

Binding between XML Schema and Java Classes

THE Java™ Architecture for XML Binding (JAXB) provides a fast and convenient way to bind between XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling XML instance documents into Java content trees, and then marshalling Java content trees back into XML instance documents. JAXB also provides a way generate XML schema from Java objects.

This chapter describes the JAXB architecture, functions, and core concepts. You should read this chapter before proceeding to Chapter 3, which provides sample code and step-by-step procedures for using JAXB.

JAXB Architecture

This section describes the components and interactions in the JAXB processing model.

Architectural Overview

Figure 2–1 shows the components that make up a JAXB implementation.

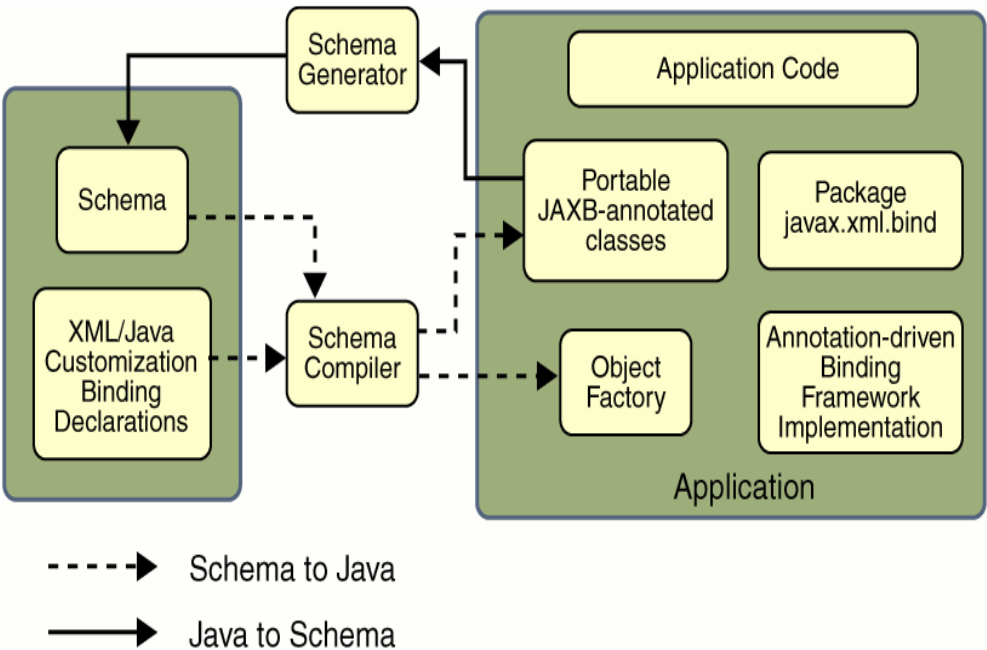


Figure 2–1 JAXB Architectural Overview

A JAXB implementation consists of the following architectural components:

- **schema compiler** : binds a source schema to a set of schema derived program elements. The binding is described by an XML-based binding language.
- **schema generator** : maps a set of existing program elements to a derived schema. The mapping is described by program annotations.
- **binding runtime framework** : provides unmarshalling (reading) and marshalling (writing) operations for accessing, manipulating and validating XML content using either schema-derived or existing program elements.

The JAXB Binding Process

Figure 2–2 shows what occurs during the JAXB binding process.

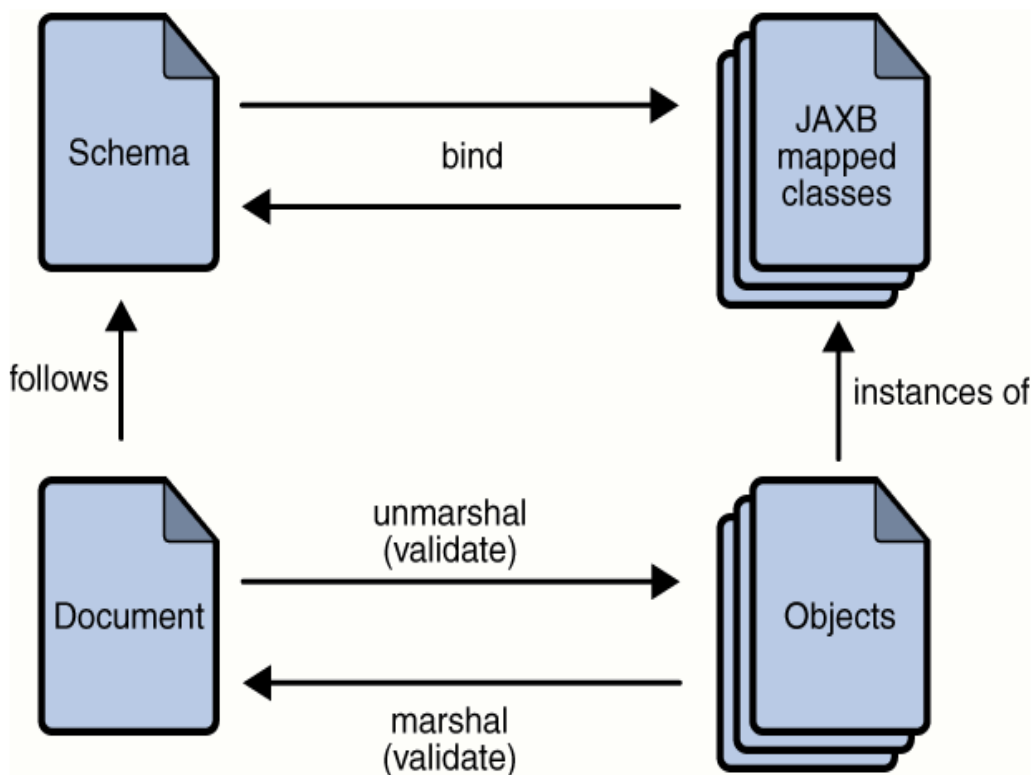


Figure 2–2 Steps in the JAXB Binding Process

Take steps from The general steps in the JAXB data binding process are:

1. **Generate classes.** An XML schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
2. **Compile classes.** All of the generated classes, source files, and application code must be compiled.
3. **Unmarshal.** XML documents written according to the constraints in the source schema are unmarshalled by the JAXB binding framework. Note that JAXB also supports unmarshalling XML data from sources other than files/documents, such as DOM nodes, string buffers, SAX Sources, and so forth.
4. **Generate content tree.** The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes; this content tree represents the structure and content of the source XML documents.

5. Validate (optional). The unmarshalling process optionally involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in Step 6, below, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
6. Process content. The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.
7. Marshal. The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.

More About Unmarshalling

Unmarshalling provides a client application the ability to convert XML data into JAXB-derived Java objects.

More About Marshalling

Marshalling provides a client application the ability to convert a JAXB-derived Java object tree back into XML data.

By default, the `Marshaller` uses UTF-8 encoding when generating XML data.

Client applications are not required to validate the Java content tree before marshalling. There is also no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data.

More About Validation

Validation is the process of verifying that an XML document meets all the constraints expressed in the schema. JAXB 1.0 provided validation at unmarshal time and also enabled on-demand validation on a JAXB content tree. JAXB 2.0 only allows validation at unmarshal and marshal time. A web service processing model is to be lax in reading in data and strict on writing it out. To meet that model, validation was added to marshal time so one could confirm that they did not invalidate the XML document when modifying the document in JAXB form.

Representing XML Content

This section describes how JAXB represents XML content as Java objects.

Java Representation of XML Schema

JAXB supports the grouping of generated classes in Java packages. A package comprises:

- A Java class name is derived from the XML element name, or specified by a binding customization.
- An `ObjectFactory` class is a factory that is used to return instances of a bound Java class.

Binding XML Schemas

This section describes the default XML-to-Java bindings used by JAXB. All of these bindings can be overridden on global or case-by-case levels by means of a custom binding declaration. See the *JAXB Specification* for complete information about the default JAXB bindings.

Simple Type Definitions

A schema component using a simple type definition typically binds to a Java property. Since there are different kinds of such schema components, the following Java property attributes (common to the schema components) include:

- Base type
- Collection type, if any
- Predicate

The rest of the Java property attributes are specified in the schema component using the `simple` type definition.

Default Data Type Bindings

Schema-to-Java

The Java language provides a richer set of data type than XML schema. Table 2–1 lists the mapping of XML data types to Java data types in JAXB.

Table 2–1 JAXB Mapping of XML Schema Built-in Data Types

XML Schema Type	Java Data Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	javax.xml.datatype.XMLGregorianCalendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	javax.xml.datatype.XMLGregorianCalendar

Table 2–1 JAXB Mapping of XML Schema Built-in Data Types (Continued)

XML Schema Type	Java Data Type
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:g	javax.xml.datatype.XMLGregorianCalendar
xsd:anySimpleType	java.lang.Object
xsd:anySimpleType	java.lang.String
xsd:duration	javax.xml.datatype.Duration
xsd:NOTATION	javax.xml.namespace.QName

JAXBElement

When XML element information can not be inferred by the derived Java representation of the XML content, a JAXBElement object is provided. This object has methods for getting and setting the object name and object value.

Java-to-Schema

Table 2–2 shows the default mapping of Java classes to XML data types.

Table 2–2 JAXB Mapping of XML Data Types to Java classes.

Java Class	XML Data Type
java.lang.String	xs:string
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.util.Calendar	xs:dateTime
java.util.Date	xs:dateTime

Table 2–2 JAXB Mapping of XML Data Types to Java classes. (Continued)

Java Class XML Data Type

javax.xml.namespace.QName	xs:QName
java.net.URI	xs:string
javax.xml.datatype.XMLGregorianCalendar	xs:anySimpleType
javax.xml.datatype.Duration	xs:duration
java.lang.Object	xs:anyType
java.awt.Image	xs:base64Binary
javax.activation.DataHandler	xs:base64Binary
javax.xml.transform.Source	xs:base64Binary
java.util.UUID	xs:string

Customizing JAXB Bindings

Schema-to-Java

Custom JAXB binding declarations also allow you to customize your generated JAXB classes beyond the XML-specific constraints in an XML schema to include Java-specific refinements such as class and package name mappings.

JAXB provides two ways to customize an XML schema:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file that is passed to the JAXB binding compiler

Code examples showing how to customize JAXB bindings are provided in Chapter 3.

Java-to-Schema

XML schema that is generated from Java objects can be customized with JAXB annotations.

Using JAXB

THIS chapter provides instructions for using the sample Java applications that are included in the `<javaee.tutorial.home>/examples/jaxb` directory. These examples demonstrate and build upon key JAXB features and concepts. It is recommended that you follow these procedures in the order presented.

After reading this chapter, you should feel comfortable enough with JAXB that you can:

- Generate JAXB Java classes from an XML schema
- Use schema-derived JAXB classes to unmarshal and marshal XML content in a Java application
- Create a Java content tree from scratch using schema-derived JAXB classes
- Validate XML content during unmarshalling and at runtime
- Customize JAXB schema-to-Java bindings

The primary goals of the Basic examples are to highlight the core set of JAXB functions using default settings and bindings. After familiarizing yourself with these core features and functions, you may wish to continue with Customizing JAXB Bindings (page 35) for instructions on using Customize examples that demonstrate how to modify the default JAXB bindings. Finally, the Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

Note: The Purchase Order schema, `po.xsd`, and the Purchase Order XML file, `po.xml`, used in the Basic and Customize samples are derived from the W3C XML Schema Part 0: Primer (<http://www.w3.org/TR/xmlschema-0/>), edited by David C. Fallside.

General Usage Instructions

This section provides general usage instructions for the examples used in this chapter, including how to build and run the applications using the Ant build tool, and provides details about the default schema-to-JAXB bindings used in these examples.

Description

This chapter describes three sets of examples:

- The Basic examples (Unmarshal Read, Modify Marshal, Unmarshal Validate, Pull Parser) demonstrate basic JAXB concepts like ummarshalling, marshalling, validating XML content, and parsing XML data.
- The Customize examples (Customize Inline, Datatype Converter, External Customize, Fix Collides) demonstrate various ways of customizing the binding of XML schemas to Java objects.
- The Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

The Basic and Customize examples are based on a *Purchase Order* scenario. With the exception of the Fix Collides example, each uses an XML document, `po.xml`, written against an XML schema, `po.xsd`.

Table 3–1 briefly describes the Basic examples.

