The Java™ Architecture for XML Binding (JAXB)

Final, V1.0
January 8th, 2003

Editors:
Joseph Fialli,
Sekhar Vajjhala
Comments to: jaxb-spec-comments@sun.com

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 USA

CONTENTS

CHAPTER 1

INTRODUCTION

XML is, essentially, a platform-independent means of structuring information. An XML document is a tree of *elements*. An element may have a set of *attributes*, in the form of key-value pairs, and may contain other elements, text, or a mixture thereof. An element may refer to other elements via *identifier* attributes, thereby allowing arbitrary graph structures to be represented.

An XML document need not follow any rules beyond the well-formedness criteria laid out in the XML 1.0 specification. To exchange documents in a meaningful way, however, requires that their structure and content be described and constrained so that the various parties involved will interpret them correctly and consistently. This can be accomplished through the use of a *schema*. A schema contains a set of rules that constrains the structure and content of a document's components, *i.e.*, its elements, attributes, and text. A schema also describes, at least informally and often implicitly, the intended conceptual meaning of a document's components. A schema is, in other words, a specification of the syntax and semantics of a (potentially infinite) set of XML documents. A document is said to be *valid* with respect to a schema if, and only if, it satisfies the constraints specified in the schema.

In what language are schemas written? The XML specification itself describes a sublanguage for writing *document-type definitions,* or DTDs. As schemas go, however, DTDs are fairly weak. They support the definition of simple constraints on structure and content, but provide no real facility for expressing datatypes or complex structural relationships. They have also prompted the creation of more sophisticated schema languages such as XDR, SOX, RELAX, TREX, and, most significantly, the XML Schema language recently defined by the World Wide Web Consortium.

This specification mandates support for a subset of the W3C XML Schema language.

1.1 Data binding

Any nontrivial application of XML will, then, be based upon one or more schemas and will involve one or more programs that create, consume, and manipulate documents whose syntax and semantics are governed by those schemas. While it is certainly possible to write such programs using the low-level SAX parser API or the somewhat higher-level DOM parse-tree API, doing so is likely to be tedious and error-prone. The resulting code is also likely to contain many redundancies that will make it difficult to maintain as bugs are fixed and as the schemas evolve.

It would be much easier to write XML-enabled programs if we could simply map the components of an XML document to in-memory objects that represent, in an obvious and useful way, the document's intended meaning according to its schema. Of what classes should these objects be instances? In some cases there will be an obvious mapping from schema components to existing classes, especially for common types such as `String`, `Date`, `Vector`, and so forth. In general, however, classes specific to the schema being used will be required. Rather than burden developers with having to write these classes we can generate the classes directly from the schema, thereby creating a Java object-level *binding* of the schema.

An *XML data-binding facility* therefore contains a *binding compiler* that binds components of a *source schema* to schema-derived Java *content classes*. Each class provides access to the content of the corresponding schema component via a set of JavaBeans-style access (*i.e.*, `get` and `set`) methods. *Binding declarations* provides a capability to customize the binding from schema components to Java representation. Such a facility also provides a *binding framework*, a runtime API that, in conjunction with the derived classes, supports three primary operations:

- The *unmarshalling* of an XML document into a tree of interrelated instances of both existing and schema-derived classes,
- The *marshalling* of such *content trees* back into XML documents, and
- The *validation* of content trees against the constraints expressed in the schema.

The unmarshalling process has the capability to check incoming XML documents for validity with respect to the schema. Similarly, a JAXB implementation provides a means to enforce the constraints expressed in the schema; some of these constraints may always be enforced, while others may only be checked upon explicit request. Validation can be used to ensure that only valid content trees are marshalled.

$$\begin{tikzcd}
\text{Schema} \arrow[r, "\textit{compile}"] & \text{Classes} \\
\text{Document} \arrow[u, "\textit{follows}"] \arrow[r, shift left, "\textit{unmarshal}"] & \text{Objects} \arrow[u, "\textit{instanceof}"'] \arrow[l, shift left, "\textit{marshal}"]
\end{tikzcd}$$

Figure 1.1 A mapping of XML to Java objects

To sum up: Schemas describe the structure and meaning of an XML document, in much the same way that a class describes an object in a program. To work with an XML document in a program we would like to map its components directly to a set of objects that reflect the document's meaning according to its schema. We can achieve this by compiling the schema into a set of derived content classes that can be marshalled, unmarshalled and validated. Data binding thus allows XML-enabled programs to be written at the same conceptual level as the documents they manipulate, rather than at the more primitive level of parser events or parse trees.

1.2 Goals

This specification aims to describe an XML data-binding facility with the following general properties:

- ***Be easy to use* –** Lower the barrier to entry to manipulating XML documents within Java programs. Programmers should be able to access and modify XML documents via a Java binding of the data, not via SAX or DOM. It should be possible for a developer who knows little about XML to compile a simple schema and immediately start making use of the classes that are produced.
- ***Be customizable* –** Provide a standard way to customize the binding of existing schema's components to Java representation of the components. Sophisticated applications sometimes require fine control over the structure and content of schema-derived classes, both for their own purposes and for that of coping with schema evolution.

- ***Portability* –** It should be possible to write a JAXB application in such a way that the JAXB implementation can be replaced without changes to the source code. Minimally, the schema would need to be submitted to the replacement JAXB implementations binding compiler and the output would need to be bundled with the application.
- ***Deliver Sooner rather than Later* –** Given the needs of the Java Community for a standardized XML data-binding solution to be delivered in a timely fashion, it was a important goal to identify a core set of functionality for this initial version of the specification that can be built upon in future versions. This document will identify the core requirements for the initial version and list the requirements and features for future consideration.

The derived classes produced by the binding compiler should, more specifically,

- ***Be natural* –** Insofar as possible, derived classes should observe standard Java API design guidelines and naming conventions. If new conventions are required then they should mesh well with existing conventions. A developer should not be astonished when trying to use a derived class.
- ***Match the conceptual level of the source schema* –** It should be straightforward to examine any content-bearing component of the source schema and identify the corresponding Java language construct in the derived classes.
- ***Hide all the plumbing* –** All the details of unmarshalling, marshalling, and validation should be completely encapsulated by schema-derived implementation classes and the runtime APIs upon which they depend. A developer should not have to think about SAX or DOM or any other XML-related API in order to perform unmarshal, marshal or validation on the schema-derived classes.
- ***Support validation on demand* –** While working with a content tree corresponding to an XML document it is often necessary to validate the tree against the constraints in the source schema. It should be possible to do this at any time, without the user having to first marshal the tree into XML.
- ***Preserve equivalence (round tripping)* –** Transforming a Java content tree to XML content and back to Java content again should result in an equivalent Java content tree before and after the transformation.

1.3 Non-Goals

- **Defining a standardized binding framework runtime system.**

  The schema-derived Java implementation classes generated by one JAXB implementation are not required to work with the runtime system of another JAXB implementation. To switch to an alternative JAXB implementations, one is required to regenerate the schema-derived implementation using the alternative JAXB implementation's binding compiler. Trying to identify a clear cut, acceptable common framework would jeopardize our "deliver sooner than later" goal. As XML processing technologies mature, we hope to identify a common framework solution in a future version of this specification. See Section 3.1, "Binding Runtime Framework Rationale," on page 24 for further details.

- **Preserving equivalence of XML document when round tripping from XML document to Java representation and back to XML document again.**

  While the JAXB specification does not require the preservation of the XML information set, it does not forbid the preservation of it.

- **Formally describing support for binding an existing JavaBean class to schema.**

  The feature will be considered for a future release but it was considered out of scope for this release.

- **Schema evolution support.**

  It is beyond the scope of the first version of the specification to address this important but difficult problem.

- **Providing support for accessing/adding of elements or attributes not initially declared in the schema.**

  The usage of `<anyAttribute>` in a schema allows an XML document to dynamically introduce data of a structure and content that was not described in the schema submitted to the binding compiler. It is not possible to generate type-safe accessors and classes for datatypes introduced by an XML document.

  A future version of the specification may provide access to dynamically introduced XML content, perhaps by returning the XML content in a generic XML representation, such as DOM.

- **Provide partial binding of an XML content root to a Java representation, skipping descendants of the XML content root that are not relevant to the task at hand.**

  If there is only a partial binding of all non-optional XML elements reachable from an XML element, it would no longer be possible to roundtrip the data back to its original XML content form. Partial mapping results in a one-way trip from the XML to a Java representation. There would be no marshal method from a Java representation back to XML since in general it would not be possible to produce a valid XML content from a partial Java representation of the XML content root and its descendants.

- **Requiring a facility described by this specification to implement every feature of the schema language it supports.**

  More precisely, a given schema-language feature need not be implemented if it is not commonly used in data-oriented applications of XML and if supporting it would unduly complicate either this specification or its implementations. This does not imply that supporting document-oriented applications is something to be avoided; it merely points out that some schema-language features that are used primarily in such applications do not always fit well into the context of an XML data-binding facility. This specification and its implementations will support document-oriented applications insofar as doing so does not interfere with achieving the above goals.

- **Explicit support for specifying the binding of DTD to a Java representation.**

  While it was desired to explicitly support binding DTD to a Java representation, it became impractical to describe both XML Schema binding and DTD binding. The existence of several conversion tools that automate the conversion of a DTD to XML Schema allows DTD users to be able to take advantage of JAXB technology by converting their existing DTDs to XML Schema.

1.4 Requirements

1. **Standardized schema input to binding compiler**

   Supported schema language:

   ❍ **Subset of W3C XML Schema.**

   All implementations are required to support the minimal required subset of W3C XML Schema. Non-required constructs are specified in Section E.2, "Not Required XML Schema concepts," on page 208. It is acceptable that an implementation support more than the minimal required subset in an implementation-dependent manner. Future versions of the specification will consider adding more complete support for W3C XML Schema.

2. **Describe default bindings from schema to Java representation**

   There must be a detailed, unambiguous description of the default mapping of schema components to Java representations in order to satisfy the portability goal. The default binding will be described from abstraction definitions of XML Schema components[XML Schema Part 1]. Each JAXB implementation must generate the same group of schema-derived interfaces and property accessors.

   ❍ Default binding from XML Schema built-in data types to Java built-in classes

   ❍ Default binding of XML Schema component, as described by abstract data model, to a Java representation.

3. **Standardized Customized Binding Schema**

   A binding schema language and its formats must be specified. There must be a means to describe the binding without requiring modification to the original schema. Additionally, the same XML Schema language must be used for the two different mechanisms for expressing a binding declaration.

4. **Capability to specify an override for default binding behavior**

   Given the diverse styles that can be used to design a schema, it is daunting to identify a single ideal default binding solution. For situations where several equally good binding alternatives exist, the specification will describe the alternatives and select one to be the default (see 3).

The binding schema must provide a means to specify an alternative binding for the scope of an entire schema. This mechanism ensures that if the default binding is not sufficient, it can easily be overridden in a portable manner.

5. **Provide ability to disable schema validation for unmarshal and marshal operations**

   There exist a significant number of scenarios that do not require validation and/or can not afford the overhead of schema validation. An application must be allowed to disable schema validation checking during unmarshal and marshal operations. The goal of this requirement is to provide the same flexibility and functionality that a SAX or DOM parser allows for. Please note that this specification can not define deterministic behavior of unmarshalling an invalid document or marshalling an invalid content tree when validation has been disabled.

1.5 Use Cases

Since the JAXB architecture provides a Java application the ability to manipulate XML content via generated Java interfaces, all of these uses cases assume the operation is occurring from within a Java application.

- Access configuration values from a properties file stored in a XML format. Tool allowing for the creation or modification to a configuration properties file represented in XML format.
- Receive data in the format of an XML document and would like to access/update it without having to write SAX event handlers or traverse a DOM parse tree.
- Validate user-inputted data, for example, from a form presented in a web browser. Form data could be mapped to an XML document. JAXB technology provides capability to validate the accuracy of the data using the validation constraints of a schema that describes the data collected from the form.
- Bind an XML document into a Java representation, update the content via Java interfaces, validate this changes against the constraints within the schema and then write the updated Java representation back to an XML document format.

- Unmarshal an XML document that it is known to already be valid, thus the application disables validation checking while unmarshalling the document to improve performance.

1.6 Conventions

Within normative prose in this specification, the words *should* and *must* are defined as follows:

- *should*
  Conforming implementations are permitted to but need not behave as described.
- *must*
  Conforming implementations are required to behave as described; otherwise they are in error.

Throughout the document, the XML namespace prefix `xs:` and `xsd:` refers to schema components in W3C XML Schema namespace as specified in [XSD Part 1] and [XSD Part 2]. Additionally, the XML namespace prefix `jaxb`: refers to the JAXB namespace, `http::/java.sun.com/xml/ns/jaxb`.

All examples in the specification are for illustrative purposes to assist in understanding concepts and are non-normative. If an example conflicts with the normative prose, the normative prose always takes precedence over the example.

1.7 Expert Group Members

The following people have contributed to this specification effort.

Arnaud Blandin, Intalio
Steve Brodsky, IBM
Christian Campo, Software AG
Kohsuke Kawaguchi, Sun
Chris Fry, BEA
Eric Johnson, TIBCO
Anjana Manian, Oracle
Ed Merks, IBM
Greg Messner, The Breeze Factor
Masaya Naito, Fujitsu
David Stephenson, HP
Keith Visco, Intalio
Scott Ziegler, BEA

1.8 Acknowledgements

This document is a derivative work of concepts and an initial draft initially led by Mark Reinhold of Sun Microsystems. Our thanks to all who were involved in pioneering that initial effort. The feedback from the Java User community on the initial JAXB technology prototype greatly assisted in identifying requirements and directions.

The data binding experiences of the expert group members have been instrumental in identifying the proper blend of the countless data binding techniques that we have considered in the course of writing this specification. We thank them for their contributions and their review feedback.

Kohsuke Kawaguchi and Ryan Shoemaker have directly contributed content to the specification and wrote the companion javadoc. The following JAXB technology team members have been invaluable in keeping the specification effort on the right track: Tom Amiro, Leonid Arbouzov, Evgueni Astigueevitch, Jennifer Ball, Carla Carlson, Patrick Curran, Scott Fordin, Omar Fung, Peter Kacandes, Dmitry Khukhro, Tom Kincaid, K. Ari Krupnikov, Ramesh Mandava, Bhakti Mehta, Ed Mooney, Ilya Neverov, Oleg Oleinik, Brian Ogata, Vivek Pandey, Cecilia Peltier, Evgueni Rouban and Leslie Schwenk. The following people, all from Sun Microsystems, have provided valuable input to this effort: Roberto Chinnici, Chris Ferris, Mark Hapner, Eve Maler, Farrukh Najmi, Eduardo Pelegri-llopart, Bill Shannon and Rahul Sharma.

The JAXB TCK software team would like to acknowledge that the NIST XML Schema test suite [NIST] has greatly assisted the conformance testing of this specification.

CHAPTER 2

ARCHITECTURE

2.1 Overview

The primary components of the XML data-binding facility described in this specification are the binding compiler, the binding framework, and the binding language.

- The *binding compiler* transforms, or *binds*, a *source schema* to a set of *content classes* in the Java programming language. As used in this specification, the term *schema* includes the W3C XML Schema as defined in the XML Schema 1.0 Recommendation[XSD Part 1][XSD Part 2].
- The *binding runtime framework* provides the interfaces for the functionality of unmarshalling, marshalling, and validation for content classes.
- The *binding language* is an XML-based language that describes the binding of a source schema to a Java representation. The binding declarations written in this language specify the details of the package, interfaces and classes derived from a particular source schema.

Figure 2.1 describes concepts to be presented in this chapter.

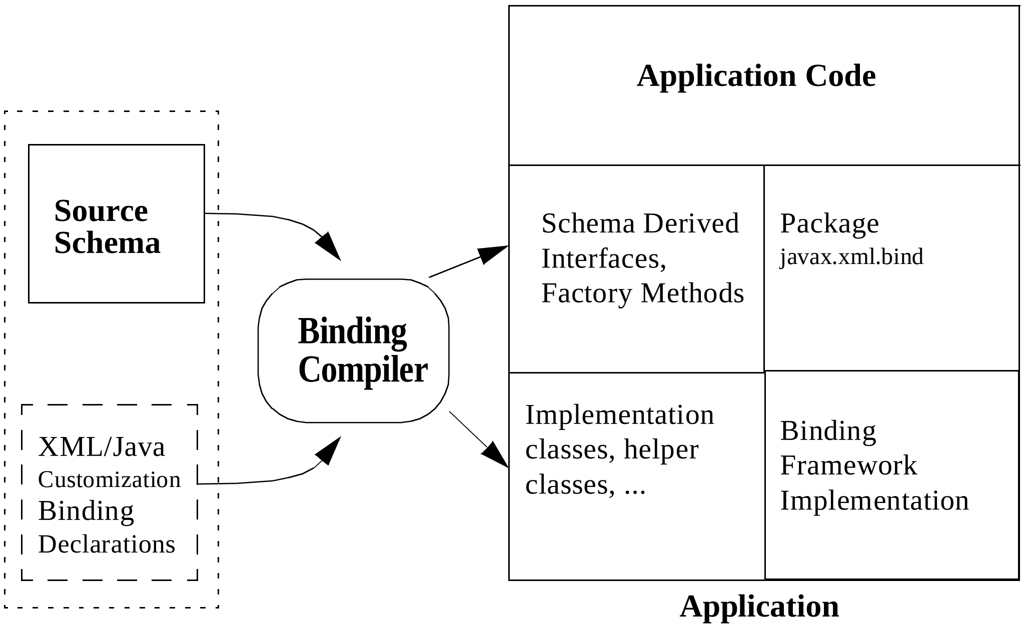
Application Code
Source Schema
Schema Derived Interfaces, Factory Methods
Package javax.xml.bind
Binding Compiler
Implementation classes, helper classes, ...
Binding Framework Implementation
XML/Java Customization Binding Declarations
Application

Figure 2.1 Non-Normative JAXB Architecture diagram

Note that the binding declarations object in the above diagram is logical. Binding declarations can either be inlined within the schema or they can appear in an external binding file that is associated with the source schema. Also, note that the application accesses only the derived content interfaces, factory methods and javax.xml.bind APIs directly. This convention is necessary to enable switching between JAXB implementations.

2.1.1 Java Representation

A coarse-grained content-bearing schema component, such as a complex type definition, is generally bound to a *content interface*. The data binding uses the Java class hierarchy between content interfaces to preserve an XML Schema's "derived by extension" type definition hierarchy.

A fine-grained schema component, such as an attribute declaration or an element declaration with a simple type, is bound directly to a *property* within a content interface. A property is *realized* in a content interface by a set of JavaBeans-style *access methods*. These methods include the usual `get` and `set` methods for retrieving and modifying a property's value; they also provide for the deletion and, if appropriate, the re-initialization of a property's value.

Properties are also used for references from one content instance to another. If an instance of a schema component *X* can occur within, or be referenced from, an instance of some other component *Y* then the content class derived from *Y* will define a property that can contain instances of *X*.

To add flexibility within the JAXB architecture, a content class is represented as both a content interface and an implementation of that interface rather than just a class. This separation enables a sophisticated users of the JAXB architecture to be able to specify their own implementation of the content interface to be used within the binding framework. Typical users will rely on the binding compiler to generate both schema-derived content interfaces and their implementations.

2.1.2 Binding Framework

The primary operations that can be performed on the set of schema-derived content interfaces and implementation classes are those of unmarshalling, marshalling, and validation.

- *Unmarshalling* is the process of reading an XML document and constructing a tree of content objects. Each content object corresponds directly to an instance in the input document of the corresponding schema component, hence this content tree reflects the document's content.
- *Marshalling* is the inverse of unmarshalling, i.e., it is the process of traversing a content tree and writing an XML document that reflects the tree's content.
- *Validation* is the process of verifying that all constraints expressed in the source schema hold for a given content tree. A *content tree* is *valid* if, and only if, marshalling the tree would generate a document that is valid with respect to the source schema.

When the unmarshalling process incorporates validation and it successfully completes without any validation errors, both the input document and the resulting content tree are guaranteed to be valid. The marshalling process, on the other hand, does not actually perform validation. If only validated content trees are marshalled, this guarantees that generated XML documents are always valid with respect to the source schema.

However, always requiring validation during unmarshalling and only allowing the marshalling of validated content trees proves to be too rigid and restrictive a requirement. Since existing XML parsers allow schema validation to be disabled, there exist a significant number of XML processing uses that disable schema validation to improve processing speed and/or to be able to process documents containing invalid or incomplete content. To enable the JAXB architecture to be used in these processing scenarios, the binding framework makes validation optional. How a JAXB technology implementation handles unmarshalling of an invalid document when validation is disabled is implementation-specific. The same holds true for marshalling an invalid content tree. It is expected that once an implementation is aware that it cannot unambiguously complete unmarshalling or marshalling, it will terminate processing with an exception.

Unmarshalling is not the only means by which a content tree may be created. Schema-derived content classes also support the programmatic construction of content trees by direct invocation of the appropriate factory methods. Once created, a content tree may be re-validated, either in whole or in part, at any time.

2.1.3 Binding Declarations

A particular binding of a given source schema is defined by a set of *binding declarations*. Binding declarations are written in a *binding language*, which is itself an application of XML. A binding declaration can occur within the annotation `appinfo` of each XML Schema component. Alternatively, binding declarations can occur in an auxiliary file.Eeach binding declaration within the auxiliary file is associated to a schema component in the source schema. It was necessary to support binding declarations external to the source schema in order to allow for customization of an XML Schemas that one prefers not to modify. The binding compiler hence actually requires two inputs, a source schema and a set of binding declarations.

Binding declarations enable one to override default binding rules, thereby allowing for user customization of the schema-derived content interfaces. Additionally, binding declarations allows for further refinements to be introduced into the binding to Java representation that could not be derived from the schema alone.

The binding declarations need not define every last detail of a binding. The binding compiler assumes *default binding declarations* for those components of the source schema that are not mentioned explicitly by binding declarations.

Default declarations both reduce the verbosity of the customization and make it more robust to the evolution of the source schema. The defaulting rules are sufficiently powerful that in many cases a usable binding can be produced with no binding declarations at all. By defining a standardized format for the binding declarations, it is envisioned that tools will be built to greatly aid the process of customizing the binding from schema components to a Java representation.

2.2 Varieties of validation

The constraints expressed in a schema fall into three general categories:

- A *type* constraint imposes requirements upon the values that may be provided by constraint facets in simple type definitions.
- A *local structural* constraint imposes requirements upon every instance of a given element type, e.g., that required attributes are given values and that a complex element's content matches its content specification.
- A *global structural* constraint imposes requirements upon an entire document, e.g., that `ID` values are unique and that for every `IDREF` attribute value there exists an element with the corresponding `ID` attribute value.

A *document* is valid if, and only if, all of the constraints expressed in its schema are satisfied. Similarly, a *content tree* is valid if, and only if, marshalling the tree would produce a valid document. It would be both inconvenient and inefficient to have to marshal a content tree just to check its validity.

The manner in which constraints are enforced in a set of derived classes has a significant impact upon the usability of those classes. All constraints could, in principle, be checked only during unmarshalling and validation. This approach would, however, yield classes that violate the *fail-fast* principle of API design: errors should, if feasible, be reported as soon as they are detected. In the context of schema-derived implementation classes, this principle ensures that violations of schema constraints are signalled when they occur rather than later on when they may be more difficult to diagnose.

With this principle in mind we see that schema constraints can, in general, be enforced in three ways:

- *Static* enforcement leverages the type system of the Java programming language to ensure that a schema constraint is checked at application's compilation time. Type constraints are often good candidates for static enforcement. If an attribute is constrained by a schema to have a boolean value, e.g., then the access methods for that attribute's property can simply accept and return values of type `boolean`.
- *Simple dynamic* enforcement performs a trivial run-time check and throws an appropriate exception upon failure. Type constraints that do not easily map directly to Java classes or primitive types are best enforced in this way. If an attribute is constrained to have an integer value between zero and 100, e.g., then the corresponding property's access methods can accept and return `int` values and its mutation method can throw a run-time exception if its argument is out of range.
- *Complex dynamic* enforcement performs a potentially costly run-time check, usually involving more than one content object, and throwing an appropriate exception upon failure. Local structural constraints are usually enforced in this way: the structure of a complex element's content, e.g., can in general only be checked by examining the types of its children and ensuring that they match the schema's content model for that element. Global structural constraints must be enforced in this way: the uniqueness of `ID` values, e.g., can only be checked by examining the entire content tree.

It is straightforward to implement both static and simple dynamic checks so as to satisfy the fail-fast principle. Constraints that require complex dynamic checks could, in theory, also be implemented so as to fail as soon as possible. The resulting classes would be rather clumsy to use, however, because it is often convenient to violate structural constraints on a temporary basis while constructing or manipulating a content tree.

Consider, e.g., a complex type definition whose content specification is very complex. Suppose that an instance of the corresponding content interface is to be modified, and that the only way to achieve the desired result involves a sequence of changes during which the content specification would be violated. If the content instance were to check continuously that its content is valid, then the only way to modify the content would be to copy it, modify the copy, and then install the new copy in place of the old content. It would be much more convenient to be able to modify the content in place.

A similar analysis applies to most other sorts of structural constraints, and especially to global structural constraints. Schema-derived classes will therefore be able to enable or disable a mode that verifies type constraints and will be able to check structural constraints upon demand.

2.2.1 Handling Validation Failures

While it would be possible to notify a JAXB application that a validation error has occurred by throwing a `JAXBException` when the error is detected, this means of communicating a validation error results in only one failure at a time being handled. Potentially, the validation operation would have to be called as many times as there are validation errors. Both in terms of validation processing and for the application's benefit, it is better to detect as many errors and warnings as possible during a single validation pass. To allow for multiple validation errors to be processed in one pass, each validation error is mapped to a validation error event. A validation error event relates the validation error or warning encountered to the location of the text or object(s) involved with the error. The stream of potential validation error events can be communicated to the application either through a registered validation event handler at the time the validation error is encountered, or via a collection of validation failure events that the application can request after the operation has completed.

Unmarshalling and on-demand validation of in-memory objects are the two operations that can result in multiple validation failures. The same mechanism is used to handle both failure scenarios. See Section 3.3, "General Validation Processing," on page 26 for further details.

2.3 An example

Throughout this specification we will refer and build upon the familiar schema from [XSD Part 0], which describes a purchase order, as a running example to illustrate various binding concepts as they are defined. Note that all schema name attributes with values in **this font** are bound by JAXB technology to either a Java interface or JavaBean-like property. Please note that the derived Java code in the example only approximates the default binding of the schema-to-Java representation.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
        <xsd:element name="shipTo"     type="USAddress"/>
        <xsd:element name="billTo"     type="USAddress"/>
        <xsd:element ref="comment"     minOccurs="0"/>
        <xsd:element name="items"      type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate"    type="xsd:date"/>
</xsd:complexType>

<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>

<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">
                        <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:element name="shipDate" type="xsd:date" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="partNum"      type="SKU" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<!-- 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>
```

Binding of purchase order schema to a Java representation:

```
import java.util.Calendar; import java.util.List;
public interface PurchaseOrderType{
    USAddress    getShipTo();              void    setShipTo(USAddress);
    USAddress    getBillTo();              void    setBillTo(USAddress);
    /** Optional to set Comment property. */
    String       getComment();             void    setComment(String);
    Items        getItems();               void    setItems(Items);
    Calendar     getOrderDate();           void    setOrderDate(Calendar);
};
public interface USAddress{
    String       getName();                void    setName(String);
    String       getStreet();              void    setStreet(String);
    String       getCity();                void    setCity(String);
    String       getState();               void    setState(String);
    int          getZip();                 void    setZip(int);
    static final String COUNTRY=”USA”;[1]
};
public interface Items {
    public interface ItemType {
        String       getProductName();     void    setProductName(String);
        /** Type constraint on Quantity setter value 0..99.[2]*/
        int          getQuantity();        void    setQuantity(int);
        float        getUSPrice();         void    setUSPrice(float);
        /** Optional to set Comment property. */
        String       getComment();         void    setComment(String);
        Calendar     getShipDate();        void    setShipDate(Calendar);
        /** Type constraint on PartNum setter value "\d{3}-[A-Z]{2}".[2]*/
        String       getPartNum();         void    setPartNum(String);
    };

    /** Local structural constraint 1 or more instances of Items.ItemType.*/
    List getItem();
}
public interface PurchaseOrder extends PurchaseOrderType, javax.xml.bind.Element {};
public interface Comment extends javax.xml.bind.Element{
                 String getValue(); void setValue(String)};

public class ObjectFactory {
    Object                newInstance(Class javaInterface);
    PurchaseOrderType     createPurchaseOrderType();
    USAddress             createUSAddress();
    Items                 createItems();
    Items.ItemType        createItemsItemType();
    PurchaseOrder         createPurchaseOrder();
    Comments              createComment();
    Comments              createComment(String value);
}
```

1. Appropriate customization required to bind a fixed attribute to a constant value.

2. Type constraint checking only performed if customization enables it and implementation supports fail-fast checking

The purchase order schema does not describe any global structural constraints.

The coming chapters will identify how these XML Schema concepts were bound to a Java representation. Just as in [XSD Part 0], additions will be made to the schema example to illustrate the binding concepts being discussed.

CHAPTER 3

THE BINDING FRAMEWORK

The *binding framework* defines APIs to access unmarshalling, validation and marshalling operations for manipulating XML data and Java content instances. The framework is presented here in overview; its full specification is available in the javadoc for the package `javax.xml.bind`.

The binding framework resides in two main packages. The `javax.xml.bind` package defines abstract classes and interfaces that are used directly with content classes. The `javax.xml.bind` package defines the `Unmarshaller, Validator`, and `Marshaller` classes, which are auxiliary objects for providing their respective operations.

The `JAXBContext` class is the entry point for a Java application into the JAXB framework. A `JAXBContext` instance manages the binding relationship between XML element names to Java content interfaces for a JAXB implementation to be used by the unmarshal, marshal and validation operations. The `javax.xml.bind.helper` package provides partial default implementations for some of the `javax.xml.bind` interfaces. Implementations of JAXB technology can extend these classes and implement the abstract methods. These APIs are not intended to be used directly by applications using the JAXB architecture. A third package, `javax.xml.bind.util`, contains utility classes that may be used directly by client applications.

Finally, the binding framework defines a rich hierarchy of validation event and exception classes for use when marshalling/unmarshalling errors occur, when constraints are violated, and when other types of errors are detected.

3.1 Binding Runtime Framework Rationale

This version of the specification does not attempt to standardize the binding runtime framework, or expose at the JAXB API layer whether the Java content classes are capable of validating, marshalling and unmarshalling themselves and whether this functionality exists external to the instance. There is not enough experience at this time to identify a single acceptable framework suitable for all.

For example, some would like to pursue reflective, dynamic solutions that provide marshalling/unmarshalling capabilities, while others would like to generate static, fixed code solutions. Some would like to use non-standard pull parsing for unmarshalling, while others would rather leverage JAXP parsing and its validation capabilities for unmarshalling.

It would prematurely restrict the exploration of possible alternative solutions to attempt to identify a common runtime framework for all implementations to conform to at this time. It is hoped that as XML processing technologies mature in the future, it will be possible to identify a common binding runtime framework in a future version of the specification.

One unfortunate result of not standardizing the binding framework runtime system is that there is a tight coupling between the schema-derived implementation classes and the JAXB implementation’s runtime framework. Users are required to regenerate the schema-derived implementation classes when changing JAXB implementations. However, note that all implementations are required to support the ability to use multiple implementations of the JAXB architecture at the same time. For example, a third party library jar that an application uses might use one JAXB implementation and the application wishes to choose a different JAXB implementation to use. Details on how this can be achieved are discussed in the next section on `JAXBContext` class.

3.2 JAXBContext

The `JAXBContext` class provides the client’s entry point to the JAXB API. It provides an abstraction for managing the XML/Java binding information necessary to implement the JAXB binding framework operations: unmarshal, marshal, and validate. Additionally, the `JAXBContext` class is designed to ensure that the correct binding framework implementation is used with Java content implementation classes.

The following summarizes the `JAXBContext` class defined in package `javax.xml.bind`.

```
public abstract class JAXBContext {
    static final String JAXB_CONTEXT_FACTORY;
    static JAXBContext newInstance(String contextPath)
    static JAXBContext newInstance(String contextPath,
                                   ClassLoader contextPathCL)
    abstract Unmarshaller createUnmarshaller();
    abstract Marshaller createMarshaller();
    abstract Validator createValidator();
}
```

A client application obtains new instances of this class via the `newInstance(String)` factory method.

