the default behavior may not meet your expectations. If you need to handle escaping more adroitly than the default manner, you can do so by doing the following:

1. Write a class that implements the `org.glassfish.jaxb.core.marshaller.CharacterEscapeHandler` interface.

2. Create a new instance of it.

3. Set that instance to the Marshaller by using this property.

See the Sample Apps sample application for more details.

2.1.5. XML Declaration Control

Property name:	`org.glassfish.jaxb.xmlDeclaration`
Type:	`boolean`
Default value:	true

This experimental JAXB RI 1.0.x property has been adopted as a standard in Eclipse Implementation of JAXB. The Eclipse Implementation of JAXB will continue to support this property, but client code should be using the Marshaller.JAXB_FRAGMENT [https://jakarta.ee/specifications/xml-binding/4.0/apidocs/jakarta.xml.bind/jakarta/xml/bind/Marshaller.html#JAXB_FRAGMENT] property instead. Please refer to the Marshaller javadoc [https://jakarta.ee/specifications/xml-binding/4.0/apidocs/jakarta.xml.bind/jakarta/xml/bind/Marshaller.html#supportedProps] for a complete description of the behavior.

In Eclipse Implementation of JAXB, calling:

```
marshaller.setProperty("org.glassfish.jaxb.xmlDeclaration", true);
```

is equivalent to calling:

```
marshaller.setProperty(Marshaller.JAXB_FRAGMENT, true);
```

Enabling fragment marshalling could be useful if you are inserting the output of the XML into another XML.

2.1.6. XML Preamble Control

Property name:	`org.glassfish.jaxb.xmlHeaders`
Type:	`java.lang.String`
Default value:	null

This property allows you to specify an XML preamble (<?xml ...> declaration) and any additional PIs, comments, DOCTYPE declaration that follows it. This property takes effect only when you are marshalling to `OutputStream`, `Writer`, or `StreamResult`. Note that this property interacts with the `Marshaller.JAXB_FRAGMENT` property. If that property is untouched or set to false, then Eclipse Implementation of JAXB would always write its XML preamble, so this property can be only used to write PIs, comments, DOCTYPE, etc. On the other hand, if it is set to true, then Jakarta XML Binding will not write its own XML preamble, so this property may contain custom XML preamble.

2.1.7. Jaxb Annotation Control

Property name:	`XmlAccessorFactory`
Type:	`boolean`
Default value:	false

This property provides support for a custom org.glassfish.jaxb.runtime.v2.runtime.reflect.Accessor implementation. It allows the user to control the access to class fields and properties.

In Eclipse Implementation of JAXB, set the property to enable:

```
marshaller.setProperty("XmlAccessorFactory", true);
```

3. XJC Customizations

3.1. Customizations

The Eclipse Implementation of JAXB provides additional customizations that are not defined by the Jakarta XML Binding specification. Note the following:

- These features may only be used when the Eclipse Implementation of JAXB XJC binding compiler is run in the `-extension` mode.
- All of the Eclipse Implementation of JAXB vendor extensions are defined in the "`http://java.sun.com/xml/ns/jaxb/xjc`" namespace.
- The namespaces containing extension binding declarations are specified to a Eclipse Implementation of JAXB processor by the occurrence of the global attribute `@jaxb:extensionBindingPrefixes` within an instance of `<xs:schema>` element. The value of this attribute is a whitespace-separated list of namespace prefixes. For more information, please refer to section 6.1.1 of the Jakarta XML Binding Specification.

