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, 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

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>
</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.

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 a `jxb:bindings` declaration specifying `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 2-4.

Figure 2-1 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.

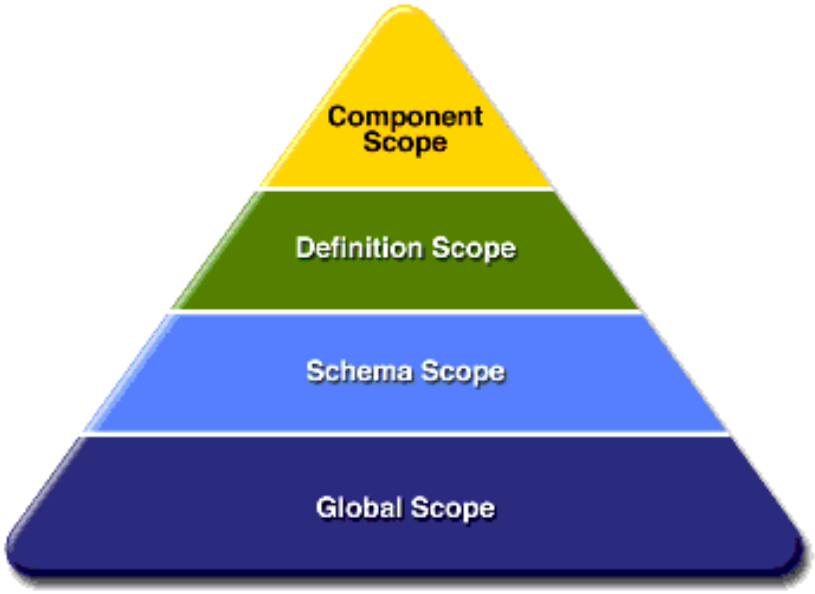

Figure 2-1 Customization Scope Inheritance and Precedence

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

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 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 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 < 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 getter 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 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.

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 &lt;b>Javadoc&lt;\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.

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 `<javaType>` 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 `<javaType>` 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 `enableFailFastCheck` 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 &lt;b>Purchase Order&lt;/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 < b>Purchase Order</b> consists of addresses and items."` The `<` is used to escape the opening bracket on the <b> 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>
        </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 `&lt;`. 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: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 `<JWSDP_HOME>/jaxb/samples/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 `<JWSDP_HOME>/jaxb/samples/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 `<jaxb: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, 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 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:

```
<xsd:annotation>
  <xsd:appinfo>
    <jxb:globalBindings
        .
        <binding_declarations>
        .
    <jxb:schemaBindings>
        .
        <binding_declarations>
        .
    </jxb:schemaBindings>
  </xsd:appinfo>
</xsd: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 `complexType` 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 run the `ant fail` command in the `<JWSDP_HOME>/jaxb/samples/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

The XML schema, `<JWSDP_HOME>/jaxb/samples/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 ant fail

Here is the complete output returned by running `ant fail` in the *<JWSDP_HOME>*`/jaxb/samples/fix-collides` directory:

```
[echo] Compiling the schema w/o external binding file
(name collision errors expected)...
[xjc] Compiling file:/C:/Sun/jwsdp-1.5/jaxb/samples/
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 <*JWSDP_HOME*>/jaxb/samples/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>
```

Running ant in the <*JWSDP_HOME*>/jaxb/samples/fix-collides directory will pass the customizations in binding.xjb 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 <*JWSDP_HOME*>/jaxb/samples/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 <*JWSDP_HOME*>/jaxb/samples/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`.