Java ™ Architecture for XML Binding readme

Implementation Version: 2.1.8 fcs

The Java ™ Architecture for XML Binding (JAXB) provides an API and tools that automate the mapping between XML documents and Java objects.

The JAXB 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

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

Documentation

Documentation for this release consists of the following:

- Release Notes
- Running the binding compiler (XJC): [command-line instructions, using the XJC Ant task]
- Running the schema generator (schemagen): [command-line instructions, using the SchemaGen Ant task]
- Unofficial JAXB User's Guide
- Javadoc API documentation (javax.xml.bind.*)
- Sample Applications
- JAXB FAQs [java.sun.com, jaxb.dev.java.net]

Software Licenses

$Revision: 1.2 $
$Date: 2008/04/16 22:45:56 $

Java ™ Architecture for XML Binding
Binding Compiler (xjc)

Implementation Version: 2.1.8 fcs

Launching xjc

The binding compiler can be launched using the appropriate xjc shell script in the bin directory for your platform. We also provide an Ant task to run the binding complier - see the instructions for using xjc with Ant.

For Solaris/Linux

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

For WindowsNT/2000/XP

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

Output

```
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        :  behave like XJC 2.0 and generate code that doesn't use any
2.1 features.
  -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
```

Execute the jaxb-xjc.jar JAR File

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

For Solaris/Linux:

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

For 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.

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 JAXB Specification. Appendix E.2 defines a set of W3C XML Schema features that are not completely supported by JAXB v1.0. 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 JAXB Vendor Extensions.

-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.

-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 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
Avoid generating code that relies on any JAXB 2.1 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 JAXB Vendor Extension.

-relaxng-compact
Treat input schemas as RELAX NG compact syntax(experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.

-dtd
Treat input schemas as XML DTD (experimental, unsupported). Support for RELAX NG schemas is provided as a JAXB Vendor Extension.

-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.

<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` 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 JAXB customization file (but with vendor extensions.)

Summary of Deprecated and Removed Command Line Options

-host & -port
These options have been deprecated and replaced with the **-httpproxy** 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 JAXB 2.0 specification has defined a portable runtime, it is no longer necessary for the JAXB RI to generate **/impl/runtime packages. Therefore, this switch is obsolete and has been removed.

-source
The -source compatibility switch was introduced in the first JAXB 2.0 Early Access release for convenience reasons. You shall not rely on this switch, because it might get removed from any future release of JAXB 2.0. If you need to generate 1.0.x code, please use an installation of the 1.0.x codebase.

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.

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:

$Revision: 1.2 $
$Date: 2008/06/20 08:36:11 $

Java ™ Architecture for XML Binding 1.0.x Release Notes

Implementation Version: 2.1.8 fcs

Overview

JAXB 2.0 is backwards compatible with JAXB 1.0 - you can deploy your existing 1.0 applications on the 2.0 runtime (provided that you also bundle the jaxb1-impl.jar) and they should run without modification. In the event that you are unable to migrate to JAXB 2.0, XJC in JAXB 2.0 is also capable of effectively becoming XJC 1.0. That is, it can compile a schema in the same way JAXB 1.0.x used to compile. This document explains this aspect of XJC.

Changelog

See the changelog document for a comprehensive list of changes in the 1.0.x releases.

Known Limitations of the bundled 1.0.x Release

The following list summarizes the known limitations of the JAXB 1.0.x runtime

Think you've found a new bug? File an issue on java.net!

- The JAXB Specification does not require full W3C XML Schema support in version 1.0. Please refer to Appendix E.2 for full details.
- Specifying large values for `maxOccurs` attributes can cause `java.lang.OutOfMemoryError`
- The `@enableFailFastCheck` optional customization of `<jaxb:globalBindings>` is not implemented yet.
- Specifying HTML markup tags within `<jaxb:javadoc>` customizations requires you to enclose the data within a CDATA section or escape all left angle brackets using "`&lt;`". See XML 1.0 2nd Edition for more detail. In addition, you must include < `body`> tags when specifying javadoc under the `<jaxb:package>` customization.
- The RI has some difficulty handling fixed attributes. For example, the validator does not detect errors when the XML data has a value different from the fixed value defined for the attribute in the schema.
- Compilations of sufficiently large and/or complex schemas may fail with `java.lang.StackOverflowError`. Try working around this problem by increasing the native stack size. For example, to increase the stack size to 2 megabytes:

  ```
  java -Xss2m
  ```

  If you run out of memory, try increasing the maximum memory size. For example, to increase the maximum memory to 128 megabytes:

  ```
  java -Xmx128m
  ```

- The JAXB Specification currently does not address the thread safety of any of the runtime classes. In the case of the Sun JAXB RI, the `JAXBContext` class **is** thread safe, but the `Marshaller`, `Unmarshaller`, and `Validator` classes **are not** thread safe.
- In rare cases, you may see: "`java.lang.Error: unable to load schema-for-schema for W3C XML Schema`" while attempting to compile your schemas. If you see this error, try updating your version of J2SE SDK.
- If you are getting error messages like:

```
[ERROR] Unable to load "CustomString" Is this class available in the classpath?
  line 15 of test.xsd
```

  Make sure that the classes you are referencing in your `<jxb:javaType>` and `<xjc:superClass>` customizations are compiled and added to the classpath via the "`-classpath`" command-line option for the `xjc` command.
- A new list of compiler restrictions has been added to the binding compiler documentation.
- Under certain conditions, xjc may give this warning:

  > Unable to validate your schema. Most likely, the JVM has loaded an incompatible XML parser implementation. You should fix this before relying on the generated code.
  >
  > Please see the release notes for details.

  The usual condition is that a version of xercesImpl.jar (as might be bundled with ant, an IDE, or an AppServer e.g.) is in your CLASSPATH before the one shipped with the Java WSDP. This deters xjc from resolving dependencies it has on internal Xerces APIs. In previous releases of JAXB, this error condition would manifest itself as an `IllegalAccessError`.

  To fix this, make sure the latest version of `xercesImpl.jar` precedes any other version of Xerces in your CLASSPATH. Otherwise, xjc can't validate your schema and you shouldn't rely on the generated code.

- The fact that `javax.xml.bind.util.JAXBSource` derives from `javax.xml.transform.sax.SAXSource` is an implementation detail. Thus in general applications are strongly discouraged from accessing methods defined on `SAXSource`. In particular, the `setXMLReader` and `setInputSource` methods shall never be called. The `XMLReader` object obtained by the `getXMLReader` method shall be used only for parsing the `InputSource` object returned by the `getInputSource` method. Similarly the `InputSource` object obtained by the `getInputSource` method shall be used only for being parsed by the `XMLReader` object returned by the `getXMLReader`.

  For `javax.xml.bind.util.JAXBResult`, applications are strongly discouraged from accessing methods defined on `javax.xml.transform.sax.SAXResult`. In particular it shall never attempt to call the `setHandler`, `setLexicalHandler`, and `setSystemId` methods.

  These limitations will be incorporated into the next version of the javadocs.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
Vendor Extensions

Implementation Version: 2.1.8 fcs

This page contains information about vendor-specific features provided by the JAXB RI.

JAXB RI Runtime Properties

This document describes JAXB RI specific properties that affect the way that the JAXB runtime library behaves.

JAXB RI Binding Customizations

This document describes additional binding customizations that can be used to control the generated source code.

JAXB RI Experimental Schema Language Support

This document describes the JAXB RI's experimental support for W3C XML Schema features not currently described in the JAXB Specification as well as support for other schema languages (RELAX NG and DTD).

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
JAXB Community

Implementation Version: 2.1.8 fcs

This page contains information about communities around the JAXB RI.

JAXB Project at java.net

Join the JAXB Community on java.net! This community is the ideal place for people interested in JAXB to share their ideas and ask each other questions. Join now!

Mailing List

This mailing list is used for discussions/questions related to JAXB usage, plugin development, and etc.

Frequently Asked Questions

We collect questions that are asked often and their answers.

User Support Forum

While we prefer mailing list-based user support, we do have a web forum where you can ask questions.