Table 3–1 Basic JAXB Examples

Example Name	Description
Unmarshal Read Example	Demonstrates how to unmarshal an XML document into a Java content tree and access the data contained within it.
Modify Marshal Example	Demonstrates how to modify a Java content tree.
Unmarshal Validate Example	Demonstrates how to enable validation during unmarshalling.
Pull Parser Example	Demonstrates how to use the StAX pull parser to parse a portion of an XML document.

Table 3–2 briefly describes the Customize examples.

Table 3–2 Customize JAXB Examples

Example Name	Description
Customize Inline Example	Demonstrates how to customize the default JAXB bindings by using inline annotations in an XML schema.
Datatype Converter Example	Similar to the Customize Inline example, this example illustrates alternate, more terse bindings of XML <code>simpleType</code> definitions to Java datatypes.
External Customize Example	Illustrates how to use an external binding declarations file to pass binding customizations for a read-only schema to the JAXB binding compiler.
Fix Collides Example	Illustrates how to use customizations to resolve name conflicts reported by the JAXB binding compiler. You should first move <code>binding.xjb</code> , the binding file, out of the application directory to see the errors reported by the JAXB binding compiler, and then look at <code>binding.xjb</code> to see how the errors were resolved. Running <code>asant</code> alone uses the binding customizations to resolve the name conflicts while compiling the schema.

Table 3–3 briefly describes the Java-to-Schema examples.

Table 3–3 Java-toSchema JAXB Examples

Example Name	Description
j2s-create-marshal	Illustrates how to marshal and unmarshal JAXB-annotated classes to XML schema. The example also shows how to enable JAXP 1.3 validation at unmarshal time using a schema file that was generated from the JAXB mapped classes.
j2s-xmlAccessorType	Illustrates how to use the <code>@XmlAccessorType</code> and <code>@XmlType.propOrder</code> mapping annotations in Java classes to control the order in which XML content is marshalled/unmarshaled by a Java type.
j2s-xmlAdapter-field	Illustrates how to use the interface <code>XmlAdapter</code> and the annotation <code>@XmlJavaTypeAdapter</code> to provide a custom mapping of XML content into and out of a <code>HashMap</code> (field) that uses an “int” as the key and a “string” as the value.
j2s-xmlAttribute-field	Illustrates how to use the annotation <code>@XmlAttribute</code> to define a property or field to be handled as an XML attribute.
j2s-xmlRootElement	Illustrates how to use the annotation <code>@XmlRootElement</code> to define an XML element name for the XML schema type of the corresponding class.
j2s-xmlSchemaType-class	Illustrates how to use the annotation <code>@XmlSchemaType</code> to customize the mapping of a property or field to an XML built-in type.
j2s-xmlType	Illustrates how to use the annotation <code>@XmlType</code> to map a class or enum type to an XML schema type.

Each Basic and Customize example directory contains several base files:

- `po.xsd` is the XML schema you will use as input to the JAXB binding compiler, and from which schema-derived JAXB Java classes will be generated. For the Customize Inline and Datatype Converter examples, this file contains inline binding customizations. Note that the Fix Collides example uses `example.xsd` rather than `po.xsd`.
- `po.xml` is the *Purchase Order* XML file containing sample XML content, and is the file you will unmarshal into a Java content tree in each example. This file is almost exactly the same in each example, with minor content

differences to highlight different JAXB concepts. Note that the Fix Collides example uses `example.xml` rather than `po.xml`.

- `Main.java` is the main Java class for each example.
- `build.xml` is an Ant project file provided for your convenience. Use Ant to generate, compile, and run the schema-derived JAXB classes automatically. The `build.xml` file varies across the examples.
- `MyDatatypeConverter.java` in the `inline-customize` example is a Java class used to provide custom datatype conversions.
- `binding.xjb` in the External Customize and Fix Collides examples is an external binding declarations file that is passed to the JAXB binding compiler to customize the default JAXB bindings.
- `example.xsd` in the Fix Collides example is a short schema file that contains deliberate naming conflicts, to show how to resolve such conflicts with custom JAXB bindings.

Using the Examples

As with all applications that implement schema-derived JAXB classes, as described above, there are two distinct phases in using JAXB:

1. Generating and compiling JAXB Java classes from an XML source schema
2. Unmarshalling, validating, processing, and marshalling XML content

In the case of these examples, you perform these steps by using `asant` with the `build.xml` project file included in each example directory.

Configuring and Running the Samples

The `build.xml` file included in each example directory is an `asant` project file that, when run, automatically performs the following steps:

1. Updates your `CLASSPATH` to include the necessary schema-derived JAXB classes.
2. For the Basic and Customize examples, runs the JAXB binding compiler to generate JAXB Java classes from the XML source schema, `po.xsd`, and puts the classes in a package named `primer.po`. For the Java-to-Schema examples runs `schemagen`, the schema generator, to generate XML schema from the annotated Java classes.

- 3. Compiles the schema-derived JAXB classes or the annotated Java code.
- 4. Runs the `Main` class for the example.

The schema-derived JAXB classes and how they are bound to the source schema is described in About the Schema-to-Java Bindings (page 19). The methods used for building and processing the Java content tree are described in Basic Examples (page 29).

JAXB Compiler Options

The JAXB XJC schema binding compiler transforms, or binds, a source XML schema to a set of JAXB content classes in the Java programming language. The compiler, `xjc`, is provided in two flavors in the Application Server: `xjc.sh` (Solaris/Linux) and `xjc.bat` (Windows). Both `xjc.sh` and `xjc.bat` take the same command-line options. You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
xjc [-options ...] < schema >
```

The `xjc` command-line options are listed in Table 3–4.

Table 3–4 `xjc` Command-Line Options

Option or Argument	Description
<code>-nv</code>	Do not perform strict validation of the input schema(s). By default, <code>xjc</code> performs strict validation of the source schema before processing. Note that this does not mean the binding compiler will not perform any validation; it simply means that it will perform less-strict validation.
<code>-extension</code>	By default, the XJC binding compiler strictly enforces the rules outlined in the Compatibility chapter of the JAXB Specification. In the default (strict) mode, you are also limited to using only the binding customizations defined in the specification. By using the <code>-extension</code> switch, you will be allowed to use the JAXB Vendor Extensions.

Table 3–4 xjc Command-Line Options (Continued)

Option or Argument Description	
-b <i>file</i>	Specify one or more external binding files to process. (Each binding file must have its 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. In addition, the ordering of the schema files and binding files on the command line does not matter.
-d <i>dir</i>	By default, xjc will generate Java content classes in the current directory. Use this option to specify an alternate output directory. The directory must already exist; xjc will not create it for you.
-p <i>package</i>	Specify an alternate output directory. By default, the XJC binding compiler will generate the Java content classes in the current directory. The output directory must already exist; the XJC binding compiler will not create it for you.
-proxy <i>proxy</i>	Specify the HTTP/HTTPS proxy. The format is [user[:password]@]proxyHost[:proxyPort]. The old -host and -port options are still supported by the Reference Implementation for backwards compatibility, but they have been deprecated.
-classpath <i>arg</i>	Specify where to find client application class files used by the <jxb:javaType> and <xjc:superClass> customizations.
-catalog <i>file</i>	Specify catalog files to resolve external entity references. Supports TR9401, XCatalog, and OASIS XML Catalog format. For more information, please read the XML Entity and URI Resolvers document or examine the catalog-resolver sample application.
-readOnly	Force the XJC binding compiler to mark the generated Java sources read-only. By default, the XJC binding compiler does not write-protect the Java source files it generates.
-npa	Suppress 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.
-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.

Table 3–4 xjc Command-Line Options (Continued)

Option or Argument	Description
-quiet	Suppress compiler output, such as progress information and warnings.
-help	Display a brief summary of the compiler switches.
-version	Display the compiler version information.
-Xlocator	Enable source location support for generated code.
-Xsync-methods	Generate accessor methods with the <code>synchronized</code> keyword.
-mark-generated	Mark the generated code with the <code>@javax.annotation.Generated</code> annotation.

JAXB Schema Generator Options

The JAXB Schema Generator, `schemagen`, creates a schema file for each namespace referenced in your Java classes. The schema generator can be launched using the appropriate `schemagen` shell script in the `bin` directory for your platform. The schema generator processes Java source files only. If your Java sources reference other classes, those sources must be accessible from your system `CLASSPATH` environment variable or errors will occur when the schema is generated. There is no way to control the name of the generated schema files.

You can display quick usage instructions by invoking the scripts without any options, or with the `-help` switch. The syntax is as follows:

```
schemagen [-options ...] [ java_source_files ]
```

The `schemagen` command-line options are listed in Table 3–5.

Table 3–5 schemagen Command-Line Options

Option or Argument	Description
-d <i>path</i>	Specifies the location of the processor- and <code>javac</code> generated class files.

About the Schema-to-Java Bindings

When you run the JAXB binding compiler against the `po.xsd` XML schema used in the basic examples (Unmarshal Read, Modify Marshal, Unmarshal Validate), the JAXB binding compiler generates a Java package named `primer.po` containing eleven classes, making a total of twelve classes in each of the basic examples:

Table 3–6 Schema-Derived JAXB Classes in the Basic Examples

Class Description	
primer/po/ Comment.java	Public interface extending <code>javax.xml.bind.Element</code> ; binds to the global schema element named <code>comment</code> . Note that JAXB generates element interfaces for all global element declarations.
primer/po/ Items.java	Public interface that binds to the schema complexType named <code>Items</code> .
primer/po/ ObjectFactory.java	Public class extending <code>com.sun.xml.bind.DefaultJAXB-ContextImpl</code> ; used to create instances of specified interfaces. For example, the <code>ObjectFactory createComment()</code> method instantiates a <code>Comment</code> object.
primer/po/ PurchaseOrder.java	Public interface extending <code>javax.xml.bind.Element</code> , and <code>PurchaseOrderType</code> ; binds to the global schema element named <code>PurchaseOrder</code> .
primer/po/ PurchaseOrderType.java	Public interface that binds to the schema complexType named <code>PurchaseOrderType</code> .
primer/po/ USAddress.java	Public interface that binds to the schema complexType named <code>USAddress</code> .
primer/po/impl/ CommentImpl.java	Implementation of <code>Comment.java</code> .
primer/po/impl/ ItemsImpl.java	Implementation of <code>Items.java</code>
primer/po/impl/ PurchaseOrderImpl.java	Implementation of <code>PurchaseOrder.java</code>

Table 3–6 Schema-Derived JAXB Classes in the Basic Examples (Continued)

Class Description

primer/po/impl/ PurchaseOrderType- Impl.java	Implementation of PurchaseOrderType.java
primer/po/impl/ USAddressImpl.java	Implementation of USAddress.java

Note: You should never directly use the generated implementation classes—that is, `*Impl.java` in the `<packagename>/impl` directory. These classes are not directly referenceable because the class names in this directory are not standardized by the JAXB specification. The `ObjectFactory` method is the only portable means to create an instance of a schema-derived interface. There is also an `ObjectFactory.newInstance(Class JAXBInterface)` method that enables you to create instances of interfaces.

These classes and their specific bindings to the source XML schema for the basic examples are described below.

Table 3–7 Schema-to-Java Bindings for the Basic Examples

XML Schema	JAXB Binding
<code><xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"></code>	
<code><xsd:element name="purchaseOrder" type="PurchaseOrderType"/></code>	<code>PurchaseOrder.java</code>
<code><xsd:element name="comment" type="xsd:string"/></code>	<code>Comment.java</code>
<code><xsd:complexType name="PurchaseOrderType"></code>	
<code><xsd:sequence></code>	
<code><xsd:element name="shipTo" type="USAddress"/></code>	
<code><xsd:element name="billTo" type="USAddress"/></code>	
<code><xsd:element ref="comment" minOccurs="0"/></code>	
<code><xsd:element name="items" type="Items"/></code>	<code>PurchaseOrder-</code>
<code></xsd:sequence></code>	<code>Type.java</code>
<code><xsd:attribute name="orderDate" type="xsd:date"/></code>	
<code></xsd:complexType></code>	

Table 3–7 Schema-to-Java Bindings for the Basic Examples (Continued)

XML Schema	JAXB Binding
<pre><xsd:complexType name="USAddress"> <xsd:sequence> <xsd:element name="name" type="xsd:string"/> <xsd:element name="street" type="xsd:string"/> <xsd:element name="city" type="xsd:string"/> <xsd:element name="state" type="xsd:string"/> <xsd:element name="zip" type="xsd:decimal"/> </xsd:sequence> <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/> </xsd:complexType></pre>	USAddress.java
<pre><xsd:complexType name="Items"> <xsd:sequence> <xsd:element name="item" minOccurs="1" maxOc- curs="unbounded"></pre>	Items.java
<pre> <xsd:complexType> <xsd:sequence> <xsd:element name="productName" type="xsd:string"/> <xsd:element name="quantity"> <xsd:simpleType> <xsd:restriction base="xsd:positiveInteger"> <xsd:maxExclusive value="100"/> </xsd:restriction> </xsd:simpleType> </xsd:element> <xsd:element name="USPrice" type="xsd:decimal"/> <xsd:element ref="comment" minOccurs="0"/> </xsd:sequence> <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/> <xsd:attribute name="partNum" type="SKU" use="required"/> </xsd:complexType> </xsd:element> </xsd:sequence> </xsd:complexType></pre>	Items.ItemType
<pre><!-- Stock Keeping Unit, a code for identifying products --> <xsd:simpleType name="SKU"> <xsd:restriction base="xsd:string"> <xsd:pattern value="\d{3}-[A-Z]{2}"/> </xsd:restriction> </xsd:simpleType> </xsd:schema></pre>	

Schema-Derived JAXB Classes

The code for the individual classes generated by the JAXB binding compiler for the Basic examples is listed below, followed by brief explanations of its functions. The classes listed here are:

- Comment.java
- Items.java
- ObjectFactory.java
- PurchaseOrder.java
- PurchaseOrderType.java
- USAddress.java

Comment.java

In Comment.java :

- The Comment.java class is part of the primer.po package.
- Comment is a public interface that extends javax.xml.bind.Element .
- Content in instantiations of this class bind to the XML schema element named comment .
- The getValue() and setValue() methods are used to get and set strings representing XML comment elements in the Java content tree.

The Comment.java code looks like this:

```
package primer.po;

public interface Comment
extends javax.xml.bind.Element
{

String getValue();
void setValue(String value);
}
```

Items.java

In `Items.java`, below:

- The `Items.java` class is part of the `primer.po` package.
- The class provides public interfaces for `Items` and `ItemType`.
- Content in instantiations of this class bind to the XML ComplexTypes `Items` and its child element `ItemType`.
- `Item` provides the `getItem()` method.
- `ItemType` provides methods for:
 - `getPartNum();`
 - `setPartNum(String value);`
 - `getComment();`
 - `setComment(java.lang.String value);`
 - `getUSPrice();`
 - `setUSPrice(java.math.BigDecimal value);`
 - `getProductName();`
 - `setProductName(String value);`
 - `getShipDate();`
 - `setShipDate(java.util.Calendar value);`
 - `getQuantity();`
 - `setQuantity(java.math.BigInteger value);`

The `Items.java` code looks like this:

```
package primer.po;

public interface Items {
    java.util.List getItem();

    public interface ItemType {
        String getPartNum();
        void setPartNum(String value);
        java.lang.String getComment();
        void setComment(java.lang.String value);
        java.math.BigDecimal getUSPrice();
        void setUSPrice(java.math.BigDecimal value);
        String getProductName();
        void setProductName(String value);
        java.util.Calendar getShipDate();
        void setShipDate(java.util.Calendar value);
```

```
java.math.BigInteger getQuantity();
void setQuantity(java.math.BigInteger value);
}
}
```

ObjectFactory.java

In ObjectFactory.java , below:

- The ObjectFactory class is part of the primer.po package.
- ObjectFactory provides factory methods for instantiating Java interfaces representing XML content in the Java content tree.
- Method names are generated by concatenating:
 - The string constant create
 - If the Java content interface is nested within another interface, then the concatenation of all outer Java class names
 - The name of the Java content interface
 - JAXB implementation-specific code was removed in this example to make it easier to read.

For example, in this case, for the Java interface primer.po.Items.ItemType , ObjectFactory creates the method createItemsItemType() .

The ObjectFactory.java code looks like this:

```
package primer.po;

public class ObjectFactory
extends com.sun.xml.bind.DefaultJAXBContextImpl {

/**
 * Create a new ObjectFactory that can be used to create
 * new instances of schema derived classes for package:
 * primer.po
 */
public ObjectFactory() {
super(new primer.po.ObjectFactory.GrammarInfoImpl());
}

/**
 * Create an instance of the specified Java content
 * interface.
 */
public Object newInstance(Class javaContentInterface)
```



```

throws javax.xml.bind.JAXBException
{
return super.newInstance(javaContentInterface);
}

/**
 * Get the specified property. This method can only be
 * used to get provider specific properties.
 * Attempting to get an undefined property will result
 * in a PropertyException being thrown.
 */
public Object getProperty(String name)
throws javax.xml.bind.PropertyException
{
return super.getProperty(name);
}

/**
 * Set the specified property. This method can only be
 * used to set provider specific properties.
 * Attempting to set an undefined property will result
 * in a PropertyException being thrown.
 */
public void setProperty(String name, Object value)
throws javax.xml.bind.PropertyException
{
super.setProperty(name, value);
}

/**
 * Create an instance of PurchaseOrder
 */
public primer.po.PurchaseOrder createPurchaseOrder()
throws javax.xml.bind.JAXBException
{
return ((primer.po.PurchaseOrder)
newInstance((primer.po.PurchaseOrder.class)));
}

/**
 * Create an instance of ItemsItemType
 */
public primer.po.Items.ItemType createItemsItemType()
throws javax.xml.bind.JAXBException
{
return ((primer.po.Items.ItemType)
newInstance((primer.po.Items.ItemType.class)));
}

```

```
/**
 * Create an instance of USAddress
 */
public primer.po.USAddress createUSAddress()
throws javax.xml.bind.JAXBException
{
    return ((primer.po.USAddress)
    newInstance((primer.po.USAddress.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment()
throws javax.xml.bind.JAXBException
{
    return ((primer.po.Comment)
    newInstance((primer.po.Comment.class)));
}

/**
 * Create an instance of Comment
 */
public primer.po.Comment createComment(String value)
throws javax.xml.bind.JAXBException
{
    return new primer.po.impl.CommentImpl(value);
}

/**
 * Create an instance of Items
 */
public primer.po.Items createItems()
throws javax.xml.bind.JAXBException
{
    return ((primer.po.Items)
    newInstance((primer.po.Items.class)));
}

/**
 * Create an instance of PurchaseOrderType
 */
public primer.po.PurchaseOrderType
createPurchaseOrderType()
throws javax.xml.bind.JAXBException
{
```

```

return ((primer.po.PurchaseOrderType)
newInstance((primer.po.PurchaseOrderType.class)));
}
}

```

PurchaseOrder.java

In PurchaseOrder.java, below:

- The PurchaseOrder class is part of the primer.po package.
- PurchaseOrder is a public interface that extends javax.xml.bind.Element and primer.po.PurchaseOrderType.
- Content in instantiations of this class bind to the XML schema element named purchaseOrder.

The PurchaseOrder.java code looks like this:

```

package primer.po;

public interface PurchaseOrder
extends javax.xml.bind.Element, primer.po.PurchaseOrderType{
}

```

PurchaseOrderType.java

In PurchaseOrderType.java, below:

- The PurchaseOrderType class is part of the primer.po package.
- Content in instantiations of this class bind to the XML schema child element named PurchaseOrderType.
- PurchaseOrderType is a public interface that provides the following methods:
 - getItems();
 - setItems(primer.po.Items value);
 - getOrderDate();
 - setOrderDate(java.util.Calendar value);
 - getComment();
 - setComment(java.lang.String value);
 - getBillTo();
 - setBillTo(primer.po.USAddress value);
 - getShipTo();
 - setShipTo(primer.po.USAddress value);

The PurchaseOrderType.java code looks like this:

```
package primer.po;

public interface PurchaseOrderType {
    primer.po.Items getItems();
    void setItems(primer.po.Items value);
    java.util.Calendar getOrderDate();
    void setOrderDate(java.util.Calendar value);
    java.lang.String getComment();
    void setComment(java.lang.String value);
    primer.po.USAddress getBillTo();
    void setBillTo(primer.po.USAddress value);
    primer.po.USAddress getShipTo();
    void setShipTo(primer.po.USAddress value);
}
```

USAddress.java

In USAddress.java , below:

- The USAddress class is part of the primer.po package.
- Content in instantiations of this class bind to the XML schema element named USAddress .
- USAddress is a public interface that provides the following methods:
 - getState();
 - setState(String value);
 - getZip();
 - setZip(java.math.BigDecimal value);
 - getCountry();
 - setCountry(String value);
 - getCity();
 - setCity(String value);
 - getStreet();
 - setStreet(String value);
 - getName();
 - setName(String value);

The USAddress.java code looks like this:

```
package primer.po;

public interface USAddress {
    String getState();
```

```

void setState(String value);
java.math.BigDecimal getZip();
void setZip(java.math.BigDecimal value);
String getCountry();
void setCountry(String value);
String getCity();
void setCity(String value);
String getStreet();
void setStreet(String value);
String getName();
void setName(String value);
}

```

Basic Examples

This section describes the Basic examples (Unmarshal Read, Modify Marshal, Unmarshal Validate) that demonstrate how to:

- Unmarshal an XML document into a Java content tree and access the data contained within it
- Modify a Java content tree
- Use the `ObjectFactory` class to create a Java content tree from scratch and then marshal it to XML data
- Perform validation during unmarshalling
- Validate a Java content tree at runtime

Unmarshal Read Example

The purpose of the Unmarshal Read example is to demonstrate how to unmarshal an XML document into a Java content tree and access the data contained within it.

1. The `<INSTALL>/examples/jaxb/unmarshal-read/` `Main.java` class declares imports for four standard Java classes plus three JAXB binding framework classes and the `primer.po` package:

```

import java.io.FileInputStream
import java.io.IOException
import java.util.Iterator
import java.util.List
import javax.xml.bind.JAXBContext
import javax.xml.bind.JAXBException

```

```
import javax.xml.bind.Unmarshaller
import primer.po.*;
```

2. A JAXBContext instance is created for handling classes generated in primer.po .

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An Unmarshaller instance is created.

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

4. po.xml is unmarshalled into a Java content tree comprising objects generated by the JAXB binding compiler into the primer.po package.

```
PurchaseOrder po =
    (PurchaseOrder)u.unmarshal(
        new FileInputStream( "po.xml" ) );
```

5. A simple string is printed to system.out to provide a heading for the purchase order invoice.

```
System.out.println( "Ship the following items to: " );
```

6. get and display methods are used to parse XML content in preparation for output.

```
USAddress address = po.getShipTo();
displayAddress(address);
Items items = po.getItems();
displayItems(items);
```

7. Basic error handling is implemented.

```
} catch( JAXBException je ) {
    je.printStackTrace();
} catch( IOException ioe ) {
    ioe.printStackTrace();
}
```

8. The USAddress branch of the Java tree is walked, and address information is printed to system.out .

```
public static void displayAddress( USAddress address ) {
    // display the address
    System.out.println( "\t" + address.getName() );
    System.out.println( "\t" + address.getStreet() );
    System.out.println( "\t" + address.getCity() +
        ", " + address.getState() +
        " " + address.getZip() );
    System.out.println( "\t" + address.getCountry() + "\n");
}
```

9. The `Items` list branch is walked, and item information is printed to `system.out` .

```
public static void displayItems( Items items ) {  
    // the items object contains a List of  
    //primer.po.ItemType objects  
    List itemTypeList = items.getItem();
```

10.Walking of the `Items` branch is iterated until all items have been printed.

```
for(Iterator iter = itemTypeList.iterator();  
    iter.hasNext();) {  
    Items.ItemType item = (Items.ItemType)iter.next();  
    System.out.println( "\t" + item.getQuantity() +  
        " copies of \"" + item.getProductName() +  
        "\"");  
}
```

Sample Output

Running `java Main` for this example produces the following output:

```
Ship the following items to:  
Alice Smith  
123 Maple Street  
Cambridge, MA 12345  
US  
  
5 copies of "Nosferatu - Special Edition (1929)"  
3 copies of "The Mummy (1959)"  
3 copies of "Godzilla and Mothra: Battle for Earth/Godzilla  
vs. King Ghidora"
```

Modify Marshal Example

The purpose of the Modify Marshal example is to demonstrate how to modify a Java content tree.

1. The `<INSTALL >/examples/jaxb/modify-marshal/`
`Main.java` class declares imports for three standard Java classes plus four JAXB binding framework classes and `primer.po` package:

```
import java.io.FileInputStream;  
import java.io.IOException;
```

```
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import primer.po.*;
```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created, and `po.xml` is unmarshalled.