```
JAXBContext jc =
       JAXBContext.newInstance( “com.acme.foo:com.acme.bar” );
```

The following ordered lookup procedure for the `newInstance()`method is used to determine which concrete implementation of `JAXBContext` to load:

- Search the context path for the first occurrence of a file named `jaxb.properties` containing the `javax.xml.bind.context.factory` property and use its value.

The `contextPath` parameter to the `newInstance` method contains a list of Java package names that contain implementation specific means for mapping XML document instances for the specified schema vocabularies to Java content instances. Typically, the XML/Java binding information is expected to be generated by the binding compiler. However, note that all implementations are required to support the ability to use multiple implementations of the JAXB architecture at the same time. By allowing multiple Java packages to be specified, the `JAXBContext` instance allows for the management of multiple schemas at one time. All Java packages specified in the `contextPath` parameter must contain XML/Java binding information from only one JAXB implementation or, if there exists an ambiguity in the multiple schemas being joined by the `JAXBContext` instance, that a `JAXBException` is thrown by the `newInstance(String)` method.

By enabling a `JAXBContext` to represent more than one schema at a time, an `Unmarshaller` created from it is capable of processing XML instance documents from more than one schema in one `unmarshal` invocation. The use case exists where an application receives an XML document instance from an external source and the application does not know the precise schema vocabulary for the document but it does know that the document is an instance of one of several schemas. This use case is the motivation for `JAXBContext` being able to represent multiple schemas at one time.

See the javadoc for `JAXBContext` for more details on this class.

3.3 General Validation Processing

Three identifiable forms of validation exist within the JAXB architecture include:

- **Unmarshal-time validation**

  This form of validation enables a client application to be notified of validation errors and warnings detected while unmarshalling XML data into a Java content tree and is completely orthogonal to the other types of validation. To enable or disable it, see the javadoc for method `Unmarshaller.setValidating(boolean)`.

- **On-demand validation**

  An application may wish to validate the correctness of the Java content tree based on schema validation constraints. This form of validation enables an application to initiate the validation process on a Java content tree at a point in time that it feels it should be valid. The application is notified about validation errors and warnings detected in the Java content tree.

- **Fail-fast validation**

  This form of validation enables a client application to receive immediate feedback about a modification to the Java content tree that violates a type constraint of a Java property. An unchecked exception is thrown if the value provided to a set method is invalid based on the constraint facets specified for the basetype of the property. This style of validation is optional in the initial version of this specification. Of the JAXB implementations that do support this type of validation, it is customization-time decision to enable or disable fail-fast validation when setting a property.

Unmarshal-time and on-demand validation use an event-driven mechanism to enable multiple validation errors and warnings to be processed during a single operation invocation. If the validation or unmarshal operation terminates with an exception upon encountering the first validation warning or error, subsequent validation errors and warnings would not be discovered until the first reported error is corrected. Thus, the validation event notification mechanism provides the application a more powerful means to evaluate validation warnings and errors as they occur and gives the application the ability to determine when a validation warning or error should abort the current operation (such as a value outside of the legal value space). Thus, an application could allow locally constrained validation problems to not terminate validation processing.

If the client application does not set an event handler on a `Validator` or `Unmarshaller` instance prior to invoking the `validate` or `unmarshal` operations, then a default event handler will receive notification of any errors or fatal errors encountered and stop processing the XML data. In other words, the default event handler will fail on the first error that is encountered.

There are three ways to handle validation events encountered during the unmarshal and validate operations:

- **Rely on the default validation event handler**
  The default handler will fail on the first error or fatal error encountered.
- **Implement and register a custom validation event handler**
  Client applications that require sophisticated event processing can implement the `ValidationEventHandler` interface and register it with the Validator or Unmarshaller instance respectively.
- **Request an error/warning event list after the operation completes**
  By registering the `ValidationEventCollector` helper, a specialized event handler, with the `setEventHandler` method, the `ValidationEvent` objects created during the unmarshal and validate operations are collected. The client application can then request the list after the operation completes.

Validation events are handled differently depending on how the client application is configured to process them as described previously. However, there are certain cases where a JAXB implementation needs to indicate that it is no longer able to reliably detect and report errors. In these cases, the JAXB implementation will set the severity of the `ValidationEvent` to

FATAL_ERROR to indicate that the unmarshal or validate operation should be terminated. The default event handler and ValidationEventCollector helper class must terminate processing after being notified of a fatal error. Client applications that supply their own ValidationEventHandler should also terminate processing after being notified of a fatal error. If not, unexpected behavior may occur.

3.4 Validator

The Validator class is responsible for controlling the validation of a content tree of in-memory objects. The following summarizes the available operations on the class.

```
public interface Validator {
    ValidationEventHandler getEventHandler()
    void setEventHandler(ValidationEventHandler)

    boolean validate(java.lang.Object subrootObject)
    boolean validateRoot(java.lang.Object rootObject)

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)
}
```

The JAXBContext class provides a factory to create a Validator instance. After an application has made a series of modifications to a Java content tree, the application validates the content tree on-demand. As far as the application is concerned, this validation takes place against the Java content instances and validation constraint warnings and errors are reported to the application relative to the Java content tree. Validation is initiated by invoking the validateRoot(Object) method on the root of the Java content tree or by invoking validate(Object) method to validate any arbitrary subtree of the Java content tree. The only difference between these two methods is global constraint checking (i.e. verifying ID/IDREF constraints.) The validateRoot(Object) method includes global constraint checking as part of its operation, whereas the validate(Object) method does not.

The validator governs the process of validating the content tree, serves as a registry for identifier references, and ensures that all local (and when appropriate, global) structural constraints are checked before the validation process is complete.

If a violation of a local or global structural constraint is detected, then the application is notified of the event with a callback passing an instance of a `ValidationEvent` as a parameter.

Design Note – The specification purposely does not state how validation is to be implemented since there exist several different approaches which have their own pros and cons. For example, the validation could be completely generated Java code. It is believed that this approach would yield the fastest validation and easiest time relating the validation errors and warnings to the Java content instances. However, this approach will take a large effort to implement for XML Schema, could result in large generated code size and would take a while to become as mature as alternative implementation approaches. An alternative implementation approach is to stream the content tree into SAX 2 events validate using one of the existing, proven XML Schema validators.

3.5 Unmarshalling

The `Unmarshaller` class governs the process of deserializing XML data into a Java content tree, capable of validating the XML data as it is unmarshalled. It provides the basic unmarshalling methods:

```
public interface Unmarshaller {
    ValidationEventHandler getEventHandler()
    void setEventHandler(ValidationEventHandler)

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)

    boolean isValidating()
    void setValidating(boolean validating)

    UnmarshallerHandler getUnmarshallerHandler()

    java.lang.Object unmarshal(java.io.File)
    java.lang.Object unmarshal(java.net.URL)
```

```
        java.lang.Object unmarshal(java.io.InputStream)
        java.lang.Object unmarshal(org.xml.sax.InputSource)
        java.lang.Object unmarshal(org.w3c.dom.Node)

        java.lang.Object unmarshal(javax.xml.transform.Source)
    }
```

The `JAXBContext` class contains a factory to create an `Unmarshaller` instance. The `JAXBContext` instance manages the XML/Java binding data that is used by unmarshalling. If the `JAXBContext` object that was used to create an `Unmarshaller` does not know how to unmarshal the XML content from a specified input source, then the `unmarshal` operation will abort immediately by throwing an `UnmarshalException`. There are six convenience methods for unmarshalling from various input sources.

An application can enable or disable unmarshal-time validation using the `setValidating()` method. The application has the option to customize validation error handling by overriding the default event handler using the `setEventHandler(ValidationEventHandler).` The default event handler aborts the unmarshalling process when the first validation error event is encountered. Validation processing options are presented in more detail in Section 3.3, “General Validation Processing.”

When the unmarshalling process detects a structural inconsistency that it is unable to recover from, it should abort the unmarshal process by throwing `UnmarshalException`.

An application has the ability to specify a SAX 2.0 parser to be used by the `unmarshal` operation using the `unmarshal(javax.xml.transform.Source)` method. Even though the JAXB provider’s default parser is not required to be SAX2.0 compliant, all providers are required to allow an application to specify their own SAX2.0 parser. Some providers may require the application to specify the SAX2.0 parser at binding compile time. See the method javadoc `unmarshal(Source)` for more detail on how an application can specify its own SAX 2.0 parser.

The `getProperty`/`setProperty` methods introduce a mechanism to associate implementation specific property/value pairs to the unmarshalling process. At this time there are no standard JAXB properties specified for the unmarshalling process.

3.6 Marshalling

The `Marshaller` class is responsible for governing the process of serializing a Java content tree into XML data. It provides the basic marshalling methods:

```
interface Marshaller {
    static final string JAXB_ENCODING;
    static final string JAXB_FORMATTED_OUTPUT;
    static final string JAXB_SCHEMA_LOCATION;
    static final string JAXB_NO_NAMESPACE_SCHEMA_LOCATION;
    <PROTENTIALLY MORE PROPERTIES...>

    java.lang.Object getProperty(java.lang.String name)
    void setProperty(java.lang.String name, java.lang.Object value)

    void setEventHandler(ValidationEventHandler handler)
    ValidationEventHandler getEventHandler()

    void marshal(java.lang.Object obj, java.io.Writer writer)
    void marshal(java.lang.Object obj, java.io.OutputStream os)
    void marshal(java.lang.Object obj, org.xml.sax.ContentHandler)
    void marshal(java.lang.Object obj, javax.xml.transform.Result)
    void marshal(java.lang.Object obj, org.w3c.dom.Node)

    org.w3c.dom.Node getNode(java.lang.Object contentTree)
}
```

The `JAXBContext` class contains a factory to create a `Marshaller` instance. Convenience method overloadings of the `marshal()` method allow for marshalling a content tree to common Java output targets and to common XML output targets of a stream of SAX2 events or a DOM parse tree.

Although each of the marshal methods accepts a `java.lang.Object` as its first parameter, JAXB implementations are not required to be able to marshal any arbitrary `java.lang.Object`. If the `JAXBContext` object that was used to create this `Marshaller` does not know how to marshal the object parameter (or any objects reachable from it), then the marshal operation will throw a `MarshalException`. Even though JAXB implementations are not required to be able to marshal arbitrary `java.lang.Object` objects, an implementation is allowed to support this type of marshalling.

The marshalling process does not validate the content tree being marshalled, but if the marshalling process detects a structural inconsistency during its process that it is unable to recover from, it should abort the marshal process by throwing `MarshalException`.

Client applications are not required to validate the Java content tree prior to calling one of the marshal APIs. Furthermore, there is no requirement that the Java content tree be valid with respect to its original schema in order to marshal it back into XML data. Different JAXB providers will support marshalling invalid Java content trees at varying levels; however, all JAXB Providers must be able to marshal a valid content tree back to XML data. A JAXB Provider must throw a `MarshalException` when it is unable to complete the marshal operation due to invalid content. Some JAXB providers could fully allow marshalling invalid content, others can fail on the first validation error.

3.6.1 Marshalling Properties

The following subsection highlights properties that can be used to control the marshalling process. These properties must be set prior to the start of a marshalling operation: the behavior is undefined if these attributes are altered in the middle of a marshalling operation. The following standard properties have been identified:

- `jaxb.encoding`: output character encoding
- `jaxb.formatted.output`:
  `true` - human readable indented xml data
  `false` - unformatted xml data
- `jaxb.schemaLocation`
  This property allows the client application to specify an `xsi:schemaLocation` attribute in the generated XML data.
- `jaxb.noNamespaceSchemaLocation`
  This property allows the client application to specify an `xsi:noNamespaceSchemaLocation` attribute in the generated XML data.

3.7 Validation Handling

Methods defined in the binding framework can cause validation events to be delivered to the client application's `ValidationEventHandler`. `Setter` methods generated in schema-derived implementation classes are capable of throwing `TypeConstraintExceptions`, all of which are defined in the binding framework.

The following list describes the primary event and constraint-exception classes:

- An instance of a `TypeConstraintException` subclass is thrown when a violation of a dynamically-checked type constraint is detected. Such exceptions will be thrown by property-set methods, for which it would be inconvenient to have to handle checked exceptions; type-constraint exceptions are therefore unchecked, *i.e*, this class extends `java.lang.RuntimeException`. The constraint check is always performed prior to the property-set method updating the value of the property, thus if the exception is thrown, the property is guaranteed to retain the value it had prior to the invocation of the property-set method with an invalid value. This functionality is optional to implement in this version of the specification. Additionally, a customization mechanism is provided to control enabling and disabling this feature.
- An instance of a `ValidationEvent` is delivered whenever a violation is detected during on-demand validation or unmarshal-time validation. Additionally, `ValidationEvents` can be discovered during marshalling such as ID/IDREF violations and print conversion failures. These violations may indicate local and global structural constraint violations, type conversion violations, type constraint violations, etc.
- Since the unmarshal operation involves reading an input document, lexical well-formedness errors may be detected or an I/O error may occur. In these cases, an `UnmarshalException` will be thrown to indicate that the JAXB provider is unable to continue the unmarshal operation.
- During the marshal operation, the JAXB provider may encounter errors in the Java content tree that prevent it from being able to complete. In these cases, a `MarshalException` will be thrown to indicate that the marshal operation can not be completed.

CHAPTER 4

JAVA REPRESENTATION OF XML CONTENT

This section defines the basic binding representation of package, content and element interfaces, properties and typesafe enum class within the Java programming language. Each section briefly states the XML Schema components that could be bound to the Java representation. A more rigorous and thorough description of possible bindings and default bindings occurs in Chapter 5, “Binding XML Schema to Java Representations” and in Chapter 6, “Customization.”

4.1 Mapping between XML Names and Java Identifiers

XML schema languages use *XML names, i.e.*, strings that match the Name production defined in XML 1.0 (Second Edition) to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. Appendix C, “Binding XML Names to Java Identifiers,” specifies an algorithm for mapping XML names to Java identifiers in a way that adheres to standard Java API design guidelines, generates identifiers that retain obvious connections to the corresponding schema, and results in as few collisions as possible. It is necessary to rigorously define a standard way to perform this mapping so all implementations of this specification perform the mapping in the same compatible manner.

4.2 Java Package

Just as the target XML namespace provides a naming context for the named type definitions, named model groups, global element declarations and global attribute declarations for a schema vocabulary, the Java package provides a naming context for Java interfaces and classes. Therefore, it is natural to map the target namespace of a schema to be the package that contains the Java content interfaces representing the structural content model of the document.

A package consists of:

- A *name*, which is either derived directly from the XML namespace URI as specified in Section C.5, “Generating a Java package name” or specified by a binding customization of the XML namespace URI as described in Section 6.6.1.1, “package.”
- A set of Java content interfaces representing the content models declared within the schema.
- A set of Java element interfaces representing element declarations occurring within the schema. Section 5.7.1, “Bind to Java Element Interface” discusses the binding of an element declaration in more detail.
- The class `ObjectFactory` containing:
  - ❍ A public no-arguments constructor.
  - ❍ An instance factory method for each Java content and element interface within the package.

    Given Java content interface named *Foo*, here is the derived factory method:

    ```
    public Foo createFoo() throws JAXBException;
    ```
  - ❍ Dynamic instance factory allocator:

    ```
    public Object newInstance(Class javaContentInterface)
                throws JAXBException;
    ```
  - ❍ Property setter/getter
    Provide the ability to associate implementation specific property/value pairs with the instance creation process.

    ```
    java.lang.Object getProperty(String name)
    void setProperty(String name, Object value)
    ```

- A set of typesafe enum classes.
- Package javadoc.

Example:

Purchase Order Schema fragment with `targetNamespace`:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:po="http://www.example.com/PO1"
              targetNamespace="http://www.example.com/PO1">
   <xs:element name="purchaseOrder" type="po:PurchaseOrderType"/>
   <xs:element name="comment"       type="xs:string"/>
   <xs:complexType name="PurchaseOrderType"/>
   ...
</xs:schema>
```

Default derived Java code:

```
import javax.xml.bind.Element;
package com.example.PO1;
interface PurchaseOrderType { .... };
interface PurchaseOrder extends PurchaseOrderType, Element;
interface Comment { String getValue(); void setValue(String); }
...
class ObjectFactory {
   PurchaseOrderType createPurchaseOrderType();
   PurchaseOrder createPurchaseOrder();
   Comment createComment(String value);
   ...
}
```

4.3 Typesafe Enum Class

A simple type definition whose value space is constrained by an enumeration is worth consideration for binding to a Java typesafe enum class. The typesafe enum design pattern is described in detail in [BLOCH]. To summarize the concept, if an application wishes to refer to the values of a class by descriptive constants and manipulate those constants in a type safe manner, it should consider binding the XML component to a typesafe enum class.

A typesafe enum class consists of:

- A *name*, which is either computed directly from an XML name or specified by a binding customization for the schema component.
- A *package name*, which is either computed from the target namespace of the schema component or specified within a binding declaration as a customization of the target namespace or a specified package name for components that are scoped to no target namespace.
- Outer Class Names is "`.`" separated list of outer class names.

  By default, if the XML component containing a typesafe enum class to be generated is scoped within a complex type as opposed to a global scope, the typesafe enum class should occur as a nested class within the Java content interface representing the complex type scope. Absolute class name is PackageName.[OuterClassNames.]Name. Note: Outer Class Name is null if interface is a top-level interface.
- A set of *enum constants.*
- A set of *enumvalue constants.*
- Class javadoc is a combination of a documentation annotation from the schema component and/or javadoc specified by customization.

An *enum constant* consists of:

- A *name*, which is either computed from the value or specified by customization.
- A *datatype* for the constant.
- A *value* for the constant.
- *Javadoc for the constant field* is a combination of a documentation annotation for an enumeration value facet and/or javadoc specified by customization.

An *enumvalue constant* consists of:

- A *name*, which is either computed from the value or specified by customization.
- A *datatype* for the constant.
- A *value* for the constant.

4.4 Java Content Interface

Complex type definitions are bound to a Java content interface. The attributes and children element content of these schema building blocks are represented as properties of the content interface that are introduced in Section 4.5, "Properties," on page 40.

A Java content interface is defined by:

- A *name*, which is either computed directly from an XML name or specified by a binding customization for the schema component.
- A *package name*, which is either computed from the target namespace of the schema component or specified by a binding customization of the target namespace or a specified package name for components that are scoped to no target namespace.
- The *outer class name* context, a dot-separated list of Java class names.

  By default, if the XML schema component for which a Java context interface is to be generated is scoped within a complex type as opposed to globally, the complex class should occur as a nested class within the Java content interface representing the complex type scope.

  The absolute class name is PackageName.[OuterClassNames.]Name. Note: The OuterClassNames is null if the interface is a top-level interface.
- A base interface that this interface extends. See Section 5.3, "Complex Type Definition," on page 67 for further details.
- A set of Java properties providing access and modification to the attributes and content model represented by the interface.
- A local structural constraint predicate representing all the structural constraints for the content of the class. The constraints include attribute occurrences and local structural constraints detailed in Section 2.2, "Varieties of validation," on page 17.
- Class-level javadoc is a combination of a documentation annotation from the schema component and/or javadoc specified within customization.
- A factory method in the package's `ObjectFactory` class (introduced in Section 4.2, "Java Package"). The factory method returns the type of the Java content interface. The name of the factory method is generated by concatenating the following components:

- ❍ 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.

For example, a Java content interface named `Foo` that is nested within Java content interface `Bar` would have the following factory method signature generated in the containing Java package's `ObjectFactory` class:

```
Bar.Foo createBarFoo()
```

4.5 Properties

The binding compiler binds local schema components to *properties* within a Java content interface.

A property is defined by:

- A *name*, which is either computed from the XML name or specified by a binding customization for the schema component.
- A *base type*, which may be a Java primitive type (*e.g.*, `int`) or a reference type.
- An optional *predicate*, which is a mechanism that tests values of the base type for validity and throws a `TypeConstraintException` if a type constraint expressed in the source schema is violated. [1]
- An optional *collection type*, which is used for properties whose values may be composed of more than one value.
- A *default value*. Schema component has a schema specified default value which is used when property's value is not set and not nil.
- Is *nillable*. A property is nillable when it represents a nillable element declaration.

[1]. Note that it is optional for a JAXB implementation to support type constraint checks when setting a property in this version of the specification.

A property is *realized* by a set of *access methods*. Several property models are identified in the following subsections; each adds additional functionally to the basic set of access methods.

A property's access methods are named in the standard JavaBeans style: the name-mapping algorithm is applied to the property name and then each method name is constructed by prefixing the appropriate verb (`get`, `set`, etc.).

A property is said to have a *set value* if that value was assigned to it during unmarshalling[2] or by invoking its mutation method. The *value* of a property is its *set value*, if defined; otherwise, it is the property's schema specified *default value*, if any; otherwise, it is the default initial value for the property's base type as it would be assigned for an uninitialized field within a Java class[3]. illustrates the states of a JAXB property and the invocations that result in state changes.

4.5.1 Simple Property

A non-collection property `prop` with a base type *Type* is realized by the two methods

```
public Type getId ();
public void setId (Type value);
```

where *`Id`* is a metavariable that represents the Java method identifier computed by applying the name mapping algorithm described in Section C.2, "The Name to Identifier Mapping Algorithm" to prop. There is one exception to this general rule in order to support the boolean property described in [BEANS]. When *Type* is boolean, the `get`*`Id`* method specified above is replaced by the method signature, *boolean* `is`*`Id`*().

- The `get` or `is` method returns the property's value as specified in the previous subsection. If *null* is returned, the property is considered to be absent from the XML content that it represents.

2. An unmarshalling implementation should distinguish between a value from an XML instance document and a schema specified defaulted value when possible. A property should only be considered to have a *set value* when there exists a corresponding value in the XML content being unmarshalled. Unfortunately, unmarshalling implementation paths do exist that can not identify schema specified default values, this situation is considered a one-time transformation for the property and the defaulted value will be treated as a *set value*.

3. Namely, a `boolean` field type defaults to `false`, `integer` field type defaults to `0`, object reference field type defaults to `null`, floating point field type defaults to `+0.0f`.

- The `set` method defines the property's *set value* to be the argument `value`. If the argument value is `null`, the property's *set value* is discarded. Prior to setting the property's value when TypeConstraint validation is enabled[4], a non-`null` value is validated by applying the property's predicate. If `TypeConstraintException` is thrown, the property retains the value it had prior to the `set` method invocation.

When the base type for a property is a primitive non-reference type, the corresponding Java wrapper class can be used as the base type to enable discarding the property's set value by invoking the set method with a null parameter. See Section 4.5.4, "isSet Property Modifier," on page 46 for an alternative to using a wrapper class for this purpose.

Example

In the purchase order schema, the `partNum` attribute of the `item` element definition is declared:

```
<xs:attribute name="partNum" type="SKU" use="required"/>
```

This element declaration is bound to a simple property with the base type `java.lang.String`:

```
public String getPartNum();
public void setPartNum(String x);
```

The `setPartNum` method could apply a predicate to its argument to ensure that the new value is legal, *i.e.*, that it is a string value that complies with the constraints for the simple type definition, SKU, and that derives by restriction from `xs:string` and restricts the string value to match the regular expression pattern `"\d{3}-[A-Z]{2}"`.

It is legal to pass `null` to the `setPartNum` method even though the `partNum` attribute declaration's attribute `use` is specified as required. The determination if `partNum` content actually has a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

4. Note that it is optional for a JAXB implementation to support type constraint checks when setting a property in this version of the specification.

4.5.2 Collection Property

A collection property may take the form of an *indexed property* or a *list property*. The base type of an indexed property may be either a primitive type or a reference type, while that of a list property must be a reference type.

4.5.2.1 Indexed Property

This property follows the indexed property design pattern for a multi-valued property from the JavaBean specification. An indexed property *prop* with base type *Type* is realized by the five methods

```
public Type [] getId();
public void setId (Type [] value);
public void setId(int index, Type value);
public Type getId(int index);
public int getIdLength();
```

regardless of whether `Type` is a primitive type or a reference type. `Id` is computed from `prop` as it was defined in simple property.

- `getId()`
  The array `getter` method returns an array containing the property's value. If the property's value has not set, then `null` is returned.
- `setId(Type [])`
  The `array setter` method defines the property's set value. If the argument itself is `null` then the property's set value, if any, is discarded. If the argument is not `null` and `TypeConstraint` validation is enabled [5] then the sequence of values in the array are first validated by applying the property's predicate, which may throw a `TypeConstraintException`. If the `TypeConstraintException` is thrown, the property retains the value it had prior to the `set` method invocation. The property's value is only modified after the `TypeConstraint` validation step.
- `setId(int, Type)`
  The indexed `set`ter method allows one to set a value within the array. The runtime exception `java.lang.ArrayIndexOutOfBoundsException`

5. Note that it is optional for a JAXB implementation to support type constraint checks when setting a property in this version of the specification.

may be thrown if the index is used outside the current array bounds. If the value argument is non-null and TypeConstraint validation is enabled[5], the value is validated against the property's predicate, which may throw an unchecked `TypeConstraintException`. If `TypeConstraintException` is thrown, the array index remains set to the same value it had before the invocation of the indexed `setter` method.

- `get`*`Id`*`(int)`
  The indexed `get`ter method returns a single element from the array. The runtime exception `java.lang.ArrayIndexOutOfBoundsException` may be thrown if the index is used outside the current array bounds. In order to change the size of the array, you must use the array set method to set a new (or updated) array.
- `get`*`Id`*`Length()`
  The indexed length method returns the length of the array. This method enables you to iterate over all the items within the indexed property using the indexed mutators exclusively. Exclusive use of indexed mutators and this method enable you to avoid the allocation overhead associated with array `getter` and `setter` methods.

The arrays returned and taken by these methods are not part of the content object's state. When an array `getter` method is invoked, it creates a new array to hold the returned values. Similarly, when the corresponding array `setter` method is invoked, it copies the values from the argument array.

To test whether an indexed property has a set value, invoke its `array getter` method and check that the result is not `null`. To discard an indexed property's set value, invoke its array `setter` method with an argument of `null`.

See the customization attribute `collectionType` in Section 6.5, "<globalBindings> Declaration" and Section 6.8, "<property> Declaration" on how to enable the generation of indexed property methods for a collection property.

Example

In the purchase order schema, we have the following repeating element occurrence of element item within `complexType` Items.

```
<xs:complexType name="Items">
   <xs:sequence>
```

```
          <xs:element name="item" minOccurs="1"maxOccurs="unbounded">
             <xs:complexType>...</xs:complexType>
          </xs:element>
   </xs:complexType>
```

The content specification of this element type could be bound to an array property realized by these four methods:

```
public Items.ItemType[] getItem();
public void setItem(Items.ItemType[] value);
public void setItem(int index, Items.ItemType value);
public Items.ItemType getItem(int index);
```

4.5.2.2 List Property

A list property *prop* with base type *Type* is realized by the method where `List`

```
public List getId();
```

is the interface `java.util.List`, *`Id`* is defined as above.

- The `get` method returns an object that implements the `List` interface, is mutable, and contains the values of type *Type* that constitute the property's value. If the property does not have a set value or a schema default value, a zero length `java.util.List` instance is returned.

The `List` returned by the `get` method is a component of the content object's state. Modifications made to this list will, in effect, be modifications to the content object. If `TypeConstraint` validation is enabled, the list's mutation methods apply the property's predicate to any non-`null` value before adding that value to the list or replacing an existing element's value with that value; the predicate may throw a `TypeConstraintException`.

The `unset` method introduced in Section 4.5.4, "isSet Property Modifier," on page 46 enables one to discard the set value for a List property.

Design Note – A future version of the Java programming language may support generic types, in which case this specification may be revised so that list-retrieval methods have the type `List<Type>`.

Example

The content specification of the `item` element type could alternatively be bound to a list property realized by one method:

```
public List getItem();
```

The list returned by the `getItem` method would be guaranteed only to contain instances of the `Item` class. As before, its length would be checked only during validation, since the requirement that there be at least one `item` in an element instance of complex type definition `Items` is a structural constraint rather than a type constraint.

4.5.3 Constant Property

An attribute use named *prop* with a schema specified fixed value can be bound to a Java constant value. *Id* is computed from `prop` as it was defined in simple

```
        static final public Type ID = <fixedValue>;
```

property. The value of the fixed attribute of the attribute use provides the *`<fixedValue>`* constant value.

The binding customization attribute `fixedAttributeToConstantProperty` enables this binding style. Section 6.5, "<globalBindings> Declaration" and Section 6.8, "<property> Declaration" describe how to use this attribute.

4.5.4 `isSet` Property Modifier

This optional modifier augments a modifiable property to enable the manipulation of the property's value as a *set value* or a *defaulted value*. Since this functionality is above and beyond the typical JavaBean pattern for a property, the method(s) associated with this modifier are not generated by default. Chapter 6, "Customization" describes how to enable this customization using the `generateIsSetMethod` attribute.

The method signatures for the `isSet` property modifier are the following:

```
public boolean isSetId();
```

where *`Id`* is defined as it was for simple and collection property.

- The `isSet` method returns `true` if the property has been set during unmarshalling or by invocation of the mutation method `setId` with a non-`null` value. [6]

  To aid the understanding of what `isSet` method implies, note that the unmarshalling process only unmarshals *set values* into XML content.

A list property and a simple property with a non-reference base type require an additional method to enable you to discard the *set value* for a property:

```
public void unsetId();
```

- The `unset` method marks the property as having no *set value*. A subsequent call to `getId` method returns the schema-specified default if it existed; otherwise, it returns the Java default initial value for `Type`.

All other property kinds rely on the invocation of their set method with a value of null to discard the set value of its property. Since this is not possible for primitive types or a List property, the additional method is generated for these cases. illustrates the method invocations that result in transitions between the possible states of a JAXB property value.

[6] A Java application does not need to distinguish between the absence of a element from the infoset and when the element occurred with nil content. Thus, in the interest of simplifying the generated API, methods were not provided to distinguish between the two. The marshalling process should always output an element with nil content for a property that is not set and it represents a required nillable element declaration.

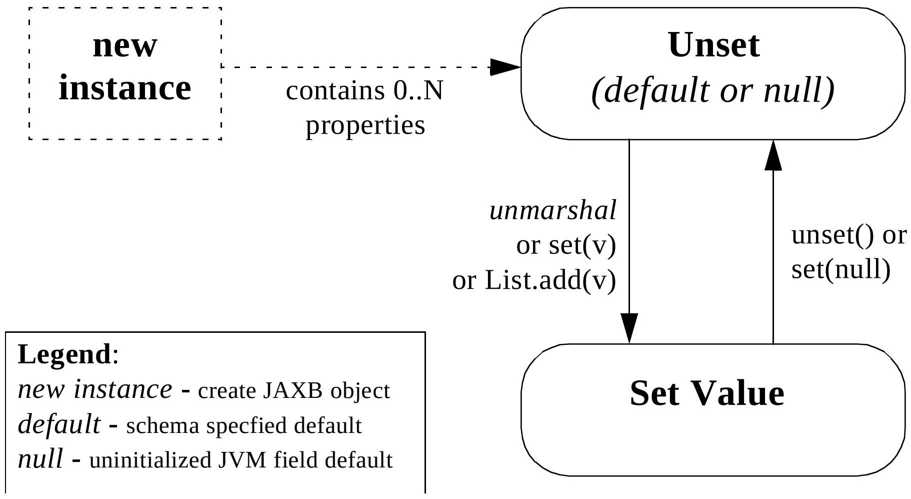
new instance
contains 0..N properties
Unset (default or null)
unmarshal or set(v) or List.add(v)
unset() or set(null)
Set Value
Legend:
new instance - create JAXB object
default - schema specfied default
null - uninitialized JVM field default

Figure 4.1 States of a Property Value

Example

In the purchase order schema, the `partNum` attribute of the element `item`'s anonymous complex type is declared:

```
<xs:attribute name="partNum" type = "SKU" use="required"/>
```

This attribute could be bound to a `isSet` simple property realized by these four methods:

```
public int getPartNum();
public void setPartNum(String skuValue);
public boolean isSetPartNum();
public void unsetPartNum();
```

It is legal to invoke the `unsetPartNum` method even though the attribute's `use` is `“required”` in the XML Schema. That the attribute actually has a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

4.5.5 Property Summary

The following core properties have been defined:

- Simple property - JavaBean design pattern for single value property
- Indexed property - JavaBean design pattern for multi-valued property
- List property - Leverages java.util.Collection
- Constant property

The methods generated for these four core property kinds are sufficient for most applications. Configuration-level binding schema declarations enable an application to request finer control than provided by the core properties. For example, the `isSet` property modifier enables an application to determine if a property's value is set or not.

4.6 Java Element Interface

Based on criteria to be identified in Section 5.7.1, "Bind to Java Element Interface," on page 82, the binding compiler binds an element declaration to a Java element interface. An element interface is defined as:

- An interface name is generated from the element declaration's name using the XML Name to Java identifier name mapping algorithm specified in Section C.2, "The Name to Identifier Mapping Algorithm," on page 189.
- If the element declaration's type definition is a:
  - Complex Type definition

    The element interface extends the Java content interface representing the complex type definition of the element declaration
  - Simple type definition

    The generated element interface has a Java property named "`value`".

    The factory method within the package's `ObjectFactory` method to create an instance of the element takes a value parameter of the Java class binding of the simple type definition.
- Scope of element class
  - Global element declarations are declared in package scope.
  - Local element declarations occur in the scope of the first ancestor complex type definition that contains the declaration.

- Each generated Element interface must extend the Java marker interface `javax.xml.bind.Element`. This enables JAXB implementations to differentiate between instances representing an XML element directly and instances representing the type of the XML element.
- A factory method is generated in the package's `ObjectFactory` class introduced in Section 4.2, "Java Package." The factory method returns the type of the Java element interface. The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.
  - If the Java element interface is nested within another interface, then the concatenation of all outer Java class names.
  - The *name* of the Java content interface.