Issue Tracker

If you find any bug in the JAXB RI, please file it as a bug in the issue tracker. A test case would be greatly appreciated.

jaxb-spec-comments@sun.com

Use this alias for feedback about the JAXB Specification and API documentation

---

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding Release Notes

Implementation Version: 2.1.8 fcs

This document contains information that should help you use this software library more effectively. See the JAXB FAQ for additional information.

The most up-to-date version of this document can be found on-line.

Java ™ 2 Platform, Standard Edition (J2SE ™) Requirements

This release of the JAXB Reference Implementation requires J2SE 5.0 or higher.

Identifying the JAR Files

Use	Description	Jars
2.x Runtime	Jars required to deploy a JAXB 2.x client	jaxb-api.jar, jaxb-impl.jar, jsr173_1.0_api.jar, activation.jar
1.0 Runtime	Jars required to deploy an older JAXB 1.0 client	2.x Runtime + jaxb1-impl.jar
Compiler	Jars required at your development environment (but not runtime)	jaxb-xjc.jar

Locating the Normative Binding Schema

You may find information about the normative binding schema defined in the JAXB Specification at http://java.sun.com/xml/ns/jaxb.

Changelog

- JAXB 2.x changelog
- Please refer to the JAXB 1.0.x Release Notes for a comprehensive list of changes in the 1.0.x releases.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
Sample Apps Readme

Implementation Version: 2.1.8 fcs

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

Using the Runtime Binding Framework

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 `javax.xml.bind` runtime framework
5. Set your CLASSPATH to include all of the JAR files
6. Compile all of your Java sources with `javac`
7. Run it!

Java-2-Schema

Java-2-Schema is the process of augmenting existing Java classes with the annotations defined in the `javax.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 `javax.xml.bind.annotation` annotations to control the binding process
3. Set your CLASSPATH to include all of 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 and the extensive sample application documentation.

Building and Running the Sample Apps with Ant

To run the sample applications, just go into each sample directory, and run `ant` without any option.

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

List of Sample Apps

dtd

This sample application illustrate some of the DTD support available in the JAXB RI's extension mode. Please refer to the Vendor Extensions page for more detail.

Universal Business Language (UBL)

This project processes a UBL (Universal Business Language) order instance and prints a report to the screen.

XML message passing via socket

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.

ClassResolver

This little DI-container-by-JAXB example demonstrates how one can avoid passing in a list of classes upfront, and instead load classes lazily.

catalog resolver

This example demonstrates how to use the "-catalog" compiler switch to handle references to schemas in external web sites.

Java to Schema Binding

This sample application demonstrates marshalling, unmarshalling and unmarshal validation with existing Java classes annotated with JAXB annotations.

Using unmarshaller (formerly SampleApp1)

This sample application demonstrates how to unmarshal an instance document into a Java content tree and access data contained within it.

Ordering Properties and Fieldes in Java to Schema Bindings

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.

Updateable Partial Binding using Binder

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.

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.

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 <tt>com.sun.xml.bind.extra.Locatable</tt>.

Ordering Properties and Fieldes in Java to Schema Bindings

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.

element-substitution

This sample application illustrates how W3C XML Schema substitution groups are supported in JAXB RI's extension mode. Please refer to the Vendor Extensions page for more detail.

datatypeconverter (formerly SampleApp7)

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.

external-customize (formerly SampleApp8)

This sample application is identical to the datatypeconverter sample application (formerly SampleApp7) except that the binding customizations are contained in an external binding file.

create-marshal (formerly SampleApp3)

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.

fix-collides (formerly part of SampleApp9)

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.

pull parser based unmarshalling

This sample app demonstrates how a pull-parser can be used with JAXB to increase the flexibility of processing.

RI-specific customizations

This example demonstrates how to use <xjc:superClass> vendor extensions provided by Sun's JAXB RI, as well as <jaxb:serializable> customization.

character-escape

This example shows how you can use the new JAXB RI Marshaller property "com.sun.xml.bind.characterEscapeHandler" to change the default character escaping behavior.

Adapters for custom marshaling/unmarshaling XML content

This sample application demonstrates the use of interface XmlAdapter and annotation XmlJavaTypeAdapter for custom marshaling/unmarshaling XML content into/out of a Java type.

Streaming Unmarshalling w/o StAX

This example illustrates a different approach to the streaming unmarshalling, which is suitable for processing a large document.

Application-driven cycle handling

JAXB RI's vendor extension "CycleRecoverable" provides application a hook to handle cycles in the object graph. Advanced.

validating unmarshaller (formerly SampleApp4)

This sample application demonstrates how to enable validation during the unmarshal operations.

Type substitutoin support

This sample app demonstrates type substitution using the W3C XML Schema Part 0: Primer international purchase order schema.

namespace-prefix

This sample application demonstrates how to use the new JAXB RI Marshaller property "com.sun.xml.bind.namespacePrefixMapper" to customize the namespace prefixes generated during marshalling.

Defining XML elements via @XmlRootElement

This sample application demonstrates the use of annotation @XmlRootElement to define a class to be an XML element.

@XmlAttribute used to define properties and fields as XML Attributes

This sample application demonstrates the use of annotation @XmlAttribute for defining Java properties and fields as XML attributes.

Generating 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.

Marshalling output customization

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.

Annotation @XmlSchemaType is used to customize the mapping of a property or field to an XML built-in type.

This sample application demonstrates the use of annotation @XmlSchemaType to customize the mapping of a property or field to an XML built-in type.

modify-marshal (formerly SampleApp2)

This sample application demonstrates how to modify a java content tree and marshal it back to XML data.

inline-customize (formerly SampleApp6)

This sample application demonstrates how to customize the default binding produced by the XJC binding compiler.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding 2.0 Changelog

Implementation Version: 2.1.8 fcs

The JAXB 2.0 RI is a major reimplementation to meet the requirements of the 2.0 specification.

Please refer to the JAXB 1.0.x changelog for older releases.

Notable Changes between 2.1.7 to 2.1.8

- Fixes to bugs reported in java.net

Notable Changes between 2.1.6 to 2.1.7

- Fixed documentation that incorrectly showed that JAXB RI is CDDL only (it's actually CDDL/GPLv2+classpath dual license)
- Fixes to bugs reported in java.net

Notable Changes between 2.1.5 to 2.1.6

- Fixes to bugs reported in java.net

Notable Changes between 2.1.4 to 2.1.5

- Fixes to bugs reported in java.net

Notable Changes between 2.1.3 to 2.1.4

- Fixes to bugs reported in java.net

Notable Changes between 2.1.2 to 2.1.3

- Fixes to bugs reported in java.net

Notable Changes between 2.1.1 to 2.1.2

- Fixes to bugs reported in java.net

Notable Changes between 2.1 First Customer Ship (FCS) to 2.1.1

- Fixes to bugs reported in java.net
- <xjc:subsitutable> customization added

Notable Changes between 2.1 Eary Access 2 to 2.1 First Customer Ship (FCS)

- Fixes to bugs reported in java.net

Notable Changes between 2.1 Eary Access to 2.1 Early Access 2

- Fixes to bugs reported in java.net
- All the changes in 2.0.x line, up to 2.0.4
    - Fixes to bugs reported in java.net
    - Bug fix: javax.xml.bind.Binder.marshal() doesn't throw expected MarshalException
    - Bug fix: javax.xml.bind.Binder.getJAXBNode(null) doesn't throw documented exception
    - More bug fixes

Notable Changes between 2.0.2 to 2.1 Early Access

- Fixes to bugs reported in java.net

Notable Changes between 2.0.2 to 2.0.3

- Fixes to bugs reported in java.net
- JavaSE 6 release is expected to include this version of the JAXB RI (certainly as of build 102.)