```
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po =
    (PurchaseOrder)u.unmarshal(
        new FileInputStream( "po.xml" ) );
```

4. `set` methods are used to modify information in the `address` branch of the content tree.

```
USAddress address = po.getBillTo();
address.setName( "John Bob" );
address.setStreet( "242 Main Street" );
address.setCity( "Beverly Hills" );
address.setState( "CA" );
address.setZip( new BigDecimal( "90210" ) );
```

5. A `Marshaller` instance is created, and the updated XML content is marshalled to `system.out`. The `setProperty` API is used to specify output encoding; in this case formatted (human readable) XML format.

```
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
    Boolean.TRUE);
m.marshal( po, System.out );
```

Sample Output

Running `java Main` for this example produces the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<purchaseOrder orderDate="1999-10-20-05:00">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
```



```
<city>Cambridge</city>
<state>MA</state>
<zip>12345</zip>
</shipTo>
<billTo country="US">
  <name>John Bob</name>
  <street>242 Main Street</street>
  <city>Beverly Hills</city>
  <state>CA</state>
  <zip>90210</zip>
</billTo>
<items>
  <item partNum="242-NO">
    <productName>Nosferatu - Special Edition (1929)</productName>
    <quantity>5</quantity>
    <USPrice>19.99</USPrice>
  </item>
  <item partNum="242-MU">
    <productName>The Mummy (1959)</productName>
    <quantity>3</quantity>
    <USPrice>19.98</USPrice>
  </item>
  <item partNum="242-GZ">
    <productName>
      Godzilla and Mothra: Battle for Earth/Godzilla vs. King Ghidora
    </productName>
    <quantity>3</quantity>
    <USPrice>27.95</USPrice>
  </item>
</items>
</purchaseOrder>
```

Unmarshal Validate Example

The Unmarshal Validate example demonstrates how to enable validation during unmarshalling (*Unmarshal-Time Validation*). Note that JAXB provides functions for validation during unmarshalling but not during marshalling. Validation is explained in more detail in [More About Validation](#) (page 4).

1. The `<INSTALL >/examples/jaxb/unmarshal-validate/Main.java` class declares imports for three standard Java classes plus seven JAXB binding framework classes and the `primer.po` package:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;
import javax.xml.bind.JAXBContext;
```

```

import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidationEvent;
import javax.xml.bind.util.ValidationEventCollector;
import primer.po.*;

```

2. A `JAXBContext` instance is created for handling classes generated in `primer.po`.

```
JAXBContext jc = JAXBContext.newInstance( "primer.po" );
```

3. An `Unmarshaller` instance is created.

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

4. The default JAXB `Unmarshaller` `ValidationEventHandler` is enabled to send validation warnings and errors to `system.out`. The default configuration causes the unmarshal operation to fail upon encountering the first validation error.

```
u.setValidating( true );
```

5. An attempt is made to unmarshal `po.xml` into a Java content tree. For the purposes of this example, the `po.xml` contains a deliberate error.

```

PurchaseOrder po =
    (PurchaseOrder)u.unmarshal(
        new FileInputStream("po.xml"));

```

6. The default validation event handler processes a validation error, generates output to `system.out`, and then an exception is thrown.

```

} catch( UnmarshalException ue ) {
    System.out.println( "Caught UnmarshalException" );
} catch( JAXBException je ) {
    je.printStackTrace();
} catch( IOException ioe ) {
    ioe.printStackTrace();
}

```

Sample Output

Running `java Main` for this example produces the following output:

```

DefaultValidationEventHandler: [ERROR]: "-1" does not satisfy
the "positiveInteger" type
Caught UnmarshalException

```

Customizing JAXB Bindings

The remainder of this chapter describes several examples that build on the concepts demonstrated in the basic examples.

The goal of this section is to illustrate how to customize JAXB bindings by means of custom binding declarations made in either of two ways:

- As annotations made inline in an XML schema
- As statements in an external file passed to the JAXB binding compiler

Unlike the examples in Basic Examples (page 29), which focus on the Java code in the respective `Main.java` class files, the examples here focus on customizations made to the XML schema *before* generating the schema-derived Java binding classes.

Note: Although JAXB binding customizations must currently be made by hand, it is envisioned that a tool/wizard may eventually be written by Sun or a third party to make this process more automatic and easier in general. One of the goals of the JAXB technology is to standardize the format of binding declarations, thereby making it possible to create customization tools and to provide a standard interchange format between JAXB implementations.

This section just begins to scratch the surface of customizations you can make to JAXB bindings and validation methods. For more information, please refer to the *JAXB Specification*(<http://java.sun.com/xml/downloads/jaxb.html>).

Why Customize?

In most cases, the default bindings generated by the JAXB binding compiler will be sufficient to meet your needs. There are cases, however, in which you may want to modify the default bindings. Some of these include:

- Creating API documentation for the schema-derived JAXB packages, classes, methods and constants; by adding custom Javadoc tool annotations to your schemas, you can explain concepts, guidelines, and rules specific to your implementation.
- Providing semantically meaningful customized names for cases that the default XML name-to-Java identifier mapping cannot handle automatically; for example:

- To resolve name collisions (as described in Appendix C.2.1 of the *JAXB Specification*). Note that the JAXB binding compiler detects and reports all name conflicts.
- To provide names for typesafe enumeration constants that are not legal Java identifiers; for example, enumeration over integer values.
- To provide better names for the Java representation of unnamed model groups when they are bound to a Java property or class.
- To provide more meaningful package names than can be derived by default from the target namespace URI.
- Overriding default bindings; for example:
 - Specify that a model group should be bound to a class rather than a list.
 - Specify that a fixed attribute can be bound to a Java constant.
 - Override the specified default binding of XML Schema built-in datatypes to Java datatypes. In some cases, you might want to introduce an alternative Java class that can represent additional characteristics of the built-in XML Schema datatype.

Customization Overview

This section explains some core JAXB customization concepts:

- Inline and External Customizations
- Scope, Inheritance, and Precedence
- Customization Syntax
- Customization Namespace Prefix

Inline and External Customizations

Customizations to the default JAXB bindings are made in the form of *binding declarations* passed to the JAXB binding compiler. These binding declarations can be made in either of two ways:

- As inline annotations in a source XML schema
- As declarations in an external binding customizations file

For some people, using inline customizations is easier because you can see your customizations in the context of the schema to which they apply. Conversely, using an external binding customization file enables you to customize JAXB

bindings without having to modify the source schema, and enables you to easily apply customizations to several schema files at once.

Note: You can combine the two types of customizations—for example, you could include a reference to an external binding customizations file in an inline annotation—but you cannot declare both an inline and external customization on the same schema element.

Each of these types of customization is described in more detail below.

Inline Customizations

Customizations to JAXB bindings made by means of inline *binding declarations* in an XML schema file take the form of `<xsd:appinfo>` elements embedded in schema `<xsd:annotation>` elements (`xsd:` is the XML schema namespace prefix, as defined in W3C *XML Schema Part 1: Structures*). The general form for inline customizations is shown below.

```
<xs:annotation>
  <xs:appinfo>
    .
    .
    binding declarations
    .
    .
  </xs:appinfo>
</xs:annotation>
```

Customizations are applied at the location at which they are declared in the schema. For example, a declaration at the level of a particular element would apply to that element only. Note that the XMLSchema namespace prefix must be used with the `<annotation>` and `<appinfo>` declaration tags. In the example above, `xs:` is used as the namespace prefix, so the declarations are tagged `<xs:annotation>` and `<xs:appinfo>` .

External Binding Customization Files

Customizations to JAXB bindings made by means of an external file containing binding declarations take the general form shown below.

```
<jxb:bindings schemaLocation = "xs:anyURI">
<jxb:bindings node = "xs:string">*
<binding declaration>
</jxb:bindings>
```

- `schemaLocation` is a URI reference to the remote schema
- `node` is an XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated.

For example, the first `schemaLocation` /`node` declaration in a JAXB binding declarations file specifies the schema name and the root schema node:

```
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
```

A subsequent `schemaLocation/node` declaration, say for a `simpleType` element named `ZipCodeType` in the above schema, would take the form:

```
<jxb:bindings node="//xs:simpleType[@name='ZipCodeType']">
```

Binding Customization File Format

Binding customization files should be straight ASCII text. The name or extension does not matter, although a typical extension, used in this chapter, is `.xjb`.

Passing Customization Files to the JAXB Binding Compiler

Customization files containing binding declarations are passed to the JAXB Binding compiler, `xjc`, using the following syntax:

```
xjc -b <file> <schema>
```

where `<file>` is the name of binding customization file, and `<schema>` is the name of the schema(s) you want to pass to the binding compiler.

You can have a single binding file that contains customizations for multiple schemas, or you can break the customizations into multiple bindings files; for example:

```
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
```

Note that the ordering of schema files and binding files on the command line does not matter, although each binding customization file must be preceded by its own `-b` switch on the command line.

For more information about `xjc` compiler options in general, see JAXB Compiler Options (page 16).

Restrictions for External Binding Customizations

There are several rules that apply to binding declarations made in an external binding customization file that do not apply to similar declarations made inline in a source schema:

- The binding customization file must begin with the `jxb:bindings` `version` attribute, plus attributes for the JAXB and XMLSchema namespaces:


```
<jxb:bindings version="1.0"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```
- The remote schema to which the binding declaration applies must be identified explicitly in XPath notation by means of `jxb:bindings` `schemaLocation` and `node` attributes:
 - `schemaLocation` – URI reference to the remote schema
 - `node` – XPath 1.0 expression that identifies the schema node within `schemaLocation` to which the given binding declaration is associated; in the case of the initial `jxb:bindings` declaration in the binding customization file, this node is typically `"/xs:schema"`

For information about XPath syntax, see *XML Path Language*, James Clark and Steve DeRose, eds., W3C, 16 November 1999. Available at <http://www.w3.org/TR/1999/REC-xpath-19991116>.

- Similarly, individual nodes within the schema to which customizations are to be applied must be specified using XPath notation; for example:

<jxb:bindings node="//xs:complexType[@name='USAddress']">

In such cases, the customization is applied to the node by the binding compiler as if the declaration was embedded inline in the node's <xs:appinfo> element.

To summarize these rules, the external binding element <jxb:bindings> is only recognized for processing by a JAXB binding compiler in three cases:

- When its parent is an <xs:appinfo> element
- When it is an ancestor of another <jxb:bindings> element
- When it is root element of a document—an XML document that has a <jxb:bindings> element as its root is referred to as an external binding declaration file

Scope, Inheritance, and Precedence

Default JAXB bindings can be customized or overridden at four different levels, or *scopes*, as described in Table 3–7.

Figure 3–1 illustrates the inheritance and precedence of customization declarations. Specifically, declarations towards the top of the pyramid inherit and supersede declarations below them. For example, Component declarations inherit from and supersede Definition declarations; Definition declarations inherit and supersede Schema declarations; and Schema declarations inherit and supersede Global declarations.

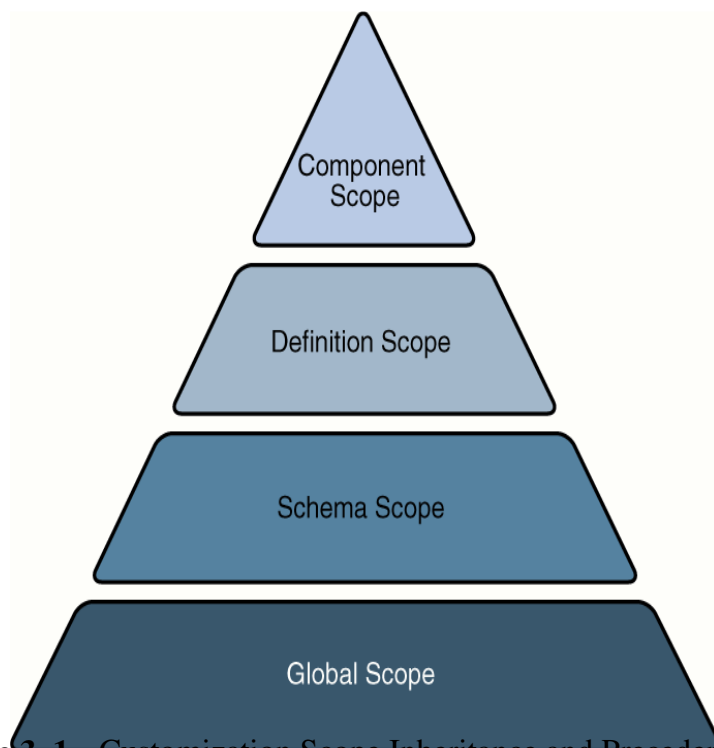


Figure 3-1 Customization Scope Inheritance and Precedence

Customization Syntax

The syntax for the four types of JAXB binding declarations, as well as the syntax for the XML-to-Java datatype binding declarations and the customization namespace prefix are described below.