  For example, a Java element interface named Foo that is nested within Java content interface Bar would have the following factory method generated in the containing Java package's `ObjectFactory` class:

  ```
  Bar.Foo createBarFoo()
  ```
- The optional methods `setNil()` and `isNil()` enable Element instances to be set to the XML concept of `nil` and to check whether an Element instance is `nil.` See Section 5.7.1, "Bind to Java Element Interface," on page 82 for details on when these methods are generated.

Example 1:

Given global XML Schema element declaration with a complex type definition:

```
<xs:complexType name="AComplexType">
   <xs:sequence>
      <xs:element name="A" type="xs:int"/>
      <xs:element name="B" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
<xs:element name="AnElement" type="AComplexType"/>
```

Its Java binding looks like this:

```
public interface AComplexType {
   void setA(int value);
   int getA();
   void setB(String value);
   String getB();
};
public interface AnElement extends
                         AComplexType, javax.xml.bind.Element {};
public class ObjectFactory {
   AnElement     createAnElement();
   AComplexType  createAComplexType();
   ... other factory methods ...
}
```

Example 2:

Given local XML Schema element declaration with a simple type definition:

```
<xs:complexType name="AComplexType" mixed=”true”>[7]
   <xs:sequence>
      <xs:element name="ASimpleElement" type="xs:int"/>
   </xs:sequence>
</xs:complexType>
```

Its Java representation:

```
public interface AComplexType {
   public interface ASimpleElement extends javax.xml.bind.Element {
      void setValue(int value);
      int getValue();
   }
   ...
```

7. Assume that this schema fragment meets one of the criteria specified in Section 5.7.1, “Bind to Java Element Interface,” on page 82 that requires that `<ASimpleElement>` element be bound to a Java element interface.

```
};
class ObjectFactory {
    AComplexType createAComplexType();
    AComplexType.ASimpleElement
        createAComplexTypeASimpleElement(int value);
    AComplexType.ASimpleElement
        createAComplexTypeASimpleElement();
    ...
}
```

4.7 Summary

The composition and relationships between the Java components introduced in this section are reflected in the following diagram.

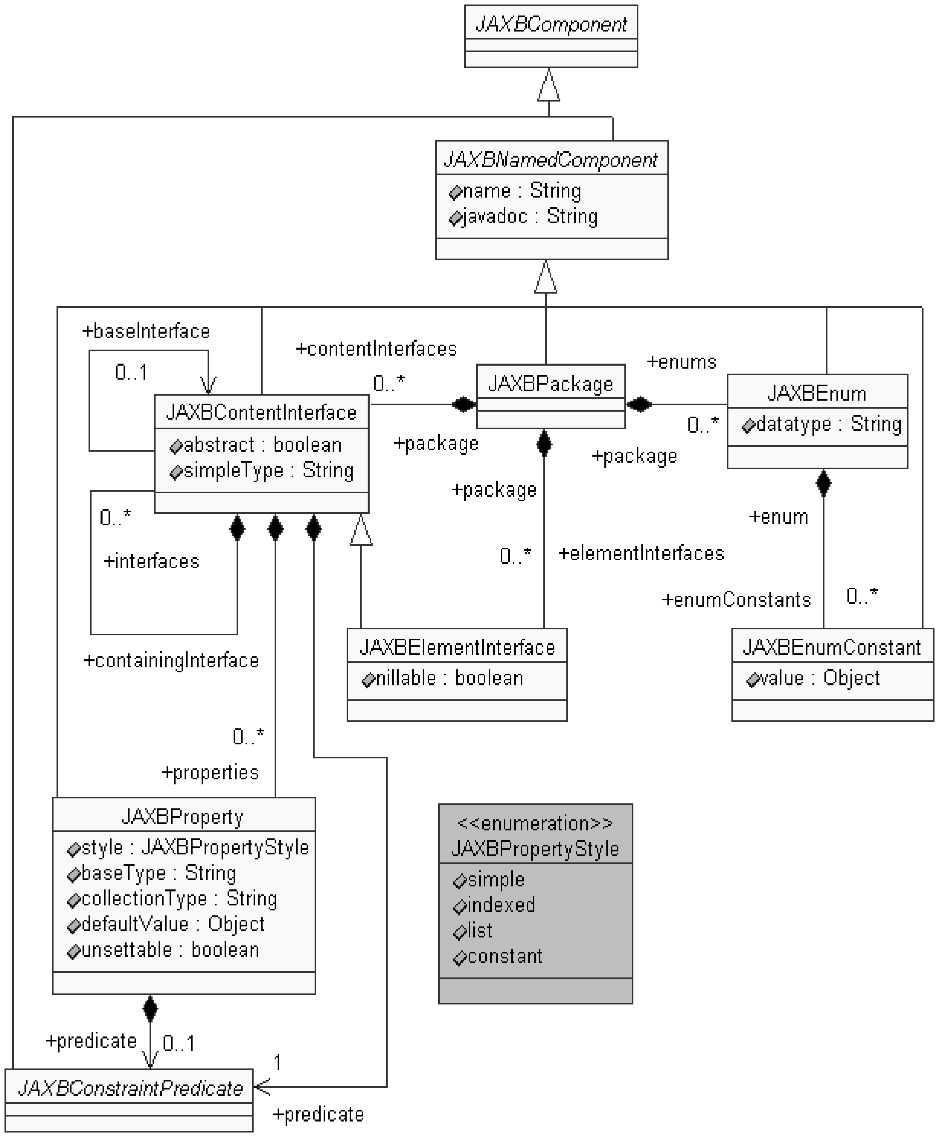

Figure 4.2 UML Diagram of Java representation

CHAPTER 5

BINDING XML SCHEMA TO JAVA REPRESENTATIONS

This chapter describes the default behavior for binding a subset of XML schema components to Java. Unsupported XML Schema components are described in Appendix E.2. The next chapter specifies how to customize the default behavior.

5.1 Overview

The abstract model described in [XSD Part 1] is used to discuss the default binding of each schema component type. Each schema component is described as a list of properties and their semantics. References to properties of a schema component as defined in [XSD Part 1] are denoted using the notation *{schema property}* throughout this section. References to properties of information items as defined in [XML-Infoset] are denoted the notation **[property]**.

All JAXB implementations are required to implement the default bindings specified in this chapter. However, users and JAXB implementors can use the global configuration capabilities of the custom binding mechanism to override the defaults in a portable manner.

All examples are non-normative.

5.2 Simple Type Definition

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) are specified here and 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.

5.2.1 Type Categorization

The simple type definitions can be categorized as:

- schema built-in datatypes [XSD PART2]
- user-derived datatypes

Conceptually, there is no difference between the two. A schema built-in datatype can be a primitive datatype. But it can also, like a user-derived datatype, be derived from a schema built-in datatype. Hence no distinction is made between the schema built-in and user-derived datatypes.

The specification of simple type definitions is based on the abstract model described in Section 4.1, “Simple Type Definition” [XSD PART2]. The abstract model defines three varieties of simple type definitions: atomic, list, union. The Java property attributes for each of these are described next.

5.2.2 Atomic Datatype

If an atomic datatype has been derived by restriction using an “enumeration” facet, the Java property attributes are defined by Section 5.2.3, “Type Safe Enumeration.” Otherwise they are defined as described here.

The base type is derived upon the XML built-in type hierarchy [XSD PART2, Section 3] reproduced below.

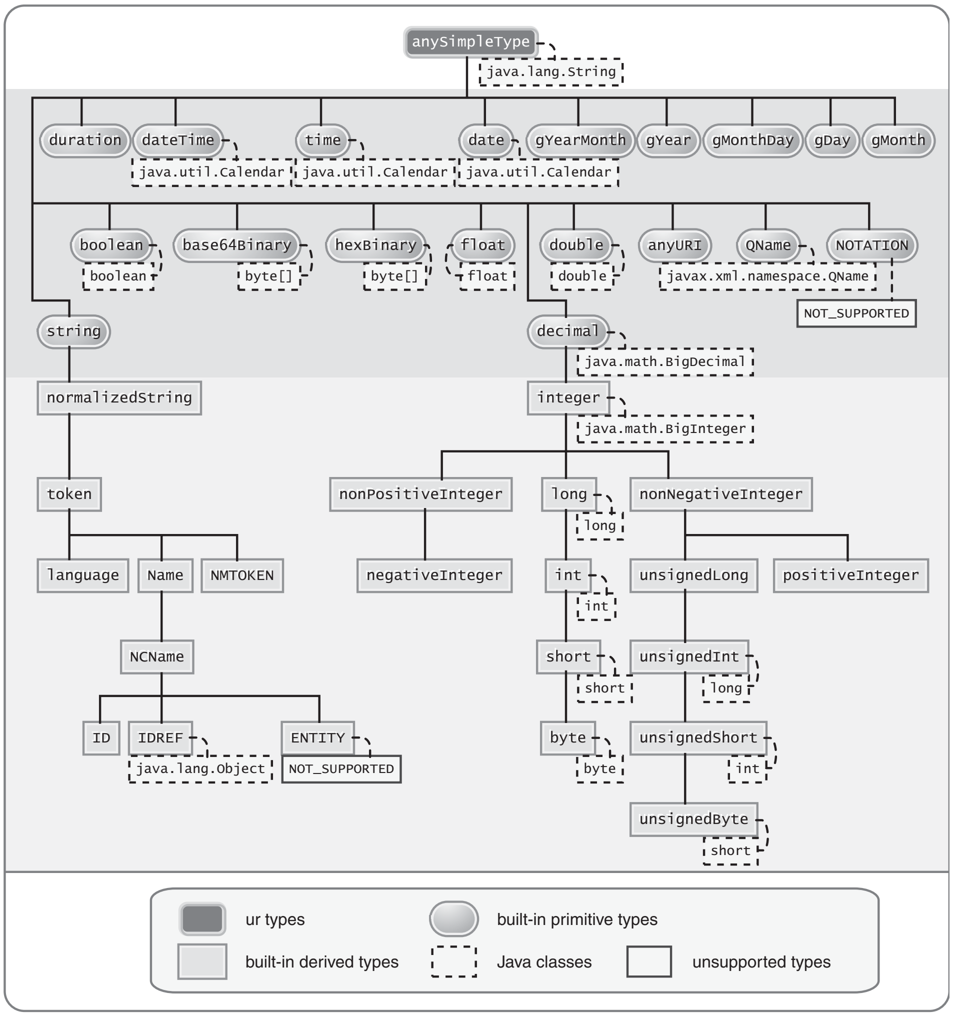

Figure 5.1 XML Built-In Type Hierarchy

The above diagram is the same as the one in [XSD PART2] except for the following:

- Only schema built-in atomic datatypes derived by restriction have been shown.
- The schema built-in atomic datatypes have been annotated with Java data types from the “Java Mapping for XML Schema Built-in Types” table below.

The following is a mapping for subset of the XML schema built-in data types to Java data types. This table is used to specify the base type later.

Table 5-1 Java Mapping for XML Schema Built-in Types

XML Schema Data 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	**java.util.Calendar**
xsd:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:time	java.util.Calendar
xsd:date	java.util.Calendar
xsd:anySimpleType	java.lang.String

The mapping shown in the table above is aligned with the default mapping of XML schema built-in atomic datatypes in [JAX-RPC]. These are indicated in bold in the above table. In addition, it also defines mappings for datatypes not specified in [JAX-RPC].

The base type is determined as follows:

1. If a mapping is defined for the simple type in Table 5.1, the base type defaults to its defined Java datatype.
2. Otherwise, the base type must be the result obtained by repeating the step 1 using the *{base type definition}*. For schema datatypes derived by

restriction, the *{base type definition}* represents the simple type definition from which it is derived. Therefore, repeating step 1 with *{base type definition}* essentially walks up the XML Schema built-in type hierarchy until a simple type definition which is mapped to a Java datatype is found.

The simple type definition `xsd:anySimpleType` is always mapped to `java.lang.String`. Since all XML simple types are derived from `xsd:anySimpleType`, a mapping for a simple type definition to `java.lang.String` is always guaranteed.

The Java property predicate must be as specified in “Simple Type Definition Validation Rules,” Section 4.1.4[XSD PART2].

Example:

The following schema fragment (taken from Section 4.3.1, “Length” [XSD PART2]):

```
<xs:simpleType name="productCode">
   <xs:restriction base="xs:string">
      <xs:length value="8" fixed="true"/>
   </xs:restriction>
</xs:simpleType>
```

The facet “length” constrains the length of a product code (represented by `productCode`) to 8 characters (see section 4.3.1 [XSD PART2] for details).

The Java property attributes corresponding to the above schema fragment are:

- There is no Java datatype mapping for `productCode.` So the Java datatype is determined by walking up the built-in type hierarchy.
- The `{base type definition}` of `productCode` is `xs:string. xs:string` is mapped to `java.lang.String` (as indicated in the table, and assuming no customization). Therefore, `productCode` is mapped to the Java datatype `java.lang.String`.
- The predicate enforces the constraints on the length.

5.2.3 Type Safe Enumeration

A named atomic type that is derived by restriction with enumeration facet(s) and whose restriction base type (represented by *{base type definition}*) is “`xsd:NCName`” or derived from it must be mapped to a typesafe enum class.

The **`[typesafeEnumBase]`** attribute customization described in Section 6.5, “<globalBindings> Declaration," enables global configuration so named atomic types derived from other restriction base types are bound by default to typesafe enumeration class. An anonymous simple type definition is never bound to a typesafe enum class by default, but it can be customized as described in Section 6.10, “<typesafeEnum> Declaration” to bind to a typesafe enum class.

The default binding described here is technically aligned with JAX-RPC specified typesafe enumeration binding but there are a few differences that are discussed in Section F.3, “Bind XML enum to a typesafe enumeration.”

5.2.3.1 Enumeration Class

A type safe enum class must be defined as specified here. An example is provided first followed by a more formal specification.

XML Schema fragment:

```
<xs:simpleType name="USState">
   <xs:restriction base="xs:NCName">
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
   </xs:restriction>
</xs:simpleType>
```

The corresponding typesafe enum class is:

```
public class USState {
   // Constructor
   protected USSate(String value) { ... }
   // one enumeration constant for each enumeration value
   public static final String _AK="AK";
   public static final USState AK= new USState(_AK);
   public static final String _AL="AL";
   public static final USState AL= new USState(_AL);
   // Gets the value for an enumerated value
   public String getValue();
   // Gets enumeration with a specific value
   // Required to throw java.lang.IllegalArgumentException if
   // any invalid value is specified
   public static USState fromValue(String value) {...}

   // Gets enumeration from a String
   // Required to throw java.lang.IllegalArgumentException if
   // any invalid value is specified
   public static USState fromString(String value){ ... }
   // Returns String representation of the enumerated value
   public String toString() { ... }
   public boolean equals(Object obj) { ... }
   public int hashCode() { ... }
}
```

5.2.3.2 Enumeration Class

The enumeration class is defined as follows:

- **name:** The default name of the enumeration class, *enumClassName,* is computed by applying the XML Name to Java identifier mapping algorithm to the *{name}* of the simple type definition. There is no mechanism to derive a name for an anonymous simple type definition, the customization must provide the **name**.
- **package name:** The package name is determined from the *{targetname space}* of the schema that directly contains the simple type definition.
- **outer class name:**
  - There is no **outer class name** for a global simple type definition.
  - The **outer class name** for an anonymous simple type definition is computed by traversing up the anonymous simple type definition's ancestor

tree until the first ancestor is found that is:
- an XML component that is mapped to a Java content interface, the **outer class name** is composed of the concatenation of this content interface's **outer class name**, ".", and its **name**.
- a global declaration or definition is reached. There is no **outer class name** for this case.

Example:

```
public class USState { ... } // Enumeration class
```

5.2.3.3 Constant Fields

For each enumeration value (represented by schema property {*value*}, there are two public, static and final constant fields in the enumeration class: *enumvalue constant* and *enum constant.*

An **enumvalue constant set** contains a enum constant for each enumeration value. Each member of the set is defined as follows:

- **name:** A name is computed as specified in Section 5.2.3.4, "XML Enumvalue-to-Java Identifier Mapping" and prefixing it with an underscore ('_').
- **type:** The Java type binding of the XML datatype, {*base_type_definition*}.
- **value:** The value is {*value*}.

An **enum constant set** contains an enum constant for each enumeration value. Each member of the set is defined as follows:

- **name:** a name that is computed as specified in Section 5.2.3.4, "XML Enumvalue-to-Java Identifier Mapping."
- **type:** The type is *enumClassName.*
- **value:** value is an instance of *enumClassName* constructed with a {*value*}. The instance is unique except in the following case.
  XSD PART 2 permits identical enumeration values to be specified in an XML enumeration. In that case, the enum constant name cannot be uniquely by default. Instead, an error must be reported.

Example:

```
public static final String _AK="AK";// enumvalue constant
public static final USState AK= new USState(_AK); // enumeration constant
```

5.2.3.4 XML Enumvalue-to-Java Identifier Mapping

Default names for enumvalue constant and enum constant are based on mapping of the XML enumeration value to a Java identifier described here.

An attempt is made to map the XML enumeration value *`{value}`* to a Java Identifier using the XML Name to Java Identifier algorithm. If one or more enumerated values in an XML enumeration cannot map to valid Java identifier (examples are “3.14”, “int”) or there is a collision among the generated constant fields **name**, then the result is determined as follows:

- If the customization option *`typesafeEnumMemberName`* is specified and set to “*`generateError`*,” an error must be reported. This is also the default behavior if *`typesafeEnumMemberName`* has not been specified.
- If the customization option, element `<jaxb:globalBindings>` **`[typesafeEnumMemberName]`** is set to the value “*`generateName,”`* the constant fields **name** is "*`VALUE_<N>"`* where *<N>* is 1 for the first enumeration value and increments by 1 to represent each value within the XML enumeration.

5.2.3.5 Methods and Constructor

`Type` is defined in enumvalue constant set in Section 5.2.3.3, “Constant Fields.” `enumClassName` is defined in Section 5.2.3.2, “Enumeration Class.”

There are three accessor methods: *getValue, fromValue* and *fromString*.

```
public Type getValue()
public enumClassName fromValue(Type value)
public enumClassName fromString(String value)
```

The *fromValue* and *fromString* method must throw a *`java.lang.IllegalArgumentException`* if *`value`* is not one of the enumeration values specified in the XML enumeration datatype.

The constructor must be declared protected as shown below:

```
protected enumClassName(Type value) { ... }
```

An enumeration class must contain the following methods which override the object methods:

```
public String toString() { ... }
public final boolean equals(Object obj) { ... }
public final int hashCode() { ... }
```

The `equals()` and `hashCode()` must be final and must invoke the `Object` methods. This ensures that no subclass of typesafe enumeration class accidentally overrides theses methods. This in turn guarantees that two equal objects of the enumeration class are also identical. [BLOCH]

5.2.4 List

A list simple type definition can only contain list items of atomic or union datatypes. The item type within the list is represented by the schema property *{item type definition}*.

The Java property attributes for a list simple type definition are:

- The *base type* is derived from the *{item type definition}* as follows. If the Java datatype for *{item type definition}* is a Java primitive type, then the base type is the wrapper class for the Java primitive type. Otherwise, the Java datatype is derived from the XML datatype as specified in Section 5.2.2, “Atomic Datatype” and Section 5.2.3, “Type Safe Enumeration.”
- The *collection type* defaults to an implementation of java.util.List. Note that this specification does not specify the default implementation for the interface java.util.List, it is implementation dependent.
- The *predicate* is derived from the “Simple Type Definition Validation Rules,” in section 4.1.4,[XSD PART2].

Example:

For the following schema fragment:

```
<xs:simpleType name="xs:USStateList">
   <xs:list itemType="xs:string"/>
</xs:simpleType>
```

The corresponding Java property attributes are:

- The *base type* is derived from *{item type definition}* which is XML datatype, *“xs:string”*, thus the Java datatype is java.util.String as specified in Section Table 5-1, “Java Mapping for XML Schema Built-in Types.”

- The *collection type* defaults to an implementation of java.util.List.
- The *predicate* only allows instances of *base type* to be inserted into the list. When failfast check is being performed[1], the list's mutation methods apply the property's predicate to any non-`null` value before adding that value to the list or replacing an existing element's value with that value; the predicate may throw a `TypeConstraintException`.

5.2.5 Union Property

A union property *prop* is used to bind a union simple type definition schema component. A union simple type definition schema component consists of union members which are schema datatypes. A union property, is therefore, realized by:

```
public Type getId();
public void setId(Type value);
```

where `Id` is a metavariable that represents the Java method identifier computed by applying the name mapping algorithm described in Section C.2, "The Name to Identifier Mapping Algorithm," on page 189 to *prop*.

The *base type* is the first common supertype of all the Java representations to which union member types are bound with `java.lang.Object` always being a common root for all Java objects. For member types that are derived by list, the *{item type}* of the list is used for determining the base type. For the purposes of determining the supertype, if a union member that is bound to a Java primitive type, the corresponding Java wrapper class is used instead. If none of the union member types are derived by list, then `Type` is *base type*. If one of the member types is derived by list, then the Union property is represented as the appropriate collection property as specified by the customization `<jaxb:globalBindings>` **@collectionType** value, specified in Section 6.5.1, "Usage."

- The `getId` method returns the set value. If the property has no set value then the value `null` is returned. The value returned is an instance of one of the union member types.

[1] Section 6.5.1, "Usage" describes the `enableFailFastCheck` customization and `Section 2.2, "Varieties of validation`" defines fail-fast checking.

- The `setId` method sets the set value. The `value` is mapped to the appropriate union member type by a JAXB technology implementation. A union schema component does not have a tag to distinguish between union member types. However, [XSD PART2] does specify the order of evaluation for a given value. Thus, for the following example,

  ```
  <xs:union memberTypes="xs:integer xs:string"/>
  ```

  the order of evaluation specified by [XSD PART2] is first “`integer`” and then “`string`”.

  The order of evaluation specified by [XSD PART2] must be followed by a JAXB implementation to map a `value` to the appropriate union member type.

  If value is `null`, the property’s *set value* is discarded. Prior to setting the property’s value when TypeConstraint validation is enabled, a non-`null` value is validated by applying the property’s predicate, which may throw a `TypeConstraintException.`

Example: Default Binding: Union

The following schema fragment:

```
<xs:complexType name="CTType">
   <xs:attribute name="state" type="ZipOrName"/>
</xs:complexType>
<xs:simpleTypename="ZipOrName"
              memberTypes="xs:integer xs:string"/>
```

is bound to the following Java representation.

```
public interface CTType {
   Object getState();
    void setState(Object value);
}
```

5.2.6 Union

A simple type definition derived by a union is bound using the union property with the following Java property attributes:

- the *base type* as specified in Section 5.2.5, “Union Property.”

- if one of the member types is derived by `<xs:list>`, then the union is bound as a Collection property.
- The *predicate* is the schema constraints specified in “Simple Type Definition Validation Rules,” Section 4.1.4 [XSD PART2].

5.3 Complex Type Definition

5.3.1 Aggregation of Java Representation

A Java representation for the entire schema is built based on aggregation. A schema component aggregates the Java representation of all the schema components that it references. This process is done until all the Java representation for the entire schema is built. Hence a general model for aggregation is specified here once and referred to in different parts of the specification.

The model assumes that there is a schema component *SP* which references another schema component *SC*. The Java representation of *SP* needs to aggregate the Java representation of *SC*. There are two possibilities:

- *SC* is bound to a property set.
- *SC* is bound to a Java datatype or a Java interface.

Each of these is described below.

5.3.1.1 Aggregation of Datatype/Interface

If a schema component *SC* is bound to a Java datatype or a Java interface, then *SP* aggregates *SC’s* Java representation as a simple property defined by:

- **name:** the name is the interface name or the Java datatype or a name determined by SP. The name of the property is therefore defined by the schema component which is performing the aggregation.
- **base type:** If SC is bound to a Java datatype, the base type is the Java datatype. If SC is bound to a Java interface, then the base type is the interface name, including a dot separated list of interface names within which SC is nested.
- **collection type:** There is no collection type.

- **predicate:** There is no predicate.

5.3.1.2 Aggregation of Property Set

If *SC* is bound to a property set, then *SP* aggregates by adding *SC's* property set to its own property set.

Aggregation of property sets can result in name collisions. A name collision can arise if two property names are identical. A binding compiler must generate an error on name collision. Name collisions can be resolved by using customization to change a property name.

5.3.2 Java Content Interface

The binding of a complex type definition to a Java content interface is based on the abstract model properties in Section E.1.3, "Complex Type Definition Schema Component," on page 204. The Java content interface must be defined as specified here.[2]

- **name:** name is the Java identifier obtained by mapping the XML name *`{name}`* using the name mapping algorithm, specified in Section C.2, "The Name to Identifier Mapping Algorithm," on page 189. For anonymous complex type definitions, see Section 5.7.2, "Binding of an anonymous complex type definition" for the specification of how a **name** value is derived from its parent element declaration.
- **package:**
  - For a global complex type definition, thederived Java content interface is generated into the Java package that represents the binding of *{target namespace}*
  - For the value of **package** for an anonymous complex type definition, see Section 5.7.2, "Binding of an anonymous complex type definition".
- **outer class name:**
  - There is no outer class name for a global complex type definition.
  - Section 5.7.2, "Binding of an anonymous complex type definition"

[2] Note that Section 5.7.2, "Binding of an anonymous complex type definition" defines the name and package property for anonymous type definitions occurring within an element declaration.

defines how to derive this property from the element declaration that contains the anonymous complex type definition.

- **base interface:** A complex type definition can derive by restriction or extension (i.e. *{derivation method}* is either "extension" or "restriction"). However, since there is no concept in Java programming similar to restriction, both are handled the same. If the *{base type definition}* is itself mapped to a Java content interface (Ci2), then the base interface must be Ci2. This must be realized as:

  ```
  public interface Ci1 extends Ci2 {
      .....
  }
  ```

  See example of derivation by extension at the end of this section.

- **property set:** The Java representation of each of the following must be aggregated into Java content interface's property set (Section 5.3.1, "Aggregation of Java Representation").
  - A subset of {attribute uses} is constructed. The subset must include the schema attributes corresponding to the `<xs:attribute>` children and the {attribute uses} of the schema attribute groups resolved by the <ref> attribute. Every attribute's Java representation (Section 5.8, "Attribute use") in the set of attributes computed above must be aggregated.
  - The Java representation for *{content type}* must be aggregated.

    For a "Complex Type Definition with complex content," the Java representation for *{content type}* is specified in Section 5.9, "Content Model - Particle, Model Group, Wildcard."
    For a complex type definition which is a "Simple Type Definition with simple content," the Java representation for *{content type}* is specified in Section 5.3.2.1, "Simple Content Binding."
  - If a complex type derives by restriction, there is no requirement that Java properties representing the attributes or elements removed by the restriction need to be disabled. This is because (as noted earlier), derivation is handled the same as derivation by restriction.

Example: Complex Type: Derivation by Extension

XML Schema Fragment (from XSD PART 0 primer):

```
<xs:complexType name="Address">
   <xs:sequence>
      <xs:element name="name"   type="xs:string"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city"   type="xs:string"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name="USAddress">
   <xs:complexContent>
      <xs:extension base="ipo:Address">
         <xs:sequence>
            <xs:element name="state" type="xs:string"/>
            <xs:element name="zip"   type="xs:integer"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Default Java binding:

```
public interface Address {
   String getName();
   void   setName(String);
   String getStreet();
   void   setStreet(String);
   void   getCity();
   void   setCity(String);
}

import java.math.BigInteger;
public interface USAdress extends Address {
   String getState();
   void   setState(String);
   BigInteger getZip();
   void   setZip(BigInteger);
}
```

5.3.2.1 Simple Content Binding

Binding to Property

By default, a complex type definition with simple content is bound to a Java property defined by:

- **name:** The property name must be "`value`".
- **base type, predicate, collection type:** As specified in [XSD Part 1], when a complex type has simple content, the content type (*{content type}*) is always a simple type schema component. And a simple type component always maps to a Java datatype (Section 5.2, "Simple Type Definition"). Values of the following three properties are copied from that Java type:
    - ❍ base type
    - ❍ predicate
    - ❍ collection type

Example: Simple Content: Binding To Property

XML Schema fragment:

```
<xs:complexType name="internationalPrice">
   <xs:simpleContent>
      <xs:extension base="xs:decimal">
         <xs:attribute name="currency" type="xs:string"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

Default Java binding:

```
interface InternationalPrice {
   /** Java property for simple content */
   java.math.BigDecimal getValue();
   void setValue(java.math.BigDecimal value);

   /** Java property for attribute*/
   String getCurrency();
   void setCurrency(String);
}
```

5.4 Attribute Group Definition

There is no default mapping for an attribute group definition. When an attribute group is referenced, each attribute in the attribute group definition becomes a part of the *[attribute uses]* property of a complex type definition. Each attribute is mapped to a Java property as described in (section, “Attribute Use”).

5.5 Model Group Definition

By default, a model group definition is not bound to a Java content interface. Rather, when a named model group is referenced, the JAXB property set representing its content model is aggregated into the Java content interface representing the complex type definition that referenced the named model group definition as illustrated in Section Figure 5.2, “Default binding for a reference to a model group definition.”

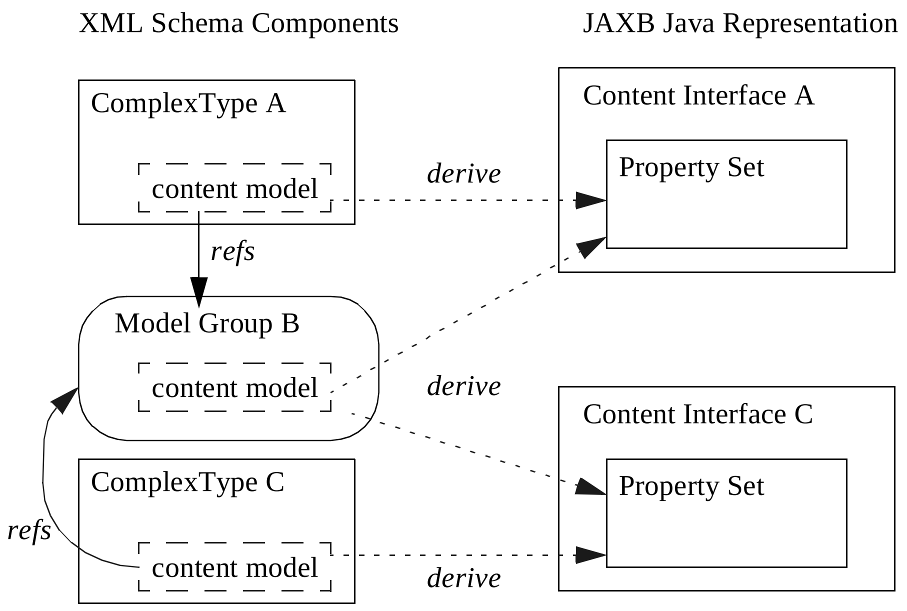

Figure 5.2 Default binding for a reference to a model group definition.

This default binding style results in the same properties occurring within both Java content interface’s A and C to represent the referenced Model Group B’s content model. However, this shared set of methods is not promoted to be a Java interface by default binding since named model group definitions are not considered part of XML Schema’s type derivation hierarchy and thus this relationship should not be reflected in the Java representation of the two complex types that reference the same model group definition.

When a model group definition's content model contains an XML Schema component that is to be bound to a content interface, element interface or typesafe enum class, it is desirable to only create a single Java representation, not one for each complex content that references the named model group definition. This default binding from a model group definition's content model is defined in Section 5.5.4, "Deriving Class Names for Named Model Group Descendants."

For those who prefer to formerly preserve model group definitions in the Java representation of a schema, a customized binding of a model group definition to a Java content interface is discussed in Section 5.5.3, "Bind to a Java content interface."

5.5.1 Bind to a set of properties

A non-repeating reference to a model group definition, when the particle referencing the group has *{max occurs}* equal to one, results in a set of content properties being generated to represent the content model. Section 5.9, "Content Model - Particle, Model Group, Wildcard" describes how a content model is bound to a set of properties and has examples of the binding.

5.5.2 Bind to a list property

When a model group definition is referenced from a particle with *{max occurs}* greater than one, it is useful to map the reference to a List property in the following manner:

- The *name* of the Java property is derived from the model group definition *{name}* property using the XML Name to Java identifier name mapping algorithm specified in Section C.2, "The Name to Identifier Mapping Algorithm," on page 189.
- The Java property's base type is `java.lang.Object`.
- The *predicate* for the Java property is all the elements/values that can be placed into the list and the ordering restrictions between elements.
- The Java property *collection type* is java.util.List.
- The property has no *default value*.