Notable Changes between 2.0.1 to 2.0.2

- Fixes to bugs reported in java.net
- Bug fix: Unmarshaller should report validation error for elements with xsi:nil="true" and content
- Bug fix: ClassCastException in JAXB when using interfaces as parameters
- More bug fixes

Notable Changes between 2.0 to 2.0.1

- Fixes to bugs reported in java.net
- More bug fixes
- The simpler and better binding mode is improved
- JAXB unofficial user's guide is available now, though it's still a work in progress

Notable Changes between 2.0 Early Access 3 and 2.0 First Customer Ship (FCS)

- Java to schema samples are added
- Added <xjc:javaType> vendor customization
- Added experimental <xjc:simple> vendor customization, which brings in a new simpler and better binding mode
- The spec has renamed `AccessType` to `XmlAccessType`, and `@AccessorType` to `@XmlAccessorType`.
- Various error handling improvements
- Experimental canonicaliztion support is added.
- The '-b' option can now take a directory and recursively search for all "*.xjb" files.
- Fixed various issues regarding using JAXB from codef inside a restricted security sandbox.
- Added more pluggability points for plugins to customize the code generation behavior.
- Some of the code is split into a separate `istack-commons` project to promote more reuse among projects.
- Made a few changes so that RetroTranslator can translate the JAXB RI (and its generated code) to run it on JDK 1.4 and earlier
- Improved the quality of the generated code by removing unnecessary imports.
- Other countless bug fixes

Notable Changes between 2.0 Early Access 2 and 2.0 Early Access 3

- Map property can be now correctly bound to XML Schema
- Default marshaller error handling behavior became draconian (previously errors were ignored.)
- @link to a parameterized type is now correctly generated
- started producing architecture document for those who want to build plugins or play with the RI internal.

- XJC now uses the platform default proxy setting by default.
- @XmlAccessorOrder, @XmlSchemaType and @XmlInlineBinaryData are implemented
- @XmlJavaTypeAdapter on a class/package is implemented
- Marshaller life-cycle events are implemented
- Integration to FastInfoset is improved in terms of performance
- XJC can generate @Generated
- The unmarshaller is significantly rewritten for better performance
- Added schemagen tool and its Ant task
- Various improvements in error reporting during unmarshalling/marshalling
- JAXB RI is now under CDDL

Notable Changes between 2.0 Early Access and 2.0 Early Access 2

- The default for @XmlAccessorType was changed to PUBLIC_MEMBER
- Optimized binary data handling enabled by callbacks in package javax.xml.bind.attachment. Standards supported include MTOM/XOP and WS-I AP 1.0 ref:swaRef.
- Unmarshal/marshal support of element defaulting
- Improved the quality of the generated Java code
- Fixed bugs in default value handling
- Runtime performance improvements, memory usage improvements
- Added support for <xjc:superInterface> vendor extension
- Migrated source code to http://jaxb2-sources.dev.java.net
- Published NetBeans project file for JAXB RI
- Added more support to the schema generator: anonymous complex types, attribute refs, ID/IDREF, etc
- Implemented `javax.xml.bind.Binder` support (not 100% done yet)
- Implemented marshal-time validation
- Improved xjc command line interface - better support for proxy options, more options for specifying schema files
- Added schema-2-Java support for simple type substitution
- Added support for the new `<jaxb:globalBindings localScoping="nested" | "toplevel">` customization which helps control deeply nested classes
- Made the default `ValidationEventHandler` more forgiving in 2.0 than it was in 1.0 (The class still behaves the same as it did when used by a 1.0 app)
- Added wildcard support for DTD
- Numerous other small changes and bugfixes....

Notable Changes between 1.0.x FCS and 2.0 Early Access

- Support for 100% W3C XML Schema (not all in EA, but planned for FCS)
- Support for binding Java to XML

- Addition of `javax.xml.bind.annotation` package for controling the binding from Java to XML
- Significant reduction in the number of generated schema-derived classes
    - Per complex type definition, generate one value class instead of an interface and implementation class.
    - Per global element declaration, generate a factory method instead of generating a schema-derived interface and implementation class.
- Replaced the validation capabilities in 1.0 with JAXP 1.3 validation API's
- Smaller runtime libraries
- Numerous other changes...

$Revision: 1.3 $
$Date: 2008/08/29 13:13:55 $

Java ™ Architecture for XML Binding
Using XJC with Ant

Implementation Version: 2.1.8 fcs

The jaxb-xjc.jar file contains the XJCTask.class file, which allows the XJC binding compiler to be invoked from the Ant 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 applications.

Synopsis

Environment Variables

- ANT_OPTS - 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.

Parameter Attributes

it is useful, if 'binding' attribute is not there.

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
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 JAXB RI 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
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
source	Specify which version of the schema compiler to use. Must be either "1.0", "2.0", or "2.1". The compiler will parse XML Schema and bind them according to the rules specified by the given version of the specification. Note that 2.1 is backward compatible with 2.0, so these two flags won't make any difference. You shall not rely on this attribute, because it might get removed from any future version of JAXB. For generating 1.0.x code, please use an installation of the 1.0.x codebase.	No, defaults to "2.1"
target	Specifies the runtime environment in which the generated code is supposed to run. This value must be smaller than the `source` attribute, if specified (IOW, you can't do `source="1.0" target="2.0"`) This allows more up-to-date versions of XJC to be used for developing applications that run on earlier JAXB versions.	No, defaults to "2.1"
language	Specifies the schema language to compile. Supported values are "WSDL", "XMLSCHEMA", and "WSDL." Case insensitive	No, defaults to "XMLSCHEMA"

Nested Elements

xjc supports the following nested element parameters.

schema

To compile more than one schema at the same time, use a nested <schema> element, which has the same syntax as <fileset> .

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> .

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" .

arg

Additional command line arguments passed to the XJC. For details about the syntax, see the relevant section 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.

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> .

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> .

xmlcatalog

The xmlcatalog element is used to resolve entities when parsing schema documents.

Generated Resource Files

Please see the xjc page for more detail.

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*.

Schema Language Support

This release of the JAXB RI 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.

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
```

$Revision: 1.2 $
$Date: 2008/06/20 08:36:11 $

Java ™ Architecture for XML Binding
Schema Generator

Implementation Version: 2.1.8 fcs

Launching schemagen

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

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 using schemagen with Ant.

For Solaris/Linux

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

For WindowsNT/2000/XP

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

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.

Command Line Options

```
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
    -episode <file>   :  generate episode file for separate compilation
    -version          :  display version information
    -help             :  Display this usage message
```

Summary of Command Line Options

-episode

Generates the "episode file", which is really just a JAXB customization file (but with vendor extensions specific to the JAXB RI, as of 2.1.) 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.

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 the schema generator ant task for that purpose.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
Using SchemaGen with Ant

Implementation Version: 2.1.8 fcs

The jaxb-xjc.jar file contains the SchemaGenTask.class file, which allows the schema generator to be invoked from the Ant 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 applications.

Synopsis

Environment Variables

- ANT_OPTS - 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.

Parameter Attributes

schemagen supports most of the attributes defined by the javac task, 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 the schemagen page.	No

Nested Elements

xjc supports all the nested elements defined by the javac task, the following nested element parameters.

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.

classpath

A path-like structure that represents the classpath. If your Java sources/classes depend on other libraries, they need to be available in the classpath.

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>
```

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
3rd Party Tools

Implementation Version: 2.1.8 fcs

Maven JAXB Plugin

A maven2 plugin for this version of JAXB is being developed at java.net. There's another independent maven2 plugin at java.net. Also, if you are using Maven, JAXB jars are available in the java.net maven repository.

Eclipse Plugin

Kirill is working on Eclipse plugin. If you are interested in this, send him feedback.

IntelliJ IDEA Plugin

Kirill is working on IntelliJ IDEA plugin. If you are interested in this, send him feedback.

JAXB Workshop

Another project from Kirill is a collection of tools that are often useful for using JAXB. See his jaxb-workshop project (and some of the screenshots.)

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 the list of plugins (if you are interested in hosting your plugin here, let us know.)