- Global Binding Declarations
- Schema Binding Declarations
- Class Binding Declarations
- Property Binding Declarations
- `<javaxType>` Binding Declarations
- Typesafe Enumeration Binding Declarations
- `<javadoc>` Binding Declarations
- Customization Namespace Prefix

Global Binding Declarations

Global scope customizations are declared with `<globalBindings>`. The syntax for global scope customizations is as follows:

```
<globalBindings>
  [ collectionType = "collectionType" ]
  [ fixedAttributeAsConstantProperty= "true" | "false" | "1" | "0" ]
  [ generateIsSetMethod= "true" | "false" | "1" | "0" ]
  [ enableFailFastCheck = "true" | "false" | "1" | "0" ]
  [ choiceContentProperty = "true" | "false" | "1" | "0" ]
  [ underscoreBinding = "asWordSeparator" | "asCharInWord" ]
  [ typesafeEnumBase = "typesafeEnumBase" ]
  [ typesafeEnumMemberName = "generateName" | "generateError" ]
  [ enableJavaNamingConventions = "true" | "false" | "1" | "0" ]
  [ bindingStyle = "elementBinding" | "modelGroupBinding" ]
  [ <javaType> ... </javaType> ]*
</globalBindings>
```

- `collectionType` can be either `indexed` or any fully qualified class name that implements `java.util.List`.
- `fixedAttributeAsConstantProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `generateIsSetMethod` can be either `true`, `false`, `1`, or `0`. The default value is `false`.
- `enableFailFastCheck` can be either `true`, `false`, `1`, or `0`. If `enableFailFastCheck` is `true` or `1` and the JAXB implementation supports this optional checking, type constraint checking is performed when setting a property. The default value is `false`. Please note that the JAXB implementation does not support failfast validation.
- `choiceContentProperty` can be either `true`, `false`, `1`, or `0`. The default value is `false`. `choiceContentProperty` is not relevant when the `bindingStyle` is `elementBinding`. Therefore, if `bindingStyle` is specified as `elementBinding`, then the `choiceContentProperty` must result in an invalid customization.
- `underscoreBinding` can be either `asWordSeparator` or `asCharInWord`. The default value is `asWordSeparator`.
- `enableJavaNamingConventions` can be either `true`, `false`, `1`, or `0`. The default value is `true`.
- `typesafeEnumBase` can be a list of QNames, each of which must resolve to a simple type definition. The default value is `xs:NCName`. See `Typesafe`

Enumeration Binding Declarations (page 47) for information about localized mapping of `simpleType` definitions to Java `typesafe enum` classes.

- `typesafeEnumMemberName` can be either `generateError` or `generateName`. The default value is `generateError`.
- `bindingStyle` can be either `elementBinding`, or `modelGroupBinding`. The default value is `elementBinding`.
- `<javaType>` can be zero or more `javaType` binding declarations. See `<javaType> Binding Declarations` (page 45) for more information.

`<globalBindings>` declarations are only valid in the `annotation` element of the top-level `schema` element. There can only be a single instance of a `<globalBindings>` declaration in any given schema or binding declarations file. If one source schema includes or imports a second source schema, the `<globalBindings>` declaration must be declared in the first source schema.

Schema Binding Declarations

Schema scope customizations are declared with `<schemaBindings>`. The syntax for schema scope customizations is:

```
<schemaBindings>
  [ <package>      package      </package> ]
  [ <nameXmlTransform>      ...      </nameXmlTransform> ]*
</schemaBindings>

<package [ name = "      packageName      " ]
  [ <javadoc>      ...      </javadoc> ]
</package>

<nameXmlTransform>
  [ <typeName [ suffix="      suffix      " ]
  [ prefix="      prefix      " ] /> ]
  [ <elementName [ suffix="      suffix      " ]
  [ prefix="      prefix      " ] /> ]
  [ <modelGroupName [ suffix="      suffix      " ]
  [ prefix="      prefix      " ] /> ]
  [ <anonymousTypeName [ suffix="      suffix      " ]
  [ prefix="      prefix      " ] /> ]
</nameXmlTransform>
```

As shown above, `<schemaBinding>` declarations include two subcomponents:

- `<package>...</package>` specifies the name of the package and, if desired, the location of the API documentation for the schema-derived classes.

- `<nameXmlTransform>...</nameXmlTransform>` specifies customizations to be applied.

Class Binding Declarations

The `<class>` binding declaration enables you to customize the binding of a schema element to a Java content interface or a Java `Element` interface. `<class>` declarations can be used to customize:

- A name for a schema-derived Java interface
- An implementation class for a schema-derived Java content interface.

The syntax for `<class>` customizations is:

```
<class [ name = "      className      "]
[ implClass= "      implClass      "] >
[ <javadoc> ... </javadoc> ]
</class>
```

- `name` is the name of the derived Java interface. It must be a legal Java interface name and must not contain a package prefix. The package prefix is inherited from the current value of `package`.
- `implClass` is the name of the implementation class for `className` and must include the complete package name.
- The `<javadoc>` element specifies the Javadoc tool annotations for the schema-derived Java interface. The string entered here must use `<CDATA>` or `<![CDATA[` to escape embedded HTML tags.

Property Binding Declarations

The `<property>` binding declaration enables you to customize the binding of an XML schema element to its Java representation as a property. The scope of customization can either be at the definition level or component level depending upon where the `<property>` binding declaration is specified.

The syntax for `<property>` customizations is:

```
<property[ name = "propertyName"]
[ collectionType = "propertyCollectionType" ]
[ fixedAttributeAsConstantProperty = "true" | "false" | "1" | "0" ]
[ generateIsSetMethod = "true" | "false" | "1" | "0" ]
[ enableFailFastCheck = "true" | "false" | "1" | "0" ]
[ <baseType> ... </baseType> ]
[ <javadoc> ... </javadoc> ]
</property>
```

```

<baseType>
  <javaType> ... </javaType>
</baseType>

```

- `name` defines the customization value `propertyName` ; it must be a legal Java identifier.
- `collectionType` defines the customization value `propertyCollectionType` , which is the collection type for the property. `propertyCollectionType` if specified, can be either `indexed` or any fully-qualified class name that implements `java.util.List` .
- `fixedAttributeAsConstantProperty` defines the customization value `fixedAttributeAsConstantProperty` . The value can be either `true` , `false` , 1 , or 0 .
- `generateIsSetMethod` defines the customization value of `generateIsSetMethod` . The value can be either `true` , `false` , 1 , or 0 .
- `enableFailFastCheck` defines the customization value `enableFailFastCheck` . The value can be either `true` , `false` , 1 , or 0 . Please note that the JAXB implementation does not support failfast validation.
- `<javadoc>` customizes the Javadoc tool annotations for the property's get-ter method.

<javaType> Binding Declarations

The `<javaType>` declaration provides a way to customize the translation of XML datatypes to and from Java datatypes. XML provides more datatypes than Java, and so the `<javaType>` declaration lets you specify custom datatype bindings when the default JAXB binding cannot sufficiently represent your schema.

The target Java datatype can be a Java built-in datatype or an application-specific Java datatype. If an application-specific datatype is used as the target, your implementation must also provide parse and print methods for unmarshalling and marshalling data. To this end, the JAXB specification supports a `parseMethod` and `printMethod` :

- The `parseMethod` is called during unmarshalling to convert a string from the input document into a value of the target Java datatype.
- The `printMethod` is called during marshalling to convert a value of the target type into a lexical representation.

If you prefer to define your own datatype conversions, JAXB defines a static class, `DatatypeConverter`, to assist in the parsing and printing of valid lexical representations of the XML Schema built-in datatypes.

The syntax for the `<javaType>` customization is:

```
<javaType name= "      javaType      "
[ xmlType= "      xmlType      " ]
[ hasNsContext = "true" | "false" ]
[ parseMethod= "      parseMethod      " ]
[ printMethod= "      printMethod      " ]>
```

- `name` is the Java datatype to which `xmlType` is to be bound.
- `xmlType` is the name of the XML Schema datatype to which `javaType` is to bound; this attribute is required when the parent of the `<javaType>` declaration is `<globalBindings>`.
- `parseMethod` is the name of the parse method to be called during unmarshalling.
- `printMethod` is the name of the print method to be called during marshalling.
- `hasNsContext` allows a namespace context to be specified as a second parameter to a print or a parse method; can be either `true`, `false`, 1, or 0. By default, this attribute is `false`, and in most cases you will not need to change it.

The `<javaType>` declaration can be used in:

- A `<globalBindings>` declaration
- An annotation element for simple type definitions, `GlobalBindings`, and `<basetype>` declarations.
- A `<property>` declaration.

See `MyDatatypeConverter Class` (page 54) for an example of how `<javaType>` declarations and the `DatatypeConverterInterface` interface are implemented in a custom datatype converter class.

Typesafe Enumeration Binding Declarations

The typesafe enumeration declarations provide a localized way to map XML `simpleType` elements to Java `typesafe enum` classes. There are two types of typesafe enumeration declarations you can make:

- `<typesafeEnumClass>` lets you map an entire `simpleType` class to `typesafe enum` classes.
- `<typesafeEnumMember>` lets you map just selected members of a `simpleType` class to `typesafe enum` classes.

In both cases, there are two primary limitations on this type of customization:

- Only `simpleType` definitions with enumeration facets can be customized using this binding declaration.
- This customization only applies to a single `simpleType` definition at a time. To map sets of similar `simpleType` definitions on a global level, use the `typesafeEnumBase` attribute in a `<globalBindings>` declaration, as described Global Binding Declarations (page 42).

The syntax for the `<typesafeEnumClass>` customization is:

```
<typesafeEnumClass[ name = "          enumClassName          " ]
  [ <typesafeEnumMember> ... </typesafeEnumMember> ]*
  [ <javadoc>          enumClassJavadoc          </javadoc> ]
</typesafeEnumClass>
```

- `name` must be a legal Java Identifier, and must not have a package prefix.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration class.
- You can have zero or more `<typesafeEnumMember>` declarations embedded in a `<typesafeEnumClass>` declaration.

The syntax for the `<typesafeEnumMember>` customization is:

```
<typesafeEnumMember name = "          enumMemberName          ">
  [ value = "          enumMemberValue          "]
  [ <javadoc>          enumMemberJavadoc          </javadoc> ]
</typesafeEnumMember>
```

- `name` must always be specified and must be a legal Java identifier.
- `value` must be the enumeration value specified in the source schema.
- `<javadoc>` customizes the Javadoc tool annotations for the enumeration constant.

For inline annotations, the `<typesafeEnumClass>` declaration must be specified in the annotation element of the `<simpleType>` element. The `<typesafeEnumMember>` must be specified in the annotation element of the enumeration member. This allows the enumeration member to be customized independently from the enumeration class.

For information about typesafe enum design patterns, see the sample chapter of Joshua Bloch's *Effective Java Programming* on the Java Developer Connection.

<javadoc> Binding Declarations

The `<javadoc>` declaration lets you add custom Javadoc tool annotations to schema-derived JAXB packages, classes, interfaces, methods, and fields. Note that `<javadoc>` declarations cannot be applied globally—that is, they are only valid as a sub-elements of other binding customizations.