Example:

Schema fragment contains a particle that references the model group definition has a *{maxOccurs}* value greater than one.

```
<xs:group name="AModelGroup">
   <xs:choice>
      <xs:element name="A" type="xs:int"/>
      <xs:element name="B" type="xs:float"/>
   </xs:choice>
</xs:group>

<xs:complexType name="foo">
   <xs:sequence>
      <xs:group ref="AModelGroup" maxOccurs="unbounded"/>
      <xs:element name="C" type="xs:float"/>
   </xs:sequence>
</xs:complexType>
```

Derived Java representation:

```
   interface AModelGroupA {
      int getValue(); void setValue(int);}
   interface AModelGroupB {
      float getValue(); void setValue(float);}

   interface Foo {
   /** A valid general content property that contains
        instances of AModelGroupA and AModelGroupB.*/
   java.util.List getAModelGroup();

   float getC();
   void setC(float value);
};
```

5.5.3 Bind to a Java content interface

With the appropriate customization specified, as defined in Section 6.7.3.2, “Model Group Definition,” a named model group is bound to a Java content interface in the following manner:

- A *name* for the content interface, which is either computed directly from an XML name or specified by the binding customization for the model group definition;
- A *package name*, which is either computed from the target namespace of the model group definition or specified by binding customization of the target namespace or a specified package name for components that are scoped to no target namespace.
- There is no outer class name context since a named model group definition must be a top-level schema component.
- There is no schema derived base interface for this Java content interface, since it is independent of the type definition derivation hierarchy.
- Set of Java properties which provide access and modification to the content model represented by the interface.
- Set of element interfaces, content interfaces and typesafe enum classes representing element declarations and anonymous type definitions occurring within content model of model group definition. These interfaces are derived from XML Schema components occurring within the content model of the model group definition either by default or due to customizations. These interfaces and classes are nested within the model group definition's Java representation as a content interface.
- A local structural constraint predicate represents all the structural constraints for the content of the class. The constraints include local structural constraints detailed in Section 2.2, "Varieties of validation," on page 17.
- A factory method is generated in the package's `ObjectFactory` class introduced in Section 4.2, "Java Package." The factory method returns the type of the Java content interface. The name of the factory method is generated by concatenating the following components:
  - The string constant `create`.
  - The *name* of the Java content interface.

  For example, if a Java content interface named `Foo` represents a model group definition `Foo`, it would have the following factory method signature generated in the containing Java package's `ObjectFactory` class:

  ```
  Foo createFoo()
  ```

All references to a model group definition bound to a Java content interface are mapped to a Java property with a base type of the Java content interface representing the model group definition. If the particle referencing the group has an occurrence greater than one, then the reference is mapped to a Collection property with a base type of the Java content interface representing the model group definition.

Note that a reference to a model group definition from a complex type definition content model with a *{content type}* of *mixed* can not be bound to a simple property with a base type of a Java content interface. It must be bound to a general content property as detailed in Section 5.9.4, “Bind mixed content.”

Example:

Given the XML Schema fragment and assume the appropriate customizations exist in an external binding file[3]:

```
<xs:group name="AModelGroup">
   <xs:sequence>
      <xs:element name="A" type="xs:int"/>
      <xs:element name="B" type="xs:float"/>
   </xs:sequence>
</xs:group>

<xs:complexType name="foo">
   <xs:sequence>
      <xs:group ref="AModelGroup"/>
      <xs:element name="C" type="xs:float"/>
   </xs:sequence>
</xs:complexType>
```

Derived Java representation:

```
public interface AModelGroup {
   void setA(int value);
   int getA();
   void setB(float value);
   float getB();
};
```

[3]. Note the actual binding declarations are not explicitly used since they are not introduced till the customization section.

```
public interface Foo {
    AModelGroup getAModelGroup();
    void setAModelGroup(AModelGroup value);
    float getC();
    void setC(float value);
};

class ObjectFactory {
    Foo createFoo();
    AModelGroup createAModelGroup();
};
```

The binding of a `<xs:choice>` to a class is specified in Section 5.9.10.1, "Bind a Choice Group to a Content Interface."

5.5.4 Deriving Class Names for Named Model Group Descendants

When a model group definition is not customized to be bound to a content interface, as described in the previous subsection, and its content model contains XML Schema components that need to be bound to a Java class or interface, this section describes how to derive the package and name for the Java content interface, typesafe enum class or element interface derived from the content model of the model group definition. The binding of XML Schema components to Java classes/interfaces is only performed once when the model group definition is processed, not each time the model group definition is referenced as is done for the property set of the model group definition.

XML Schema components occurring within a model group definition's content model that are specified by this chapter and the customization chapter to be bound to an interface or typesafe enum class are bound as specified with the following naming exceptions:

- *package*: The element interface, content interface or typesafe enum class is bound in the Java package that represents the target namespace containing the model group definition.
- *name*: The name of the interface or class is generated as previously specified with one additional step to promote uniqueness between interfaces/classes promoted from a model group definition to be bound

to a top-level class within a Java package. A prefix for the interface/class name is computed from the model group definition's *{name}* using the XML name to Java identifier algorithm.

For example, given a model group definition named *Foo* containing an element declaration named *bar* with an anonymous complex type definition, the anonymous complex type definition is bound to a Java content interface with the name *FooBarType*. The following figure illustrates this example.

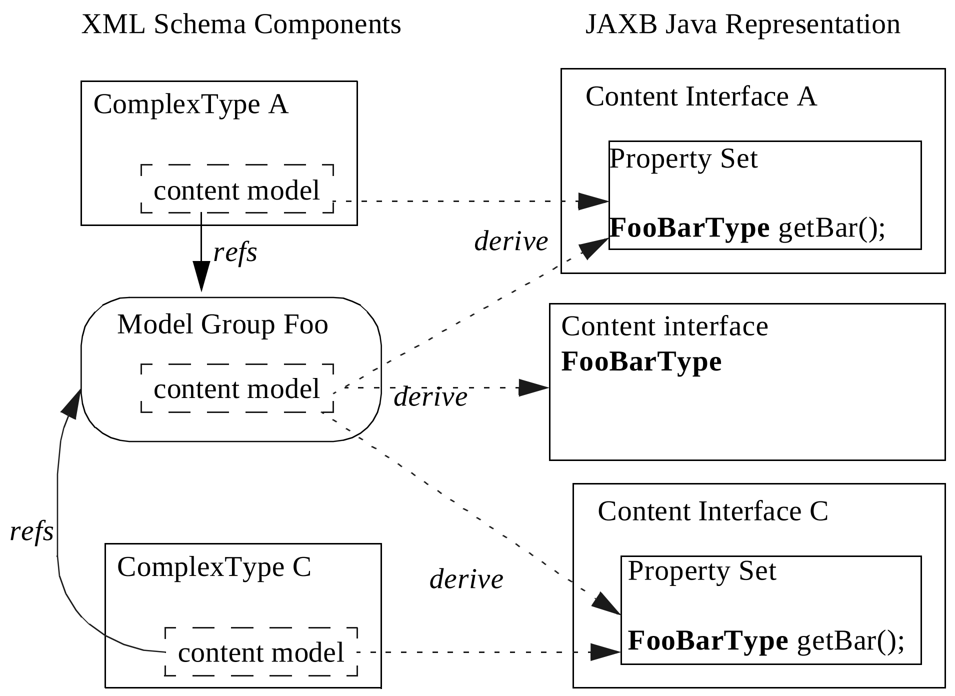

Figure 5.3 Default binding for anonymous type def within a model group definition.

Note that even customization specified interface or typesafe enum class names are prepended with the model group definition's name. Thus, if a model group definition named `Foo` contains an anonymous simple type definition with a typesafe enum class customization name of `Colors,` the typesafe enum class name is `FooColors`.

5.6 Attribute Declaration

An attribute declaration is bound to a Java property when it is referenced or declared, as described in Section 5.8, “Attribute use,” from a complex type definition.

5.7 Element Declaration

This section describes the binding of an XML element declaration to a Java representation. It also introduces why a JAXB technology user would want to use instances of a Java Element interface as opposed to instances of Java datatypes or content interfaces when manipulating XML content.

An XML element declaration is composed of two key components:

- its qualified name is *{target namespace}* and *{name}*
- its value is an instance of the Java class binding of its *{type definition}*

A Java Element interface is generated to represent both of these components. An instance of a Java content interface or a Java class represents only the value of an element. Commonly in JAXB binding, the Java representation of XML content enables one to manipulate just the value of an XML element, not an actual element instance. The binding compiler statically associates the XML element qualified name to a content property and this information is used at unmarshal/marshal time. The following schema/derived Java code example illustrates this point.

Example:

Given the XML Schema fragment:

```
<xs:complexType name="chair_kind">
   <xs:sequence>
      <xs:element name="has_arm_rest" type="xs:boolean"/>
   </xs:sequence>
</xs:complexType>
```

Schema-derived Java content interface:

```
public interface ChairKind {
    boolean isHasArmRest();
    void setHasArmRest(boolean value);
}
```

A user of the Java interface `ChairKind` never has to create a Java instance that both has the value of local element `has_arm_rest` and knows that its XML element name is `has_arm_rest`. The user only provides the value of the element to the content-property `hasArmRest`. A JAXB implementation associates the content-property `hasArmRest` with XML element name `has_arm_rest` when marshalling an instance of `ChairKind`.

The next schema/derived Java code example illustrates when XML element information can not be inferred by the derived Java representation of the XML content. Note that this example relies on binding described in Section 5.9.5, “Bind wildcard schema component.”

Example:

```
<xs:complexType name="chair_kind">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
</xs:complexType>

public interface ChairKind {
    java.lang.Object getAny();
    void setAny(java.lang.Object elementOrValue);
}
```

For this example, the user can provide an Element instance to the `any` content-property that contains both the value of an XML element and the XML element name since the XML element name could not be statically associated with the content-property `any` when the Java representation was derived from its XML Schema representation. The XML element information is dynamically provided by the application for this case. Section 5.9, “Content Model - Particle, Model Group, Wildcard,” on page 89 cover additional circumstances when one can use instances of Element interface.

5.7.1 Bind to Java Element Interface

The characteristics of the generated Java Element interface are derived in terms of the properties of the "Element Declaration Schema Component" on page 205 as follows:

- The *name* of the generated Java Element interface is derived from the element declaration *{name}* using the XML Name to Java identifier mapping algorithm for class names.
- If the element declaration's *{type definition}* is a
  - Complex Type definition

    The derived Java Element interface extends the Java content interface representing the *{type definition}*.
  - Simple type definition

    The generated element interface has a Java simple content-property named `"value"`.

    `ObjectFactory` method to create an instance of the Element interface takes a value parameter of the Java class binding of the simple type definition.
- If *{scope}* is
  - **Global:** The derived Element interface is generated into the Java package that represents the binding of *{target namespace}*.
  - **A Complex Type Definition:** The derived Element interface is generated within the Java content interface represented by the complex type definition value of *{scope}*.
- Each generated Element interface must extend the Java marker interface `javax.xml.bind.Element`. This enables JAXB implementations to differentiate between instances representing a XML element directly and instances representing the type of the XML element.
- If *{nillable}* is `"true"`, the methods `setNil()` and `isNil()` are generated.
- Optional *{value constraint}* property with pair of `default` or `fixed` and a value.
  If a default or fixed value is specified, the data binding system must substitute the default or fixed value if an empty tag for the element declaration occurs in the XML content.

- If an element declaration schema component has an *{abstract}* property of `"true"`, an `ObjectFactory` factory method must not be generated for it.

Note – Substitution properties are not covered since support is not required in this version of the specification as stated in Section E.2, “Not Required XML Schema concepts,” on page 208.

Default binding rules require an element declaration to be bound to derived Element interface under the following conditions:

- All element declarations with global *{scope}* are bound to a derived Java Element interface. The rationale is that any global element declaration can occur within a wildcard context and one might want to provide element instances, not instances of the element’s type, the element’s value, for this case.
- All local element declarations, having a *{scope}* of a complex type definition, occurring within content that is mapped to a general content property must have derived Java Element interfaces generated. General content property is specified in Section 5.9.3, “General content property” An example of when a content model is mapped to a general content property, forcing the generation of element declarations is at Section 5.9.3.1, “Examples.”

5.7.2 Binding of an anonymous complex type definition

An anonymous complex type definition of an element declaration is mapped to a content interface[4]. The naming characteristics of the generated Java Content interface is derived in terms of the properties of the “Element Declaration Schema Component” on page 205 as follows:

- The *name* of the generated Java Content interface is derived from the element declaration *{name}* using the XML Name to Java identifier with a “Type” suffix appended, by default. If there exists a customization for adding a prefix or suffix to anonymous type definitions that are bound to

[4] See Section 5.5, “Model Group Definition” for how anonymous complex type definitions with a named model group as an ancestor are handled.

a Java class or interface, the default “Type” suffix is not added. Section 6.6, “<schemaBindings> Declaration” specifies the element `<jaxb:anonymousTypeName>` to describe the customization.

- The *package* of the generated Java Content interface is the same as the package derived from the element declaration’s *{target namespace}*.
- The *outer class names* of the generated Java Content interface is determined by the element declaration’s *{scope}*. If *{scope}* is:
  - Global
    There is no outer class name.
  - A Complex Type Definition
    The derived Content interface is generated nested within the Java content interface represented by the complex type definition value of *{scope}*.

Section 5.3, “Complex Type Definition” defines how the remaining Java content interface properties are derived from the anonymous complex type definition.

Example:

Given XML Schema fragment:

```
<xs:element name="foo">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="bar" type="xs:int"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Derived Java code:

```
/** Java content interface generated
   from anonymous complex type definition of element foo. */
interface FooType {
   int  getBar();
   void setBar(int value);
}
/** Java Element interface. */
interface Foo extends javax.xml.bind.Element, FooType {};
```

5.7.3 Bind to a Property

- Local element declaration

  Map local element declaration with a fixed *{value constraint}* to a Java constant property.

- If an element declaration has a *{nillable}* property that is `“true”` and its *{type definition}* is mapped by default to a non-referenceable primitive Java type, the base type for the Java property is mapped to the corresponding Java wrapper class for the Java primitive type. Setting the property to the `null` value indicates that the property has been set to the XML Schema concept of `nil=’true`’.

5.8 Attribute use

A ‘required’ or ‘optional’ attribute use is bound by default to a Java property as described in Section 4.5, “Properties,” on page 40. The characteristics of the Java property are derived in terms of the properties of the “Attribute Use Schema Component” on page 207 and “Attribute Declaration Schema Component” on page 206 as follows:

- The *name* of the Java property is derived from the *{attribute declaration}* property’s *{name}* property using the XML Name to Java Identifier mapping algorithm described in Section C.2, “The Name to Identifier Mapping Algorithm,” on page 189.
- A *base type* for the Java property is derived from the `{attribute declaration}` property’s `{type definition}` property as described in binding of Simple Type Definition in Section 5.2, “Simple Type Definition.”
- An optional *predicate* for the Java property is constructed from the `{attribute declaration}` property’s `{type definition}` property as described in the binding of simple type definition to a Java representation.
- An optional *collection type* for the Java property is derived from the `{attribute declaration}` property’s `{type definition}` property as described in the binding of simple type definition to a Java representation.

- The *default value* for the Java property is the *value* from the attribute use's *{value constraint}* property. If the optional *{value constraint}* is absent, the default value for the Java property is the Java default value for the base type.

This Java property is a member of the Java content interface that represents the binding of the complex type definition containing the attribute use.

Design Note – Since the target namespace is not being considered when mapping an attribute to a Java property, two distinct attributes that have the same *{name}* property but not the same *{target namespace}* will result in a Java property naming collision. As specified generically in Section C.2.1, "Collisions and conflicts," on page 192, the binding compiler detect this name collision between the two distinct properties and report the error. The user can provide a customization that provides an alternative Java property name to resolve this situation.

Example:

Given XML Schema fragment:

```
<xs:complexType name="USAddress">
   <xs:attribute name="country" type="xs:string"/>
</xs:complexType>
```

Default derived Java code:

```
public interface USAddress {
   public String getCountry();
   public void setCountry(String value);
}
```

5.8.1 Bind to a Java Constant property

An attribute use with a `fixed` *{value constraint}* property can be bound to a Java Constant property. This mapping is not performed by default since `fixed` is a validation constraint. Since validation is not required to unmarshal or marshal, XML content can have an alternative value for an attribute than the fixed value. The user must set the binding declaration attribute `fixedAttributeToConstantProperty` on `<jaxb:globalBinding>` element as specified in Section 6.5.1, "Usage," on page 121 or on

`<jaxb:property>` element as specified in Section 6.8.1, “Usage,” on page 134 to enable this mapping.

Example:

Given XML Schema fragment:

```
<xs:annotation><xs:appinfo>
  <jaxb:globalBindings fixedAttributeAsConstantProperty="true"/>
</xs:appinfo></xs:annotation>
<xs:complexType name="USAddress">
   <xs:attribute name="country" type="xs:NMTOKEN" fixed="US"/>
</xs:complexType>
```

If the appropriate binding schema customization enables mapping a fixed XML value to Java constant property, the following Java code fragment is generated.

```
public interface USAddress {
   public static final String COUNTRY="US";
   ...
}
```

5.8.1.1 Contributions to Local Structural Constraint

If the attribute use’s *{required}* property is true, the local structural constraint for an instance of the Java content interface requires that the corresponding Java property to be set when the Java content interface instance is validated.

5.8.2 Binding an IDREF component to a Java property

An element or attribute with a type of `xs:IDREF` refers to the element in the instance document that has an attribute with a type of `xs:ID` or derived from type `xs:ID` with the same value as the `xs:IDREF` value. Rather than expose the Java programmer to this XML Schema concept, the default binding of an `xs:IDREF` component maps it to a Java property with a base type of java.lang.Object. The caller of the property setter method must be sure that its parameter is identifiable. An object is considered identifiable if one of its properties is derived from an attribute that is or derives from type `xs:ID`. There is an expectation that all instances provided as values for properties’ representing an `xs:IDREF` should have the Java property representing the `xs:ID` of the instances set before the content tree containing both the `xs:ID` and `xs:IDREF` is (1) globally validated or (2) marshalled. If a property representing an `xs:IDREF` is set with an object that does not have its `xs:ID` set, the `NotIdentifiableEvent` is reported by (1) validation or (2) marshalling.

- The *name* of the Java property is derived from the *{name}* property *of the attribute or element* using the XML Name to Java Identifier mapping algorithm described in Section C.2, "The Name to Identifier Mapping Algorithm," on page 189.
- A *base type* for the Java property is java.lang.Object.
- There is no *predicate* for a property representing an `xs:IDREF`.
- An optional *collection type*
- Default and fixed values can not be supported for an attribute with type `xs:IDREF`.

Example:

Given XML Schema fragment:

```
<xs:complexType name="Book">
   <xs:sequence>
      <xs:element name="author" type="xs:IDREF"/>
      <!-- ... -->
   </xs:sequence>
</xs:complexType>
<xs:complexType name="AuthorBio">
   <xs:sequence><!-- ... --> </xs:sequence>
   <xs:attribute name="name" type="xs:ID"/>
</xs:complexType>
```

Schema-derived Java content interfaces:

```
public interface Book {
    java.lang.Object getAuthor();

    /** Parameter referencedObj should have an attribute or
     * child element with base type of xs:ID by validation
     * or marshal time.
     */
    void setAuthor(java.lang.Object referencedObj);
}
public interface AuthorBio{
    String getName();
    void setName(String value);
}
```

Demonstration of a Java content instance referencing another instance:

```
Book book = ...;
AuthorBio authorBio = ...;
book.setAuthor(authorBio);
authorBio.setName("<some author’s name>");
// The content instance root used to validate or marshal book must
// also include "authorBio" as a child element somewhere.
// A Java content instance is not included
```

Note that ID and IDREF mechanisms does not incorporate the type definitions that can be referenced. A binding declaration customization could specify that the base type for the author property of content interface Book should be AuthorBio instead of java.lang.Object to make for a more meaningful binding.

5.9 Content Model - Particle, Model Group, Wildcard

This section describes the possible Java bindings for the content model of a complex type definition schema component with a *{content type}* property of `mixed` or `element-only`. The possible element content(s) and the valid orderings between those contents are constrained by the *{particles}* describing the complex type definition’s content model. The Java binding of a content model is realized by the derivation of one or more content-properties to represent the element content constrained by the model group. Section 5.9 introduces to logically different binding styles. Section 5.9.1 through 5.9.7 describes the default binding style that is referred to as the *element binding* of the content model. From Section 5.9.8 till the end of Section 5.9, *model group binding* style is defined. Note that model group binding is only enabled by customizations described in Section 6.

5.9.1 Element binding style

The ideal Java binding would be to map each uniquely named element declaration occurring within a content model to a single Java content-property. The model group schema component constraint, element declarations consistent, specified in [XSD-Part 1] ensures that all element declarations/ references having the same {target namespace} and {name} must have the same top-level type definition. This model allows the JAXB technology user to specify only the content and the JAXB implementation infers the valid ordering between the element content based on the *{particles}* constraints in the source schema. However, there do exist numerous scenarios that this ideal binding is not possible for parts of the content model or potentially the entire content model. For these cases, default binding has a fallback position of representing the element content and the ordering between the content using a *general content model*. The scenarios where one must fallback to the general content model will be identified later in this subsection.

5.9.2 Bind each element declaration name to a content property

This approach relies on the fact that a model group merely provide constraints on the ordering between children elements and the user merely wishes to provide the content. It is easiest to introduce this concept without allowing for repeating occurrences of model groups within a content model. Conceptually, this approach presents all element declarations within a content model as a set of element declaration *{name}*'s. Each one of the *{name}*'s is mapped to a content-property. Based on the element content that is set by the JAXB application via setting content-properties, the JAXB implementation can compute the order between the element content using the following methods.

Computing the ordering between element content within **[children]** of an element information item

- Schema constrained fixed ordering or semantically insignificant ordering

  The sequence in the schema represents an ordering between children elements that is completely fixed by the schema. Schema-constrained ordering is not exposed to the Java programmer when mapping each element in the sequence to a Java property. However, it is necessary for the marshal/unmarshal process to know the ordering. No new ordering constraints between children elements can be introduced by an XML document or Java application for this case. Additionally, the Java application does not need to know the ordering between children elements. When the compositor is `all`, the ordering between element content is not specified semantically and any ordering is okay. So this additional case can be handled the same way.

- Schema only constrains content and does not significantly constrain ordering

  If the ordering between the children elements is significant and must be accessible to the Java application, then the ordering is naturally preserved in Java representation via a collection. Below are examples where schema provides very little help in constraining order based on content.

  ```
  <xs:choice maxOccurs="unbounded"> ... </choice>
  <xs:sequence maxOccurs="unbounded"> ... </sequence>
  ```

- Schema constrained partial ordering

  The ordering between children elements is constrained by a combination of constraints between content specified in the schema and the actual content within the XML content. The schema provides constraints on ordering for this case that is computed based on the content assigned from the XML document during unmarshalling or from the set values by the Java application. There exists a significant number of cases where the ordering constraints can be computed based on the *set value* content and partial ordering between elements specified in the schema.

Below is an example demonstrating the ordering of children elements using partially schema constrained ordering. Given that the following schema is mapped to four Java properties: A, B, C and D,

```
 <xs:choice>
    <xs:sequence>
       <xs:element ref="A"/>
       <xs:element ref="C"/>
       <xs:element ref="D"/>
    </xs:sequence>
    <xs:sequence>
       <xs:element ref="B"/>
       <xs:choice>
          <xs:element ref="C"/>
          <xs:element ref="D"/>
       </xs:choice>
    </xs:sequence>
</xs:choice>
```

one can compute if only the properties for A, C and D are set, that the content should be marshalled out in the order constrained by the first choice sequence. If the content is set for either B and C or B and D, then the second choice sequence ordering constraint between elements should be followed.

Example:

Given XML Schema fragment:

```
<xs:complexType name="USAddress"/>
<xs:complexType name="Items"/>
<xs:element name="comment" type="xs:string"/>
<xs:complexType name="PurchaseOrderType">
   <xs:sequence>
      <xs:choice>
         <xs:group   ref="shipAndBill"/>
         <xs:element name="singleUSAddress" type="USAddress"/>
      </xs:choice>
      <xs:element ref="comment" minOccurs="0"/>
      <xs:element name="items"  type="Items"/>
   </xs:sequence>
   <xs:attribute name="orderDate" type="xs:date"/>
</xs:complexType>
<xs:group name="shipAndBill">
   <xs:sequence>
      <xs:element name="shipTo" type="USAddress"/>
      <xs:element name="billTo" type="USAddress"/>
   </xs:sequence>
</xs:group>
```

Generate following Java code and assume `USAddress` is a complex type definition that is bound to a Java content interface `USAddress`.

```
public interface PurchaseOrderType {
    void setShipTo(USAddress );
    USAddress getShiptTo();
    void setBillTo(USAddress );
    USAddress getBillTo();
    void setSingleUSAddress(USAddress );
    USAddress getSingleUSAddress();
    void setComment(String);
    String getComment();
    void setOrderDate(java.util.Calendar);
    java.util.Calendar getOrderDate();
    void setItems(Items);
    Items getItems();
}
```

User is responsible for knowing that a valid content model requires either property `singleUSAddress` to be set or for properties `shipTo` and `billTo` must be set. Note that the user does not have to concern themselves with the ordering between properties. A JAXB implementation is responsible for inferring the order between elements based on what content is set. If the system is unable to infer the ordering at validation time, a validation event is thrown. The marshalling of invalid content is not specified so it is non-deterministic what a system does for that case.

5.9.3 General content property

A general content property is, as its name implies, the most general of all content properties. Such a property can be used with any content specification, no matter how complex. A general content property is represented as a List property as introduced in Section 4.5.2.2, “List Property,” on page 45. Unlike the prior approach where the JAXB implementation must infer ordering between the element content, this approach always requires the JAXB technology user to specify a valid ordering of element and text content. This approach has the benefit of providing the application with more control over setting and knowing the order between element content.

A general content property is capable of representing both element information items and character data items occurring within **[children]** of an element information item. Character data is inserted into the list as java.lang.String values. Element data is added to the list as instances of Java Element interfaces.

5.9.3.1 Examples

Example 1: Complex content model of Elements with primitive types

```
<xs:complexType name="Base">
   <xs:choice maxOccurs="unbounded">
      <xs:element name="A" type="xs:string"/>
      <xs:element name="B" type="xs:string"/>
      <xs:element name="C" type="xs:int"/>
   </xs:choice>
</xs:complexType>

public interface Base {
   interface A extends javax.xml.bind.Element {
      String getValue(); void setValue(String);}
   interface B extends javax.xml.bind.Element {
      String getValue(); void setValue(String);}
   interface C extends javax.xml.bind.Element {
      int getValue(); void setValue(int);}

   /**
    * A general content list that can contain
    * element instances of Base.A,Base.B and Base.C.
    * <insert appropriate schema fragment here>
    */
   List getAOrBOrC();
}
```

Example 2: XML Schema element declaration with Complex Type Definition

XML Schema fragment:

```
<xs:complexType name="AType"/>
<xs:complexType name="BType"/>
<xs:complexType FooBar>
   <xs:choice maxOccurs="unbounded">
      <xs:element name="foo" type="AType"/>
      <xs:element name="bar" type="BType"/>
   </xs:choice>
</xs:complexType>
```

Default derived Java code:

```
interface AType { ... }
interface BType { ... }

interface FooBar {
   interface Foo extends AType, javax.xml.bind.Element {...}
   interface Bar extends BType, javax.xml.bind.Element {...}
   /**
   '* A valid general content list contains instances of
    * Foo and/or Bar.
    */
   List getFooOrBar();
};
```

5.9.4 Bind mixed content

When a complex type definition's *{content type}* is "mixed," its character and element information content is bound to general content list as described in Section 5.9.3, "General content property." Character information data is inserted as instances of `java.lang.String` into a `java.util.List` instance. The local structural constraints of the *{content type}* particles is propagated up to the Java content interface representing the complex type definition.

Example:

Schema fragment loosely derived from mixed content example from [XSD Part 0].

```
<xs:element name="letterBody">
   <xs:complexType mixed="true">
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="quantity" type="xs:positiveInteger"/>
         <xs:element name="productName" type="xs:string"/>
         <!-- etc. -->
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Derived Java code:

```
import java.math.BigInteger;
interface LetterBodyType {
   interface Name extends javax.xml.bind.Element {
      String getValue(); void setValue(String); }
   interface Quantity extends javax.xml.bind.Element {
      BigInteger getValue(); void setValue(BigInteger); }
   interface ProductName extends javax.xml.bind.Element {
      String getValue(); void setValue(String);}

   /** Mixed content can contain instances of Element interfaces
      Name, Quantity and ProductName. Text data is represented as
      java.util.String for text.
   */
   List getContent();
}
public interface LetterBody extends
                     javax.xml.bind.Element, LetterBodyType { };
```

The following instance document

```
<letterBody>
Dear Mr.<name>Robert Smith</name>
Your order of <quantity>1</quantity> <productName>Baby
Monitor</productName> shipped from our warehouse. ....
</letterBody>
```

could be constructed using JAXB API.

```
LetterBody lb = ObjectFactory.createLetterBody();
List gcl = lb.getContent();
gcl.add("Dear Mr.");
gcl.add(ObjectFactory.createLetterBodyName("Robert Smith"));
gcl.add("Your order of ");
gcl.add(ObjectFactory.
          createLetterBodyQuantity(new BigInteger("1")));
gcl.add(ObjectFactory.createLetterBodyProductName("Baby Monitor"));
gcl.add("shipped from our warehouse");
```

Note that if any element instance is placed into the general content list, *gcl*, that is not an instance of LetterBody.Name, LetterBody.Quantity or LetterBody.ProductName, validation would detect the invalid content model. With the fail fast customization enabled, element instances of the wrong type are detected when being added to the general content list, *gcl*.

5.9.5 Bind wildcard schema component

A wildcard is mapped to a simple content-property with:

- Content-property *name* set to the constant “`any`”. A binding schema customization could provide a more semantically meaningful content-property name.
- Content-property *base type* set to `java.lang.Object` by default. Wildcard content encountering during unmarshalling is supported if global XML element tags occurring in “strict” or “lax” wildcard context are known to the instance of `javax.xml.bind.JAXBContext`, meaning that the schema(s) describing the element content occurring in the wildcard context is registered with the `JAXBContext` instance, see Section 3.2, “JAXBContext,” on page 24 on how bindings are registered with a `JAXBContext` instance. A JAXB implementation is only required to be able to marshall and unmarshal global element content to/from “strict”/“lax” wildcard context that is registered and valid[5] according to the schema(s) registered to `JAXBContext`. The specification does not specify how a JAXB implementation handles element content that it does not know how to map to a Java representation.
- See content-property predicate for a wildcard.
- If the `maxOccurs` is greater than one, the content property is mapped to a collection property. The default collection property is a List property.
- These is no *default value*.

Note that the default base type being the marker class for an XML element indicates that a wildcard content handled by default as an instance of an XML Element. Since the schema does not contain any information about the element content of a wildcard content, even the content-property, by default, can not infer an XML element tag for wildcard element content.

[5] The wildcard content must conform to the schema(s) registered with `JAXBContext`.

5.9.6 Bind a repeating occurrence model group

A choice or sequence model group with a repeating occurrence, `maxOccurs` attribute greater than one, is bound to a general content property in the following manner:

- Content-property *name* is derived in following ways:
  - ❍ If a named model group definition is being referenced, the value of its *{name}* property is mapped to a Java identifier for a method using the algorithm specified in Section C.2, “The Name to Identifier Mapping Algorithm,” on page 189.
  - ❍ To derive a content property *name* for unnamed model group, see Section C.4, “Deriving an identifier for a model group,” on page 194.
- Content-property *base type* set to `java.lang.Object`. A binding schema customization could provide a more specialized java class.
- Content-property *predicate* validates the order between element instances in the list and whether the occurrence constraints for each element instance type is valid according to the schema.
- Since the `maxOccurs` is always greater than one, the content property is mapped to a collection property. The default collection property is a List property.
- These is no *default value*.

Local structural Constraints

The list content property’s value must satisfy the content specification of the model group. The ordering and element contents must satisfy the constraints specified by the model group.

5.9.7 Content Model Default Binding

The following rules define *element* binding style for a complex type definition’s content model.

1. If *{content type}* is mixed, bind the entire content model to a general content property with the content-property name “`content`”. See Section 5.9.4, “Bind mixed content” for more details.
2. If (1) a particle has *{max occurs}* >1 and (2) its *{term}* is a model group, then that particle and its descendants are mapped to one general content

property that represents them. See Section 5.9.6, “Bind a repeating occurrence model group” for details.

3. Process all the remaining particles (1) whose *{term}* are wildcard particles and (2) that did not belong to a repeating occurrence model group bound in step. 2. If there is only one wildcard, bind it as specified in Section 5.9.5, “Bind wildcard schema component.” If there is more than one, then fallback to representing the entire content model as a single general content property.

4. Process all particles (1) whose *{term}* are element declarations and (2) that do not belong to a repeating occurrence model group bound in step.2.

   First, we say a particle has a label *L* if it refers to an element declaration whose *{name}* is *L*. Then, for all the possible pair of particles *P* and *P’* in this set, ensure the following constraints are met:

   a. If *P* and *P’* have the same label, then they must refer to the same element declaration.

   b. If *P* and *P’* refer to the same element reference, then its closest common ancestor particle may not have sequence as its *{term}*.

   If either of the above constraints are violated, then the binding compiler must report a property naming collision that can be corrected via customization.

   Create a content property for each label *L* as follows:

   ❍ The content property *name* is derived from label name *L*.

   ❍ The *base type* will be the Java type to which the referenced element declaration maps.

   ❍ The content property *predicate* reflects the occurrence constraint.

   ❍ The content property *collection type* defaults to ‘`list`’ if there exist a particle with label *L* that has *{maxOccurs}* > 1.

   ❍ For the default value, if all particles with label *L* has a *{term}* with the same *{value constraint}* default or fixed value, then this value. Otherwise none.

Note – Note: Binding schema customization can be used to give particles a different name to avoid the fallback.