RDBMS Persistence

Lexi is working on the hyperjaxb3 project, which is the RDBMS persistence support for the JAXB RI. If you are interested in this, send him feedback.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

XML Entity and URI Resolvers

Version 1.1

Norman Walsh, Staff Engineer
Sun Microsystems, XML Technology Center

05 Nov 2001

Table of Contents

- Finding Resources on the Net
  - Resolver Classes Version 1.1
- What's Wrong with System Identifiers?
- Naming Resources
  - Public Identifiers
  - Uniform Resource Names
- Resolving Names
- Catalog Files
- Understanding Catalog Files
  - OASIS XML Catalogs
  - OASIS TR9401 Catalogs
  - XCatalogs
  - Resolution Semantics
- Controlling the Catalog Resolver
- Using Catalogs with Popular Applications
- Adding Catalog Support to Your Applications
- Catalogs In Action
  - Using resolver
  - Using xparse
- May All Your Names Resolve Successfully!

Finding Resources on the Net

It's very common for web resources to be related to other resources: documents rely on DTDs and schemas, schemas

are derived from other schemas, stylesheets are often customizations of other stylesheets, documents refer to the schemas and stylesheets with which the expect to be processed, etc. These relationships are expressed using URIs, most often URLs.

Relying on URLs to directly identify resources to be retrieved often causes problems for end users:

1. If they're absolute URLs, they only work when you can reach them[1]. Relying on remote resources makes XML processing susceptible to both planned and unplanned network downtime.

   The URL “http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd” isn't very useful if I'm on an airplane at 35,000 feet.

2. If they're relative URLs, they're only useful in the context where the were initially created.

   The URL “../../xml/dtd/docbookx.xml” isn't useful *anywhere* on my system. Neither, for that matter, is “/export/home/fred/docbook412/docbookx.xml”.

One way to avoid these problems is to use an entity resolver (a standard part of SAX) or a URI Resolver (a standard part of JAXP). A resolver can examine the URIs of the resources being requested and determine how best to satisfy those requests.

The best way to make this function in an interoperable way is to define a standard format for mapping system identifiers and URIs. The OASIS Entity Resolution Technical Committee is defining an XML representation for just such a mapping. These “catalog files” can be used to map public and system identifiers and other URIs to local files (or just other URIs).

Resolver Classes Version 1.1

The Resolver classes that are described in this article greatly simplify the task of using Catalog files to perform entity resolution. Many users will want to simply use these classes directly “out of the box” with their applications (such as Xalan and Saxon), but developers may also be interested in the JavaDoc API Documentation.

Changes from Version 1.0

The most important change in this release is the availability of both source and binary forms under a generous license agreement.

Other than that, there have been a number of minor bug fixes and the introduction of system properties in addition to the `CatalogManager.properties` file to control the resolver.

What's Wrong with System Identifiers?

The problems associated with system identifiers (and URIs in general) arise in several ways:

1. I have an XML document that I want to publish on the web or include in the distribution of some piece of software. On my system, I keep the doctype of the document in some local directory, so my doctype declaration reads:

   ```
   <!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
                    "file:///n:/share/doctypes/docbook/xml/docbookx.dtd">
   ```

   As soon as I distribute this document, I immediately begin getting error reports from customers who can't read the document because they don't have DocBook installed at the location identified by the URI in my document.

2. Or I remember to change the URI before I publish the document:

   ```
   <!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
                    "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
   ```

   And the next time I try to edit the document, *I get errors* because I happen to be working on my laptop on a plane somewhere and can't get to the net.

3. Just as often, I get tripped up this way: I'm working collaboratively with a colleague. She's created initial drafts of some documents that I'm supposed to review and edit. So I grab them and find that I can't open or publish them because I don't have the same network connections she has or I don't have my applications installed in the same place. And if I change the system identifiers so they work on my system, she has the same problems when I send them back to her.

4. These problems aren't limited to editing applications. If I write a special stylesheet for formatting our collaborative document, it will include some reference to the “main” stylesheet:

   ```
   <xsl:import href="/path/to/real/stylesheet.xsl"/>
   ```

   But this won't work on my colleague's machine because she has the main stylesheet installed somewhere else.

Public identifiers offer an effective solution to this problem, at least for documents. They provide global, unique names for entities independent of their storage location. Unfortunately, public identifiers aren't used very often because many users find that they cannot rely on applications resolving them in an interoperable manner.

For XSLT, XML Schemas, and other applications that rely on URIs without providing a mechanism for associating public identifiers with them, the situation is a little more irksome, but it can still be addressed using a URI Resolver.

Naming Resources

In some contexts, it's more useful to refer to a resource by name than by address. If I want the version 3.1 of the DocBook DTD, or the 1911 edition of Webster's dictionary, or *The Declaration of Independence*, that's what I want, irrespective of its location on the net (or even if it's available on the net). While it is possible to view a URL as an address, I don't think that's the natural interpretation.

There are currently two ways that I might reasonably assign an address-independent name to an object: public identifiers or Uniform Resource Names (URNs)[2].

Public Identifiers

Public identifiers are part of XML 1.0. They can occur in any form of external entity declaration. They allow you to give a globally unique name to any entity. For example, the XML version of DocBook V4.1.2 is identified with the following public identifier:

```
-//OASIS//DTD DocBook XML V4.1.2//EN
```

You'll see this identifier in the two doctype declarations I used earlier. This identifier gives no indication of where the resource (the DTD) may be found, but it does uniquely name the resource. That public identifier, now and forever refers to the XML version of DocBook V4.1.2.

Uniform Resource Names

URNs are a form of URI. Like public identifiers, they give a location-neutral, globally unique name to an entity. For example, OASIS might choose to identify the XML version of DocBook V4.1.2 with the following URN:

```
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

Like a public identifier, a URN can now and forever refer to a specific entity in a location-independent manner.

The publicid URN Namespace

Public identifiers don't fit very well into the web architecture (they are not, for example, always valid URIs). This problem can be addressed by the `publicid` URN namespace defined by RFC 3151.

This namespace allows public identifiers to be easily represented as URNs. The OASIS XML Catalog specification accords special status to URNs of this form so that catalog resolution occurs in the expected way.

Resolving Names

Having extolled the virtues of location-independent names, it must be said that a name isn't very useful if you can't find the thing it refers to. In order to do that, you must have a name resolution mechanism that allows you to determine what resource is referred to by a given name.

One important feature of this mechanism is that it can allow resources to be distributed, so you don't have to go to http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd to get the XML version of DocBook V4.1.2, if you have a local copy.

There are a few possible resolution mechanisms:

- The application just “knows”. Sure, it sounds a little silly, but this is currently the mechanism being used for namespaces. Applications know what the semantics of namespaced elements are because they recognize the namespace URI.

- OASIS Catalog files provide a mechanism for mapping public and system identifiers, allowing resolution to both local and distributed resources. This is the resolution scheme we're going to consider for the balance of this column.

- Many other mechanisms are possible. There are already a few for URNs, including at least one built on top of DNS, but they aren't widely deployed.

Catalog Files

Catalog files are straightforward text files that describe a mapping from names to addresses. Here's a simple one:

Example 1. An Example Catalog File

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

<public publicId="-//OASIS//DTD XML DocBook V4.1.2//EN"
      uri="docbook/xml/docbookx.dtd"/>

<system systemId="urn:x-oasis:docbook-xml-v4.1.2"
      uri="docbook/xml/docbookx.dtd"/>

<delegatePublic publicIdStartString="-//Example//"
       catalog="http://www.example.com/catalog"/>
</catalog>
```

This file maps both the public identifier and the URN I mentioned earlier to a local copy of DocBook on my system. If the doctype declaration uses the public identifier for DocBook, *I'll get DocBook* regardless of the (possibly bogus) system identifier! Likewise, my local copy of DocBook will be used if the system identifier contains the DocBook URN.

The delegate entry instructs the resolver to use the catalog “`http://www.example.com/catalog`” for any public identifier that begins with “-//Example//”. The advantage of delegate in this case is that I don't have to parse that catalog file unless I encounter a public identifier that I reasonably expect to find there.

Understanding Catalog Files

The OASIS Entity Resolution Technical Committee is actively defining the next generation XML-based catalog file format. When this work is finished, it is expected to become the official XML Catalog format. In the meantime, the existing OASIS Technical Resolution TR9401 format is the standard.

OASIS XML Catalogs

OASIS XML Catalogs are being defined by the Entity Resolution Technical Committee. This article describes the 01 Aug 2001 draft. Note that this draft is labelled to reflect that it is "not an official committee work product and may not reflect the consensus opinion of the committee."