The syntax for the `<javadoc>` customization is:

```
<javadoc>
  Contents in <b>Javadoc<b> format.
</javadoc>
```

or

```
<javadoc>
  <![CDATA[
    Contents in <b>Javadoc<b> format
  ]]>
</javadoc>
```

Note that documentation strings in `<javadoc>` declarations applied at the package level must contain `<body>` open and close tags; for example:

```
<jxb:package name="primer.myPo">
  <jxb:javadoc><![CDATA[<body>Package level documentation
for generated package primer.myPo.</body>]]>
</jxb:javadoc>
</jxb:package>
```

Customization Namespace Prefix

All standard JAXB binding declarations must be preceded by a namespace prefix that maps to the JAXB namespace URI (<http://java.sun.com/xml/ns/jaxb>). For example, in this sample, `jxb:` is used. To this end, any schema you want to

customize with standard JAXB binding declarations *must* include the JAXB namespace declaration and JAXB version number at the top of the schema file. For example, in `po.xsd` for the Customize Inline example, the namespace declaration is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
jxb:version="1.0">
```

A binding declaration with the `jxb` namespace prefix would then take the form:

```
<xsd:annotation>
<xsd:appinfo>
<jxb:globalBindings                binding declarations        />
<jxb:schemaBindings>
.
.
binding declarations
.
.
</jxb:schemaBindings>
</xsd:appinfo>
</xsd:annotation>
```

Note that in this example, the `globalBindings` and `schemaBindings` declarations are used to specify, respectively, global scope and schema scope customizations. These customization scopes are described in more detail in Scope, Inheritance, and Precedence (page 40).

Customize Inline Example

The Customize Inline example illustrates some basic customizations made by means of inline annotations to an XML schema named `po.xsd`. In addition, this example implements a custom datatype converter class, `MyDatatypeConverter.java`, which illustrates print and parse methods in the `<javaxType>` customization for handling custom datatype conversions.

To summarize this example:

1. `po.xsd` is an XML schema containing inline binding customizations.
2. `MyDatatypeConverter.java` is a Java class file that implements print and parse methods specified by `<javaxType>` customizations in `po.xsd`.

- 3. `Main.java` is the primary class file in the Customize Inline example, which uses the schema-derived classes generated by the JAXB compiler.

Key customizations in this sample, and the custom `MyDatatypeConverter.java` class, are described in more detail below.

Customized Schema

The customized schema used in the Customize Inline example is in the file `<JAVA_HOME>/jaxb/samples/inline-customize/po.xsd`. The customizations are in the `<xsd:annotation>` tags.

Global Binding Declarations

The code below shows the `globalBindings` declarations in `po.xsd`:

```
<jxb:globalBindings
  fixedAttributeAsConstantProperty="true"
  collectionType="java.util.Vector"
  typesafeEnumBase="xsd:NCName"
  choiceContentProperty="false"
  typesafeEnumMemberName="generateError"
  bindingStyle="elementBinding"
  enableFailFastCheck="false"
  generateIsSetMethod="false"
  underscoreBinding="asCharInWord"/>
```

In this example, all values are set to the defaults except for `collectionType`.

- Setting `collectionType` to `java.util.Vector` specifies that all lists in the generated implementation classes should be represented internally as vectors. Note that the class name you specify for `collectionType` must implement `java.util.List` and be callable by `newInstance`.
- Setting `fixedAttributeAsConstantProperty` to `true` indicates that all fixed attributes should be bound to Java constants. By default, fixed attributes are just mapped to either simple or collection property, which ever is more appropriate.
- Please note that the JAXB implementation does not support the `enable-FailFastCheck` attribute.
- If `typesafeEnumBase` to `xsd:string` it would be a global way to specify that all simple type definitions deriving directly or indirectly from

`xsd:string` and having enumeration facets should be bound by default to a `typesafe enum`. If `typesafeEnumBase` is set to an empty string, `""`, no simple type definitions would ever be bound to a `typesafe enum` class by default. The value of `typesafeEnumBase` can be any atomic simple type definition except `xsd:boolean` and both binary types.

Note: Using typesafe enums enables you to map schema enumeration values to Java constants, which in turn makes it possible to do compares on Java constants rather than string values.

Schema Binding Declarations

The following code shows the schema binding declarations in `po.xsd` :

```
<jxb:schemaBindings>
<jxb:package name="primer.myPo">
<jxb:javadoc>
<![CDATA[<body> Package level documentation for
generated package primer.myPo.</body>]]>
</jxb:javadoc>
</jxb:package>
<jxb:nameXmlTransform>
<jxb:elementName suffix="Element"/>
</jxb:nameXmlTransform>
</jxb:schemaBindings>
```

- `<jxb:package name="primer.myPo"/>` specifies the `primer.myPo` as the package in which the schema-derived classes should be generated.
- `<jxb:nameXmlTransform>` specifies that all generated Java element interfaces should have `Element` appended to the generated names by default. For example, when the JAXB compiler is run against this schema, the element interfaces `CommentElement` and `PurchaseOrderElement` will be generated. By contrast, without this customization, the default binding would instead generate `Comment` and `PurchaseOrder`.

This customization is useful if a schema uses the same name in different symbol spaces; for example, in global element and type definitions. In such cases, this customization enables you to resolve the collision with one declaration rather than having to individually resolve each collision with a separate binding declaration.

- `<jxb:javadoc>` specifies customized Javadoc tool annotations for the `primer.myPo` package. Note that, unlike the `<javadoc>` declarations at the class level, below, the opening and closing `<body>` tags must be included when the `<javadoc>` declaration is made at the package level.

Class Binding Declarations

The following code shows the class binding declarations in `po.xsd` :

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:annotation>
    <xsd:appinfo>
      <jxb:class name="POType">
        <jxb:javadoc>
          A <b>Purchase Order</b> consists of
          addresses and items.
        </jxb:javadoc>
      </jxb:class>
    </xsd:appinfo>
  </xsd:annotation>
  .
  .
  .
</xsd:complexType>
```

The Javadoc tool annotations for the schema-derived `POType` class will contain the description "A **Purchase Order** consists of addresses and items." The `<` is used to escape the opening bracket on the `` HTML tags.

Note: When a `<class>` customization is specified in the `appinfo` element of a `complexType` definition, as it is here, the `complexType` definition is bound to a Java content interface.

Later in `po.xsd` , another `<javadoc>` customization is declared at this class level, but this time the HTML string is escaped with `CDATA`:

```
<xsd:annotation>
  <xsd:appinfo>
    <jxb:class>
      <jxb:javadoc>
        <![CDATA[ First line of documentation for a
        <b>USAddress</b>.]>
      </jxb:javadoc>
    </xsd:appinfo>
  </xsd:annotation>
```

```
</jxb:javadoc>
</jxb:class>
</xsd:appinfo>
</xsd:annotation>
```

Note: If you want to include HTML markup tags in a `<jaxb:javadoc>` customization, you must enclose the data within a CDATA section or escape all left angle brackets using `<`. See *XML 1.0 2nd Edition* for more information (<http://www.w3.org/TR/2000/REC-xml-20001006#sec-cdata-sect>).

Property Binding Declarations

Of particular interest here is the `generateIsSetMethod` customization, which causes two additional property methods, `isSetQuantity` and `unsetQuantity`, to be generated. These methods enable a client application to distinguish between schema default values and values occurring explicitly within an instance document.

For example, in `po.xsd` :

```
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="1"
      maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity" default="10">
            <xsd:annotation>
              <xsd:appinfo>
                <jxb:property generateIsSetMethod="true"/>
              </xsd:appinfo>
            </xsd:annotation>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The `@generateIsSetMethod` applies to the `quantity` element, which is bound to a property within the `Items.ItemType` interface. `unsetQuantity` and `isSetQuantity` methods are generated in the `Items.ItemType` interface.

MyDatatypeConverter Class

The `<INSTALL>/examples/jaxb/inline-customize/MyDatatypeConverter` class, shown below, provides a way to customize the translation of XML datatypes to and from Java datatypes by means of a `<javaType>` customization.

```
package primer;
import java.math.BigInteger;
import javax.xml.bind.DatatypeConverter;

public class MyDatatypeConverter {

    public static short parseIntegerToShort(String value) {
        BigInteger result =
            DatatypeConverter.parseInteger(value);
        return (short)(result.intValue());
    }

    public static String printShortToInteger(short value) {
        BigInteger result = BigInteger.valueOf(value);
        return DatatypeConverter.printInteger(result);
    }

    public static int parseIntegerToInt(String value) {
        BigInteger result =
            DatatypeConverter.parseInteger(value);
        return result.intValue();
    }

    public static String printIntToInteger(int value) {
        BigInteger result = BigInteger.valueOf(value);
        return DatatypeConverter.printInteger(result);
    }
};
```

The following code shows how the `MyDatatypeConverter` class is referenced in a `<javaType>` declaration in `po.xsd` :

```
<xsd:simpleType name="ZipCodeType">
  <xsd:annotation>
    <xsd:appinfo>
      <jxb:javaType name="int"
        parseMethod="primer.MyDatatypeConverter.parseIntegerToInt"
        printMethod="primer.MyDatatypeConverter.printIntTo Integer" />
    </xsd:appinfo>
  </xsd:annotation>
```

```

<xsd:restriction base="xsd:integer">
<xsd:minInclusive value="10000"/>
<xsd:maxInclusive value="99999"/>
</xsd:restriction>
</xsd:simpleType>

```

In this example, the `jxb:javaType` binding declaration overrides the default JAXB binding of this type to `java.math.BigInteger`. For the purposes of the Customize Inline example, the restrictions on `ZipCodeType`—specifically that legal US ZIP codes are limited to five digits—make it so all valid values can easily fit within the Java primitive datatype `int`. Note also that, because `<jxb:javaType name="int"/>` is declared within `ZipCodeType`, the customization applies to all JAXB properties that reference this `simpleType` definition, including the `getZip` and `setZip` methods.

Datatype Converter Example

The Datatype Converter example is very similar to the Customize Inline example. As with the Customize Inline example, the customizations in the Datatype Converter example are made by using inline binding declarations in the XML schema for the application, `po.xsd`.

The global, schema, and package, and most of the class customizations for the Customize Inline and Datatype Converter examples are identical. Where the Datatype Converter example differs from the Customize Inline example is in the `parseMethod` and `printMethod` used for converting XML data to the Java `int` datatype.

Specifically, rather than using methods in the custom `MyDataTypeConverter` class to perform these datatype conversions, the Datatype Converter example uses the built-in methods provided by `javax.xml.bind.DatatypeConverter`:

```

<xsd:simpleType name="ZipCodeType">
<xsd:annotation>
<xsd:appinfo>
<jxb:javaType name="int"
parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
</xsd:appinfo>
</xsd:annotation>
<xsd:restriction base="xsd:integer">

```

```
<xsd:minInclusive value="10000"/>
<xsd:maxInclusive value="99999"/>
</xsd:restriction>
</xsd:simpleType>
```

External Customize Example

The External Customize example is identical to the Datatype Converter example, except that the binding declarations in the External Customize example are made by means of an external binding declarations file rather than inline in the source XML schema.

The binding customization file used in the External Customize example is

```
<INSTALL >/examples/jaxb/external-customize/binding.xjb .
```

This section compares the customization declarations in `bindings.xjb` with the analogous declarations used in the XML schema, `po.xsd`, in the Datatype Converter example. The two sets of declarations achieve precisely the same results.

- JAXB Version, Namespace, and Schema Attributes
- Global and Schema Binding Declarations
- Class Declarations

JAXB Version, Namespace, and Schema Attributes

All JAXB binding declarations files must begin with:

- JAXB version number
- Namespace declarations
- Schema name and node

The version, namespace, and schema declarations in `bindings.xjb` are as follows:

```
<jxb:bindings version="1.0"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">
.
<binding_declarations>
```



```

</jxb:bindings>
<!-- schemaLocation="po.xsd" node="/xs:schema" -->
</jxb:bindings>

```

JAXB Version Number

An XML file with a root element of `<jxb:bindings>` is considered an external binding file. The root element must specify the JAXB version attribute with which its binding declarations must comply; specifically the root `<jxb:bindings>` element must contain either a `<jxb:version>` declaration or a `version` attribute. By contrast, when making binding declarations inline, the JAXB version number is made as attribute of the `<xsd:schema>` declaration:

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
jxb:version="1.0">

```

Namespace Declarations

As shown in JAXB Version, Namespace, and Schema Attributes (page 56), the namespace declarations in the external binding declarations file include both the JAXB namespace and the XMLSchema namespace. Note that the prefixes used in this example could in fact be anything you want; the important thing is to consistently use whatever prefixes you define here in subsequent declarations in the file.