Below is an example demonstrating violation of rules 4(a) and 4(b) specified above.

```
<xs:sequence>
   <xs:choice>
      <xs:element ref="ns1:bar"/> (A)
      <xs:element ref="ns2:bar"/> (B)
   </xs:choice>
   <xs:element ref="ns1:bar"/> (C)
</xs:sequence>
```

The pair *(A,B)* violates the first clause because they both have the label “bar” but they refer to different element declarations. The pair *(A,C)* violates the second clause because their nearest common ancestor particle is the outermost `<sequence>`.

5.9.7.1 Default binding of content model “derived by extension”

If a content-property naming collision occurs between a content-property that exists in an base complex type definition and a content-property introduced by a “derive by extension” derived complex type definition, the content-properties from the colliding property on are represented by a general content property with the default property name `rest`.

Example:
derivation by extension content model with a content-property collision.

Given XML Schema fragment:

```
<xs:complexType name="Base">
   <xs:sequence>
      <xs:element name="A" type="xs:int"/>
      <xs:element name="B" type="xs:int"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="Derived">
   <xs:complexContent>
      <xs:extension base="Base">
         <xs:sequence>
            <xs:element name="A" type="xs:int"/>
         </xs:sequence>
      </xs:extension>
```

```
        </xs:complexContent>
    </xs:complexType>
```

Default binding derived Java code:

```
    interface Base {
        int getA(); void setA(int);
        int getB(); void setB(int);
    }

    interface Derived extends Base {
        interface A extends javax.xml.bind.Element {
            int getValue();
            void setValue(int value);
        }

        /**
         * Instances of Derived.A must be placed in this general
         * content propert that represents the rest of the content
         * model. 6 */
        List getRest();
    }
```

5.9.8 Model group binding style

An alternative binding approach to treating the content model as just a list of elements is to map model groups nested in the content model to Java content interfaces. The benefit of this binding approach is the generated content interfaces and content properties capture the semantics of model groups, aiding the user in constructing valid content. Additionally, the additional content interfaces allow this style of binding to rely a lot less on the general content model, only mixed content models have to be represented as a general content property. Unfortunately, this approach does result in an increase in the number of generated Java content interfaces. Additionally, this approach benefits from binding schema customizations that provide semantically meaningful names to represent the content interfaces generated to represent nested choice and sequence model groups. Thus, it was not considered as good a candidate for default binding but it is considered a valuable alternative binding option.

6. Specifying a customization of the local element declaration A within Derived complex type to a different property name than A would avoid the fallback position for this case.

5.9.9 Bind Top-level Model Group to a Property Set

The contents of non-repeating top-level model group[7] is bound to a property set as specified in Section 5.9.11, "Model Group binding algorithm." This property set aggregates into either to a Java content interface derived from a complex type definition in Section 5.3.2, "Java Content Interface" or to a model group definition as described in Section 5.5.3, "Bind to a Java content interface.

There is one exception case that must be excluded from this style of binding. If the top-level model group is a choice for a complex type definition with attributes, then the top-level model group should be bound as if it were a nested model group. The rationale for this exception case is that there exist a relationship between the properties of a choice content property, described in Section 5.9.10.1, "Bind a Choice Group to a Content Interface." that does not accommodate simply aggregating the properties representing attributes into the same interface.

5.9.10 Bind Nested Model Group

This subsection describes how a repeating top-level model group or a model group nested[6] within a content model is bound to a content interface. Chapter 6, "Customization" describes the various customizations that enable this binding.

The characteristics of the content interface are derived in terms of the properties of the "Model Group Schema Component" on page 208[8] as follows:

- The *content interface name* for a unnamed model group is either derived as specified in Section C.4, "Deriving an identifier for a model group," on page 194 or specified explicitly by customization.
- A *package name* which is the same as the package name for the primary schema component containing the nested model group.
- For a nested model group, the *outer class name context* is a dot-separated list of Java class names representing the first ancestor of the model group that was mapped to a content interface. The ancestor of the model group

7. A top-level model group represents the entire content model of a complex type definition or model group definition. Any model group occurring within a top-level model group is referred to as a *nested model group*.

8. A model group reference can not be bound to a content interface, only the model group definition it is referencing can be bound to a content interface.

for this case is either a complex type definition or a model group that was also mapped to a content interface.

- By default, the content interface does not have a *base interface type*.
- Set of Java properties which provide access and modification of the content model represented by the interface.
  - When the model group's *{compositor}* is `choice`, Section 5.9.10.1, "Bind a Choice Group to a Content Interface" defines the property set.
  - When the model group's *{compositor}* is `all` or `sequence`, Section 5.9.11, "Model Group binding algorithm" defines how to compute the property set.
- A local structural constraint predicate represents all the structural constraints for the content of the class. The constraints include attribute occurrences and local structural constraints detailed in Section 2.2, "Varieties of validation," on page 17.
- A factory method is generated in the package's `ObjectFactory` class introduced in Section 4.2, "Java Package." The factory method returns the type of the Java content interface. The name of the factory method is generated by concatenating the following components:
  - 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.

5.9.10.1 Bind a Choice Group to a Content Interface

A choice group in XML Schema specifies one or more particles and where only one can occur in content. A choice group could be accessed either as a single entity or as a set of Java properties, only one of which is ever set at one time. The following subsections describe two styles of binding a choice group to its Java representation.

A <class> binding declaration customization of a choice group indicates that its content model should be represented by a generated content interface that encapsulates all of its properties and also allows for access of the choice as a single entity. The customization is specified in Section 6.7.3.3, "Model Group." All of the characteristics necessary to generate a choice content interface are defined in the previous subsection except for the specification of choice property set.

The property set that represents a choice group consists of the following property kinds:

- Set of choice properties (one for each significant[9] particle in choice model group). At a given point in time, at most one of these choice properties is ever set.
- A read-only `content` property. This convenience property represents the current state of the choice content interface.

The choice property set is defined by the following method signatures:

- Identify the properties for the choice model group using Section 5.9.11, “Model Group binding algorithm.” The choice content interface adds the following behaviors to the Java properties specified in Section 4.
    - `isSetId` method returns true if the choice property is specified by the particle corresponding to `Id`. At any point in time, only one choice property in a choice content class is true, all others will be false.
    - If the choice property represents a Simple or Indexed Property:
      (i) `getId` method returns the current value of the choice property if the choices content is specified by `Id`; otherwise, return `null`. The method returns a Java primitive type when appropriate.
      (ii) `setId` method sets the given value of the choice property. This is a mutually exclusive set. It logically unsets the previously set value for the choice and makes this only set property for the choice content interface.
    - Else if the choice property represents a List property:
      (i) `getID` method returns a `java.util.List`. When this property is not the set choice for the choice group, i.e. `isSetID` returns `false`, this method returns a zero-length `java.util.List`.
      (ii) The `java.util.List` methods that allow adding to the list are the setters for a list property. When a choice list property has its first element added to it, the previously set choice property is considered to be unset. If the previously set choice property was a list property, its list content is cleared so the list is a zero-length `java.util.List`.
- The `getContent` method returns the current value for the choice content interface.

[9] A insignificant particle is defined in the Model group binding algorithm as the sequence nested within another sequence or a choice nested within another choice.

The return type for the `getContent` method signature is determined in the following manner:

a. If all choice property's `getter` methods have the same Java datatype as a return type, then this datatype is the `getContent` method's return type and we are finished. Otherwise, proceed to next step.

b. Compute the most common base type of the return type over all the choice property's `getter` accessor methods, considering Java primitive datatypes as their corresponding Java wrapper class.

For example,

❍ If all choice item properties bind to a List property, the return type for `getContent` is `java.util.List.`
❍ If all choice item properties bind to an indexed property, the return type is an array of the most common Java supertype shared among the choice item properties base types.
❍ If all choice item properties are different Java numeric primitive datatypes, the `getContent` method has a return type of `java.lang.Number`.
❍ If some choice item properties are collection properties and some are content values or Java primitive datatypes, the return type is `java.lang.Object`.

The actual instance returned by `getContent` method is the same as if the specific `getter` method for the currently set choice property was invoked. The exception being when that property would have returned a primitive datatype and step (b) above promoted it to its wrapper class. Additionally, if no choice properties are considered set (when `isSetContent`() is `false`), then the JVM default value for an uninitialized field is returned by an invocation of the `getContent` method.

- The `isSetContent` method returns `true` if one of the N choice properties has a current value.
- The `unsetContent` method discards the property's given value, if any.

Example:

XML Schema fragment:

```
<xs:complexType name="SomeComplexType"/>
   <xs:choice maxOccurs="unbounded">
      <xs:element name="foo" type="xs:int"/>
      <xs:element name="bar" type="xs:string"/>
   </xs:choice>
</xs:complexType>
```

Derived Java interfaces:

```
public interface SomeComplexType {
   /** class generated to represent <insert choice fragment here>*/
   public interface FooOrBar {
      /** Setting Foo implies all other properties are not set and
      *   and only isSetFoo() will return true.*/
      void setFoo(int value);
      int getFoo();
      boolean isSetFoo();

      /** Setting Bar implies all other properties are not set.*/
      void setBar(String value);
      String getBar();
      boolean isSetBar();

      /** returns an instance of java.lang.Integer or String.*/
      java.lang.Object getContent();
      /** returns true if isSetFoo or isSetBar is true.*/
      boolean isSetContent();
      void unsetContent();
   }

   List getFooOrBar();
}

class ObjectFactory {
   ....
   SomeComplexType createSomeComplexType();
   SomeComplexType.FooOrBar createSomeComplexTypeFooOrBar();
}
```

5.9.10.2 Bind choice group to a choice content property

Setting the `choiceContentProperty` attribute of `<jaxb:globalBindings>` as specified in Section 6.5.1, "Usage," on page 121 enables this customized binding option.

A non-repeating choice model group is bound to a simple property. A repeating choice model group is bound to a collection property. A choice content property is derived from a choice model group as follows:

- The choice content property name is either the referenced model group definition *{name}* or obtained using the algorithm specified in Section C.4, "Deriving an identifier for a model group," on page 194.
- The choice content property `base type` is the first common supertype of all items within the choice model group, with `java.lang.Object` always being a common root for all Java objects.[10]
- The predicate
- The collection type defaults to List if the choice model group has *{max occurs}* greater than one.
- No default value.

A choice property consists of the following methods:

- The `getChoiceID` method returns the set value. If the property has no set value then the value `null` is returned. Note that a set value of a primitive Java type is returned as an instance of the corresponding Java wrapper class.
- The `setChoiceID` method has a single parameter that is the type of the choice content property `base type`.

The `globalBindings` and property customization attribute, `choiceContentProperty,` enables this customized binding`.` The customization is specified in Section 6.5, "<globalBindings> Declaration."

[10] Note that primitive Java types must be represented by their Java wrapper classes when *base type* is used in the choice content property method signatures. Also, all sequence descendants of the choice are treated as either a general content property or are mapped to their own Java content interface.

Example:

XML Schema representation of a choice model group.

```
<xs:choice>
   <xs:element name="foo" type="xs:int"/>
   <xs:element name="bar" type="xs:string"/>
</xs:choice>
```

Derived choice content property method signatures:

```
void setFooOrBar(Object);
Object getFooOrBar();
```

5.9.11 Model Group binding algorithm

The following rules describe how to bind a content model to its Java representation when customizations specify that a content model or nested model group be bound using the model group binding style:

1. When *{content type}* is
   a. `mixed` - Bind the entire content model to a general content property with the content-property name "content". See Section 5.9.3, "General content property" for more details.
   b. `element-only` - Apply all binding declaration customizations on model groups within the content model.
2. Normalize unnecessary nested, non-repeating model groups remaining after applying previous step.

   Given particle *T* that contains a particle *N*, (1) if the *{term}* for both particle *T* and *N* represent the same compositor, either `<sequence>` or `<choice>` and (2) particle *N* has *{max occurs}* == 1, then one can flatten all the particles from particle N's *{term}* model group into the particle T's *{term}* model group.

   This process should be repeated until the top level particle only contains
   a. choice groups containing nested, non-repeating sequences
   b. sequence groups containing nested, non-repeating choices
   c. directly or indirectly, repeating occurrence model groups
3. Bind all repeating occurrence model groups remaining after applying the

previous steps in the following manner:

a. Bind each sequence or choice group to the appropriate Java content interface.

b. Represent the multiple occurrences of the model group as a List property with base type of the Java content interface derived in step 3(a).

4. Bind all non-repeating model groups remaining after applying previous steps in the following manner:

    a. Bind each non-repeating top-level model group to a Java content interface that also represents its parent complex type definition or model group definition.

    b. Bind each nested model group to a Java content interface.

    c. Map each model group to a simple property with a base type of the Java content interface derived in step 4(b).

5. Bind elements occurring within the remaining sequences to the appropriate Java property.

5.10 Default Binding Rule Summary

Note that this summary is non-normative and all default binding rules specified previously in the chapter take precedence over this summary.

- Bind the following to Java package:
    - XML Namespace URI
- Bind the following XML Schema components to Java content interface:
    - Named complex type
    - Anonymous inlined type definition of an element declaration
- Bind to typesafe enum class:
    - A named simple type definition with a basetype that derives from "`xs:NCName`" and has enumeration facets.
- Bind the following XML Schema components to a Java Element interface

  - A global element declaration to a Element interface.
  - Local element declaration that can be inserted into a general content list.
- Bind to Java property
  - Attribute use
  - Particle with a term that is an element reference or local element declaration.
- Bind model group and wildcard content with a repeating occurrence and complex type definitions with `mixed` *{content type}* to:
  - A general content property - a List content-property that holds Java instances representing element information items and character data items.

CHAPTER 6

CUSTOMIZATION

The default binding of source schema components to derived Java representation by a binding compiler sometimes may not meet the requirements of a JAXB application. In such cases, the default binding can be customized using a *binding declaration*. Binding declarations are specified by a *binding language*, the syntax and semantics of which are defined in this chapter.

All JAXB implementations are required to provide customization support specified here unless explicitly stated as optional.

6.1 Binding Language

The binding language is an XML based language which defines constructs referred to as *binding declarations*. A binding declaration can be used to customize the default binding between an XML schema component and its Java representation.

The schema for binding declarations is defined in the namespace `http://java.sun.com/xml/ns/jaxb`. This specification uses the namespace prefix "`jaxb`" to refer to the namespace of binding declarations. For example,

```
<jaxb: binding declaration >
```

A binding compiler interprets the binding declaration relative to the source schema and a set of default bindings for that schema. Therefore a source schema need not contain a binding declarations for every schema component. This makes the job of a JAXB application developer easier.

There are two ways to associate a binding declaration with a schema element:

- as part of the source schema (*inline annotated schema*)
- external to the source schema in an *external binding declaration.*

The syntax and semantics of the binding declaration is the same regardless of which of the above two methods is used for customization.

A binding declaration itself does not identify the schema component to which it applies. A schema component can be identified in several ways:

- explicitly - e.g. QName, XPath expressions etc.
- implicitly - based on the context in which the declaration occurs.

It is this separation which allows the binding declaration syntax to be shared between inline annotated schema and the external binding.

6.1.1 Extending the Binding Language

In recognition that there will exist a need for additional binding declarations than those currently specified in this specification, a formal mechanism is introduced so all JAXB processors are able to identify *extension binding declarations*. An extension binding declaration is not specified in the *jaxb:* namespace, is implementation specific and its use will impact portability. Therefore, binding customization that must be portable between JAXB implementations should not rely on particular customization extensions being available.

The namespaces containing extension binding declarations are specified to a JAXB processor by the occurrence of the global attribute `<jaxb:extensionBindingPrefixes>` within an instance of `<xs:schema>` element. The value of this attribute is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as a customization declaration namespace. Prefixes are resolved on the `<xs:schema>` element that carries this attribute. It is an error if the prefix fails to resolve. This feature is quite similar to the extension-element-prefixes attribute in [XSLT 1.0] `http://www.w3.org/TR/xslt10/#extension`, introduces extension namespaces for extension instructions and functions for XSLT 1.0.

This specification does not define any mechanism for creating or processing extension binding declarations and does not require that implementations support any such mechanism. Such mechanisms, if they exist, are implementation-defined.

6.1.2 Inline Annotated Schema

This method of customization utilizes on the `<appinfo>` element specified by the XML Schema [XSD PART 1]. A binding declaration is embedded within the `<appinfo>` element as illustrated below.

```
<xs:annotation>
   <xs:appinfo>
      <binding declaration>
   </xs:appinfo>
</xs:annotation>
```

The inline annotation where the binding declaration is used identifies the schema component.

6.1.3 External Binding Declaration

The external binding declaration format enables customized binding without requiring modification of the source schema. Unlike inline annotation, the remote schema component to which the binding declaration applies must be identified explicitly. The `<jaxb:bindings>` element enables the specification of a remote schema context to associate its binding declaration(s) with. Minimally, an external binding declaration follows the following format.

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

The attributes *schemaLocation* and *node* are used to construct a reference to a node in a remote schema. The binding declaration is applied to this node by the binding compiler as if the binding declaration was embedded in the node's `<xs:appinfo>` element. The attribute values are interpreted as follows:

- *schemaLocation* - It is a URI reference to a remote schema.
- *node* - It is an XPath 1.0 expression that identifies the schema node within schemaLocation to associate binding declarations with.

An example external binding declaration can be found in Section D.1, "Example."

6.1.3.1 Restrictions

- The external binding element `<jaxb:bindings>` is only recognized for processing by a JAXB processor when its parent is an `<xs:appinfo>` element, it is an ancestor of another `<jaxb:bindings>` element, or when it is root element of a document. An XML document that has a `<jaxb:bindings>` element as its root is referred to as an external binding declaration file.
- The top-most `<jaxb:binding>` element within an `<xs:appinfo>` element or the root element of an external binding file must have its `schemaLocation` attribute set.

6.1.4 Version Attribute

The normative binding schema specifies a global `version` attribute. It is used to identify the version of the binding declarations. For example, a future version of this specification may use the version attribute to specify backward compatibility. For this version of the specification, the `version` must always `"1.0".` If any other version is specified, it must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations."

The `version` attribute must be specified in one of the following ways:

- If customizations are specified in inline annotations, the `version` attribute must be specified in `<xs:schema>` element of the source schema. For example,

  ```
  <xs:schema jaxb:version="1.0">
  ```

- If customizations are specified in an external binding file, then the `jaxb:version` attribute must be specified in the root element `<jaxb:bindings>` in the external binding file. Alternately, a local `version` attribute may be used. Thus the version can be specified either as

```
<jaxb:bindings version="1.0" ... />
```

or

```
<jaxb:bindings jaxb:version="1.0" ... />
```

Specification of both `version` and `<jaxb:version>` must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations."

6.1.5 Invalid Customizations

A *non conforming* binding declaration is a binding declaration in the `jaxb` namespace but does not conform to this specification. A non conforming binding declaration results in a *customization error*. The binding compiler must report the customization error. The exact error is not specified here. For additional requirements see Chapter 7, "Compatibility."

The rest of this chapter assumes that non conforming binding declarations are processed as indicated above and their semantics are not explicitly specified in the descriptions of individual binding declarations.

6.2 Notation

The source and binding-schema fragments shown in this chapter are meant to be illustrative rather than normative. The normative syntax for the binding language is specified in Appendix , "Normative Binding Schema Syntax." in addition to the other normative text within this chapter. All examples are non-normative.

- Metavariables are in *italics*.
- Optional attributes are enclosed in `[ square="bracket" ]`.
- Optional elements are enclosed in `[ <elementA> ... </elementA> ]`.
- Other symbols: '`,`" denotes a sequence, '`|`' denotes a choice, '`+`' denotes one or more, '`*`' denotes zero or more.

- The prefix `xs:` is used to refer to schema components in W3C XML Schema namespace.
- In examples, the binding declarations as well as the customized code are shown in bold like this: **<appinfo> <annotation>** or **getAddress()**.

6.3 Naming Conventions

The naming convention for XML names in the binding language schema are:

- The first letter of the first word in a multi word name is in lower case.
- The first letter of every word except the first one is in upper case.

For example, the XML name for the Java property basetype is baseType.

6.4 Customization Overview

A binding declaration customizes the default binding of a schema element to a Java representation. The binding declaration defines one or more *customization values* each of which customizes a part of Java representation.

6.4.1 Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies. If a customization value applies to a schema element, then the schema element is said to be *covered* by the scope of the customization value. The scopes are:

- **global scope:** A customization value defined in `<globalBindings>` has *global scope*. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
- **schema scope:** A customization value defined in <`schemaBindings`> has *schema scope*. A schema scope covers all the schema elements in the target namespace of a schema.

- **definition scope:** A customization value in binding declarations of a type definition or global declaration has *definition scope.* A definition scope covers all schema elements that reference the type definition or the global declaration. This is more precisely specified in the context of binding declarations later on in this chapter.
- **component scope:** A customization value in a binding declaration has *component scope* if the customization value applies only to the schema element that was annotated with the binding declaration.

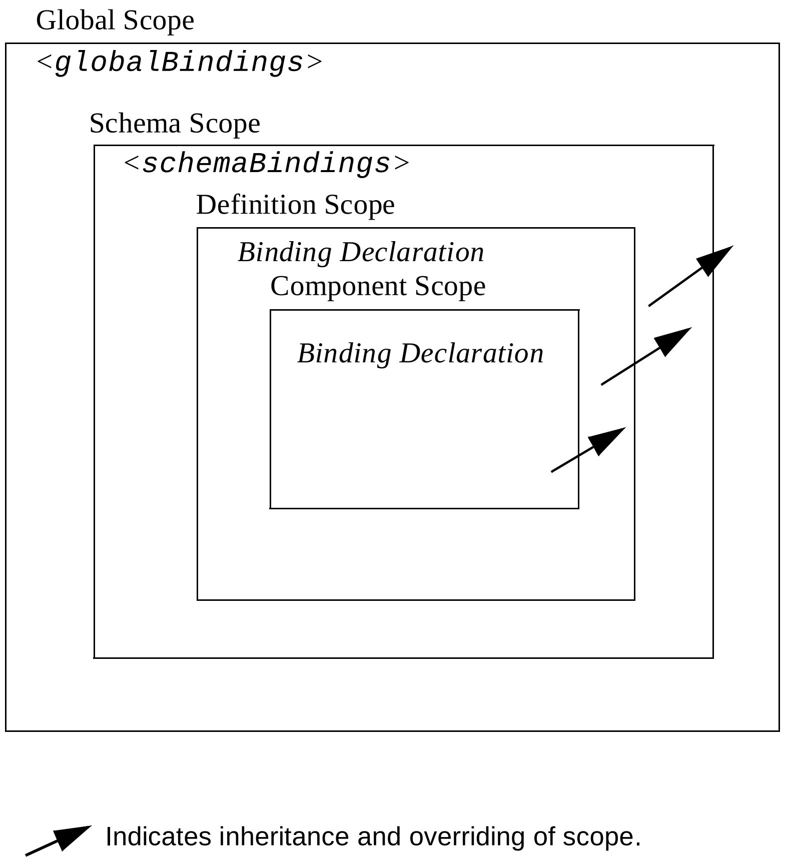

Figure 6.1 Scoping Inheritance and Overriding For Binding Declarations

The different scopes form a taxonomy. The taxonomy defines both the inheritance and overriding semantics of customization values. A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- a schema element in schema scope inherits a customization value defined in global scope.
- a schema element in definition scope inherits a customization value defined in schema or global scope.
- a schema element in component scope inherits a customization value defined in definition, schema or global scope.

Likewise, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- value in schema scope overrides a value inherited from global scope.
- value in definition scope overrides a value inherited from schema scope or global scope.
- value in component scope overrides a value inherited from definition, schema or global scope.

6.4.2 XML Schema Parsing

Chapter 5 specified the bindings using the abstract schema model. Customization, on the other hand, is specified in terms of XML syntax not abstract schema model. The XML Schema specification [XSD PART 1] specifies the parsing of schema elements into abstract schema components. This parsing is assumed for parsing of annotation elements specified here. In some cases, [XSD PART 1] is ambiguous with respect to the specification of annotation elements. Section 6.12, “Annotation Restrictions” outlines how these are addressed.

Design Note – The reason for specifying using the XML syntax instead of abstract schema model is as follows. For most part, there is a one-to-one mapping between schema elements and the abstract schema components to which they are bound. However, there are certain exceptions: local attributes and particles. A local attribute is mapped to two schema components: {attribute declaration} and {attribute use}. But the XML parsing process associates the annotation with the {attribute declaration} not the {attribute use}. This is tricky and not obvious. Hence for ease of understanding, a choice was made to specify customization at the surface syntax level instead.

6.5 `<globalBindings>` Declaration

The customization values in “`<globalBindings>`” binding declaration have global scope. This binding declaration is therefore useful for customizing at a global level.

6.5.1 Usage

```
<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>
```

The following customization values are defined in global scope:

- *`collectionType`* if specified, must be either “`indexed"` or any fully qualified class name that implements *`java.util.List.`*

- *`fixedAttributeAsConstantProperty`* if specified , defines the customization value *`fixedAttributeAsConstantProperty`*. The value must be one of `"true", false", "1" or"0".` The default value is `"false"`.
- *`generateIsSetMethod`* if specified, defines the customization value of *`generateIsSetMethod.`* The value must be one of `"true", false", "1" or"0".` The default value is `"false"`.
- *`enableFailFastCheck`* if specified, defines the customization value *`enableFailFastCheck.`* The value must be one of `"true", "false", "1" or"0".` If *`enableFailFastCheck`* is `"true"` or `"1"` and the JAXB implementation supports this optional checking, type constraint checking when setting a property is performed as described in Section 4.5, “Properties". The default value is `"false"`.
- *`choiceContentProperty`* if specified,defines the customization value *`choiceContentProperty`* . The value must be one of `"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 as specified in Section 6.1.5, “Invalid Customizations".
- *`underscoreBinding`* if specified, defines the customization value *`underscoreBinding`*. The value must be one of `"asWordSeparator" or "asCharInWord".` The default value is `"asWordSeparator"`.
- *`enableJavaNamingConventions`* if specified, defines the customization value *`enableJavaNamingConventions`*. The value must be one of `"true", false", "1" or"0".` The default value is `"true"`.
- *`typesafeEnumBase`* if specified, defines the customization value *`typesafeEnumBase.`* The value must be a list of QNames, each of which must resolve to a simple type definition. Only simple type definitions with an enumeration facet and a restriction base type listed in *`typesafeEnumBase`* or derived from a type listed in *`typesafeEnumBase`* is bound to a *`typesafeEnumClass`* by default as specified in Section 5.2.3, “Type Safe Enumeration". The default value of *`typesafeEnumBase`* is `"xs:NCName".`

  The*`typesafeEnumBase`* cannot contain the following simple types and therefore a JAXB implementation is not required to support the binding of the these types to typesafe enumeration class: *"xs:QName", "xs:base64Binary", "xs:hexBinary", "xs:date", "xs:time", "xs:dateTime", "xs:duration", "xs:gDay", "xs:gMonth", "xs:gYear", "xs:gMonthDay", "xs:YearMonth".* If any of them are specified, it must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations." JAXB implementation must be capable of binding any other simple type listed in *typesafeEnumBase* to a typesafe enumeration class.

- *typesafeEnumMemberName* if specified, defines the customization value *typesafeEnumMemberName.* The value must be one of "generateError" or "generateName". The default value is "generateError".
- *bindingStyle* if specified, defines the customization value *bindingStyle*. The value must be one of "elementBinding", or "modelGroupBinding". The default value is "elementBinding".
  If the value is "elementBinding", the binding style specified in Section 5.9.7, "Content Model Default Binding" is used. If the value is "modelGroupBinding" then the binding style specified in Section 5.9.8, "Model group binding style" is selected.
- zero or more *javaType* binding declarations. Each binding declaration must be specified as described in Section 6.9, "<javaType> Declaration," on page 151."

The semantics of the above customization values, if not specified above, are specified when they are actually used in the binding declarations.

For inline annotation, a <globalBindings> is a valid only in the annotation element of the <schema> element. There must only be a single instance of a <globalBindings> declaration in the annotation element of the <schema> element.

If one source schema includes or imports a second source schema then the <globalBindings> declaration must be declared in the first source schema.

6.5.2 Customized Name Mapping

A customization value can be used to specify a name for a Java object (e.g. class name, package name etc.). In this case, a customization value is referred to as a *customization name*.

A customization name is always a legal Java identifier (this is formally specified in each binding declaration where the name is specified). Since customization deals with customization of a Java representation to which an XML schema element is bound, requiring a customization name to be a legal Java identifier rather than an XML name is considered more meaningful.

A customization name may or may not conform to the recommended Java language naming conventions. [JLS - Java Language Specification, Second Edition, Section 6.8, "Naming Conventions"]. The customization value *enableJavaNamingConventions* determines if a customization name is mapped to a Java identifier that follows Java language naming conventions or not.

If *enableJavaNamingConventions* is defined and the value is `"true"` or `"1"`, then the customization name (except for constant name) specified in the section from where this section is referenced must be mapped to Java identifier which follows the Java language naming conventions as specified in Section C.6, "Conforming Java Identifier Algorithm"; otherwise the customized name must be used as is.

6.5.3 Underscore Handling

This section applies only when XML names are being mapped to a legal Java Identifier by default. In this case, the treatment of underscore ('_') is determined by *`underscoreBinding.`*

If *`underscoreBinding`* is `"asWordSeparator",` then underscore ('_') must be treated as a punctuation character; otherwise if *`underscoreBinding`* is `"asCharInWord",` then underscore ('_') must be treated as a character in the word. The default value for *`underscoreBinding`* is `"asWordSeparator"`.

6.6 <schemaBindings> Declaration

The customization values in `<schemaBindings>` binding declaration have schema scope. This binding declaration is therefore useful for customizing at a schema level.

6.6.1 Usage

```
<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>
```

For readability, the `<nameXmlTransform>` and `<package>` elements are shown separately. However, they are local elements within the `<schemaBindings>` element.

The semantics of the customization value are specified when they are actually used in the binding declarations.

For inline annotation, a `<schemaBindings>` is valid only in the annotation element of the `<schema>` element. There must only be a single instance of a `<schemaBindings>` declaration in the annotation element of the `<schema>` element.

If one source schema includes (via the include mechanism specified by XSD PART 1) a second source schema, then the `<schemaBindings>` declaration must be declared in the first including source schema. It should be noted that there is no such restriction on `<schemaBindings>` declarations when one source schema imports another schema since the scope of `<schemaBindings>` binding declaration is schema scope.

6.6.1.1 package

Usage

- *`name`* if specified, defines the customization value *`packageName`*. *`packageName`* must be a valid Java package name.
- <*`javadoc`*> if specified, customizes the package level Javadoc. <*`javadoc`*> must be specified as described in Section 6.11, “<javadoc> Declaration.” The Javadoc must be generated as specified in Section 6.11.3, “Javadoc Customization.” The Javadoc section customized is the `package section.`

Design Note – The word “package” has been prefixed to *`name`* used in the binding declaration. This is because the attribute or element tag names “name” is not unique by itself across all scopes. For e.g., “name” attribute can be specified in the <property> declaration. The intent is to disambiguate by reference such as “`packageName`”.

The semantics of the *`packageName`* is specified in the context where it is used. If neither *`packageName`* nor the <*`javadoc`*> element is specified, then the binding declaration has no effect.

Example: Customizing Package Name

```
<jaxb:schemaBindings>
   <jaxb:package name = "org.example.po" />
</jaxb:schemaBindings>
```

specifies “`org.example.po`” as the package to be associated with the schema.

6.6.1.2 nameXmlTransform

The use case for this declaration is the UDDI Version 2.0 schema. The UDDI Version 2.0 schema contains many declarations of the following nature:

```
<xs:element name="bindingTemplate" type="uddi:bindingTemplate"/>
```

The above declaration results in a name collision since both the element and type names are the same - although in different XML Schema symbol spaces. Normally, collisions are supposed to be resolved using customization. However, since there are many collisions for the UDDI V2.0 schema, this is not a convenient solution. Hence the binding declaration `nameXmlTransform` is being provided to automate name collision resolution.

The `nameXmlTransform` allows a *`suffix`* and a *`prefix`* to be specified on a per symbol space basis. The following symbol spaces are supported:

- `<typeName>` for the symbol space "type definitions"
- `<elementName>` for the symbol space "element definitions"
- `<modelGroupName>` for the symbol space "model group definitions."
- `<anonymousTypeName>` for customizing Java content interface to which an anonymous type is bound.[1]

If *`suffix`* is specified, it must be appended to all the default XML names in the symbol space. The *`prefix`* if specified, must be prepended to the default XML name. Furthermore, this XML name transformation must be done before the XML name to Java Identifier algorithm is applied to map the XML name to a Java identifier. The XML name transformation must not be performed on customization names.

By using a different *`prefix`* and/or *`suffix`* for each symbol space, identical names in different symbol spaces can be transformed into non-colliding XML names.

1. XML schema does not associate anonymous types with a specific symbol space. However, `nameXmlTransform` is used since it provides a convenient way to customize the Java content interface to which an anonymous type is bound.

`anonymousTypeName`

As specified in Section 5.7.2, “Binding of an anonymous complex type definition”, by default a “Type” suffix is added to the name of the Java content interface to which an anonymous type is bound. The `<anonymousTypeName>` declaration can be used to customize the suffix and prefix for the Java content interface. If *`suffix`* is specified, it must replace the “Type” suffix in the Java content interface name. If *`prefix`* is specified, then it must be prepended to the Java content interface name for the anonymous type.