The document element for OASIS XML Catalogs is `catalog`. The official namespace name for OASIS XML Catalogs is "`urn:oasis:names:tc:entity:xmlns:xml:catalog`".

There are eight elements that can occur in an XML Catalog: `group`, `public`, `system`, `uri`, `delegatePublic`, `delegateSystem`, `delegateURI`, and `nextCatalog`:

`<catalog prefer="public|system" xml:base="uri-reference">`

> The `catalog` element is the root of an XML Catalog.
>
> The `prefer` setting determines whether or not public identifiers specified in the catalog are to be used in favor of system identifiers supplied in the document. Suppose you have an entity in your document for which both a public identifier and a system identifier has been specified, and the catalog only contains a mapping for the public identifier (e.g., a matching `public` catalog entry). If the current value of `prefer` is "public", the URI supplied in the matching `public` catalog entry will be used. If it is "system", the system identifier in the document will be used. (If the catalog contained a matching `system` catalog entry giving a mapping for the system identifier, that mapping would have been used, the public identifier would never have been considered, and the setting of override would have been irrelevant.)
>
> Generally, the purpose of catalogs is to override the system identifiers in XML documents, so `prefer` should usually be "public" in your catalogs.
>
> The `xml:base` URI is used to resolve relative URIs in the catalog as described in the XML Base specification.

`<group prefer="public|system" xml:base="uri-reference">`

> The `group` element serves merely as a wrapper around one or more other entries for the purpose of establishing the preference and base URI settings for those entries.

`<public publicId="pubid" uri="systemuri"/>`

> Maps the public identifier *`pubid`* to the system identifier *`systemuri`*.

`<system systemId="sysid" uri="systemuri"/>`

> Maps the system identifier *`sysid`* to the alternate system identifier *`systemuri`*.

`<uri name="uri" uri="alternateuri"/>`

The `uri` entry maps a *`uri`* to an *`alternateuri`*. This mapping, as might be performed by a JAXP URIResolver, for example, is independent of system and public identifier resolution.

`<delegatePublic publicIdStartString="pubid-prefix" catalog="cataloguri"/>`,
`<delegateSystem systemIdStartString="sysid-prefix" catalog="cataloguri"/>`,
`<delegateURI uriStartString="uri-prefix" catalog="cataloguri"/>`

The delegate entries specify that identifiers beginning with the matching prefix should be resolved using the catalog specified by the *`cataloguri`*. If multiple delegate entries of the same kind match, they will each be searched, starting with the longest prefix and continuing with the next longest to the shortest.

The delegate entries differs from the `nextCatalog` entry in the following way: alternate catalogs referenced with a `nextCatalog` entry are parsed and included in the current catalog. Delegated catalogs are only considered, and consequently only loaded and parsed, if necessary. Delegated catalogs are also used *instead of* the current catalog, not as part of the current catalog.

`<rewriteSystem systemIdStartString="sysid-prefix" rewritePrefix="new-prefix"/>`,
`<rewriteURI uriStartString="uri-prefix" rewritePrefix="new-prefix"/>`

Supports generalized rewriting of system identifiers and URIs. This allows all of the URI references to a particular document (which might include many different fragment identifiers) to be remapped to a different resource).

`<nextCatalog catalog="cataloguri"/>`

Adds the catalog file specified by the *`cataloguri`* to the end of the current catalog. This allows one catalog to refer to another.

OASIS TR9401 Catalogs

These catalogs are officially defined by OASIS Technical Resolution TR9401.

A Catalog is a text file that contains a sequence of entries. Of the 13 types of entries that are possible, only six are commonly applicable in XML systems: BASE, CATALOG, OVERRIDE, DELEGATE, PUBLIC, and SYSTEM:

BASE *`uri`*

Catalog entries can contain relative URIs. The BASE entry changes the base URI for subsequent relative URIs. The initial base URI is the URI of the *catalog* file.

In XML Catalogs, this functionality is provided by the closest applicable `xml:base` attribute, usually on the surrounding `catalog` or `group` element.

CATALOG *cataloguri*

This entry serves the same purpose as the `nextCatalog` entry in XML Catalogs.

OVERRIDE *YES|NO*

This entry enables or disables overriding of system identifiers for subsequent entries in the catalog file.

In XML Catalogs, this functionality is provided by the closest applicable `prefer` attribute on the surrounding `catalog` or `group` element.

An override value of “yes” is equivalent to “prefer="public"”.

DELEGATE *pubid-prefix cataloguri*

This entry serves the same purpose as the `delegate` entry in XML Catalogs.

PUBLIC *pubid systemuri*

This entry serves the same purpose as the `public` entry in XML Catalogs.

SYSTEM *sysid systemuri*

This entry serves the same purpose as the `system` entry in XML Catalogs.

XCatalogs

The Resolver classes also understand the XCatalog format supported by Apache.

Resolution Semantics

Resolution is performed in roughly the following way:

1. If a system entry matches the specified system identifier, it is used.
2. If no system entry matches the specified system identifier, but a rewrite entry matches, it is used.
3. If a public entry matches the specified public identifier and either `prefer` is public or no system identifier is provided, it is used.
4. If no exact match was found, but it matches one or more of the partial identifiers specified in delegate entries, the delegated catalogs are searched for a matching identifier.

For a more detailed description of resolution semantics, including the treatment of multiple catalog files and the complete rules for delegation, consult the XML Catalog standard.

Controlling the Catalog Resolver

The Resolver classes uses either Java system properties or a standard Java properties file to establish an initial environment. The property file, if it is used, must be called `CatalogManager.properties` and must be somewhere on your `CLASSPATH`. The following properties are supported:

System property `xml.catalog.files`; CatalogManager property `catalogs`

A semicolon-delimited list of catalog files. These are the catalog files that are initially consulted for resolution.

Unless you are incorporating the resolver classes into your own applications, and subsequently establishing an initial set of catalog files through some other means, at least one file must be specified, or all resolution will fail.

System property `xml.catalog.prefer`; CatalogManager property `prefer`

The initial prefer setting, either `public` or `system`.

System property `xml.catalog.verbosity`; CatalogManager property `verbosity`

An indication of how much status/debugging information you want to receive. The value is a number; the larger the number, the more information you will receive. A setting of 0 turns off all status information.

System property `xml.catalog.staticCatalog`; CatalogManager property `static-catalog`

In the course of processing, an application may parse several XML documents. If you are using the built-in `CatalogResolver`, this option controls whether or not a new instance of the resolver is constructed for each parse. For performance reasons, using a value of `yes`, indicating that a static catalog should be used for all parsing, is probably best.

System property `xml.catalog.allowPI`; CatalogManager property `allow-oasis-xml-catalog-pi`

This setting allows you to toggle whether or not the resolver classes obey the `<?oasis-xml-catalog?>` processing instruction.

System property `xml.catalog.className`; CatalogManager property `catalog-class-name`

If you're using the convenience classes `com.sun.resolver.tools.*`), this setting allows you to specify an alternate class name to use for the underlying catalog.

CatalogManager property `relative-catalogs`

If `relative-catalogs` is `yes`, relative catalogs in the `catalogs` property will be left relative; otherwise they will be made absolute with respect to the base URI of the `CatalogManager.properties` file. This setting has no effect on catalogs loaded from the `xml.catalogs.files` system property (which are always returned unchanged).

System property `xml.catalog.ignoreMissing`

By default, the resolver will issue warning messages if it cannot find a `CatalogManager.properties` file, or if resources are missing in that file. However if *either* `xml.catalog.ignoreMissing` is `yes`, or catalog files are specified with the `xml.catalog.catalogs` system property, this warning will be suppressed.