3.1.1. Index of Customizations

- SCD Support
- Extending a Common Super Class - Extending a Common Super Class
- Extending a Common Super Interface - Extending a Common Super Interface
- Enhanced <jaxb:javaType> - Enhanced <jaxb:javaType> customization
- Experimental simpler & better binding mode - Experimental simpler & better binding mode
- Alternative Derivation-by-restriction Binding Mode - Alternative derivation-by-restriction binding mode
- Allow separate compilations to perform element substitutions - Allow separate compilations to perform element substitutions

3.1.2. SCD Support

The Eclipse Implementation of JAXB supports the use of schema component designator [http://www.w3.org/TR/2005/WD-xmlschema-ref-20050329/] as a means of specifying the customization target (of all standard Jakarta XML Binding customizations as well as vendor extensions explained below.) To use this feature, use the `scd` attribute on <bindings> element instead of the `schemaLocation` and `node` attributes.

```
<bindings xmlns:tns="http://example.com/myns"
          xmlns="https://jakarta.ee/xml/ns/jaxb" version="3.0">
    <bindings
            ...
            scd="tns:foo">
        <!-- this customization applies to the global element declaration -->
        <!-- 'foo' in the http://example.com/myns namespace -->
        <class name="FooElement"/>
    </bindings>
    <bindings
```

```
            ...
            scd="~tns:bar">
        <!-- this customization applies to the global type declaration -->
        <!-- 'bar' in the http://example.com/myns namespace -->
        <class name="BarType"/>
    </bindings>
</bindings>
```

Compared to the standard XPath based approach, SCD allows more robust and concise way of identifying a target of a customization. For more about SCD, refer to the scd example. Note that SCD is a W3C working draft, and may change in the future.

3.1.3. Extending a Common Super Class

The `<xjc:superClass>` customization allows you to specify the fully qualified name of the Java class that is to be used as the super class of all the generated implementation classes. The `<xjc:superClass>` customization can only occur within your `<jaxb:globalBindings>` customization on the `<xs:schema>` element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="3.0">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings>
                <xjc:superClass
                        name="org.acme.RocketBooster"/>
            </jaxb:globalBindings>
        </xs:appinfo>
    </xs:annotation>

    ...

</xs:schema>
```

In the sample above, the `<xjc:superClass>` customization will cause all of the generated implementation classes to extend the named class, `org.acme.RocketBooster`.

3.1.4. Extending a Common Super Interface

The `<xjc:superInterface>` customization allows you to specify the fully qualified name of the Java interface that is to be used as the root interface of all the generated interfaces. The `<xjc:superInterface>` customization can only occur within your `<jaxb:globalBindings>` customization on the `<xs:schema>` element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="3.0">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings>
                <xjc:superInterface
```

```
                        name="org.acme.RocketBooster"/>
                </jaxb:globalBindings>
            </xs:appinfo>
        </xs:annotation>

        ...

</xs:schema>
```

In the sample above, the `<xjc:superInterface>` customization will cause all of the generated interfaces to extend the named interface, `org.acme.RocketBooster`.

3.1.5. Enhanced <jaxb:javaType>

The <xjc:javaType> customization can be used just like the standard <jaxb:javaType> customization, except that it allows you to specify an `XmlAdapter`-derived class, instead of parse&print method pair.

This customization can be used in all the places <jaxb:javaType> is used, but nowhere else:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="3.0">

    ...

    <xsd:simpleType name="LayerRate_T">
        <xsd:annotation>
            <xsd:appinfo>
                <xjc:javaType name="org.acme.foo.LayerRate"
                              adapter="org.acme.foo.LayerRateAdapter"/>
            </xsd:appinfo>
        </xsd:annotation>

        ... gory simple type definition here ...

    </xsd:simpleType>
</xsd:schema>
```

In the above example, `LayerRate_T` simple type is adapted by `org.acme.foo.LayerRateAdapter`, which extends from `XmlAdapter`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="3.0">

    <xsd:annotation>
        <xsd:appinfo>
            <jaxb:globalBindings>
                <xjc:javaType name="org.acme.foo.MyDateType"
                              xmlType="xsd:dateTime"
                              adapter="org.acme.foo.MyAdapterImpl"/>
            </jaxb:globalBindings>
        </xsd:appinfo>
    </xsd:annotation>
```

```
    ...

</xsd:schema>
```

In the above example, all the use of `xsd:dateTime` type is adapter by `org.acme.foo.MyAdapterImpl` to `org.acme.foo.MyDateType`

3.1.6. Experimental simpler & better binding mode

This experimental binding mode can be enabled as a part of the global binding. See below:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="3.0">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings generateValueClass="false">
                <xjc:simple/>
            </jaxb:globalBindings>
        </xs:appinfo>
    </xs:annotation>

    ...