Schema Name and Schema Node

The fourth line of the code in JAXB Version, Namespace, and Schema Attributes (page 56) specifies the name of the schema to which this binding declarations file will apply, and the schema node at which the customizations will first take effect. Subsequent binding declarations in this file will reference specific nodes within the schema, but this first declaration should encompass the schema as a whole; for example, in `bindings.xjb`:

```

<jxb:bindings schemaLocation="po.xsd" node="/xs:schema">

```

Global and Schema Binding Declarations

The global schema binding declarations in `bindings.xjb` are the same as those in `po.xsd` for the Datatype Converter example. The only difference is that because the declarations in `po.xsd` are made inline, you need to embed them in

<xs:appinfo> elements, which are in turn embedded in <xs:annotation> elements. Embedding declarations in this way is unnecessary in the external bindings file.

```
<jxb:globalBindings
  fixedAttributeAsConstantProperty="true"
  collectionType="java.util.Vector"
  typesafeEnumBase="xs:NCName"
  choiceContentProperty="false"
  typesafeEnumMemberName="generateError"
  bindingStyle="elementBinding"
  enableFailFastCheck="false"
  generateIsSetMethod="false"
  underscoreBinding="asCharInWord"/>
<jxb:schemaBindings>
<jxb:package name="primer.myPo">
<jxb:javadoc><![CDATA[<body>Package level
documentation for generated package primer.myPo.</body>]]>
</jxb:javadoc>
</jxb:package>
<jxb:nameXmlTransform>
<jxb:elementName suffix="Element"/>
</jxb:nameXmlTransform>
</jxb:schemaBindings>
```

By comparison, the syntax used in po.xsd for the Datatype Converter example is:

```
<xs:annotation>
<xs:appinfo>
<jxb:globalBindings
.
    <binding_declarations>
.
<jxb:schemaBindings>
.
    <binding_declarations>
.
</jxb:schemaBindings>
</xs:appinfo>
</xs:annotation>
```

Class Declarations

The class-level binding declarations in `bindings.xjb` differ from the analogous declarations in `po.xsd` for the Datatype Converter example in two ways:

- As with all other binding declarations in `bindings.xjb`, you do not need to embed your customizations in schema `<xsd:appinfo>` elements.
- You must specify the schema node to which the customization will be applied. The general syntax for this type of declaration is:

```
<jxb:bindings node="//< node_type >[@name='< node_name >']">
```

For example, the following code shows binding declarations for the complex-Type named `USAddress`.

```
<jxb:bindings node="//xs:complexType[@name='USAddress']">
<jxb:class>
<jxb:javadoc>
<![CDATA[First line of documentation for a <b>USAddress</b>..]]>
</jxb:javadoc>
</jxb:class>

<jxb:bindings node="//xs:element[@name='name']">
<jxb:property name="toName"/>
</jxb:bindings>

<jxb:bindings node="//xs:element[@name='zip']">
<jxb:property name="zipCode"/>
</jxb:bindings>
</jxb:bindings>
<!-- node="//xs:complexType[@name='USAddress']" -->
```

Note in this example that `USAddress` is the parent of the child elements `name` and `zip`, and therefore a `</jxb:bindings>` tag encloses the `bindings` declarations for the child elements as well as the class-level `javadoc` declaration.

Fix Collides Example

The Fix Collides example illustrates how to resolve name conflicts—that is, places in which a declaration in a source schema uses the same name as another declaration in that schema (namespace collisions), or places in which a declaration uses a name that does translate by default to a legal Java name.

Note: Many name collisions can occur because XSD Part 1 introduces six unique symbol spaces based on type, while Java only has only one. There is a symbols space for type definitions, elements, attributes, and group definitions. As a result, a valid XML schema can use the exact same name for both a type definition and a global element declaration.

For the purposes of this example, it is recommended that you remove the `binding` parameter to the `xjc` task in the `build.xml` file in the `<INSTALL>/examples/jaxb/fix-collides` directory to display the error output generated by the `xjc` compiler. The XML schema for the Fix Collides, `example.xsd`, contains deliberate name conflicts.

Like the External Customize example, the Fix Collides example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customizations.

- The `example.xsd` Schema
- Looking at the Conflicts
- Output From Running the `ant` Task Without Using a Binding Declarations File
- The `binding.xjb` Declarations File
- Resolving the Conflicts in `example.xsd`

The `example.xsd` Schema

The XML schema, `<INSTALL>/examples/jaxb/fix-collides/example.xsd`, used in the Fix Collides example illustrates common name conflicts encountered when attempting to bind XML names to unique Java identifiers in a Java package. The schema declarations that result in name conflicts are highlighted in bold below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  jxb:version="1.0">

  <xs:element name="Class" type="xs:int"/>
  <xs:element name="FooBar" type="FooBar"/>
  <xs:complexType name="FooBar">
    <xs:sequence>
      <xs:element name="foo" type="xs:int"/>
      <xs:element ref="Class"/>
    
```

```

    <xs:element name="zip" type="xs:integer"/>
  </xs:sequence>
  <xs:attribute name="zip" type="xs:string"/>
</xs:complexType>
</xs:schema>

```

Looking at the Conflicts

The first conflict in `example.xsd` is the declaration of the element `name Class` :

```

<xs:element name="Class" type="xs:int"/>

```

`Class` is a reserved word in Java, and while it is legal in the XML schema language, it cannot be used as a name for a schema-derived class generated by JAXB.

When this schema is run against the JAXB binding compiler with the `-ant fail` command, the following error message is returned:

```

[xjc] [ERROR] Attempt to create a property having the same
name as the reserved word "Class".
[xjc] line 6 of example.xsd

```

The second conflict is that there are an `element` and a `complexType` that both use the name `FooBar` :

```

<xs:element name="FooBar" type="FooBar"/>
<xs:complexType name="FooBar">

```

In this case, the error messages returned are:

```

[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd

```

The third conflict is that there are an `element` and an `attribute` both named `zip` :

```

<xs:element name="zip" type="xs:integer"/>
<xs:attribute name="zip" type="xs:string"/>

```

The error messages returned here are:

```
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 22 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 20 of example.xsd
```

Output From Running the ant Task Without Using a Binding Declarations File

Here is the output that is returned if you run the `ant` task in the `<INSTALL >/examples/jaxb/fix-collides` directory without specifying the `binding` parameter to the `xjc` task in the `build.xml` file:

```
[echo] Compiling the schema w/o external binding file
(name collision errors expected)...
[xjc] Compiling file:/C:/javaetutorial5/examples/jaxb/
fix-collides/example.xsd
[xjc] [ERROR] Attempt to create a property having the same
name as the reserved word "Class".
[xjc] line 14 of example.xsd
[xjc] [ERROR] A property with the same name "Zip" is
generated from more than one schema component.
[xjc] line 17 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from this schema component.
[xjc] line 15 of example.xsd
[xjc] [ERROR] A class/interface with the same name
"generated.FooBar" is already in use.
[xjc] line 9 of example.xsd
[xjc] [ERROR] (Relevant to above error) another one is
generated from here.
[xjc] line 18 of example.xsd
```

The binding.xjb Declarations File

The `<INSTALL >/examples/jaxb/fix-collides/binding.xjb` binding declarations file resolves the conflicts in `examples.xsd` by means of several customizations.

Resolving the Conflicts in example.xsd

The first conflict in `example.xsd`, using the Java reserved name `Class` for an element name, is resolved in `binding.xjb` with the `<class>` and `<property>` declarations on the schema element node `Class`:

```
<jxb:bindings node="//xs:element[@name='Class']">
  <jxb:class name="Clazz"/>
  <jxb:property name="Clazz"/>
</jxb:bindings>
```

The second conflict in `example.xsd`, the namespace collision between the element `FooBar` and the complexType `FooBar`, is resolved in `binding.xjb` by using a `<nameXmlTransform>` declaration at the `<schemaBindings>` level to append the suffix `Element` to all element definitions.

This customization handles the case where there are many name conflicts due to systemic collisions between two symbol spaces, usually named type definitions and global element declarations. By appending a suffix or prefix to every Java identifier representing a specific XML symbol space, this single customization resolves all name collisions:

```
<jxb:schemaBindings>
  <jxb:package name="example"/>
  <jxb:nameXmlTransform>
    <jxb:elementName suffix="Element"/>
  </jxb:nameXmlTransform>
</jxb:schemaBindings>
```

The third conflict in `example.xsd`, the namespace collision between the element `zip` and the attribute `zip`, is resolved in `binding.xjb` by mapping the attribute `zip` to property named `zipAttribute`:

```
<jxb:bindings node="//xs:attribute[@name='zip']">
  <jxb:property name="zipAttribute"/>
</jxb:bindings>
```

If you add the `binding` parameter you removed back to the `xjc` task in the `build.xml` file and then run `ant` in the `<INSTALL>/examples/jaxb/fix-collides` directory, the customizations in `binding.xjb` will be passed to the `xjc` binding compiler, which will then resolve the conflicts in `example.xsd` in the schema-derived Java classes.

Bind Choice Example

The Bind Choice example shows how to bind a `choice` model group to a Java interface. Like the External Customize and Fix Collides examples, the Bind Choice example uses an external binding declarations file, `binding.xjb`, to define the JAXB binding customization.

The schema declarations in `<INSTALL >/examples/jaxb/bind-choice/example.xsd` that will be globally changed are highlighted in bold below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
jxb:version="1.0">

  <xs:element name="FooBar">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="foo" type="xs:int"/>
        <xs:element ref="Class"/>
        <xs:choice>
          <xs:element name="phoneNumber" type="xs:string"/>
          <xs:element name="speedDial" type="xs:int"/>
        </xs:choice>
      <xs:group ref="ModelGroupChoice"/>
    </xs:sequence>
    <xs:attribute name="zip" type="xs:string"/>
  </xs:complexType>
</xs:element>

  <xs:group name="ModelGroupChoice">
    <xs:choice>
      <xs:element name="bool" type="xs:boolean"/>
      <xs:element name="comment" type="xs:string"/>
      <xs:element name="value" type="xs:int"/>
    </xs:choice>
  </xs:group>
</xs:schema>
```

Customizing a choice Model Group

The `<INSTALL >/examples/jaxb/bind-choice/binding.xjb` binding declarations file demonstrates one way to override the default derived names for `choice`

model groups in `example.xsd` by means of a `<jxb:globalBindings>` declaration:

```
<jxb:bindings schemaLocation="example.xsd" node="/xs:schema">
<jxb:globalBindings bindingStyle="modelGroupBinding"/>
<jxb:schemaBindings/>
<jxb:package name="example"/>
</jxb:schemaBindings>
</jxb:bindings
</jxb:bindings>
```

This customization results in the `choice` model group being bound to its own content interface. For example, given the following `choice` model group:

```
<xs:group name="ModelGroupChoice">
<xs:choice>
<xs:element name="bool" type="xs:boolean"/>
<xs:element name="comment" type="xs:string"/>
<xs:element name="value" type="xs:int"/>
</xs:choice>
</xs:group>
```

the `globalBindings` customization shown above causes JAXB to generate the following Java class:

```
/**
 * Java content class for model group.
 */
public interface ModelGroupChoice {
int getValue();
void setValue(int value);
boolean isSetValue();

java.lang.String getComment();
void setComment(java.lang.String value);
boolean isSetComment();

boolean isBool();
void setBool(boolean value);
boolean isSetBool();

Object getContent();
boolean isSetContent();
void unSetContent();
}
```

Calling `getContent` returns the current value of the `Choice` content. The setters of this `choice` are just like radio buttons; setting one unsets the previously set one. This class represents the data representing the `choice`.

Additionally, the generated Java interface `FooBarType`, representing the anonymous type definition for element `FooBar`, contains a nested interface for the choice model group containing `phoneNumber` and `speedDial`.

Java-toSchema Examples

The Java-to-Schema examples show how to use annotations to map Java classes to XML schema.

j2s-create-marshal Example

The `j2s-create-marhal` example illustrates Java to schema databinding. It demonstrates marshalling and unmarshalling of JAXB annotated classes. The example also shows how to enable JAXP 1.3 validation at unmarshal time using a schema file that was generated from the JAXB mapped classes.

The schema file, `bc.xsd`, was generated with the following commands:

```
% schemagen src/cardfile/*.java
% cp schema1.xsd bc.xsd
```

Note that `schema1.xsd`, was copied to `bc.xsd`; `schemagen` does not allow you to specify a schema name of your choice.

j2s-xmlAccessorOrder Example

The `j2s-xmlAccessorOrder` example shows how to use the `@XmlAccessorOrder` and `@XmlType.propOrder` annotations to dictate the order in which XML content is marshalled/unmarshalled by a Java type.

Java-to-Schema maps a JavaBean's properties and fields to an XML Schema type. The class elements are mapped to either an XML Schema complex type or an XML Schema simple type. The default element order for a generated schema type is currently unspecified because Java reflection does not impose a return order. The lack of reliable element ordering negatively impacts application portability. You can use two annotations, `@XmlAccessorOrder` and `@XmlType.pro-`

pOrder, to define schema element ordering for applications that need to be portable across JAXB Providers.