My `CatalogManager.properties` file looks like this:

Example 2. Example CatalogManager.properties File

```
#CatalogManager.properties

verbosity=1

relative-catalogs=yes

# Always use semicolons in this list
catalogs=./xcatalog;/share/doctypes/catalog;/share/doctypes/xcatalog

prefer=public

static-catalog=yes

allow-oasis-xml-catalog-pi=yes

catalog-class-name=com.sun.resolver.Resolver
```

Using Catalogs with Popular Applications

A number of popular applications provide easy access to catalog resolution:

Xalan

Recent development versions of Xalan include new command-line switches for setting the resolvers. You can use them directly with the `com.sun.resolver.tools` classes:

```
-URIRESOLVER com.sun.resolver.tools.CatalogResolver
-ENTITYRESOLVER com.sun.resolver.tools.CatalogResolver
```

Saxon

Similarly, Saxon supports command-line access to the resolvers:

```
-x com.sun.resolver.tools.ResolvingXMLReader
-y com.sun.resolver.tools.ResolvingXMLReader
-r com.sun.resolver.tools.CatalogResolver
```

The *-x* class is used to read source documents, the *-y* class is used to read stylesheets.

XP

To use XP, simply use the included `com.sun.xp.xml.sax.Driver` class instead of the default XP driver.

XT

Similarly, for XT, use the `com.sun.xt.xsl.sax.Driver` class.

Adding Catalog Support to Your Applications

If you work with Java applications using a parser that supports the SAX1 `Parser` interface or the SAX2 `XMLReader` interface, adding Catalog support to your applications is a snap. The SAX interfaces include an `entityResolver` hook designed to provide an application with an opportunity to do this sort of indirection. The Resolver classes implements the full OASIS Catalog semantics and provide an appropriate class that implements the SAX `entityResolver` interface.

All you have to do is setup a `com.sun.resolver.tools.CatalogResolver` on your parser's `entityResolver` hook. The code listing in Example 3 demonstrates how straightforward this is:

Example 3. Adding a CatalogResolver to Your Parser

```
import com.sun.resolver.tools.CatalogResolver;
...
   CatalogResolver cr = new CatalogResolver();
...
   yourParser.setEntityResolver(cr)
```

The system catalogs are loaded from the `CatalogManager.properties` file on your `CLASSPATH`. (For all the gory details about these classes, consult the API documentation.) You can explicitly parse your own catalogs (perhaps taken from command line arguments or a Preferences dialog) instead of or in addition to the system catalogs.

Catalogs In Action

The Resolver distribution includes a couple of test programs, **resolver** and **xparse**, that you can use to see how this all works.

Using resolver

The **resolver** application simply performs a catalog lookup and returns the result. Given the following catalog:

Example 4. An Example XML Catalog File

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

<public publicId="-//Example//DTD Example V1.0//EN"
      uri="example.dtd"/>

</catalog>
```

A demonstration of public identifier resolution can be achieved like this:

Example 5. Resolving Identifiers

```
$ java com.sun.resolver.apps.resolver -d 2 -c example/catalog.xml \
  -p "-//Example//DTD Example V1.0//EN" public
Loading catalog: ./catalog
Loading catalog: /share/doctypes/catalog
Resolve PUBLIC (publicid, systemid):
  public id: -//Example//DTD Example V1.0//EN
Loading catalog: file:/share/doctypes/entities.cat
Loading catalog: /share/doctypes/xcatalog
Loading catalog: example/catalog.xml
Result: file:/share/documents/articles/sun/2001/01-resolver/example/example.dtd
```

Using xparse

The **xparse** command simply sets up a catalog resolver and then parses a document. Any external entities encountered during the parse are resolved appropriately using the catalogs provided.

In order to use the program, you must have the `resolver.jar` file on your `CLASSPATH` and you must be using JAXP. In the examples that follow, I've already got these files on my `CLASSPATH`.

The file we'll be parsing is shown in Example 6.

Example 6. An xparse Example File

```
<!DOCTYPE example PUBLIC "-//Example//DTD Example V1.0//EN"
                "file:///dev/this/does/not/exist/example.dtd">
```

```
<example>
<p>This is just a trivial example.</p>
</example>
```

First let's look at what happens if you try to parse this document without any catalogs. For this example, I deleted the `catalogs` entry on my `CatalogManager.properties` file. As expected, the parse fails:

Example 7. Parsing Without a Catalog

```
$ java com.sun.resolver.apps.xparse -d 2 example.xml
Attempting validating, namespace-aware parse
Fatal error:example.xml:2:External entity not found:
   "file:///dev/this/does/not/exist/example.dtd".
Parse failed with 1 error and no warnings.
```

With an appropriate catalog file, we can map the public identifier to a local copy of the DTD. We could have mapped the system identifier instead (or as well), but the public identifier is probably more stable.

Using a command-line option to specify the catalog, I can now successfully parse the document:

Example 8. Parsing With a Catalog