</xs:schema>
```

When enabled, XJC produces Java source code that are more concise and easier to use. Improvements include:

1. Some content model definitions, such as `A,B,A`, which used to cause an XJC compilation error and required manual intervention, now compile out of the box without any customization.

2. Some content model definitions that used to bind to a non-intuitive Java class now binds to a much better Java class:

```
<!-- schema -->
<xs:complexType name="foo">
    <xs:choice>
        <xs:sequence>
            <xs:element name="a" type="xs:int"/>
            <xs:element name="b" type="xs:int"/>
        </xs:sequence>
        <xs:sequence>
            <xs:element name="b" type="xs:int"/>
            <xs:element name="c" type="xs:int"/>
        </xs:sequence>
    </xs:choice>
</xs:complexType>
```

```
// before
class Foo {
    List<JAXBElement<Integer>> content;
}

// in <xjc:simple> binding
class Foo {
```

```
    Integer a;
    int b; // notice that b is effectively mandatory, hence primitive
    Integer c;
}
```

3. When repetable elements are bound, the method name will become plural.

```
<!-- schema -->
<xs:complexType name="person">
    <xs:sequence>
        <xs:element name="child" type="xs:string"
                    maxOccurs="unbounded"/>
        <xs:element name="parent" type="xs:string"
                    maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

```
// before
public class Person {
    protected List<String> child;
    protected List<String> parent;
}

// in <xjc:simple> binding
public class Person {
    protected List<String> children;
    protected List<String> parents;
}
```

Once again, readers are warned that this is an **experimental binding mode**, and therefore the binding is subject to change in future versions of the Eclipse Implementation of JAXB without notice. Please send feedbacks on this binding to jaxb-impl-dev@eclipse.org [https://accounts.eclipse.org/mailing-list/jaxb-impl-dev]

3.1.7. Alternative Derivation-by-restriction Binding Mode

Normally, the Jakarta XML Binding specification requires that a derivation-by-restriction be mapped to an inheritance betwee n two Java classes. This is necessary to preserve the type hierarchy, but one of the downsides is that the derived class does not really provide easy-to-use properties that reflect the restricted content model.

This experimental <xjc:treatRestrictionLikeNewType> changes this behavior by not preserving the type inheritance to Java. Instead, it generates two unrelated Java classes, both with proper properties. For example, given the following schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           xmlns:jaxb="https://jakarta.ee/xml/ns/jaxb"
           jaxb:version="3.0"
           elementFormDefault="qualified">

    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings>
                <xjc:treatRestrictionLikeNewType/>
            </jaxb:globalBindings>
        </xs:appinfo>
    </xs:annotation>
```

```
    <xs:complexType name="DerivedType">
        <xs:complexContent>
            <xs:restriction base="ResponseOptionType">
                <xs:sequence>
                    <xs:element name="foo" type="xs:string"/>
                </xs:sequence>
            </xs:restriction>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="ResponseOptionType">
        <xs:sequence>
            <xs:element name="foo" type="xs:string"
                        maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

The generated `Derived` class will look like this (comment and annotations removed for brevity):

```
public class DerivedType {
    protected String foo;

    public String getFoo() { return foo; }
    public void setFoo(String value) { this.foo = value; }
}
```

In contrast, without this customization the `Derived` class would look like the following:

```
public class DerivedType extends ResponseOptionType {

    // it simply inherits List<String> ResponseOptionType.getFoo()

}
```

3.1.8. Allow separate compilations to perform element substitutions

In an attempt to make the generated code easier to use, the Jakarta XML Binding specification sometimes choose bindings based on how certain feature is used. One of them is element substitution feature. If no actual element substitution happens in the schema, Jakarta XML Binding assumes that the element is not used for substitution, and generates code that assumes it.