The @XmlAccessorType annotation imposes one of two element ordering algorithms, AccessorOrder.UNDEFINED or AccessorOrder.ALPHABETICAL. AccessorOrder.UNDEFINED is the default setting. The order is dependent on the system's reflection implementation. AccessorOrder.ALPHABETICAL orders the elements in lexicographic order as determined by `java.lang.String.CompareTo(String anotherString)`.

You can define the @XmlAccessorType annotation for annotation type ElementType.PACKAGE on a class object. When the @XmlAccessorType annotation is defined on a package, the scope of the formatting rule is active for every class in the package.

When defined on a class, the rule is active on the contents of that class.

There can be multiple @XmlAccessorType annotations within a package. The order of precedence is the innermost (class) annotation takes precedence over the outer annotation. For example, if @XmlAccessorType(AccessorOrder.ALPHABETICAL) is defined on a package and @XmlAccessorType(AccessorOrder.UNDEFINED) is defined on a class in that package, the contents of the generated schema type for the class would be in an unspecified order and the contents of the generated schema type for every other class in the package would be alphabetical order.

The @XmlType annotation can be defined for a class. The annotation element propOrder() in the @XmlType annotation allows you to specify the content order in the generated schema type. When you use the @XmlType.propOrder annotation on a class to specify content order, all public properties and public fields in the class must be specified in the parameter list. Any public property or field that you want to keep out of the parameter list must be annotated with @XmlAttribute or @XmlTransient.

The default content order for @XmlType.propOrder is {} or {}, not active. In such cases, the active @XmlAccessorType annotation takes precedence. When class content order is specified by the @XmlType.propOrder annotation, it takes precedence over any active @XmlAccessorType annotation on the class or package. If the @XmlAccessorType and @XmlType.propOrder(A, B, ...) annotations are specified on a class, the propOrder always takes precedence regardless of the order of the annotation statements. For example, in the code

below, the `@XmlAccessorType` annotation precedes the `@XmlType.propOrder` annotation.

```
@XmlAccessorType(AccessorOrder.ALPHABETICAL)
@XmlType(propOrder={"name", "city"})
public class USAddress {
:
public String getCity() {return city;}
public void setCity(String city) {this.city = city;}

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

In the code below, the `@XmlType.propOrder` annotation precedes the `@XmlAccessorType` annotation.

```
@XmlType(propOrder={"name", "city"})
@XmlAccessorType(AccessorOrder.ALPHABETICAL)
public class USAddress {
:
public String getCity() {return city;}
public void setCity(String city) {this.city = city;}

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

In both scenarios, `propOrder` takes precedence and the identical schema content shown below will be generated.

```
<xs:complexType name="usAddress">
<xs:sequence>
  <xs:element name="name" type="xs:string" minOccurs="0"/>
  <xs:element name="city" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
```

The purchase order code example demonstrates the affects of schema content ordering using the `@XmlAccessorType` annotation at the package and class level, and the `@XmlType.propOrder` annotation on a class.

Class `package-info.java` defines `@XmlAccessorType` to be ALPHABETICAL for the package. The public fields `shipTo` and `billTo` in class `Purchase-`

`OrderType` will be affected in the generated schema content order by this rule. `Class USAddress` defines the `@XmlType.propOrder` annotation on the class. This demonstrates user-defined property order superseding ALPHABETICAL order in the generated schema.

The generated schema file can be found in directory `schemas` .

j2s-xmlAdapter-field Example

The `j2s-xmlAdapter-field` example demonstrates how to use the `XmlAdapter` interface and the `@XmlJavaTypeAdapter` annotation to provide a custom mapping of XML content into and out of a `HashMap` (field) that uses an “int” as the key and a “string” as the value.

Interface `XmlAdapter` and annotation `@XmlJavaTypeAdapter` are used for special processing of datatypes during unmarshalling/marshalling. There are a variety of XML datatypes for which the representation does not map easily into Java (for example, `xs:DateTime` and `xs:Duration`), and Java types which do not map conveniently into XML representations, for example implementations of `java.util.Collection` (such as `List`) and `java.util.Map` (such as `HashMap`) or for non-`JavaBean` classes. It is for these cases that

The `XmlAdapter` interface and the `@XmlJavaTypeAdapter` annotation are provided for cases such as these. This combination provides a portable mechanism for reading/writing XML content into and out of Java applications.

The `XmlAdapter` interface defines the methods for data reading/writing.

```

/*
 * ValueType - Java class that provides an XML representation
 * of the data. It is the object that is used for
 * marshalling and unmarshalling.
 *
 * BoundType - Java class that is used to process XML content.
 */

public abstract class XmlAdapter<ValueType,BoundType> {
    // Do-nothing constructor for the derived classes.
    protected XmlAdapter() {}

    // Convert a value type to a bound type.
    public abstract BoundType unmarshal(ValueType v);

    // Convert a bound type to a value type.
    public abstract ValueType marshal(BoundType v);
}

```

You can use the `@XmlJavaTypeAdapter` annotation to associate a particular `XmlAdapter` implementation with a `Target` type, `PACKAGE`, `FIELD`, `METHOD`, `TYPE`, or `PARAMETER`.

The `j2s-xmlAdapter-field` example demonstrates an `XmlAdapter` for mapping XML content into and out of a (custom) `HashMap`. The `HashMap` object, `basket`, in class `KitchenWorldBasket`, uses a key of type “int” and a value of type “String”. We want these datatypes to be reflected in the XML content that is read and written. The XML content should look like this.

```

<basket>
<entry key="9027">glasstop stove in black</entry>
<entry key="288">wooden spoon</entry>
</basket>

```

The default schema generated for Java type `HashMap` does not reflect the desired format.

```

<xs:element name="basket">
<xs:complexType>
<xs:sequence>
<xs:element name="entry" minOccurs="0"
maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>

```

```

<xs:element name="key" minOccurs="0"
type="xs:anyType"/>
<xs:element name="value" minOccurs="0"
type="xs:anyType"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>

```

In the default HashMap schema, key and value are both elements and are of datatype `anyType`. The XML content will look like this:

```

<basket>
<entry>
<key>9027</>
<value>glasstop stove in black</>
</entry>
<entry>
<key>288</>
<value>wooden spoon</>
</entry>
</basket>

```

To resolve this issue, we wrote two Java classes, `PurchaseList` and `PartEntry`, that reflect the needed schema format for unmarshalling/marshalling the content. The XML schema generated for these classes is as follows:

```

<xs:complexType name="PurchaseListType">
<xs:sequence>
<xs:element name="entry" type="partEntry"
nillable="true" maxOccurs="unbounded"
minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="partEntry">
<xs:simpleContent>
<xs:extension base="xs:string">
<xs:attribute name="key" type="xs:int"
use="required"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>

```

Class `AdapterPurchaseListToHashMap` implements the `XmlAdapter` interface. In class `KitchenWorldBasket`, the `@XmlJavaTypeAdapter` annotation is used to pair `AdapterPurchaseListToHashMap` with field `HashMap basket`. This pairing will cause the marshal/unmarshal method of `AdapterPurchaseListToHashMap` to be called for any corresponding marshal/unmarshal action on `KitchenWorldBasket`.

j2s-xmlAttribute-field Example

The `j2s-xmlAttribute-field` example shows how to use the `@XmlAttribute` annotation to define a property or field to be treated as an XML attribute.

The `@XmlAttribute` annotation maps a field or JavaBean property to an XML attribute. The following rules are imposed:

- A static final field is mapped to a XML fixed attribute.
- When the field or property is a collection type, the items of the collection type must map to a schema simple type.
- When the field or property is other than a collection type, the type must map to a schema simple type.

When following the JavaBean programming paradigm, a property is defined by a “get” and “set” prefix on a field name.

```
int zip;
public int getZip(){return zip;}
public void setZip(int z){zip=z;}
```

Within a bean class, you have the choice of setting the `@XmlAttribute` annotation on one of three components: the field, the setter method, or the getter method. If you set the `@XmlAttribute` annotation on the field, the setter method will need to be renamed or there will be a naming conflict at compile time. If you set the `@XmlAttribute` annotation on one of the methods, it must be set on either the setter or getter method, but not on both.

The `j2s-xmlAttribute-field` example shows how to use the `@XmlAttribute` annotation on a static final field, on a field rather than on one of the corresponding bean methods, on a bean property (method), and on a field that is other than a collection type. In class `USAddress`, fields, `country`, and `zip` are tagged as attributes. The `setZip` method was disabled to avoid the compile error. Property state was tagged as an attribute on the setter method. You could have used the getter method instead. In class `PurchaseOrderType`, field `cCardVendor` is a

non-collection type. It meets the requirement of being a simple type; it is an enum type.

j2s-xmlRootElement Example

The j2s-xmlRootElement example demonstrates the use of the `@XmlRootElement` annotation to define an XML element name for the XML schema type of the corresponding class.

The `@XmlRootElement` annotation maps a class or an enum type to an XML element. At least one element definition is needed for each top-level Java type used for unmarshalling/marshalling. If there is no element definition, there is no starting location for XML content processing.

The `@XmlRootElement` annotation uses the class name as the default element name. You can change the default name by using the annotation attribute `name`. If you do, the specified name will then be used as the element name and the type name. It is common schema practice for the element and type names to be different. You can use the `@XmlType` annotation to set the element type name.

The namespace attribute of the `@XmlRootElement` annotation is used to define a namespace for the element.

j2s-xmlSchemaType-class Example

The j2s-XmlSchemaType-class example demonstrates the use of the annotation `@XmlSchemaType` to customize the mapping of a property or field to an XML built-in type.

The `@XmlSchemaType` annotation can be used to map a Java type to one of the XML built-in types. This annotation is most useful in mapping a Java type to one of the nine date/time primitive datatypes.

When the `@XmlSchemaType` annotation is defined at the package level, the identification requires both the XML built-in type name and the corresponding Java type class. A `@XmlSchemaType` definition on a field or property takes precedence over a package definition.

The j2s-XmlSchemaType-clasexample shows how to use the `@XmlSchemaType` annotation at the package level, on a field and on a property. `FileTrackingOrder` has two fields, `orderDate` and `deliveryDate`, which are defined to be of type `XMLGregorianCalendar`. The generated schema will define these elements to be

of XML built-in type `gMonthDay`. This relationship was defined on the package in the file `package-info.java`. Field `shipDate` in file `TrackingOrder` is also defined to be of type `XMLGregorianCalendar`, but the `@XmlSchemaType` annotation statements override the package definition and specify the field to be of type `date`. Property method `getTrackingDuration` defines the schema element to be defined as primitive type `duration` and not Java type `String`.

j2s-xmlType Example

The `j2s-xmlType` example demonstrates the use of annotation `@XmlType`. Annotation `@XmlType` maps a class or an enum type to a XML Schema type.

A class must have either a public zero arg constructor or a static zero arg factory method in order to be mapped by this annotation. One of these methods is used during unmarshalling to create an instance of the class. The factory method may reside within in a factory class or the existing class. There is an order of precedence as to which method is used for unmarshalling.

- If a factory class is identified in the annotation, a corresponding factory method in that class must also be identified and that method will be used.
- If a factory method is identified in the annotation but no factory class is identified, the factory method must reside in the current class. The factory method is used even if there is a public zero arg constructor method present.
- If no factory method is identified in the annotation, the class must contain a public zero arg constructor method.

In this example a factory class provides zero arg factory methods for several classes. The @XmlType annotation on class `OrderContext` references the factory class. The unmarshaller will use the identified factory method in this class.

```
public class OrderFormsFactory {
    public OrderContext newOrderInstance() {
        return new OrderContext()
    }

    public PurchaseOrderType newPurchaseOrderType() {
        return new newPurchaseOrderType();
    }
}

@XmlType(name="oContext", factoryClass="OrderFormsFactory",
factoryMethod="newOrderInstance")
public class OrderContext {
    public OrderContext(){ ..... }
}
```

In this example, a factory method is defined in a class, which also contains a standard class constructure. Because the `factoryMethod` value is defined and no `factoryClass` is defined, the factory method `newOrderInstance` is used during unmarshalling.

```
@XmlType(name="oContext", factoryMethod="newOrderInstance")
public class OrderContext {

    public OrderContext(){ ..... }

    public OrderContext newOrderInstance() {
        return new OrderContext();
    }
}
```