```
$ java com.sun.resolver.apps.xparse -d 2 -c catalog.xml example.xml
Loading catalog: catalog.xml
Attempting validating, namespace-aware parse
Resolved public: -//Example//DTD Example V1.0//EN
    file:/share/documents/articles/sun/2001/01-resolver/example/example.dtd
Parse succeeded (0.32) with no errors and no warnings.
```

The additional messages in each of these examples arise as a consequence of the debugging option, *`-d 2`*. In practice, you can make resolution silent.

May All Your Names Resolve Successfully!

We hope that these classes become a standard part of your toolkit. Incorporating this code allows you to utilize public identifiers in XML documents with the confidence that you will be able to move those documents from one system to another and around the Web.

The Author: Sun Microsystems supports Norm's active participation in a number of standards efforts worldwide, including the XML Core, XSL and XML Schema Working Groups of the World Wide Web Consortium, the OASIS XSLT Conformance and RELAX NG Committees, the OASIS Entity Resolution Committee, for which he is the editor, and the OASIS DocBook Technical Committee, which he chairs.

---

[1] It is technically possible to use a proxy to transparently cache remote resources, thus making the cached resources available even when the real hosts are unreachable. In practice, this requires more technical skill (and system administration access) than many users have available. And I don't know of any such proxies that can be configured to provide preferential caching to the specific resources that are needed. Without such preferential treatment, its difficult to be sure that the resources you need are actually in the cache.

[2] URIs that rely on the domain name system to identify objects (in other words, all URLs) are addresses, not names, even though the domain name provides a level of indirection and the illusion of a stable name.

Java ™ Architecture for XML Binding 1.0.x Changelog

Implementation Version: 2.1.8 fcs

Please refer to the JAXB 2.0.x changelog for the most recent changes.

Changes between 1.0.6 FCS and 2.0 FCS

No change is made to the 1.0-compatible compiler in JAXB 2.0 FCS since 1.0.6

Changes between 1.0.5 FCS and 1.0.6 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0.3 FCS release.

- The license was changed from Sun Binary Code License (BCL) to Common Development and Distribution License (CDDL), allowing free-er redistribution.
- JAXB 1.0.6 jars are made available to the java.net maven repository.

Changes between 1.0.4 FCS and 1.0.5 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0.3 FCS release.

- Issue 55 - build id not propagated to localize message bundles
- Issue 62 - Add customizations support
- Issue 64 - Generate getEnumMap() method in enum classes
- Issue 65 - Add xdoclets support to JDocComment
- Issue 74 - Allow arbitrary extensions/customizations in bindings
- Issue 82 - Attribute-driven fields are not customized
- Fixed a bug in handling of inner class in the model group binding mode.
- Other minor bug fixes that we lost track of

Changes between 1.0.3 FCS and 1.0.4 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0.3 FCS release.

- Bug Fixes
  - IssueTracker:
    - 15 - can't parse W3 SOAP 2003-05 envelope XML schema: "xml:lang"
    - 18 - Performance Improvement in NamespaceContextImpl.java
    - 19 - Performance Improvement in NamespaceContextImpl.java
    - 24 - XJC doesn't work with directories with spaces in them
    - 28 - ValidatorImpl does not initialize DatatypeConverter
  - fixed an NPE that affected Acord and UBL 1.0 schemas
  - found and fixed a StackOverflowError in XJC
  - xjc handles non-existant ouput directories more gracefully
  - found and removed some dependencies on jdk1.4
  - fixed a bug in the entity resolver handling
- Other
  - updated UBL sample app to UBL 1.0 schema (Visit the UBL interest page for more information).

Changes between 1.0.2 FCS and 1.0.3 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0.2 FCS release.

- Bug Fixes
  - Improved command line error handling
  - Fixed bug where JAXB crashed when using JAXBResult
  - Improved line number reporting for DTD compilation messages
  - Improved naming collision error messages
  - Improved DTD wildcard handling
  - Improved error reporting when <javaType> contains an undefined simple type
  - Fixed a couple of bugs related to XJC error handling with external binding files
  - Fixed a bug that was causing an exception when Unmarshalling an element with int list as value
  - Fixed a bug related to the compilation of multiple schemas that import a common base schema
- Performance Enhancements
  - Fixed a memory leak in XSOM
  - Modified XJC to use less call stack to avoid StackOverflowErrors
  - Generated ObjectFactory classes now specify an initialCapaicty on their defaultImpl HashMaps. This should obviate the need to rehash at runtime for very large schemas.
  - Improved memory usage in XJC
  - Force string interning during unmarshalling to improve runtime performance
  - Reuse AttributesImpl object rather than reallocate
  - Lazily instantiate ListImpl classes in the generated code to save on memory usage
- Other
  - Added a new JAXB RI vendor customization that allows for partial-code generation (reduced functionality bindings)

- Began posting weekly binary and source bundles to java.net
- Added a -quiet option on the XJC command line to supress verbose output

Changes between 1.0.1 FCS and 1.0.2 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0.1 FCS release.

- Additional schema support (available only in "extension mode")
  - JAXB now supports `xsi:type` (XSD type substitution).
  - JAXB now supports `block`, `abstract`, `final`, `blockDefault`, and `finalDefault` attributes for the `<complexType>`, `<element>`, and `<schema>` elements.
- Enhancements
  - `XJCTask` no longer fails if the directory specified by an enclosed `<produces>` doesn't exist.
  - Marshaller error messages have been revised to be more informative.
  - The marshaller now supports marshalling of `QName`s.
  - `xjc:superClass` now supported for DTD.
  - `javaType` customization now supported for RELAX NG.
  - A new vendor extension suppresses `<?xml?>`.
  - A new vendor extension allows you to set `<!DOCTYPE>`
  - `NamespacePrefixMapper` now allows namespaces to be declared within greater scope.
  - `xjc` is now more tolerant of different XML parser implementations.
  - Ad hoc performance enhancements
  - <xs:documentation>s are now also picked up for javadoc comments for classes
- Bug fixes
  - `xjc` now rejects a `javaType` customization not enclosed by a `property` tag.
  - `xjc` now emits a warning when compiling a schema with no global element declaration.
  - The `propagateEvent` method in `ErrorHandlerAdaptor` now passes the right boolean to `handleEvent`.
  - A condition where a schema wouldn't validate even after being revised to be valid has been fixed.
  - `xjc` now handles highly cyclic complex types better.
  - `xjc` now provides an application specific error in a case where it had been throwing an `InternalError`.
  - `JAXBException.printStackTrace(PrintWriter)` now works.
  - `xjc` now generates javadoc for `xsd:attribute@use` and `xsd:attribute@default`.
  - `xjc` now honors vendor extensions in an external binding file.
  - `xjc` now processes `javadoc` customizations within a `property`.
  - A condition where `xjc` wasn't resolving a reference to a type in an included fragment has been fixed.
  - A condition where the runtime would marshal an attribute with an empty namespace prefix has been fixed.
  - A condition where the runtime wouldn't validate an instance with a derived attribute has been fixed.

- JAXB now handles the time zone offset properly.
- Validation no longer fails when an `xsd:choice` has attributes defined by an `xsd:element` type.

Changes between 1.0 FCS and 1.0.1 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the JAXB version 1.0 FCS release.

1. JAR file changes & additions:

`$JWSDP_HOME/jaxb/libs/jaxb-ri.jar`	Renamed `$JWSDP_HOME/jaxb/libs/jaxb-impl.jar`
`$JWSDP_HOME/jwsdp-shared/lib/relaxngDatatype.jar`	New JAR file *
`$JWSDP_HOME/jwsdp-shared/lib/xsdlib.jar`	New JAR file *
* The contents of the `relaxngDatatype.jar` and `xsdlib.jar` files used to be contained within the `$JWSDP_HOME/jaxb/libs/jaxb-libs.jar` file, but were broken out so they could be shared with other Java WSDP component technologies.	

2. There are several new compiler switches. Please read about them in the xjc notes.
3. Many of the JAXB runtime classes that used to be contained in the `jaxb-ri.jar` file are now generated by the binding compiler as part of the client package. You can find these classes under one of the generated packages. In the following example, the new generated runtime classes are contained in `primer.po.impl.runtime`:

```
parsing a schema...
compiling a schema...
primer\po\impl\CommentImpl.java
primer\po\impl\ItemsImpl.java
primer\po\impl\JAXBVersion.java
primer\po\impl\PurchaseOrderImpl.java
primer\po\impl\PurchaseOrderTypeImpl.java
primer\po\impl\USAddressImpl.java
primer\po\impl\runtime\Discarder.java
primer\po\impl\runtime\AbstractUnmarshallingEventHandlerImpl.java
primer\po\impl\runtime\XMLSerializable.java
primer\po\impl\runtime\AbstractGrammarInfoImpl.java
primer\po\impl\runtime\MarshallerImpl.java
```

```
primer\po\impl\runtime\ContentHandlerAdaptor.java
primer\po\impl\runtime\ValidatableObject.java
primer\po\impl\runtime\DefaultJAXBContextImpl.java
primer\po\impl\runtime\ErrorHandlerAdaptor.java
primer\po\impl\runtime\ValidatingUnmarshaller.java
primer\po\impl\runtime\GrammarInfoFacade.java
primer\po\impl\runtime\XMLSerializer.java
primer\po\impl\runtime\ValidationContext.java
primer\po\impl\runtime\Util.java
primer\po\impl\runtime\UnmarshallingEventHandler.java
primer\po\impl\runtime\UnmarshallingContext.java
primer\po\impl\runtime\UnmarshallableObject.java
primer\po\impl\runtime\NamespaceContext2.java
primer\po\impl\runtime\NamespaceContextImpl.java
primer\po\impl\runtime\MSVValidator.java
primer\po\impl\runtime\ValidatorImpl.java
primer\po\impl\runtime\UnmarshallingEventHandlerAdaptor.java
primer\po\impl\runtime\SAXMarshaller.java
primer\po\impl\runtime\UnmarshallerImpl.java
primer\po\impl\runtime\SAXUnmarshallerHandlerImpl.java
primer\po\impl\runtime\GrammarInfo.java
primer\po\impl\runtime\PrefixCallback.java
primer\po\impl\runtime\SAXUnmarshallerHandler.java
primer\po\Comment.java
primer\po\Items.java
primer\po\ObjectFactory.java
primer\po\PurchaseOrder.java
primer\po\PurchaseOrderType.java
primer\po\USAddress.java
primer\po\bgm.ser
primer\po\jaxb.properties
```

If you are compiling multiple schemas and would like to share the generated runtime classes across all of the packages, use the "`-use-runtime`" compiler switch.

4. The JAXB RI has added vendor-specific properties that allow you to control the behavior of some of the runtime classes. See the JAXB Vendor Extension documentation for additional information.
5. Additional features supported (element substitution, DOM customization, enhanced pretty printing, better I18N support).
6. The `$JWSDP_HOME/jaxb/examples` directory was renamed `$JWSDP_HOME/jaxb/samples` to follow the same directory naming conventions as the other Java WSDP component technologies. Also, all of the `SampleApp #` directories were give more descriptive names - see the Sample Apps page for more information.
7. Improved XJC Ant Task support.
8. `javax.xml.namespace.QName` has been updated. See the QName javadocs

Changes between 1.0 Beta and 1.0 FCS

The following list summarizes significant changes made to the JAXB Reference Implementation since the beta release.

1. Factory methods in schema-derived `ObjectFactory` classes were static in Beta and are now instance methods in JAXB v1.0.

   Class `ObjectFactory` is specified in section 4.2 of JAXB v1.0 specification. The rationale for this change was to enable associating implementation specific property/value pairs with object creation.

   Illustration of change:

   Object Factory Creation fragment in Beta:

   ```
   Comment comment = ObjectFactory.createComment("foo");
   ```

   Object Factory Creation in FCS:

   ```
   ObjectFactory of = new ObjectFactory();
   Comment comment = of.createComment("foo");
   ```

2. A JAXB property representing an XML Schema element reference was not bound to Java accessor methods as specified. The basetype of the JAXB property was the referenced element's Java element interface in the Beta release, when it should have been the Java type representing the XML element's datatype. The rationale for this change is that the Java method signatures should not be impacted by whether XML element content is represented as a local element declaration or an element reference.

   This change manifests itself as a compilation error when a JAXB property should be passing in an instance of a builtin Java datatype, the XML element's type, rather than its element instance. All global element's with XML complex types are not impacted by this change, only references to global elements where the XML global element's type is an XML Schema builtin datatype that is bound to a Java builtin type require any change.

   Illustration of change:

   For example, given schema fragments:

   ```
   <xs:complexType name="AComplexType"/>
   <xs:element name="GlobalElement" type="AComplexType"/>
   <xs:element name="Comment" type="xs:string"/>
   <xs:complexType name="refCommentExample">
      <xs:sequence>
   ```