Most of the time this is fine, but when you expect other "extension" schemas to be compiled later on top of your base schema, and if those extension schemas do element substitutions, this binding causes a problem (see example [https://github.com/eclipse-ee4j/jaxb-ri/issues/289].)

<xjc:substitutable> customization is a work around for this issue. It explicitly tells XJC that a certain element is used for element substitution head, even though no actual substitution might be present in the current compilation. This customization should be attached in the element declaration itself, like this:

```
<xs:element name="Model" type="Model">
    <xs:annotation>
        <xs:appinfo>
            <xjc:substitutable/>
        </xs:appinfo>
    </xs:annotation>
</xs:element>
```

4. DTD

4.1. DTD

The Eclipse Implementation of JAXB is shipped with experimental DTD support, which lets you compile XML DTDs.

To compile a DTD `test.dtd`, run the XJC binding compiler as follows:

```
$ xjc.sh -dtd test.dtd
```

All the other command-line options of the XJC binding compiler can be applied. Similarly, the **xjc** ant [http://ant.apache.org/] task supports DTD. The generated code will be no different from what is generated from W3C XML Schema. You'll use the same Jakarta XML Binding API to access the generated code, and it is portable in the sense that it will run on any Jakarta XML Binding implementation.

4.1.1. Customization

The customization syntax for DTD is roughly based on the ver.0.21 working draft of the Jakarta XML Binding specification, which is available at xml.coverpages.org [http://xml.coverpages.org/jaxb0530spec.pdf]. The deviations from this document are:

- The `whitespace` attribute of the `conversion` element takes " `preserve`", " `replace`", and " `collapse`" instead of " `preserve`"," `normalize`", and " `collapse`" as specified in the document.
- The `interface` customization just generates marker interfaces with no method.

5. Develop Plugins

This document describes how to write an XJC plugin to extend the code generation of XJC.

5.1. What Can A Plugin Do?

An XJC plugin participates in the code generation from a schema. It can define its own customizations that users can use to control it, it can access the code that the Eclipse Implementation of JAXB generates, it can generate additional classes/methods/fields/annotations/comments, and it can also replace some of the pluggability points in the compilation process, such as XML name -> Java name conversion.

As a show case of what a plugin can do, take a look at plugins hosted at JAXB2-commons.

5.1.1. Quick Start

To write a plugin, do the following simple steps.

1. Write a class, say, `org.acme.MyPlugin` by extending `com.sun.tools.xjc.Plugin`. See javadoc for how to implement methods.
2. Write the name of your plugin class in a text file and put it as `/META-INF/services/com.sun.tools.xjc.Plugin` in your jar file.

Users can then use your plugins by declaring an XJC ant task with your jar files.

```
<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
    <classpath>
        <fileset dir="jaxb-ri/lib" includes="*.jar"/>
        <fileset dir="your-plugin" includes="*.jar"/>
    </classpath>
</taskdef>
```

5.1.2. Resources

Although we will do our best to maintain the compatibility of the interfaces, it is still subject to change at this point.

Frequently Asked Questions

1. JAXB 2.0

Q: Which version of Java SE does Eclipse Implementation of JAXB 4.0.0 require?

A: Java SE 11 or higher.

Q: Can I run my existing JAXB 1.x/2.x applications on the Eclipse Implementation of JAXB runtime?

A: This is not supported.

Q: What if I want to port my JAXB 1.x/2.x application to Jakarta XML Binding runtime?

A: You need to replace references to `javax.xml.bind` package by `jakarta.xml.bind` package, recompile your schema with the newer xjc and modify your application code to work with the new bindings.

Q: Where are `schemagen` and `xjc` command line scripts available?

A: They are included only in the zip distribution.

Q: Are the Jakarta XML Binding runtime API's thread safe?