6.7 `<class>` Declaration

This binding declaration can be used to customize the binding of a schema element to a Java content interface or a Java Element interface. The customizations can be used to specify:

- a name for the derived Java interface.
- an implementation class for the derived Java content interface. An implementation cannot be specified for a Java Element interface.

Specification of an alternate implementation for a Java content interface allows implementations generated by a tool (e.g. based on UML) to be used in place of the default implementation generated by a JAXB provider.

The implementation class may have a dependency upon the runtime of the binding framework. Since a runtime is not specified in this version of the specification, the implementation class may not be portable across JAXB provider implementations. Hence one JAXB provider implementation is not required to support the implementation class from another JAXB provider.

6.7.1 Usage

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

- *`className`* is the name of the derived Java interface, if specified. 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`* if specified, is the name of the implementation class for *`className`* and must include the complete package name.
- *`<javadoc>`* element, if specified customizes the Javadoc for the derived Java interface. *`<javadoc>`* must be specified as described in Section 6.11, "<javadoc> Declaration."

6.7.2 Customization Overrides

When binding a schema element's Java representation to a Java content interface or a Java Element interface, the following customization values override the defaults specified in Chapter 5. It is specified in a common section here and referenced from Section 6.7.3, "Customizable Schema Elements."

- **name:** The name is *`className`* if specified.
- **package name:** The name of the package is *`packageName`* inherited from a scope that covers this schema element.

  NOTE: The *`packageName`* is only set in the <package> declaration. The scope of *`packageName`* is schema scope and is thus inherited by all schema elements within the schema.
- **javadoc:** The Javadoc must be generated as specified in section Section 6.11.3, "Javadoc Customization." The Javadoc section customized is the `class/interface section.`

6.7.3 Customizable Schema Elements

6.7.3.1 Complex Type Definition

When `<class>` customization specified in the annotation element of the complex type definition, the complex type definition must be bound to a Java content interface as specified in Section 5.3.2, "Java Content Interface" applying the customization overrides as specified in Section 6.7.2, "Customization Overrides."

Example: Class Customization: Complex Type Definition To Java Content Interface

XML Schema fragment:

```
<xs:complexType name="USAddress">
   <xs:annotation> <xs:appinfo>
      <jaxb:class name="MyAddress" />
   </xs:appinfo></xs:annotation>
   <xs:sequence>...</xs:sequence>
   <xs:attribute name="country" type="xs:string"/>
</xs:complexType>
```

Customized code:

```
// public interface USAddress { // Default Code
public interface MyAddress {  // Customized Code
   public String getCountry();
   public void setCountry(String value);
   ...
}
```

6.7.3.2 Model Group Definition

When a `<class>` declaration is specified in the annotation element of a model group definition, the model group definition must be bound to a Java content interface as specified in Section 5.5.3, “Bind to a Java content interface” applying the customization overrides as specified in Section 6.7.2, “Customization Overrides.”

Example: Class Customization: Model Group Definition To Class

XML Schema Fragment:

```
<xs:group name="AModelGroup">
   <xs:annotation> <xs:appinfo>
      <jaxb:class name="MyModelGroup" />
   </xs:appinfo></xs:annotation>
   <xs:choice>
      <xs:element name="A" type="xs:int"/>
      <xs:element name="B" type="xs:float"/>
   </xs:choice>
</xs:group>
```

Customized code:

```
// interface AModelGroup // Default code
interface MyModelGroup { // Customized code ( customized class name )
    void setA(int value);
    int getA();

    void setB(float value);
    float getB();
}
```

6.7.3.3 Model Group

When a <`class`> customization is specified in the annotation element of the model group's compositor, the model group must be bound to a Java content interface as specified in Section 5.9.10, "Bind Nested Model Group" applying the customization overrides as specified in Section 6.7.2, "Customization Overrides."

6.7.3.4 Global Element Declaration

A <`class`> declaration is allowed in the annotation element of the global element declaration. However, the *`implClass`* attribute is not allowed. The global element declaration must be bound as specified in Section 5.7.1, "Bind to Java Element Interface" applying the customization overrides A specified in Section 6.7.2, "Customization Overrides."

Example: Class Customization: Global Element to Class

XML Schema Fragment:

```
<xs:complexType name="AComplexType">
    <xs:sequence>
        <xs:element name="A" type="xs:int"/>
        <xs:element name="B" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="AnElement" type="AComplexType">
    <xs:annotation><xs:appinfo>
        <jaxb:class name="MyElement"/>
    </xs:appinfo></xs:annotation>
</xs:element>
```

Customized code:

```
    public interface AComplexType {
        void setA(int value);
        int getA();
        void setB(String value);
        String getB();
    };
    // following interface would be generated by default
    // public interface AnElement extends AComplexType,
    //                                       javax.xml.jaxb.Element {};

    // following interface is generated because of customization
    public interface MyElement extends AComplexType,
                                          javax.xml.jaxb.Element {};
    public class ObjectFactory {
        // AnElement  createAnElement(); // Default code
        AnElement     createMyElement(); // Customized code
        AComplexType  createAComplexType();
        ... other factory methods ...

}
```

6.7.3.5 Local Element

A local element is a schema element that occurs within a complex type definition. A local element is one of:

- local element reference (using the “ref” attribute) to a global element declaration.
- local element declaration (“ref” attribute is not used).

A `<class>` declaration is allowed in the annotation element of a local element. Section 6.12, “Annotation Restrictions” contains more information regarding the annotation element for a local element reference. However, the *`implClass`* attribute is not allowed.

A `<class>` customization on local element reference must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations” since a local element reference is never bound to a Java Element interface.

A `<class>` customization on local element declaration applies only when a local element declaration is bound to a Java Element interface. Otherwise it must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.” If applicable, a local element must be bound as specified in

Section 5.7.1, “Bind to Java Element Interface” applying the customization overrides as specified in Section 6.7.2, “Customization Overrides.”

Example: Class Customization: Local Element Declaration To Java Element Interface

The following example is from Section 5.9.3.1, “Examples.”

XML Schema fragment:

```
<xs:complexType name="Base">
   <xs:choice maxOccurs="unbounded">
      <xs:element name="A" type="xs:string">
         <xs:annotation><xs:appinfo>
            <jaxb:class name="Bar"/>
         </xs:appinfo></xs:annotation>
      </xs:element>
      <xs:element name="B" type="xs:string"/>
      <xs:element name="C" type="xs:int"/>
   </xs:choice>
</xs:complexType>
```

Customized code:

```
import javax.xml.bind.Element;
interface Base {
   // interface A extends Element {...} // Default code
   interface Bar extend Element {...}// Customized code
   interface B extends Element {...}
   interface C extends Element {...}

   /**
    * A general content list that can contain
    * element instances of Base.Bar,Base.B and Base.C.
    */
   List getAOrBOrC();
}
```

6.8 `<property>` Declaration

This binding declaration allows the customization of a binding of an XML schema element to its Java representation as a property. This section identifies all XML schema elements that can be bound to a Java property and how to customize that binding.

The scope of customization value can either be definition scope or component scope depending upon which XML schema element the <*property*> binding declaration is specified.

6.8.1 Usage

```
<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>
```

For readability, the `<baseType>` element is shown separately. However, it can be used only as a local element within the `<property>` element.

The use of this declaration is subject to the constraints specified in Section 6.8.1.2, "Usage Constraints."

The customization values defined are:

- *`name`* if specified , defines the customization value *`propertyName;`* it must be a legal Java identifier.
- *`collectionType`* if specified, defines the customization value *`propertyCollectionType`* which is the collection type for the property. *`propertyCollectionType`* if specified, must be either "`indexed"` or any fully qualified class name that implements *`java.util.List.`*
- *`fixedAttributeAsConstantProperty`* if specified , defines the customization value *`fixedAttributeAsConstantProperty`*. The value must be one of `"true", false", "1" or"0".`

- *`generateIsSetMethod`* if specified, defines the customization value of *`generateIsSetMethod.`* The value must be one of `"true", false", "1" or"0".`
- *`enableFailFastCheck`* if specified, defines the customization value *`enableFailFastCheck.`* The value must be one of `"true", false", "1" or"0".`
- *`<javadoc>`* element, if specified customizes the Javadoc for the property's getter method. *`<javadoc>`* must be specified as described in Section 6.11, "<javadoc> Declaration."

6.8.1.1 baseType

The `<baseType>` element is intended to allow the customization of a base type a property. Specifically, it allows the customization of a simple type at the point of reference to the simple type. A future version of this specification my define additional uses for `<baseType>`. The syntax for `<baseType>` is designed to accommodate both the current and the intended future uses.

The <*`javaType`*>, if specified, defines the customization value *`javaType`* and must be specified as specified in Section 6.9, "<javaType> Declaration." The customization value defined has component scope.

If `<javaType>` is specified with a XML schema element not listed above, it must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations."

6.8.1.2 Usage Constraints

The usage constraints on `<property>` are specified below. Any constraint violation must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations." The usage constraints are:

1. The `<baseType>` is only allowed with the following XML schema elements from the Section 6.8.3, "Customizable Schema Elements":
   a. Local Element, Section 6.8.3.4, "Local Element."
   b. Local Attribute, Section 6.8.3.2, "Local Attribute."
   c. ComplexType with simpleContent, Section 6.8.3.8, "ComplexType."
2. The *`fixedAttributeAsConstantProperty`* is only allowed with a local attribute, Section 6.8.3.2, "Local Attribute" , that is fixed.

3. If a `<property>` declaration is associated with the `<complexType>`, then a `<property>` customization cannot be specified on the following schema elements that are scoped to `<complexType>`:

    a. Local Element

    b. Model group

    c. Model Group Reference

    The reason is that a `<property>` declaration associated with a complex type binds the content model of the complex type to a general content property. If a `<property>` declaration is associated with a schema element listed above, it would create a conflicting customization.

Design Note – A Local Attribute is excluded from the list above. The reason is that a local attribute is not part of the content model of a complex type. This allows a local attribute to be customized (using a <property> declaration) independently from the customization of a complex type's content model.

Example: Property Customization: simple type customization

```
<xs:complexType name="internationalPrice">
    ....
    <xs:attribute name="currency" type="xs:string">
        <xs:annotation> <xs:appinfo>
            <jaxb:property>
                <jaxb:baseType>
                    <jaxb:javaType name="java.math.BigDecimal"
    parseMethod="javax.xml.bind.DatatypeConverter.parseInteger"
    printMethod="javax.xml.bind.DatatypeConverter.printInteger"/>
                </jaxb:baseType>
            </jaxb:property>
        </xs:appinfo></xs:annotation>
    </xs:attribute>
</xs:complexType>
```

The code generated is:

```
public interface InternationalPrice {
    // String getCurrency(); default
    java.math.BigDecimal getCurrency() ; // customized
    public void setCurrency(java.math.BigDecimal val); // customized
}
```

6.8.2 Customization Overrides

When binding a schema element's Java representation to a property, the following customization values override the defaults specified in Chapter 5. It is specified in a common section here and referenced from Section 6.8.3, "Customizable Schema Elements."

- **name:** If *propertyName* is defined, then it is the name obtained by mapping the name as specified in Section 6.5.2, "Customized Name Mapping."
- **base type:** The basetype is *`propertyBaseType`* if defined. The *propertyBaseType* is defined by a XML schema element in Section 6.8.3, "Customizable Schema Elements."
- **collection type:** The collection type is *`propertyCollectionType`* if specified; otherwise it is the *`propertyCollectionType`* inherited from a scope that covers this schema element.
- **javadoc:** The Javadoc must be generated as specified in section Section 6.11.3, "Javadoc Customization." The Javadoc section customized is the `method section.`
- If *`propertyBaseType`* is a Java primitive type and *`propertyCollectionType`* is a class that implements java.util.List, then the primitive type must be mapped to its wrapper class.

The following does not apply if local attribute is being bound to a constant property as specified in Section 6.8.3.2, "Local Attribute":

- If *`generateIsSetMethod`* is `"true"` or `"1",` then additional methods as specified in Section 4.5.4, "isSet Property Modifier" must be generated.
- If *`enableFailFastCheck`* is `"true"` or `"1"` then the type constraint checking when setting a property is enforced by the JAXB implementation. Support for this feature is optional for a JAXB implementation in this version of the specification.

6.8.3 Customizable Schema Elements

6.8.3.1 Global Attribute Declaration

A <*`property`*> declaration is allowed in the annotation element of the global attribute declaration.

The binding declaration does not bind the global attribute declaration to a property. Instead it defines customization values that have definition scope. The definition scope covers all local attributes (Section 6.8.3.2, “Local Attribute”) that can reference this global attribute declaration. This is useful since it allows the customization to be done once when a global attribute is defined instead of at each local attribute that references the global attribute declaration.

6.8.3.2 Local Attribute

A local attribute is an attribute that occurs within an attribute group definition, model group definition or a complex type. A local attribute can either be a

- local attribute reference (using the “ref” attribute) to a global attribute declaration.
- local attribute declaration (“ref” attribute is not used).

A <*`property`*> declaration is allowed in the annotation element of a local attribute.Chapter 6, “Annotation Restrictions” contains more information regarding the annotation element for a local attribute reference. The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope.

If *`javaType`* is defined, then the *`propertyBaseType`* is defined to be Java datatype specified in the `"name"` attribute of the *`javaType`*.

- If *`fixedAttributeAsConstantProperty`* is “*`true"`* or “*`1"`* and the local attribute is a fixed, the local attribute must be bound to a Java Constant property as specified in Section 5.8.1, “Bind to a Java Constant property” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.” The *`generateIsSetMethod, choiceContentProperty`* and *`enableFailFastCheck`* must be considered to have been set to false.

- Otherwise, it is bound to a Java property as specified in Section 5.8, “Attribute use” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.”

Example: Customizing Java Constant Property

XML Schema fragment:

```
<xs:complexType name="USAddress">
   <xs:attribute name="country" type="xs:NMTOKEN" fixed="US">
      <xs:annotation><xs:appinfo>
         <jaxb:property name="MY_COUNTRY"
                        fixedAttributeAsConstantProperty="true"/>
      </xs:appinfo></xs:annotation>
   </xs:attribute>
</xs:complexType>
```

Customized derived code:

```
public interface USAddress {
   public static final String MY_COUNTRY = "US"; // Customized Code
}
```

Example 2: Customizing to other Java Property

XML Schema fragment:

```
<xs:complexType name="USAddress">
   <xs:attribute name="country" type="xs:string">
      <xs:annotation><xs:appinfo>
         <jaxb:propertyname="MyCountry"/>
      </xs:appinfo></xs:annotation>
   </xs:attribute>
</xs:complexType>
```

Customized derived code:

```
public interface USAddress {
   // public getString getCountry();     // Default Code
   // public void setCountry(string value);// Default Code
   public String getMyCountry();           // Customized Code
   public void setMyCountry(String value); // Customized Code
}
```

Example 3: Generating IsSet Methods

XML Schema fragment:

```
<xs:attribute name="account" type = "xs:int">
   <xs:annotation><xs:appinfo>
      <jaxb:property generateIsSetMethod="true"/>
   </xs:appinfo></xs:annotation>
</xs:attribute>
```

Customized code:

```
public int getAccount();
public void setAccount(int account);
public boolean isSetAccount(); // Customized code
public void unsetAccount();    // Customized code
```

6.8.3.3 Global Element Declaration

A *<property>* declaration is allowed in the annotation element of a global element declaration. However, the usage is constrained as follows:

The binding declaration does not bind the global element declaration to a property. Instead it defines customization values that have definition scope. The definition scope covers all local elements (Section 6.8.3.4, “Local Element”) that can reference this global element declaration. This is useful since it allows the customization to be done once when a global element is defined instead of at each local element that references the global element declaration.

6.8.3.4 Local Element

A local element is a schema element that occurs within a complex type definition. A local element is one of:

- local element reference (using the “ref” attribute) to a global element declaration.
- local element declaration (“ref” attribute is not used).

A <property> declaration is allowed in the annotation element of a local element. Section 6.12, “Annotation Restrictions” contains more information regarding the annotation element for a local element reference.

The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope.

If *javaType* is defined, then the *propertyBaseType* is defined to be Java datatype specified in the "name" attribute of the *javaType*.

The local element must be bound as specified in Section 5.9.7, “Content Model Default Binding” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.”

See example in “Example 3: Property Customization: Model Group To Content Property Set” in section Section 6.8.3.6, “Model Group.”

Relationship To Element Binding

The <property> declaration takes precedence over elementBinding, if specified and must therefore be processed before element binding is attempted.

Example: Property Customization: Element Binding

The following schema fragment

```
<xs:complexType name="Bar">
    <xs:choice>
        <xs:sequence>
            <xs:element name = "A" type = "xs:int">
                <xs:annotation><xs:appinfo>
                    <jaxb:property name="foo"/> <!--customization-->
                </xs:appinfo></xs:annotation>
            </xs:element>
        </xs:sequence>
        <xs:sequence>
            <xs:element name = "B" type = "xs:string"/>
            <xs:element name = "A" type = "xs:int"/>
        </xs:sequence>
    </xs:choice>
</xs:complexType>
```

The generated code is:

```
interface Bar {
   public int getA();
   public void setA(int value); // accessor for element A
   public String setB();
   public void setB(String value);
   public int getFoo();     // another accessor for element A
   public void setFoo(int value);
}
```

Since <property> declaration taking precedence, this can lead to generation of multiple accessor and mutator methods for local elements. If `bindingStyle` is `elementBinding,` either mutator method may be used to set the content. In the above example, the content for the second sequence may be set in one of the following ways:

```
// one way to set content for second sequence
setB();
setA();

// another way to set content for second sequence
setB();
setFoo();
```

When `elementBinding` is specified, the multiple accessors/mutators can be eliminated by associating the same `<property>` declaration with multiple occurrences of local elements. For e.g.,

```
<xs:complexType name="bar">
   <xs:choice>
      <xs:sequence>
         <xs:element name = "A" type = "xs:int">
            <xs:annotation><xs:appinfo>
               <jaxb:property name="foo"/>
            </xs:appinfo></xs:annotation>
         </xs:element>
      </xs:sequence>
      <xs:sequence>
         <xs:element name = "B" type = "xs:string"/>
         <xs:element name = "A" type = "xs:int">
            <xs:annotation><xs:appinfo>
               <jaxb:property name="foo"/>
            </xs:appinfo></xs:annotation>
            </xs:element>
        </xs:sequence>
    </xs:choice>
</xs:complexType>
```

6.8.3.5 Wildcard

A `<property>` declaration is allowed in the annotation element of the wildcard schema component. The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope.

The wildcard schema component must be bound to a property as specified in Section 5.9.5, “Bind wildcard schema component” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.”

Example: The following schema example is from UDDI V2.0

```
<xs:complexType name="businessEntityExt">
    <xs:sequence>
        <xs:any namespace="##other"
            processContents="strict"
            minOccurs="1" maxOccurs="unbounded">
            <xs:annotation><xs:appinfo>
                <jaxb:property name="Extension"/>
            </xs:appinfo></xs:annotation>
        </xs:any>
        ....
    </xs:sequence>
</xs:complexType>
```

Customized derived code:

```
public interface BusinessEntityExt {
    ...
    // List getAny(); // Default Code
    List getExtension(); // Customized Code
}
```

6.8.3.6 Model Group

A `<property>` binding declaration is allowed in the annotation element of the compositor (i.e. `<choice>`, `<sequence>` or `<all>`). The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope.

The customized binding of a model group is determined by the following:

- `modelGroupBinding` attribute in `<globalBindings>`.
- `choiceContentProperty` attribute in `<globalBindings>`.
- any `<class>` customization associated with the model group in addition to this `<property>` declaration.

The following table shows the binding based on all three of the above. The table is non-normative and is intended for ease of understanding of the model group customization. Normative text follows the table.

Table 6-1 Model group binding with customizations (Non-normative)

ID	**globalBinding. bindingStyle**	**globalBinding. choiceContent Property**	**property**	**class**	**Binding**
1	Element Binding	true/false/not set (value not used)	not set	not set	model group bound to general content property.
2	Element Binding	true / false/ not set (value not used)	set	not set	model group bound to a general content property;
3	Element Binding	true / false / not set (value not used)	not set	set (allows class customization)	model group is bound to a class;
4	Element Binding	true / false / not set (value not used)	set (allows property customiza tion)	set (allows class customization)	model group bound to a content interface;
5	Model Group Binding	false or not set	not set	not set	choice binds to choice content interface; other model groups bind to content interface.

Table 6-1 Model group binding with customizations (Non-normative) *(Continued)*

6	Model Group Binding	true	not set	not set	choice binds to choice content property; other model groups bind to content interface.
7	Model Group Binding	false / true / not set (value not used)	not set	set (allows content interface customization)	model group binds to content interface;
8	Model Group Binding	false / true / not set (value not used)	set (allows property customiza tion)	not set	choice binds to choice content property; other model groups bind to content interface;
9	Model Group Binding	false / true / not set (value not used)	set (allows property customiza tion)	set (allows content interface customization)	model group - including choice - bind to content interface;

If `elementBinding` is specified then the binding algorithm specified in section Section 5.9.7, “Content Model Default Binding” is in effect. The value of the `choiceContentProperty`, if specified has no effect when this binding style is in effect. A model group must be bound as follows:

1. Assume *propertySet* and *propertyBaseType* are both undefined to start with.
2. If a `<class>` declaration is also associated with the model group, then the model group must be bound to Java content interface as specified in Section 6.7.3.3, “Model Group.” The *propertyBaseType* is defined to be the Java content interface to which the model group is bound.
3. Otherwise if there is a `<property>` declaration, then the model is bound to a general content property as specified in Section 5.9.6, “Bind a repeating occurrence model group” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.” The *propertySet* is defined to be the general content property to which the model group is bound.

4. If *propertyBaseType* is defined, and a `<property>` declaration is also present, then the customization overrides specified in Section 6.8.2, “Customization Overrides” must be applied by the model group’s parent schema element to the property used to aggregate the Java content interface.
5. If *propertySet* is defined, then the model group’s parent schema element must aggregate the property set as specified in Section 5.3.1.2, “Aggregation of Property Set.”

When `bindingStyle=modelGroupBinding`, the model group is binding is in effect. The value of the `choiceContentProperty`, if specified does impact the model group binding in this case. A model must be bound as follows following the steps in order.

1. Assume *propertySet* and *propertyBaseType* are both undefined to start with.
2. If `<class>` declaration is associated with the model group, then the model group is bound to Java content interface as specified in Section 6.7.3.3, “Model Group.” The *propertyBaseType* is defined to be the Java content interface to which the model group is bound.
3. Otherwise if there is a `<property>` declaration and `choiceContentProperty` is `"true"` and the model group is a choice model group, then the choice model group is bound to a choice content property as specified in Section 5.9.10.2, “Bind choice group to a choice content property” applying customization overrides specified in Section 6.8.2, “Customization Overrides.” The *propertySet* is defined to be the set of properties generated for the choice content property.
4. Otherwise, if there is `<property>` declaration, then the model group is bound to a class as specified in Section 5.9.10, “Bind Nested Model Group.” The *propertyBaseType* is defined to be the Java content interface to which the model group is bound.
5. If *propertyBaseType* is defined and a `<property>` declaration is also present, then the customization overrides specified in Section 6.8.2, “Customization Overrides” must be applied by the model group’s parent schema element to the property used to aggregate the Java content interface.
6. If *propertySet* is defined, then the model group’s parent schema element

must aggregate the property set as specified in Section 5.3.1.2, “Aggregation of Property Set.”

Example1: Property Customization: Model Group To ChoiceContent Property

XML Schema fragment

```
<xs:annotation><xs:appinfo>
   <jaxb:globalBindings bindingStyle="modelGroupBinding"
                        choiceContentProperty="true"/>
</xs:appinfo></xs:annotation>
<xs:complexType name=”AType”>
   <xs:choice>
      <xs:element name="foo" type="xs:int"/>
      <xs:element name="bar" type="xs:string"/>
   </xs:choice>
</xs:complexType>
```

Customized derived code:

```
interface AType {
   interface Foo extends javax.xml.bind.Element { ... }
   interface Bar extends javax.xml.bind.Element { ... }

   void setFooOrBar(java.lang.Object o);  //customized code
   Object getFooOrBar();                  // customized code
}
```

The `choiceContentProperty` is required to bind the choice model group to a choice content property. The `bindingStyle` attribute is also required since `choiceContentProperty` is only applicable when the binding style is `modelGroupBinding.`

Example 2: Property Customization: Model Group To General Content Property

XML Schema fragment:

```
<xs:complexType name="Base">
    <xs:choice maxOccurs="unbounded">
        <xs:annotation><xs:appinfo>
            <jaxb:property name="items" />
        </xs:appinfo></xs:annotation>
        <xs:element name="A" type="xs:string"/>
        <xs:element name="B" type="xs:string"/>
        <xs:element name="C" type="xs:int"/>
    </xs:choice>
</xs:complexType>
```

Customized derived code:

```
interface Base {
    interface A extends javax.xml.bind.Element {...}
    interface B extends javax.xml.bind.Element {...}
    interface C extends javax.xml.bind.Element {...}
    /**
     * A general content list that can contain
     * instances of Base.A, Base.B and Base.C.
     */
    // List getAOrBOrC(); - default
    List getItems();// Customized Code
}
```

Example 3: Property Customization: Model Group To Content Property Set

XML Schema fragment:

```
<xs:complexType name="USAddress"/>
<xs:complexType name="PurchaseOrderType">
   <xs:sequence>
      <xs:choice>
         <xs:group   ref="shipAndBill"/>
         <xs:element name="singleUSAddress" type="USAddress">
            <xs:annotation><xs:appinfo>
               <jaxb:property name="address"/>
            </xs:appinfo></xs:annotation>
         </xs:element>
      </xs:choice>
   </xs:sequence>
</xs:complexType>
<xs:group name="shipAndBill">
   <xs:sequence>
      <xs:element name="shipTo" type="USAddress">
         <xs:annotation><xs:appinfo>
            <jaxb:property name="shipAddress"/>
         </appinfo></annotation>
      </xs:element>
      <xs:element name="billTo" type="USAddress">
         <xs:annotation><xs:appinfo>
            <jaxb:property name="billAddress"/>
         </xs:appinfo></xs:annotation>
      </xs:element>
   </xs:sequence>
</xs:group>
```

Customized derived code:

```
public interface PurchaseOrderType {
   USAddress getShipAddress(); void setShipAddress(USAddress);
   USAddress getBillAddress(); void setBillAddress(USAddress);
   USAddress getAddress();     void setAddress(USAddress);
}
```

6.8.3.7 Model Group Reference

A model group reference is a reference to a model group using the "ref" attribute. A property customization is allowed on the annotation property of the model group reference. Section Chapter 6, “Annotation Restrictions” contains more information regarding the annotation element for a model group reference.

The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope. A model group reference is bound to a Java property set or a list property as specified in Chapter 5, “Content Model Default Binding” applying customization overrides as specified in Section 6.8.2, “Customization Overrides.”

6.8.3.8 ComplexType

A `<property>` customization is allowed on the annotation element of a complex type. The customization values must be defined as specified in Section 6.8.1, “Usage” and have component scope. The result of this customization depends upon the content type of the complex type.

- If the content type of the content model is simple content, then the content model must be bound to a property as specified in Section 5.3.2.1, “Simple Content Binding.” applying the customization overrides as specified in Section 6.8.2, “Customization Overrides.” If *`javaType`* is defined, then the *`propertyBaseType`* is defined to be Java datatype specified in the `"name"` attribute of the *`javaType`*.
- For all other content types, the content model is bound to a property as specified by one of the two binding algorithms:
  - If the `bindingStyle` is `elementBinding`, then the content model must be bound as specified in step 1. of Section 5.9.7, “Content Model Default Binding” applying the customization overrides as specified in Section 6.8.2, “Customization Overrides".
  - If the `bindingStyle` is `modelGroupBinding`, then the content model is bound as specified in step 1a. of Section 5.9.11, “Model Group binding algorithm.” applying the customization overrides as specified in Section 6.8.2, “Customization Overrides".

Design Note – The <property> declaration is not allowed on an annotation element of attribute group definition. However, attributes within the attribute group definition can themselves be customized as described in the “Local Attribute” section above. Section 6.8.3.2, “Local Attribute.”

6.9 <javaType> Declaration

A <javaType> declaration provides a way to customize the binding of an XML schema atomic datatype to a Java datatype, referred to as the *target Java datatype*. The target Java datatype can be a Java built-in data type or an application specific Java datatype. This declaration also provides two additional methods: a *parse method* and a *print method*.

The parse method converts a lexical representation of the XML schema datatype into a value of the target Java datatype. The parse method is invoked by a JAXB provider's implementation during unmarshalling.

The print method converts a value of the target Java datatype into its lexical representation of the XML schema datatype. The print method is invoked by a JAXB provider's implementation during marshalling.

An application specific datatype used as a target Java datatype must provide an implementation of both the parse method and print method.

6.9.1 Usage

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

The binding declaration can be used in one of the following:

- a <globalBindings> declaration.
- annotation element of one of the XML schema elements specified in Section 6.9.6, “Customizable Schema Elements.”
- in a <property> declaration. See Section 6.8, “<property> Declaration.” This can be used for customization at the point of reference to a simple type.

When used in a <globalBindings> declaration, <javaType> defines customization values with global scope. When used in an annotation element of one of the schema elements specified in Section 6.9.6, “Customizable Schema Elements.” the customization values have component scope.

6.9.1.1 name

The *`javaType`*, if specified, is the Java datatype to which *`xmlType`* is to be bound. Therefore, *`javaType`* must be a legal Java type name, which may include a package prefix. If the package prefix is not present, then the Java type name must be one of the Java built-in primitive types [JLS - Java Language Specification, Second Edition, Section 4.2, “Primitive Types and Values”]. (For example, “`int`”) or a Java class in the unnamed package.

6.9.1.2 xmlType

The *`xmlType`*, if specified, is the name of the XML Schema datatype to which *`javaType`* is to bound. If specified, *`xmlType`* must be a XML atomic datatype derived from restriction. The use of the *`xmlType`* is further constrained as follows.

The purpose of the *`xmlType`* attribute is to allow the global customization of a XML schema to Java datatype. Hence *`xmlType`* attribute is required when `<javaType>` declaration’s parent is `<globalBindings>`. If absent, it must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.” Otherwise, the *`xmlType`* attribute must not be present since the XML datatype is determined from the XML schema element with which the annotation element containing `<javaType>` declaration or the `<baseType>` (containing the `<javaType>`) is associated. If present, it must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.”

Examples can be found in “Example: javaType Customization: Java Built-in Type” and “Example: javaType Customization: User Specified Parse Method”

6.9.1.3 hasNsContext

The `hasNsContext`, if specified, must be one of `"true"`, `"false"`,`"1"`, or `"0"` and defines the customization value `hasNsContext`. The default value is `false`.

The purpose of `hasNsContext` attribute is to allow a namespace context to be specified as a second parameter to a print or a parse method. The rationale for passing the namespace context and its usage is explained in Section 6.9.1.4, “Namespace Context.”

6.9.1.4 Namespace Context

A namespace context may need to be passed to a parse or a print method for two reasons: QNames and XPath expressions.

A QName consists of a namespace prefix and a local part. The scope of a namespace prefix is the document in which it is declared. However, there are instances, where the name space prefix is of value to applications and needs to be preserved. However, the namespace prefix cannot be interpreted without a namespace context.

Element and attribute values can contain XPath expression which can also contain QNames and/or XPath name functions. A namespace context is also needed for such XPath expressions.

If *`hasNsContext`* is `"true"`, then the JAXB implementation must pass a namespace context to the namespace context parameter of the user specified parse method or the print method. The namespace context passed must implement the `NamespaceContext` interface as specified in the Javadoc for `java.xml.namespace.NamespaceContext`.

Note – The `javax.xml.namespace` package contains `NamespaceContext` interface and `QName` class. The `javax.xml.namespace.NamespaceContext` interface is specified by JAXB technology. Since a `NamespaceContext` represents a mapping between a XML namespace URI and XML namespace prefixes, the interface is of use to other XML enabling Java technologies. Hence the version of `javax.xml.namespace.NamespaceContext` interface specified here is an interim version until such time a common `NamespaceContext` interface is defined in a JSR common to XML enabling technologies.

The `javax.xml.namespace.QName` class is specified by JAX-RPC V1.0 specification.

6.9.1.5 `parseMethod`

The parse method if specified, must be applied during unmarshalling in order to convert a string from the input document into a value of the target Java datatype. The parse method must be invoked as follows:

- The parse method may be specified as `new` provided *`javaType`* is not a Java primitive type such as (`"int"`). If *`javaType`* is a Java primitive type, then this must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.” Otherwise, the binding compiler must assume that the target type is a class that defines a constructor as follows:

- ❍ `String` as the first parameter of the constructor.
- ❍ `java.xml.namespace.NamespaceContext` as a second parameter, when *`hasNsContext`* is `"true"`.

To apply the conversion to a string it must generate code that invokes this constructor, passing it the input string and namespace context (if `hasNsContext` is `"true"`) as specified in Section 6.9.1.3, “hasNsContext.”