```
        <xs:element ref="Comment"/>
        <xs:element ref="GlobalElement"/>
      </xs:sequence>
    </xs:complexType>
```

Beta generated accessors for JAXB properties, `Comment` and `GlobalElement`, with the base type corresponding to the Java element interface, not the Java datatype for the XML element's type.

The following methods were generated with the Beta version:

```
interface AClass {
  Comment getComment();
  void setComment(Comment value);
  GlobalElement getGlobalElement();
  void setGlobalElement(GlobalElement value);
}
```

The following methods were generated with the FCS version:

```
interface AClass {
  String getComment();
  void setComment(String value);
  AComplexType getGlobalElement();
  // Parameter "value" can be an instance of an XML element's type Or
  // an instance of an XML element's interface.
  // Thus, one can pass an instanceof GlobalElement interface OR an
  // instance of AComplexType to setGlobalElement.
  void setGlobalElement(AComplexType value);
}
```

3. The Customization: javaType `printMethod` and `parseMethod` attributes are required to be specified more frequently in the FCS release than they had to be in the Beta release. This specification change was the result of not being able to properly handle all the defaulting cases in the Beta release. See Section 6.9 "<javaType> Declaration" in JAXB v1.0 Specification for more details.
4. Additional APIs:
    - `getProperty()` / `setProperty()` added to `javax.xml.bind.Unmarshaller`
    - `getProperty()` / `setProperty()` added to `javax.xml.bind.Validator`
    - `getProperty()` / `setProperty()` added to all of the generated `ObjectFactory` classes
    - `getNode()` added to `javax.xml.bind.Marshaller`

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
Vendor Customizations

Implementation Version: 2.1.8 fcs

Marshaller Properties

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

Index of Marshaller Properties

- Namespace Prefix Mapping
- Indentation
- Character Escaping Control
- XML Declaration Control
- Jaxb Annotation Control

Namespace Prefix Mapping

Property name:	`com.sun.xml.bind.namespacePrefixMapper`
Type:	`com.sun.xml.bind.marshaller.NamespacePrefixMapper`
Default value:	null

The JAXB RI 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 `com.sun.xml.bind.marshaller.NamespacePrefixMapper`.
2. This class is then set on the marshaller via the RI specific property `com.sun.xml.bind.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 `com.sun.xml.bind.marshaller.NamespacePrefixMapper` class has the following method that you need to implement:

```
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>
     * <?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>
     * </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>
     * <?xml version="1.0"?>
     * <root xmlns:ns1="urn:foo">
     *   <ns1:child> ... </ns1:child>
     *   <ns1:child> ... </ns1:child>
     *   <ns1:child> ... </ns1:child>
     *   ...
     * </root>
     * </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>
     *   <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>
     * </pre>
     * <!-- TODO: shall we relax this constraint? -->
     *
     * @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 namespace-prefix sample application for a detailed example.

Indentation

Property name:	`com.sun.xml.bind.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 javax.xml.bind.Marshaller interface for details of this property.

Character Escaping Control

Property name:	`com.sun.xml.bind.characterEscapeHandler`
Type:	`com.sun.xml.bind.marshaller.CharacterEscapeHandler`
Default value:	null

By default, the marshaller implementation of the JAXB RI 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 com.sun.xml.bind.marshaller.CharacterEscapeHandler interface.
2. Create a new instance of it.
3. Set that instance to the Marshaller by using this property.

See the character-escape sample application for more details.

XML Declaration Control

Property name:	`com.sun.xml.bind.xmlDeclaration`
Type:	`boolean`
Default value:	true

This experimental JAXB RI 1.0.x property has been adopted as a standard in JAXB 2.0. The 2.0 RI will continue to support this property, but client code should be using the Marshaller.JAXB_FRAGMENT property instead. Please refer to the Marshaller javadoc for a complete description of the behavior.

In JAXB 2.0, calling:

```
marshaller.setProperty("com.sun.xml.bind.xmlDeclaration", true);
```

is equivalent to calling:

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

JAXB 1.0 generated code and clients will continue to work exactly the same on the JAXB 2.0 runtime as they did on the JAXB 1.0 runtime.

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

XML Preamble Control

Property name:	`com.sun.xml.bind.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 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 JAXB will not write its own XML preamble, so this property may contain custom XML preamble.

Jaxb Annotation Control

Property name:	`com.sun.xml.bind.XmlAccessorFactory`
Type:	`boolean`
Default value:	false

This property provides support for a custom com.sun.xml.bind.v2.runtime.reflect.Accessor implementation. It allows the user to control the access to class fields and properties.

In JAXB 2.1, set the property to enable:

```
marshaller.setProperty("com.sun.xml.bind.XmlAccessorFactory", true);
```

$Revision: 1.4 $
$Date: 2008/04/16 22:18:01 $

Java ™ Architecture for XML Binding
Vendor Customizations

Implementation Version: 2.1.8 fcs

Customizations

The JAXB RI provides additional customizations that are not defined by the JAXB specification. Note the following:

- These features may only be used when the JAXB XJC binding compiler is run in the `"-extension"` mode.
- All of the JAXB RI 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 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 JAXB Specification.

Index of Customizations

- Identifying customization targets by Schema Component Designator
- <xjc:superClass> - Extending a Common Super Class
- <xjc:superInterface> - Extending a Common Super Interface
- <xjc:javaType> - Enhanced <jaxb:javaType> customization
- <xjc:simple> - Experimental simpler&better binding mode
- <xjc:treatRestrictionLikeNewType> - Alternative derivation-by-restriction binding mode
- <xjc:substitutable> - Allow separate compilations to perform element substitutions

SCD Support

The JAXB RI supports the use of schema component designator as a means of specifying the customization target (of all standard JAXB 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="http://java.sun.com/xml/
ns/jaxb" version="2.1">
  <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.

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="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="2.0">

    <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`.

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="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="2.0">

    <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`.

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="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="2.0">
  .
  .
  .
  <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="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="2.0">
  <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`

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="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:extensionBindingPrefixes="xjc"
           jaxb:version="2.0">

    <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 definitoins, 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 JAXB RI without notice. Please send feedbacks on this binding to `users@jaxb.dev.java.net`

Alternative Derivation-by-restriction Binding Mode

Normally, the JAXB 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="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0"
  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()
}
```

Allow separate compilations to perform element substitutions

In an attempt to make the generated code easier to use, the JAXB 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, JAXB 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.)

<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>
```

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
DTD Support

Implementation Version: 2.1.8 fcs

DTD

The JAXB RI 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 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.

Customization

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. 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.

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $

Java ™ Architecture for XML Binding
Developing JAXB RI Extensions

Implementation Version: 2.1.8 fcs

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

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 JAXB RI 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.

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>
```

Resources

See this page for more detailed, up-to-date information.

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

$Revision: 1.1 $
$Date: 2007/12/05 00:49:15 $