A: The Jakarta XML Binding Specification currently does not address the thread safety of any of the runtime classes. In the case of the Eclipse Implementation of JAXB, the `JAXBContext` class **is** thread safe, but the `Marshaller`, `Unmarshaller`, and `Validator` classes **are not** thread safe.

For example, suppose you have a multi-thread server application that processes incoming XML documents by Jakarta XML Binding. In this case, for the best performance you should have just one instance of `JAXBContext` in your whole application like this:

```
class MyServlet extends HttpServlet {
    static final JAXBContext context = initContext();

    private static JAXBContext initContext() {
        return JAXBContext.newInstance("....",
 MyServlet.class.getClassLoader());
    }
}
```

And each time you need to unmarshal/marshal/validate a document. Just create a new `Unmarshaller`/`Marshaller`/`Validator` from this context, like this:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) {
    Unmarshaller u = context.createUnmarshaller();
    u.unmarshal(...);
}
```

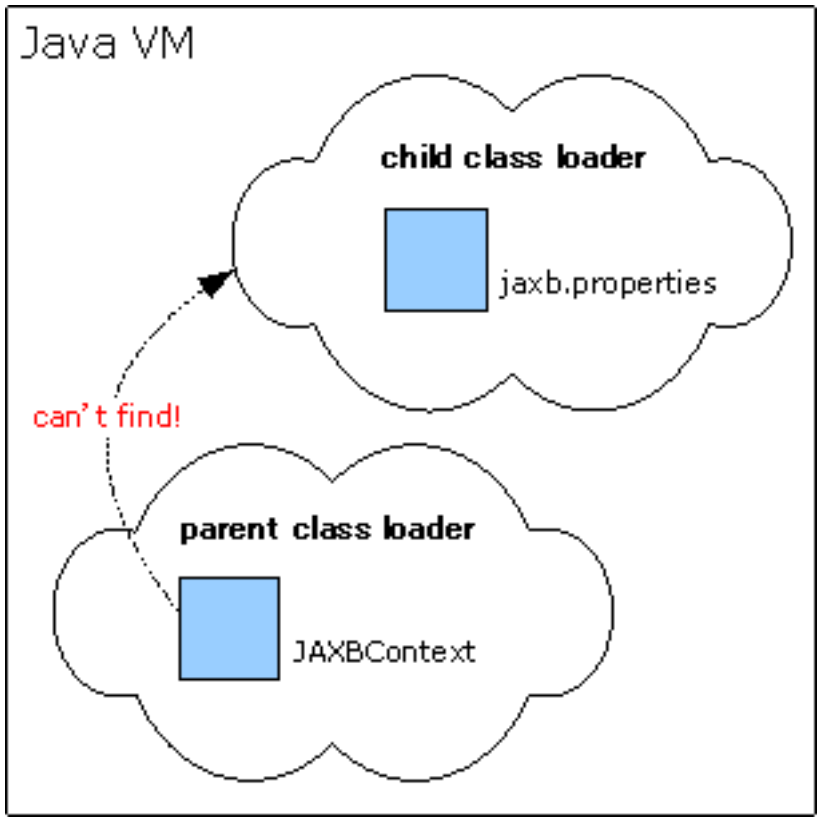
This is the simplest safe way to use the Eclipse Implementation of JAXB from multi-threaded applications.

If you really care about the performance, and/or your application is going to read a lot of small documents, then creating `Unmarshaller` could be relatively an expensive operation. In that case, consider pooling `Unmarshaller` objects. Different threads may reuse one `Unmarshaller` instance, as long as you don't use one instance from two threads at the same time.

Q: Why can't I cast the unmarshalled object into the generated type.

A: When you invoke `JAXBContext.newInstance("aaa.bbb.ccc")`, it tries to load classes and resources using the same classloader used to load the `JAXBContext` class itself. This classloader may be different from the classloader which was used to load your application (see the picture Parent/Child classloader). In this case, you'll see the above error. This problem is often seen with application servers, Jakarta EE containers, Ant, JUnit, and other applications that use sophisticated class loading mechanisms.

Figure 1. Parent/Child classloader



With some applications, things get even more complicated when the Jakarta XML Binding-generated code can be loaded by either classloader. In this case, `JAXBContext.newInstance("aaa.bbb.ccc")` will work but the JVM ends up loading two copies of the generated classes for each class loader. As a result, unmarshalling works

but an attempt to cast the returned object into the expected type will fail, even though its `getClass().getName()` returns the expected name.

The solution for both situations is to pass your curent class loader like this:

```
JAXBContext.newInstance("aaa.bbb.ccc", this.getClass().getClassLoader());
```

In general, if you are writing code that uses Jakarta XML Binding, it is always better to explicitly pass in a class loader, so that your code will work no matter where it is deployed.

Q: Which jar files do I need to distribute with my application that uses the Eclipse Implementation of JAXB?

A:

```
$JAXB_HOME/mod/jakarta.xml.bind-api.jar
$JAXB_HOME/mod/jakarta.activation-api.jar
$JAXB_HOME/mod/angus-activation.jar
$JAXB_HOME/mod/jaxb-core.jar
$JAXB_HOME/mod/jaxb-impl.jar
```

Q: How can I cause the `Marshaller` to generate `CDATA` blocks?

A: This functionality is not available from Eclipse Implementation of JAXB directly, but you can configure an Apache Xerces-J `XMLSerializer` to produce `CDATA` blocks. Please review the JaxbCDATASample.java [download/JaxbCDATASample.java] sample app for more detail.

Q: Can I access <xs:any/> as a DOM node?

A: In Eclipse Implementation of JAXB, <xs:any/> is handled correctly without any customization.

1. If it's `strict`, it will map to `Object` or `List<Object>` and when you unmarshal documents, you'll get objects that map to elements (such as `JAXBElements` or classes that are annotated with `XmlRootElement`).

2. If it's `skip`, it will map to `org.w3c.dom.Element` or `List<Element>` and when you unmarshal documents, you'll get DOM elements.

3. If it's `lax`, it will map to the same as with `strict`, and when you unmarshal documents, you'll get either:

   a. `JAXBElement`s

   b. classes that are annotated with `XmlRootElement`

   c. DOM elements

Q: How do I find out which version of the Eclipse Implementation of JAXB I'm using?

A: Run the following command

```
$ java -jar jaxb-xjc.jar -version
```

Alternatively, each Eclipse Implementation of JAXB jar has version information in its `META-INF/MANIFEST.MF`, such as this:

```
Manifest-Version: 1.0
Specification-Title: Jakarta XML Binding
Specification-Version: 4.0
```

```
Specification-Vendor: Eclipse Foundation
Implementation-Title: Eclipse Implementation of JAXB
Implementation-Version: 4.0.0
Implementation-Vendor: Eclipse Foundation
Implementation-Vendor-Id: org.eclipse
Build-Id: 2022-05-18 22:33
Class-Path: jaxb-core.jar jaxb-impl.jar
```

Related Articles

Table of Contents

1. Introductory

- Exchanging Data with XML and JAXB, Part I [https://blogs.oracle.com/CoreJavaTechTips/entry/exchanging_data_with_xml_and] and Part II [https://blogs.oracle.com/CoreJavaTechTips/entry/exchanging_data_with_xml_and1] by Jennie Hall

2. Blogs

- Aleksei Valikov's blog [http://lexicore.blogspot.com/]

3. Interesting articles

- JAXB 1.0.5 and RELAX NG support [http://www.devx.com/xml/Article/28784/0/page/4]
- JAXB and XInclude - Alternative to XIncluder [http://amazing-development.com/archives/2005/12/08/xml-with-schema-and-xinclude-in-java/]
- Kohmori Reports now using JAXB! [http://www.jroller.com/gmazza/entry/kohmori_reports_now_using_jaxb]