- The parse method may be specified in the form *ClassName.methodName,* where the *ClassName* is a fully qualified class name that includes the package name. A compiler must assume that the class *ClassName* exists and that it defines a static method named *methodName* that takes:

  - ❍ `String` as the first argument.
  - ❍ `java.xml.namespace.NamespaceContext` as a second parameter, when `hasNsContext` is `"true"`.

  To apply the conversion to a string it must generate code that invokes this method, passing it the input string and namespace context (if `hasNsContext` is `"true"`) as specified in Section 6.9.1.3, “hasNsContext.”

- The parse method may be specified in the form *methodName* provided *`javaType`* is not a Java primitive type (such as `"int"`). If *`javaType`* is Java primitive type, then this must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.” Otherwise, the binding compiler must assume that *methodName* is a method in the class *`javaType.`* The binding compiler must therefore prefix the *`javaType`* to the *methodName* and process *`javaType.`methodName* as specified in above.

The string passed to parse method can be any lexical representation for `xmlType` as specified in [XSD PART2].

6.9.1.6 `printMethod`

The print method if specified, must be applied during marshalling in order to convert a value of the target type into a lexical representation:

- The print method is specified in the form *methodName* provided `javaType` is not a Java primitive type (such as `"int"`). If `javaType` is Java primitive type, then this must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.” Otherwise, the compiler must assume that the target type is a class or an interface that defines a zero-argument instance method named *methodName* that returns a `String`. To apply the conversion it must generate code to invoke this method upon an instance of the target Java datatype.
- If the print method is specified in the form *ClassName.methodName* then the compiler must assume that the class *ClassName* exists and that it defines a static method named *methodName* that returns a string that takes the following:
    - ❍ the first parameter is the target Java datatype.
    - ❍ `java.xml.namespace.NamespaceContext` as a second parameter, when `hasNsContext` is `"true"`.

  To apply the conversion to a string it must generate code that invokes this method, passing it a value of the target Java datatype and namespace context (if `hasNsContext` is `"true"`) as specified in Section 6.9.1.3, “hasNsContext.”

The lexical representation to which the value of the target type is converted can be any lexical representation for `xmlType` as specified in [XSD PART2].

6.9.2 `DatatypeConverter`

Writing customized parse and print methods can be difficult for a Java programmer. This requires a programmer to understand the lexical representations of XML schema datatypes. To make it easier, an interface, `DatatypeConverterInterface,` and a class `DatatypeConverter` are defined to expose the parse and print methods of a JAXB implementation. These can be invoked by user defined parse and print methods. This shifts the burden of dealing with lexical spaces back to the JAXB implementation.

The `DatatypeConverterInterface` defines parse and print methods for XML schema datatypes. There is one parse and print method for each of XML schema datatype specified in Table 5-1, “Java Mapping for XML Schema Built-in Types,” on page 58. The interface is fully specified by the Javadoc specified in `java.xml.bind.DatatypeConverterInterface`.

The `DatatypeConverter` class defines a static parse and print method corresponding to each parse and print method respectively in the `DatatypeConverterInterface` interface. The property `javax.xml.bind.DatatypeConverter` can be used to select the name of a class that provides an implementation of the parse and print methods. The name specified in the property must be a fully qualified class name and must implement the interface `DatatypeConverterInterface`. The class is fully specified by the Javadoc specified in `java.xml.bind.DatatypeConverter`.

6.9.2.1 Usage

The following example demonstrates the use of the `DatatypeConverter` class for writing a customized parse and print method.

Example: javaType Customization: User Specified Parse Method

This example shows the binding of XML schema type `"xs:date"` is bound to a Java datatype `long` using user specified print and parse methods.

```
<jaxb:globalBindings>
    <jaxb:javaType name="long" xmlType="xs:date"
                parseMethod="pkg.MyDatatypeConverter.myParseDate"
                printMethod="pkg.MyDatatypeConverter.myPrintDate"/>
    </jaxb:javaType>
</jaxb:globalBindings>

package pkg;
import javax.xml.bind.DatatypeConverter;
public class MyDatatypeConverter {
    public static long myParseDate(String s) {
        java.util.Calendar d = DatatypeConverter.parse(s);
        long result= cvtCalendarToLong(d) ; // user defined method
        return result;
    }
    public static String myPrintDate(long l) {
        java.util.Calendar d = cvtLongToCalendar(l); //user defined
        return DatatypeConverter.print(d);
        }
    }
```

The implementation of the print methods (*parseDate* and *printDate*) are provided by the user.

The customization is applied during the processing of XML instance document. During unmarshalling, the JAXB implementation invokes *myParseDate*. If *myParseDate* method throws a *ParseException*, then the JAXB implementation code catches the exception, and generate a *parseConversionEvent*.

6.9.2.2 Lexical And Value Space

[XSD PART 2] specifies both a value space and a lexical space for an schema datatypes. There can be more than one lexical representation for a given value.

Examples of multiple lexical representations for a single value are:

- For boolean, the value `true` has two lexical representations `"true"` and `"1"`.
- For integer, the value `1` has two lexical representations `"1.0"` and `"1"`.

XSD PART 2 also specifies a canonical representation for all XML schema atomic datatypes.

The requirements on the parse and print methods are as follows:

- A JAXB implementation of a parse method in `DatatypeConverterInterface` must be capable of a processing all lexical representations for a value as specified by [XSD PART 2]. This ensures that an instance document containing a value in any lexical representation specified by [XSD PART 2] can be marshalled.
- A JAXB implementation of a print method in `DatatypeConverterInterface` must convert a value into any lexical representation of the XML schema datatype to which the parse method applies, as specified by [XSD PART 2] and which is valid with respect to the application's schema.

Design Note – The print methods that are exposed may not be portable. The only requirement on a print method is that it must output a lexical representation that is valid with respect to the schema. So two vendors can choose to output different lexical representations. However, there is value in exposing them despite being non portable. Without the print method, a user would have to be knowledgeable about how to output a lexical representation for a given schema datatype, which is not desirable.

6.9.3 Built-in Conversions

As a convenience to the user, this section specifies some built-in conversions. A built-in conversion is one where the parse and the print method may be omitted by a user. The built-in conversions leverage the narrowing and widening conversions defined in [JLS - Java Language Specification, Second Edition], Section 5.1.2, "Widening Primitive Conversion" and Section 5.1.3, "Narrowing Primitive Conversions." For example:

```
<xs:simpleType name="foo" type="xs:long">
   <xs:annotation><xs:appinfo>
      <jjaxb:avaType name="int"/>
   </xs:appinfo></xs:annotation>
</xs:simpleType>
```

If the parse method is omitted, then a JAXB implementation must perform the following steps:

a. if *`javaType`* is not one of the primitive types or its corresponding wrapper class as shown in Table 6-2, "Built-In Conversions," on page 159, then it must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations." Skip steps b through d.

b. bind *`xmlType`* to its default Java datatype using the parse method for the *`xmlType`* defined in `DatatypeConverter`. If *`javaType`* is the same as the default Java datatype or its wrapper class, then skip steps b and c.

c. If default Java datatype in step a. is not found in Column 1, "Default Javatype" of Table 6-2, "Built-In Conversions," on page 159, then this must result in an invalid binding customization as specified in Section 6.1.5, "Invalid Customizations. Skip step d.

d. Convert the default Java datatype from step a. to value of type `javaType` using a method in the Java package wrapper class for `javaType` as shown in Table 6-2, "Built-In Conversions," on page 159.

The following is split into two tables for formatting purposes but is logically a single table.

Table 6-2 Built-In Conversions

Default JavaType	byte	short	int	long
byte	N/A	Byte. shortValue()	Byte. intValue()	Byte. longValue()
short	Short. byteValue()	N/A	Short. intValue()	Short. longValue()
int	Integer. byteValue()	Integer. shortValue()	N/A	Integer. longValue()
long	Long. byteValue()	Long. shortValue()	Long. intValue()	N/A
double	Double. byteValue()	Double. shortValue()	Double. intValue()	Double. longValue()
float	Float. byteValue()	Float. shortValue()	Float. intValue()	Float. doubleValue()

Default JavaType	double	float
byte	Byte. doubleValue()	Byte. floatValue()
short	Short. doubleValue()	Short. floatValue()
int	Integer. doubleValue()	Integer. floatValue()

long	Long. doubleValue()	Long. floatValue()
double	N/A	Double. floatValue()
float	Float. doubleValue()	N/A

Example: javaType Customization: Java Built-in Type

This example illustrates how to bind a XML schema type to a Java type different from the default one.

XML Schema fragment:

```
<xs:element name="partNumber" type="xs:int"/>
```

Customization:

```
<jaxb:globalBindings>
   ....
   <jaxb:javaTypename="long"
            xmlType="xs:int"/>
</jaxb:globalBindings>
```

Since a Java built-in is specified, a parse or a print method need not be specified. A JAXB implementation uses the parse and print methods defined in `DatatypeConverter` class for converting between lexical representations and values. A JAXB implementation unmarshalls an input value using the following methods:

```
long i = DataTypeConverter.parseLong(string);
int j = (new java.lang.Long(i)).intValue();
```

6.9.4 Events

The parse method *`parseMethod`* may fail, since it is only defined on those strings that are valid representations of target Java datatype values and it can be applied to arbitrary strings. A parse method must indicate failure by throwing an exception of whatever type is appropriate, though it should never throw a `TypeConstraintException`. A JAXB implementation must ensure that an exception thrown by a parse method is caught and a `parseConversionEvent` event is generated.

The print method *`printMethod`* usually does not fail. If it does, then the JAXB implementation must ensure that the exception thrown by a print method is caught and a `printConversionEvent` is generated.

6.9.5 Customization Overrides

The `<javaType>` overrides the default binding of *`xmlType`* to the Java datatype specified in Table 5-1, “Java Mapping for XML Schema Built-in Types,” on page 58.

6.9.6 Customizable Schema Elements

6.9.6.1 Simple Type Definition

A `<javaType>` binding declaration is allowed in the annotation element of the of a simple type definition. The *`javaType`* overrides the default binding of *`xmlType`* to the Java datatype specified in Table 5-1, “Java Mapping for XML Schema Built-in Types,” on page 58. The customization values defined have definition scope and thus covers all references to this simple type definition.

6.9.6.2 `GlobalBindings`

A `<javaType>` binding declaration is allowed as part of `<globalBindings>.` The *`javaType`* overrides the default binding of *`xmlType`* to the Java datatype specified in Table 5-1, “Java Mapping for XML Schema Built-in Types,” on page 58. The customization values defined have global scope.

6.9.6.3 <baseType> declaration

A `<javaType>` binding declaration is allowed as part of `<baseType>` in the `<property>` binding declaration. The *`javaType`* overrides the default binding of *`xmlType`* to the Java datatype specified in Table 5-1, “Java Mapping for XML Schema Built-in Types,” on page 58. Additional semantics are specified in Section 6.8.1.1, “baseType” also apply.

6.10 `<typesafeEnum>` Declaration

This binding declaration allows the customization of a binding of an XML schema element to its Java representation as a typesafe enumeration class [BLOCH]. Only simple type definitions with enumeration facets can be customized using this binding declaration.

6.10.1 Usage

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

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

There are two binding declarations *<typesafeEnumClass>* and *<typesafeEnumMember>*. The two binding declarations allow the enumeration members of an enumeration class and enumeration class itself to be customized independently.

The <*typesafeEnumClass*> declaration defines the following customization values:

- *name* defines a customization value *enumClassName*, if specified. *enumClassName* must be a legal Java Identifier; it must not have a package prefix.

  For an anonymous simple type, the `name` attribute must be present. If absent, it must result in an invalid customization as specified in Section 6.1.5, “Invalid Customizations.”

- *<javadoc>* element, if specified customizes the Javadoc for the enumeration class. *<javadoc>* defines the customization value *enumClassjavadoc* if specified as described in Section 6.11, “<javadoc> Declaration.”

- Zero or more *<typesafeEnumMember>* declarations. The customization values are as defined as specified by the *<typesafeEnumMember>* declaration.

The `<typesafeEnumMember>` declaration defines the following customization values:

- *`name`* must always be specified and defines a customization value *`enumMemberName`*. *`enumMemberName`* must be a legal Java identifier.
- *`value`* defines a customization value *`enumMemberValue`*, if specified. *`enumMemberValue`* must be the enumeration value specified in the source schema. The usage of *`value`* is further constrained as specified in Section 6.10.2, "value Attribute."
- *`<javadoc>`* if specified, customizes the Javadoc for the enumeration constant. *`<javadoc>`* defines a customization value *`enumMemberjavadoc`* if specified as described in Section 6.11, "<javadoc> Declaration."

For inline annotation, the `<typesafeEnumClass>` 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.

6.10.2 `value` Attribute

The purpose of the *`value`* attribute is to support customization of an enumeration value using an external binding syntax. When the `<typesafeEnumMember>` is used in an inline annotation, the enumeration value being customized can be identified by the annotation element with which it is associated. However, when an external binding declaration is used, while possible, it is not desirable to use XPath to identify an enumeration value.

So when customizing using external binding syntax, the `value` attribute must be provided. This serves as a key to identify the enumeration value to which the `<typesafeEnumMember>` applies. It's use is therefore further constrained as follows:

- When `<typesafeEnumMember>` is specified in the annotation element of the enumeration member or when XPath refers directly to a single enumeration facet, then the value attribute must be absent. If present, it must result in must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations."

- When `<typesafeEnumMember>` is scoped to the `typesafeEnumClass` declaration, the value attribute must be present. If absent, it must result in must result in an invalid customization as specified in Section 6.1.5, "Invalid Customizations." The *enumMemberValue* must be used to identify the enumeration member to which the `<typesafeEnumMember>` applies.

An example of external binding syntax can be found in "Example 2: typesafeEnum Customization: External Binding Declaration."

6.10.3 Inline Annotations

There are two ways to customize an enumeration class:

- split inline annotation
- combined inline annotation

In split inline annotation, the enumeration value and the enumeration class are customized separately i.e. the `<typesafeEnumMember>` is used independently not as a child element of `<typesafeEnumClass>.` An example of this is shown in "Example 1: typesafeEnum Customization: Split Inline Annotation."

In combined inline annotation, the enumeration value and the enumeration class are customized together i.e. the `<typesafeEnumMember>` is used as a child element of `<typesafeEnumClass>.`This is similar to the customization used in external binding declaration. In this case the `value` attribute must be present in the `<typesafeEnumMember>` for reasons noted in Section 6.10.2, "value Attribute." An example of this customization is shown in "Example 3: typesafeEnum Customization: Combined Inline Annotation."

6.10.4 Customization Overrides

When binding a schema element's Java representation to a typesafe enumeration class, the following customization values override the defaults specified in Chapter 5. It is specified in a common section here and referenced from Section 6.8.3, "Customizable Schema Elements."

- **name:** If *enumClassName* is defined, then the name obtained by mapping *enumClassName* as specified in Section 6.5.2, "Customized Name Mapping."

- **package name:** The name obtained by inheriting *packgeName* from a scope that covers this schema element and mapping *packageName* as specified in Section 6.5.2, “Customized Name Mapping.”
- **enumclass javadoc:** *enumClassJavaDoc* if defined, customizes the `class/interface section (` Section 6.11.1, “Javadoc Sections”) for the enumeration class, as specified in Section 6.11.3, “Javadoc Customization.”
- **enum constant set:** Each member of the set is computed as follows:
    - **name:** If *enumMemberName* is defined, the name obtained by mapping *enumMemberName* as specified in Section 6.5.2, “Customized Name Mapping.”
    - **javadoc:** *enumMemberJavaDoc* if defined, customizes the `field section (` Section 6.11.1, “Javadoc Sections”) for the enumeration class, as specified in Section 6.11.3, “Javadoc Customization.”
- **enumvalue constant set:** Each member of the set is computed as follows:
    - **name:** If *enumMemberValueName* is defined, the name obtained by mapping *enumMemberValueName* as specified in Section 6.11.3, “Javadoc Customization” and prefixing the obtained name with an underscore (‘_’).

6.10.5 Customizable Schema Elements

Any XML Schema simple type which has an enumeration facet can be customized.

Example 1: typesafeEnum Customization: Split Inline Annotation

XML Schema fragment:

```
<xs:simpleType name="USState">
    <xs:annotation><xs:appinfo>
        <jaxb:typesafeEnumClass name="USStateAbbr"/>
    </xs:appinfo></xs:annotation>
    <xs:restrictionbase="xs:NCName">
        <xs:enumeration value="AK">
            <xs:annotation><xs:appinfo>
                <jaxb:typesafeEnumMember name="STATE_AK"/>
            </xs:appinfo></xs:annotation>
        </xs:enumeration>
        <xs:enumeration value="AL">
            <xs:annotation><xs:appinfo>
                <jaxb:typesafeEnumMember name="STATE_AL"/>
            </xs:appinfo></xs:annotation>
        </xs:enumeration>
    </xs:restriction>
</xs:simpleType>
```

Customized derived code:

```
public class USStateAbbr {
    protected USStateAbbr(String value) { ... }
    public static final String _STATE_AL="AL";
    public static final USStateAbbr STATE_AL=
                          new USStateAbbr(_STATE_AL);

    public static final String _STATE_AK="AK";
    public static final USStateAbbr STATE_AK=
                              new USStateAbbr(_STATE_AK);

    public String getValue();
    public static USStateAbbr fromValue(String value) {...}
    public static USStateAbbr fromString(String value){ ... }
    public String toString() { ... }
    public boolean equals(Object "obj) { ... }
    public int hashCode() { ... }
}
```

Example 2: typesafeEnum Customization: External Binding Declaration

The following example shows how to customize the above XML schema fragment using an external binding syntax.

```
<jaxb:typesafeEnumClass name="USStateAbbr">
   <jaxb:typesafeEnumMember name="STATE_AK" value="AK">
   <jaxb:typesafeEnumMember name="STATE_AL" value="AL"/>
</jaxb:typesafeEnumClass>
```

The attribute `value` must be specified for `<typesafeEnumMember>`. This identifies the enumeration member to which `<typesafeEnumMember>` applies.

Example 3: typesafeEnum Customization: Combined Inline Annotation

The following example shows how to customize the above XML schema fragment using inline annotation which does not split the external binding syntax.

```
<xs:simpleType name="USState">
   <xs:annotation><xs:appinfo>
      <jaxb:typesafeEnumClass name="USStateAbbr">
         <jaxb:typesafeEnumMember name="STATE_AK" value="AK"/>
         <jaxb:typesafeEnumMember name="STATE_AL" value="AL"/>
      </jaxb:typesafeEnumClass>
   </xs:appinfo></xs:annotation>
   <xs:restriction base="xs:NCName">
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
   </xs:restriction>
</xs:simpleType>
```

The attribute value must be specified for `typesafeEnumMember`. This identifies the enumeration member to which the binding declaration applies.

6.11 `<javadoc>` Declaration

The `<javadoc>` declaration allows the customization of a javadoc that is generated when an XML schema element is bound to its Java representation.

This binding declaration is not a global XML element. Hence it can only be used as a local element within the content model of another binding declaration. The binding declaration in which it is used determines the *section* of the Javadoc that is customized.

6.11.1 Javadoc Sections

The terminology used for the javadoc sections is derived from “Requirements for Writing Java API Specifications” which can be found online at `//java.sun.com/j2se/javadoc/writingapispecs/index.html.`

The following sections are defined for the purposes for customization:

- package section (corresponds to package specification)
- class/interface section (corresponds to class/interface specification)
- method section (corresponds to method specification)
- field section (corresponds to field specification)

6.11.2 Usage

Note that the text content of a `<javadoc>` element must use `CDATA` or `&lt;` to escape embedded HTML tags.

```
<javadoc>
   Contents in &lt;b>Javadoc&lt;\b> format.
</javadoc>
```

or

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

6.11.3 Javadoc Customization

The Javadoc must be generated from the `<javadoc>` element if specified. The Javadoc section depends upon where `<javadoc>` element is used. JAXB providers may generate additional provider specific Javadoc information (for example, contents of the `<xs:documentation>` element).

6.12 Annotation Restrictions

[XSD PART 1] allows an annotation element to be specified for most elements but is ambiguous in some cases. The ambiguity and the way they are addressed are described here.

The source of ambiguity is related to the specification of an annotation element for a reference to a schema element using the “ref” attribute. This arises in three cases:

- A local attribute references a global attribute declaration (using the “ref” attribute).
- A local element in a particle references a global element declaration using the “ref” attribute.
- A model group in a particle references a model group definition using the “ref” attribute.

For example in the following schema fragment (for brevity, the declaration of the global element “Name” has been omitted).

```
<xs:element name = "Customer">
    <xs:complexType>
        <xs:element ref = "Name"/>
        <xs:element ref = "Address" />
    </xs:complexType>
</xs:element>
```

XML Schema spec is ambiguous on whether an annotation element can be specified at the reference to the “Name” element.

The restrictions on annotation elements has been submitted as an issue to the W3C Schema Working Group along with JAXB technology requirements (which is that annotations should be allowed anywhere). Pending a resolution, the semantics of annotation elements where the XML spec is unclear are assumed as specified as follows.

This specification assumes that an annotation element can be specified in each of the three cases outlined above. Furthermore, an annotation element is assumed to be associated with the abstract schema component as follows:

- The annotation element on an element ref is associated with {Attribute Use}
- The annotation element on a model group ref or an element reference is associated with the {particle}.

CHAPTER 7

COMPATIBILITY

This section describes the conformance requirements for an implementor of this specification. A JAXB implementation must implement these constraints, without exception, to provide a predictable environment for application development and deployment.

This section explicitly lists the high level requirements of this specification. Additional requirements can be found in other sections of this specification and the associated javadoc for package `javax.xml.bind` and its subpackages. If any requirements listed here conflict with requirements listed elsewhere in the specification, the requirements here take precedence and replace the conflicting requirements.

For the purpose of portability, all operating modes of a JAXB implementation must support all the XML Schema-to-Java bindings described in this specification.Specifically, other operating modes must not implement a default binding for XML Schema-to-Java bindings as an alternative to those specified in Section 5, “Binding XML Schema to Java Representations” nor alternative interpretations for the standard customizations described in Section 6, “Customization.”

The default operating mode for a JAXB implementation MUST report an error to users when an XML Schema contains constructs for which the Java Binding has not been specified by this specification as summarized in Appendix E.2.1,“Concepts detected at binding compilation time.”

The default operating mode for a JAXB implementation MUST report an error when extension binding declaration is encountered. All operating modes for a JAXB implementation MUST report an error if an invalid binding customization is detected as defined in Section 6. An extension binding declaration must be introduced in the following cases:

1. to alter a binding customization that is allowed to be associated with a

schema element as specified in Section 6, “Customization.”

2. to associate a binding customization with a schema element where it is disallowed as specified in Section 6, “Customization.”

The default operating mode for a JAXB binding compiler MUST report an error when processing a schema that does not comply with the 2001 W3C Recommendation for XML Schema, [XSD Part 1] and [XSD Part 2].

A JAXB implementation MAY support non-default operating modes that are capable of generating bindings for XML Schema constructs not required by this specification.

A JAXB implementation exposes the JAXP 1.1 or higher APIs.

A JAXB compiler MAY support non-default operating modes for binding schema languages other than XML Schema.

A JAXB implementation MUST be able to generate Java classes that are able to run on at least one J2SE Java Runtime Environment.

CHAPTER 8

REFERENCES

[XSD Part 0] XML Schema Part 0: Primer,
W3C Recommendation 2 May 2001
Available at `http://www.w3.org/TR/xmlschema-0/`
(schema fragments borrowed from this widely used source)

[XSD Part 1] XML Schema Part 1: Structures,
W3C Recommendation 2 May 2001
Available at `http://www.w3.org/TR/xmlschema-1/`

[XSD Part 2] XML Schema Part 2: Datatypes,
W3C Recommendation 2 May 2001
Available at `http://www.w3.org/TR/xmlschema-2/`

[XMl-Infoset] XML Information Set, John Cowan and Richard Tobin, eds., W3C, 16 March 2001. Available at `http://www.w3.org/TR/2001/WD-xml-infoset-20010316/`

[XML 1.0] Extensible Markup Language (XML) 1.0 (Second Edition),
W3C Recommendation 6 October 2000.
Available at `http://www.w3.org/TR/2000/REC-xml-20001006`.

[Namespaces in XML] Namespaces in XML
W3C Recommendation 14 January 1999.
Available at `http://www.w3.org/TR/1999/REC-xml-names-19990114`

[XPath], XML Path Language, James Clark and Steve DeRose, eds., W3C, 16 November 1999. Available at `http://www.w3.org/TR/1999/REC-xpath-19991116`

[XSLT 1.0] XSL Transformations (XSLT), Version 1.0, James Clark, W3C Recommendation 16 November 1999. Available at http://www.w3.org/TR/1999/REC-xslt-19991116.

[BEANS] JavaBeans(TM), Version 1.01, July 24, 1997. Available at `http://java.sun.com/beans`.

[XSD Primer] XML Schema Part 0: Primer,
W3C Recommendation 2 May 2001
Available at `http://www.w3.org/TR/xmlschema-0/`

[BLOCH] Joshua Bloch, Effective Java, Chapter 3, Typesafe Enums
`http://developer.java.sun.com/developer/Books/shiftintojavapage1.html#replaceenum`

[RFC2396] Uniform Resource Identifiers (URI): Generic Syntax, `http://www.ietf.org/rfc/rfc2396.txt.`

[JAX-RPC] Java^a API for XML-based RPC JAX-RPC 1.0, `http://java.sun.com/xml/downloads/jaxrpc.html`.

[JLS] The Java Language Specification, 2nd Edition, Gosling, Joy, Steele.Bracha.
Available at .`http://java.sun.com/docs/books/jls`.

[NIST] NIST XML Schema Test Suite, `http://xw2k.sdct.itl.nist.gov/xml/page4.html.`

APPENDIX **A**

PACKAGE JAVAX.XML.BIND

<Available as a separate document.>

APPENDIX **B**

NORMATIVE BINDING SCHEMA SYNTAX

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schematargetNamespace = "http://java.sun.com/xml/ns/jaxb"
   xmlns:jaxb = "http://java.sun.com/xml/ns/jaxb"
   xmlns:xs = "http://www.w3.org/2001/XMLSchema"
   elementFormDefault = "qualified"
   attributeFormDefault = "unqualified">
   <xs:annotation><xs:documentation>
      Schema for binding schema.  JAXB Version 1.0
   </xs:documentation></xs:annotation>
   <xs:group name = "declaration">
      <xs:annotation>
         <xs:documentation>
            Model group that represents a binding declaration.
            Each new binding declaration added to the jaxb
            namespace that is not restricted to globalBindings
            should be added as a child element to this model group.
         </xs:documentation>
         <xs:documentation>
             Allow for extension binding declarations.
         </xs:documentation>
      </xs:annotation>
      <!-- each new binding declaration, not restricted to
          globalBindings, should be added here -->
      <xs:choice>
         <xs:element ref = "jaxb:globalBindings"/>
         <xs:element ref = "jaxb:schemaBindings"/>
         <xs:element ref = "jaxb:class"/>
         <xs:element ref = "jaxb:property"/>
         <xs:element ref = "jaxb:typesafeEnumClass"/>
         <xs:element ref = "jaxb:javaType"/>
```

```
            <xs:element ref = "jaxb:typesafeEnumMember"/>
            <xs:any namespace = "##other" processContents = "lax"/>
        </xs:choice>
    </xs:group>
    <xs:attribute name = "version" type="xs:token" >
        <xs:annotation><xs:documentation>
            Used to specify the version of the binding schema on the
            schema element for inline annotations or jaxb:bindings
            for external binding.
        </xs:documentation></xs:annotation>
    </xs:attribute>
    <xs:attributeGroup name = "propertyAttributes">
        <xs:annotation>
            <xs:documentation>
                Attributes used for property customization. The
                attribute group can be referenced either from the
                globalBindings declaration or from the property
                declaration.
                The following defaults are defined by the JAXB
                specification in global scope only. Thus they apply
                when the propertyAttributes group is referenced
                from the globalBindings declaration but not when
                referenced from the property declaration.
                    collectionType class that implements
                                  java.util.List. The class is
                                  JAXB implementation dependent.
                    fixedAttributeAsConstantProperty false
                    enableFailFastCheck      false
                    generateIsSetMethod      false
            </xs:documentation>
        </xs:annotation>
        <xs:attribute name = "collectionType"
                         type="jaxb:referenceCollectionType"/>
        <xs:attribute name = "fixedAttributeAsConstantProperty"
                    type = "xs:boolean"/>
        <xs:attribute name = "enableFailFastCheck"
                    type = "xs:boolean"/>
        <xs:attribute name = "generateIsSetMethod"
                    type = "xs:boolean"/>
    </xs:attributeGroup>
    <xs:attributeGroup name = "XMLNameToJavaIdMappingDefaults">
        <xs:annotation>
            <xs:documentation>
                Customize XMlNames to Java id mapping
            </xs:documentation>
        </xs:annotation>
        <xs:attribute name = "underscoreBinding"
                    default = "asWordSeparator"
                    type = "jaxb:underscoreBindingType"/>
        <xs:attribute name = "typesafeEnumMemberName"
                    default = "generateError"
                    type = "jaxb:typesafeEnumMemeberNameType"/>
    </xs:attributeGroup>
    <xs:attributeGroup name = "typesafeEnumClassDefaults">
        <xs:attribute name = "typesafeEnumBase"
                    default = "xs:NCName"
                    type = "jaxb:typesafeEnumBaseType"/>
    </xs:attributeGroup>
    <xs:element name = "globalBindings">
        <xs:annotation>
            <xs:documentation>
                Customization values defined in global scope.
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence minOccurs = "0">
                <xs:element ref = "jaxb:javaType"
                            minOccurs = "0" maxOccurs = "unbounded"/>
                <xs:any namespace = "##other" processContents = "lax">
                    <xs:annotation> <xs:documentation>
                        allows extension binding declarations to be
                        specified.
                    </xs:documentation></xs:annotation>
                </xs:any>
            </xs:sequence>
            <xs:attributeGroup ref = "jaxb:XMLNameToJavaIdMappingDefaults"/>
            <xs:attributeGroup ref = "jaxb:typesafeEnumClassDefaults"/>
            <xs:attributeGroup ref = "jaxb:propertyAttributes"/>
            <xs:attribute name = "enableJavaNamingConventions"
                        default = "true"
                        type = "xs:boolean"/>
            <xs:attribute name = "bindingStyle"
                        default = "elementBinding"
                        type = "jaxb:bindingStyleType"/>
            <xs:attribute name = "choiceContentProperty"
                        default = "false"
                        type = "xs:boolean"/>
        </xs:complexType>
```

```
      </xs:element>
      <xs:element name = "schemaBindings">
         <xs:annotation>
            <xs:documentation>
               Customization values with schema scope
            </xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:all>
               <xs:element name = "package" type = "jaxb:packageType"
                             minOccurs = "0"/>
               <xs:element  name = "nameXmlTransform"
                          type = "jaxb:nameXmlTransformType"
                          minOccurs = "0"/>
            </xs:all>
         </xs:complexType>
      </xs:element>
      <xs:element name = "class">
         <xs:annotation>
            <xs:documentation>Customize interface and implementation
   class.</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:sequence>
               <xs:element name = "javadoc" type = "xs:string"
                       minOccurs = "0"/>
            </xs:sequence>
            <xs:attribute name = "name"
                      type = "jaxb:javaIdentifierType">
               <xs:annotation><xs:documentation>
                  Java class name without package prefix.
               </xs:documentation></xs:annotation>
            </xs:attribute>
            <xs:attribute name = "implClass" type = "jaxb:javaIdentifierType">
               <xs:annotation><xs:documentation>
                  Implementation class name including packageprefix.
               </xs:documentation></xs:annotation>
            </xs:attribute>
         </xs:complexType>
      </xs:element>
      <xs:element name = "property">
         <xs:annotation><xs:documentation>
            Customize property.
         </xs:documentation></xs:annotation>
```

```
          <xs:complexType>
             <xs:all>
                <xs:element name = "javadoc" type = "xs:string"
                       minOccurs = "0"/>
                <xs:element name = "baseType"
                       type = "jaxb:propertyBaseType"
                       minOccurs="0"/>
             </xs:all>
             <xs:attribute name = "name"
                      type = "jaxb:javaIdentifierType"/>
                      <xs:attributeGroup ref =
"jaxb:propertyAttributes"/>
          </xs:complexType>
       </xs:element>
       <xs:element name = "javaType">
          <xs:annotation><xs:documentation>
                Data type conversions; overriding builtins
          </xs:documentation></xs:annotation>
          <xs:complexType>
             <xs:attribute name = "name" use = "required"
                      type = "jaxb:javaIdentifierType">
                <xs:annotation><xs:documentation>
                   name of the java type to which xml type is to be
                   bound.
                </xs:documentation></xs:annotation>
             </xs:attribute>
             <xs:attribute name = "xmlType" type = "QName">
                <xs:annotation><xs:documentation>
                   xml type to which java datatype has to be bound.
                   Must be present when javaType is scoped to
                   globalBindings.
                </xs:documentation></xs:annotation>
             </xs:attribute>
             <xs:attribute name = "parseMethod"
                      type = "jaxb:javaIdentifierType"/>
             <xs:attribute name = "printMethod"
                      type = "jaxb:javaIdentifierType"/>
              <xs:attribute name = "hasNsContext" default = "false"
type = "xs:boolean" >
                 <xs:annotation>
                   <xs:documentation>
If true, the parsMethod and printMethod must reference a method
signtature that has a second parameter of type NamespaceContext.
                </xs:documentation>
```

```
            </xs:annotation>
        </xs:attribute>
        </xs:complexType>
    </xs:element>
    <xs:element name = "typesafeEnumClass">
        <xs:annotation><xs:documentation>
            Bind to a type safe enumeration class.
        </xs:documentation></xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name = "javadoc" type = "xs:string"
                                minOccurs = "0"/>
                <xs:element ref = "jaxb:typesafeEnumMember"
                        minOccurs = "0" maxOccurs = "unbounded"/>
            </xs:sequence>
            <xs:attribute name = "name"
                        type = "jaxb:javaIdentifierType"/>
        </xs:complexType>
    </xs:element>
    <xs:element name = "typesafeEnumMember">
        <xs:annotation><xs:documentation>
            Enumeration member name in a type safe enumeration
            class.
        </xs:documentation></xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name = "javadoc" type = "xs:string"
                                  minOccurs = "0"/>
            </xs:sequence>
            <xs:attribute name = "value" type = "xs:string"/>
            <xs:attribute name = "name" use = "required"
                        type = "jaxb:javaIdentifierType"/>
        </xs:complexType>
    </xs:element>

    <!-- TYPE DEFINITIONS -->
    <xs:complexType name = "propertyBaseType">
        <xs:annotation><xs:documentation>
            Customize the base type of a property.
        </xs:documentation></xs:annotation>
        <xs:all>
            <xs:element ref = "jaxb:javaType" minOccurs = "0"/>
        </xs:all>
    </xs:complexType>
```

```
<xs:simpleType name = "bindingStyleType">
   <xs:annotation><xs:documentation>
      Allows selection of a binding algorithm
   </xs:documentation></xs:annotation>
   <xs:restriction base = "xs:string">
      <xs:enumeration value = "elementBinding"/>
      <xs:enumeration value = "modelGroupBinding"/>
   </xs:restriction>
</xs:simpleType>

<xs:complexType name = "packageType">
   <xs:sequence>
      <xs:element name = "javadoc" type = "xs:string"
              minOccurs = "0"/>
   </xs:sequence>
   <xs:attribute name = "name" type="jaxb:javaIdentifierType"/>
</xs:complexType>
<xs:simpleType name = "underscoreBindingType">
   <xs:annotation><xs:documentation>
      Treate underscore  in XML Name to Java identifier mapping.
   </xs:documentation></xs:annotation>
   <xs:restriction base = "xs:string">
      <xs:enumeration value = "asWordSeparator"/>
      <xs:enumeration value = "asCharInWord"/>
   </xs:restriction>
</xs:simpleType>
<xs:simpleType name = "typesafeEnumBaseType">
   <xs:annotation><xs:documentation>
      XML types or types derived from them which have
      enumeration facet(s) which are be mapped to
      typesafeEnumClass by default.
      The following types cannot be specified in this list:
      "xs:QName", "xs:base64Binary", "xs:hexBinary",
      "xs:date", "xs:time", "xs:dateTime", "xs:duration",
      "xs:gDay", "xs:gMonth", "xs:gYear", "xs:gMonthDay",
      "xs:YearMonth"
   </xs:documentation></xs:annotation>
      <xs:list itemType = "xs:QName"/>
</xs:simpleType>
<xs:simpleType name = "typesafeEnumMemeberNameType">
   <xs:annotation><xs:documentation>
      Used to customize how to handle name collisions.
      i.  generate VALUE_1, VALUE_2... if  generateName.
```

```
          ii.  generate an error  if value is generateError.
             This is JAXB default behavior.
      </xs:documentation></xs:annotation>
      <xs:restriction base = "xs:string">
         <xs:enumeration value = "generateName"/>
         <xs:enumeration value = "generateError"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name = "javaIdentifierType">
      <xs:annotation><xs:documentation>
         Type to indicate Legal Java identifier.
      </xs:documentation></xs:annotation>
      <xs:restriction base = "xs:NCName"/>
   </xs:simpleType>
   <xs:complexType name = "nameXmlTransformRule">
      <xs:annotation><xs:documentation>
         Rule to transform an Xml name into another Xml name
      </xs:documentation></xs:annotation>
      <xs:attribute name = "prefix" type = "xs:string">
         <xs:annotation><xs:documentation>
            prepend the string to QName.
         </xs:documentation></xs:annotation>
      </xs:attribute>
      <xs:attribute name = "suffix" type = "xs:string">
         <xs:annotation><xs:documentation>
            Append the string to QName.
         </xs:documentation></xs:annotation>
      </xs:attribute>
   </xs:complexType>
   <xs:complexType name = "nameXmlTransformType">
      <xs:annotation><xs:documentation>
         Allows transforming an xml name into another xml name. Use
         case UDDI 2.0 schema.
      </xs:documentation></xs:annotation>
      <xs:all>
         <xs:element name = "typeName"
                 type = "jaxb:nameXmlTransformRule"
                    minOccurs = "0">
            <xs:annotation><xs:documentation>
               Mapping rule for type definitions.
            </xs:documentation></xs:annotation>
         </xs:element>
         <xs:element name = "elementName"
                 type = "jaxb:nameXmlTransformRule"
               minOccurs = "0">
           <xs:annotation><xs:documentation>
              Mapping rule for elements
           </xs:documentation></xs:annotation>
       </xs:element>
       <xs:element name = "modelGroupName"
              type = "jaxb:nameXmlTransformRule"
              minOccurs = "0">
           <xs:annotation><xs:documentation>
              Mapping rule  for model group
           </xs:documentation></xs:annotation>
       </xs:element>
       <xs:element name = "anonymousTypeName"
               type = "jaxb:nameXmlTransformRule">
           <xs:annotation><xs:documentation>
              Mapping rule for class names generated for an
              anonymous type.
           </xs:documentation></xs:annotation>
       </xs:element>
    </xs:all>
</xs:complexType>
<xs:attribute name = "extensionBindingPrefixes">
    <xs:annotation><xs:documentation>
       A binding compiler only processes this attribute when it
       occurs on an instance of xs:schema element.  The value of
       this attribute is a whitespace-separated list of namespace
       prefixes.  The namespace bound to each of the prefixes is
       designated as a customization declaration namespace.
    </xs:documentation></xs:annotation>
    <xs:simpleType>
       <xs:list itemType = "xs:normalizedString"/>
    </xs:simpleType>
</xs:attribute>
<xs:element name = "bindings">
    <xs:annotation><xs:documentation>
       Binding declaration(s) for a remote schema.
       If attribute node is set, the binding declaraions
       are associated with part of the remote schema
       designated by schemaLocation attribute. The node
       attribute identifies the node in the remote schema
       to associate the binding declaration(s) with.
    </xs:documentation></xs:annotation>
    <!-- a <bindings> element can contain arbitrary number of
       binding declarations or nested <bindings> elements -->
```

```
        <xs:complexType>
            <xs:sequence>
                <xs:choice minOccurs = "0" maxOccurs = "unbounded">
                    <xs:group ref = "jaxb:declaration"/>
                    <xs:element ref = "jaxb:bindings"/>
                </xs:choice>
            </xs:sequence>
            <xs:attribute name = "schemaLocation" type = "xs:anyURI">
                <xs:annotation><xs:documentation>
                    Location of the remote schema to associate binding
                    declarations with.
                </xs:documentation></xs:annotation>
            </xs:attribute>
            <xs:attribute name = "node" type = "xs:string">
                <xs:annotation><xs:documentation>
                    The value of the string is an XPATH 1.0 compliant
                    string that resolves to a node in a remote schema
                    to associate binding declarations with. The remote
                    schema is specified by the schemaLocation
                    attribute occuring in the current element or in a
                    parent of this element.
                </xs:documentation></xs:annotation>
            </xs:attribute>
            <xs:attribute name = "version" type = "xs:token">
                <xs:annotation><xs:documentation>
                    Used to indicate the version of binding
                    declarations. Only valid on root level bindings
                    element. Either this or "jaxb:version" attribute
                    but not both may be specified.
                </xs:documentation> </xs:annotation>
            </xs:attribute>
            <xs:attribute ref = "jaxb:version">
                <xs:annotation><xs:documentation>
                    Used to indicate the version of binding
                    declarations. Only valid on root level bindings
                    element. Either this attribute or "version"
                    attribute but not both may be specified.
                </xs:documentation> </xs:annotation>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
    <xs:simpleType name="referenceCollectionType">
        <xs:union>
            <xs:simpleType>
```

```
                <xs:restriction base="xs:string">
                    <xs:enumeration value="indexed"/>
                </xs:restriction>
            </xs:simpleType>
            <xs:simpleType>
                <xs:restriction base="jaxb:javaIdentifierType"/>
            </xs:simpleType>
        </xs:union>
    </xs:simpleType>
</xs:schema>
```

APPENDIX C

BINDING XML NAMES TO JAVA IDENTIFIERS

C.1 Overview

This section provides default mappings from:

- XML Name to Java identifier
- Model group to Java identifier
- Namespace URI to Java package name

C.2 The Name to Identifier Mapping Algorithm

Java identifiers typically follow three simple, well-known conventions:

- Class and interface names always begin with an upper-case letter. The remaining characters are either digits, lower-case letters, or upper-case letters. Upper-case letters within a multi-word name serve to identify the start of each non-initial word, or sometimes to stand for acronyms.
- Method names and components of a package name always begin with a lower-case letter, and otherwise are exactly like class and interface names.
- Constant names are entirely in upper case, with each pair of words separated by the underscore character (‘_’, \u005F, LOW LINE).

XML names, however, are much richer than Java identifiers: They may include not only the standard Java identifier characters but also various punctuation and special characters that are not permitted in Java identifiers. Like most Java identifiers, most XML names are in practice composed of more than one natural-language word. Non-initial words within an XML name typically start with an upper-case letter followed by a lower-case letter, as in Java language, or are prefixed by punctuation characters, which is not usual in the Java language and, for most punctuation characters, is in fact illegal.

In order to map an arbitrary XML name into a Java class, method, or constant identifier, the XML name is first broken into a *word list*. For the purpose of constructing word lists from XML names we use the following definitions:

- A *punctuation character* is one of the following:
  - A hyphen ('-', \u002D, HYPHEN-MINUS),
  - A period ('.', \u002E, FULL STOP),
  - A colon (':', \u003A, COLON),
  - An underscore ('_', \u005F, LOW LINE),
  - A dot ('.', \u00B7, MIDDLE DOT),
  - \u0387, GREEK ANO TELEIA,
  - \u06DD, ARABIC END OF AYAH, or
  - \u06DE, ARABIC START OF RUB EL HIZB.

These are all legal characters in XML names.

- A *letter* is a character for which the `Character.isLetter` method returns `true`, *i.e.*, a letter according to the Unicode standard. Every letter is a legal Java identifier character, both initial and non-initial.
- A *digit* is a character for which the `Character.isDigit` method returns `true`, *i.e.*, a digit according to the Unicode Standard. Every digit is a legal non-initial Java identifier character.
- A *mark* is a character that is in none of the previous categories but for which the `Character.isJavaIdentifierPart` method returns `true`. This category includes numeric letters, combining marks, non-spacing marks, and ignorable control characters.

Every XML name character falls into one of the above categories. We further divide letters into three subcategories:

- An *upper-case letter* is a letter for which the `Character.isUpperCase` method returns `true`,

- A *lowercase letter* is a letter for which the `Character.isLowerCase` method returns `true`, and
- All other letters are *uncased*.

An XML name is split into a word list by removing any leading and trailing punctuation characters and then searching for *word breaks*. A word break is defined by three regular expressions: A prefix, a separator, and a suffix. The prefix matches part of the word that precedes the break, the separator is not part of any word, and the suffix matches part of the word that follows the break. The word breaks are defined as:

Table C-1 XML Word Breaks

Prefix	Separator	Suffix	Example
`[^punct]`	`punct+`	`[^punct]`	`foo|--|bar`
`digit`		`[^digit]`	`foo22|bar`
`[^digit]`		`digit`	`foo|22`
`lower`		`[^lower]`	`foo|Bar`
`upper`		`upper lower`	`FOO|Bar`
`letter`		`[^letter]`	`Foo|\u2160`
`[^letter]`		`letter`	`\u2160|Foo`
`uncased`		`[^uncased]`	
`[^uncased]`		`uncased`	

(The character `\u2160` is ROMAN NUMERAL ONE, a numeric letter.)

After splitting, if a word begins with a lower-case character then its first character is converted to upper case. The final result is a word list in which each word is either

- A string of upper- and lower-case letters, the first character of which is upper case,
- A string of digits, or
- A string of uncased letters and marks.

Given an XML name in word-list form, each of the three types of Java identifiers is constructed as follows:

- A class or interface identifier is constructed by concatenating the words in the list,

- A method identifier is constructed by concatenating the words in the list. A prefix verb (`get`, `set`, *etc.*) is prepended to the result.
- A constant identifier is constructed by converting each word in the list to upper case; the words are then concatenated, separated by underscores.

This algorithm will not change an XML name that is already a legal and conventional Java class, method, or constant identifier, except perhaps to add an initial verb in the case of a property access method.

Example

Table C-2 XML Names and Java Class, Method, and Constant Names

XML Name	Class Name	Method Name	Constant Name
mixedCaseName	MixedCaseName	getMixedCaseName	MIXED_CASE_NAME
Answer42	Answer42	getAnswer42	ANSWER_42
name-with-dashes	NameWithDashes	getNameWithDashes	NAME_WITH_DASHES
other_punct-chars	OtherPunctChars	getOtherPunctChars	OTHER_PUNCT_CHARS

C.2.1 Collisions and conflicts

It is possible that the name-mapping algorithm will map two distinct XML names to the same word list. This will result in a *collision* if, and only if, the same Java identifier is constructed from the word list and is used to name two distinct generated classes or two distinct methods or constants in the same generated class. Collisions are not permitted by the binding compiler and are reported as errors; they may be repaired by revising XML name within the source schema or by specifying a customized binding that maps one of the two XML names to an alternative Java identifier.

A class name must not conflict with the generated JAXB class, `ObjectFactory`, section 4.2 on page 36, that occurs in each schema-derived Java package. Method names are forbidden to conflict with Java keywords or literals, with methods declared in `java.lang.Object`, or with methods declared in the binding-framework classes. Such conflicts are reported as errors and may be repaired by revising the appropriate schema or by specifying an appropriate customized binding that resolves the name collision.

Design Note – The likelihood of collisions, and the difficulty of working around them when they occur, depends upon the source schema, the schema language in which it is written, and the binding declarations. In general, however, we expect that the combination of the identifier-construction rules given above, together with good schema-design practices, will make collisions relatively uncommon.

The capitalization conventions embodied in the identifier-construction rules will tend to reduce collisions as long as names with shared mappings are used in schema constructs that map to distinct sorts of Java constructs. An attribute named `foo` is unlikely to collide with an element type named `foo` because the first maps to a set of property access methods (`getFoo`, `setFoo`, *etc.*) while the second maps to a class name (`Foo`).

Good schema-design practices also make collisions less likely. When writing a schema it is inadvisable to use, in identical roles, names that are distinguished only by punctuation or case. Suppose a schema declares two attributes of a single element type, one named `Foo` and the other named `foo`. Their generated access methods, namely `getFoo` and `setFoo`, will collide. This situation would best be handled by revising the source schema, which would not only eliminate the collision but also improve the readability of the source schema and documents that use it.

C.3 Deriving a legal Java identifier from `xs:string`

The XML Name to Java identifier algorithm needs to be extended to accommodate generating Java identifiers from `xs:string` due to the binding customization `typesafeEnumBase` enabling enumeration values that are not `xs:NCName` to need to be mapped to a Java identifier by default.

Leading and trailing white space for the value of the `xs:string` is dropped. All characters that return false for `isJavaIdentifierPart()` are dropped.

C.4 Deriving an identifier for a model group

XML Schema has the concept of a group of element declarations. Occasionally, it is convenient to bind the grouping as a Java content property or a Java content interface. When a semantically meaningful name for the group is not provided within the source schema or via a binding declaration customization, it is necessary to generate a Java identifier from the grouping. Below is an algorithm to generate such an identifier.

A name is computed for an unnamed model group by concatenating together the first 3 element declarations and/or wildcards that occur within the model group. Each XML *{name}* is mapped to a Java identifier for a method using the XML Name to Java Identifier Mapping algorithm. Since wildcard does not have a {name} property, it is represented as the Java identifier “`Any`”. The Java identifiers are concatenated together with the separator “`And`” for sequence and all compositor and “`Or`” for choice compositors. For example, a sequence of element `foo` and element `bar` would map to “`FooAndBar`” and a choice of element `foo` and element `bar` maps to “`FooOrBar`.” Lastly, a sequence of wildcard and element `bar` would map to the Java identifier “`AnyAndBar`”.

Example:

Given XML Schema fragment:

```
<xs:choice>
   <xs:sequence>
      <xs:element name="A"/>
      <xs:any processContents="strict"/>
   </xs:sequence>
   <xs:element name="C"/>
</xs:choice>
```

The generated Java identifier would be `AAndAnyOrC`.

C.5 Generating a Java package name

This section describes how to generate a package name to hold the derived Java representation. The motivation for specifying a default means to generate a Java package name is to increase the chances that a schema can be processed by a binding compiler without requiring the user to specify customizations.

If a schema has a target namespace, the next subsection describes how to map the URI into a Java package name. If the schema has no target namespace, there is a section that describes an algorithm to generate a Java package name from the schema filename.

C.5.1 Mapping from a Namespace URI

An XML namespace is represented by a URI. Since XML Namespace will be mapped to a Java package, it is necessary to specify a default mapping from a URI to a Java package name. The URI format is described in [RFC2396].

The following steps describe how to map a URI to a Java package name. The example URI, `http://www.acme.com/go/espeak.xsd` , is used to illustrate each step.

1. Remove the scheme and `":"` part from the beginning of the URI, if present.
   Since there is no formal syntax to identify the optional URI scheme, restrict the schemes to be removed to case insensitive checks for schemes “`http`” and “`urn`”.

   ```
   //www.acme.com/go/espeak.xsd
   ```
2. Remove the trailing file type, one of `.??` or `.???` or `.html`.

   ```
   //www.acme.com/go/espeak
   ```
3. Parse the remaining string into a list of strings using `’/’` and `‘:’` as separators. Treat consecutive separators as a single separator.

   ```
   {"www.acme.com", "go", "espeak" }
   ```
4. For each string in the list produced by previous step, unescape each escape sequence octet.

   ```
   {"www.acme.com", "go", "espeak" }
   ```
5. Apply algorithm described in Section 7.7 “Unique Package Names” in [JLS] to derive a unique package name from the potential internet domain name contained within the first component. The internet domain name is reversed, component by component. Note that a leading “`www.`” is not considered part of an internet domain name and must be dropped.

If the first component does not contain either one of the top-level domain names, for example, com, gov, net, org, edu, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981, this step must be skipped.

```
{“com”, “acme”, “go”, “espeak”}
```

6. For each string in the list, convert each string to be all lower case.

```
{"com”, “acme”, "go", "espeak" }
```

7. For each string remaining, the following conventions are adopted from [JLS] Section 7.7, “Unique Package Names.”
   a. If the sting component contains a hyphen, or any other special character not allowed in an identifier, convert it into an underscore.
   b. If any of the resulting package name components are keywords then append underscore to them.
   c. If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.

```
{"com”, “acme”, "go", "espeak" }
```

8. Concatenate the resultant list of strings using `’.’` as a separating character to produce a package name.

```
Final package name: "com.acme.go.espeak".
```

Section C.2.1, “Collisions and conflicts,” on page 192, specifies what to do when the above algorithm results in an invalid Java package name.

C.6 Conforming Java Identifier Algorithm

This section describes how to convert a legal Java identifier which may not conform to Java naming conventions to a Java identifier that conforms to the standard naming conventions. Since a legal Java identifier is also a XML name, this algorithm is the same as Section C.2, “The Name to Identifier Mapping Algorithm” with the following exception: constant names must not be mapped to a Java constant that conforms to the Java naming convention for a constant. The reason is that this algorithm is used to map legal Java identifiers specified in customization referred to as a customization name. As specified in the Chapter 6, “Customization”, customization names that are not mapped to constants that conform to the Java naming conventions.

APPENDIX D

EXTERNAL BINDING DECLARATION

D.1 Example

Example: Consider the following schema and external binding file:

Source Schema: `A.xsd:`

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:ens="http://example.com/ns"
          targetNamespace="http://example.com/ns">
   <xs:complexType name="aType">
      <xs:sequence>
         <xs:element name="foo" type="xs:int"/>
      </xs:sequence>
      <xs:attribute name="bar" type="xs:int"/>
   </xs:complexType>
   <xs:element name="root" type="ens:aType"/>
</xs:schema>
```

External binding declarations file:

```
<jaxb:bindingsxmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             version="1.0">
   <jaxb:bindings schemaLocation=”A.xsd”>
      <jaxb:bindings node="//xs:complexType[@name=’aType’]”>
         <jaxb:class name="customNameType"/>
         <jaxb:bindings node=”.//xs:element[@name=’foo’]”>
            <jaxb:property name="customFoo"/>
         </jaxb:bindings>
```

```
            <jaxb:bindings node=”./xs:attribute[@name=’bar’]”>
               <jaxb:property name="customBar"/>
            </jaxb:bindings>
         </jaxb:bindings>
      </jaxb:bindings>
   </jaxb:bindings>
```

Conceptually, the combination of the source schema and external binding file above are the equivalent of the following inline annotated schema.

```
   <xs:schema        xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:ens="http://example.com/ns"
                     targetNamespace="http://example.com/ns"
                     xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
                     jaxb:version="1.0">
      <xs:complexType name="aType">
         <xs:annotation>
            <xs:appinfo>
               <jaxb:class name="customNameType"/>
            </xs:appinfo>
         </xs:annotation>
         <xs:sequence>
            <xs:element name="foo" type="xs:int">
               <xs:annotation>
                  <xs:appinfo>
                     <jaxb:property name="customFoo"/>
                  </xs:appinfo>
               </xs:annotation>
            </xs:element>
         </xs:sequence>
         <xs:attribute name="bar" type="xs:int">
            <xs:annotation>
               <xs:appinfo>
                  <jaxb:property name="customBar"/>
               </xs:appinfo>
            </xs:annotation>
         </xs:attribute>
      </xs:complexType>
      <xs:element name="root" type="ens:aType"/>
   </xs:schema>
```

D.2 Transformation

The intent of this section is to describe the transformation of external binding declarations and their target schemas into a set of schemas annotated with JAXB binding declarations. ready for processing by a JAXB compliant binding compiler.

This transformation must be understood to work on XML data model level. Thus, this transformation is applicable even for those schemas which contain semantic errors.

The transformation is applied as follows:

1. Gather all the top-most `<jaxb:bindings>` elements from all the schema documents and all the external binding files that participate in this process. *Top-most* `<jaxb:bindings>` are those `<jaxb:bindings>` elements that are either a root element in a document or whose parent is an `<xs:appinfo>` element. We will refer to these trees as “external binding forest.”
2. Collect all the namespaces used in the elements inside the external binding forest, except the taxi namespace, `"http://java.sun.com/xml/ns/jaxb”,`and the no namespace. Allocate an unique prefix for each of them and declare the namespace binding at all the root `<xs:schema>` elements of each schema documents.
   Then add a `jaxb:extensionBindingPrefix` attribute to each `<xs:schema>` element with all those allocated prefixes. If an `<xs:schema>` element already carries this attribute, prefixes are just appended to the existing attributes.

   Note: The net effect is that all “foreign” namespaces used in the external binding forest will be automatically be considered as extension customization declaration namespaces.
3. For each `<jaxb:bindings>` element, we determine the “target element” that the binding declaration should be associated with. This process proceeds in a top-down fashion as follows:
   a. Let `p` be the target element of the parent `<jaxb:bindings>`. If it is the top most `<jaxb:bindings>`, then let `p` be the <`jaxb:bindings>` element itself.

b. Identify the "target element" using `<jaxb:bindings>` attributes. (i) If the `<jaxb:bindings>` has a `@schemaLocation`, the value of the attribute should be taken as an URI and be absolutized with the base URI of the `<jaxb:bindings>` element. Then the target element will be the root node of the schema document identified by the absolutized URI. If there's no such schema document in the current input, it is an error. Note: the root node of the schema document is not the document element.

   (ii) If the `<jaxb:bindings>` has `@node`, the value of the attribute should be evaluated as an XPath 1.0 expression. The context node in this evaluation should be `p` as we computed in the previous step. It is an error if this evaluation results in something other than a node set that contains exactly one element. Then the target element will be this element.

   (iii) if the `<jaxb:bindings>` has neither `@schemaLocation` nor `@node`, then the target element will be `p` as we computed in the previous step. Note: `<jaxb:bindings>` elements can't have both `@schemaLocation` and `@node` at the same time.

We define the target element of a binding declaration to be the target element of its parent `<jaxb:bindings>` element. It is an error if a target element of a binding declaration doesn't belong to the *"http://wwww.w3.org/2001/XMLSchema"* namespace.

4. Next, for each target element of binding declarations, if it doesn't have any `<xs:annotation> <xs:appinfo>` in its children, one will be created and added as the first child of the target.

   After that, we move each binding declaration under the target node of its parent `<jaxb:bindings>`. Consider the first `<xs:appinfo>` child of the target element. The binding declaration element will be moved under this `<xs:appinfo>` element.

APPENDIX E

XML SCHEMA

E.1 Abstract Schema Model

The following summarization abstract schema component model has been extracted from [XSD Part 1] as a convenience for those not familiar with XML Schema component model in understanding the binding of XML Schema components to Java representation. One must refer to [XSD Part 1] for the complete normative description for these components.

E.1.1 Simple Type Definition Schema Component

Table E-1 Simple Type Definition Schema Components

Component	**Description**
`{name}`	Optional. An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{base type definition}`	A simple type definition
`{facets}`	A set of constraining facets.
`{fundamental facets}`	A set of fundamental facets.
`{final}`	A subset of {extension, list, restriction, union}.

Table E-1 Simple Type Definition Schema Components *(Continued)*

Component	Description	
`{variety}`	One of {atomic, list, union}. Depending on the value of {variety}, further properties are defined as follows:	
	atomic `{primitive type definition}`	A built-in primitive simple type definition.
	list `{item type definition}`	A simple type definition.
	union `{member type definitions}`	A non-empty sequence of simple type definitions.
`{annotation}`	Optional. An annotation.	

E.1.2 Enumeration Facet Schema Component

Table E-2 Enumeration Facet Schema Components

Component	Description
`{value}`	The actual value of the value. (Must be in value space of base type definition.)
`{annotation}`	Optional annotation.

E.1.3 Complex Type Definition Schema Component

Table E-3 Complex Type Definition Schema Components

Component	Description
`{name}`	Optional. An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{base type definition}`	Either a simple type definition or a complex type definition.
`{derivation method}`	Either extension or *restriction.*
`{final}`	A subset of {extension, restriction}.
`{abstract}`	A boolean
`{attribute uses}`	A set of attribute uses.
`{attribute wildcard}`	Optional. A wildcard.

Table E-3 Complex Type Definition Schema Components *(Continued)*

Component	Description
`{content type}`	One of *empty*, a *simple type definition*, or a pair consisting of a ·content model· and one of *mixed*, *element-only*.
`{prohibited substitutions}`	A subset of {extension, restriction}.
`{annotations}`	A set of annotations.

E.1.4 Element Declaration Schema Component

Table E-4 Element Declaration Schema Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name
`{type definition}`	Either a simple type definition or a complex type definition.
`{scope}`	Optional. Either global or a complex type definition.
`{value constraint}`	Optional. A pair consisting of a value and one of default, fixed.
`{nillable}`	A boolean.
`{identity-constraint definitions}`	A set of constraint definitions.
`{substitution group affiliation}`	Optional. A top-level element definition.
`{substitution group exclusions}`	A subset of {extension, restriction}.
`{disallowed substitution}`	A subset of {substitution,extension,restriction}.
`{abstract}`	A boolean.
`{annotation}`	Optional. An annotation.

E.1.5 Attribute Declaration Schema Component

Table E-5 Attribute Declaration Schema Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name
`{type definition}`	A simple type definition.
`{scope}`	Optional. Either global or a complex type definition.
`{value constraint}`	Optional. A pair consisting of a value and one of default, fixed.
`{annotation}`	Optional. An annotation.

E.1.6 Model Group Definition Schema Component

Table E-6 Model Group Definition Schema Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{model group}`	A model group.
`{annotation}`	Optional. An annotation.

E.1.7 Identity-constraint Definition Schema Component

Table E-7 Identity-constraint Definition Schema Components

Component	Description
`{name}`	An NCName as defined by [XML-Namespaces].
`{target namespace}`	Either ·absent· or a namespace name.
`{identity-constraint category}`	One of key, keyref or unique.
`{selector}`	A restricted XPath ([XPath]) expression.
`{fields}`	A non-empty list of restricted XPath ([XPath]) expressions.

Table E-7 Identity-constraint Definition Schema Components *(Continued)*

Component	Description
`{referenced key}`	Required if {identity-constraint category} is keyref, forbidden otherwise. An identity-constraint definition with {identity-constraint category} equal to key or unique.
`{annotation}`	Optional. An annotation.

E.1.8 Attribute Use Schema Component

Table E-8 Attribute Use Schema Components

Component	Description
`{required}`	A boolean.
`{attribute declaration}`	An attribute declaration.
`{value constraint}`	Optional. A pair consisting of a value and one of default, fixed.

E.1.9 Particle Schema Component

Table E-9 Particle Schema Components

Component	Description
`{min occurs}`	A non-negative integer.
`{max occurs}`	Either a non-negative integer or unbounded.
`{term}`	One of a model group, a wildcard, or an element declaration.

E.1.10 Wildcard Schema Component

Table E-10 Wildcard Schema Components

Component	Description
`{namespace constraint}`	One of any; a pair of not and a namespace name or ·absent·; or a set whose members are either namespace names or ·absent·.
`{process contents}`	One of skip, lax or strict.
`{annotation}`	Optional. An annotation.

E.1.11 Model Group Schema Component

Table E-11 Model Group Components

Component	Description
`{compositor}`	One of *all, choice* or *sequence*.
`{particles}`	A list of particles.
`{annotation}`	An annotation.

E.2 Not Required XML Schema concepts

A JAXB implementation is not required to support the following XML Schema concepts for this version of the specification. A JAXB implementation may choose to support these features in an implementation dependent manner. For the purposes of compatibility, all JAXB technology implementations must have a strict operating mode that reports when non-required XML schema concepts are encountered by the binding compiler.

E.2.1 Concepts detected at binding compilation time

A binding compilation running in the strict conforming operating mode must report an error or warning when encountering any of these XML Schema components.

E.2.1.1 Binding Compilation Errors

No Java representation in the form of interfaces, classes, or properties can be derived for these schema components.

- **Redefinition of declaration**

  This infrequently used feature of XML Schema does not have an obvious data binding, thus, it is not supported in this initial release.

- **Schema component: identity-constraint definition** *(*key, keyref, unique*)*

Due to complexities surrounding supporting this feature, specify in a future version. No JAXB properties are derived from the schema to enable users of the JAXB technology to access and/or update the **key** and **keyref** attributes. The **unique** attribute is not required to be enforced by the JAXB technology validation.

- **Notation declaration**

  No Java representation is generated for Notation declaration.

- **Schema component: attribute wildcard (anyAttribute)**

  The schema derived Java representation provides no generated method signatures that enable attribute wildcards from an XML document to be manipulated by a Java application. An implementation should maintain the wildcard attribute content it parsed at unmarshal time and it can write it out at marshal time.

- **Substitution group**

  Any XML Schema concepts indicating that substitution group support is a necessary component of the schema should be reported as an error. A binding compiler must not report the following two cases as errors.

  a. `<xs:element abstract=”false”>`

  b. `<xs:element substitutionGroup=””>`

E.2.1.2 Binding compilation warnings

The existence of the following XML schema concepts are not considered an error. A JAXB implementation ignores the value of these attributes and treats them in a specific way that is reported as a warning.

- **“block” feature**

  Attributes:
  `schema.blockDefault,element.block`, `complexType.block`

  The values of these attributes are ignored and the binding compiler treats them each one as if it was set to “`#all`”.

- **“final”** feature

  `schema.finalDefault,element.final,`
  `complexType.final`

The values of these attributes are ignored and the binding compiler treats them as if they were set to “`#all`”.

- The value of attribute `complexType.abstract` is ignored and the binding compiler treats it as if it were set to the value “`false`”.
- **Skip wildcard content - `<xs:any processContents=”skip”>`**

  A warning should be issued that there is no specified standard way to bind non-schema constrained but well-formed XML content to a Java representation.

E.2.2 Not supported while manipulating the XML content

- **Schema component: wildcard (any)**

  A JAXB implementation always generates a property method for XML content as described in Section 5.9.5, “Bind wildcard schema component,” on page 98.

  JAXB implementations are not required to unmarshal or marshal XML content that does not conform to a schema that is registered with `JAXBContext`. However, wildcard content must be handled as detailed in Section 5.9.5, “Bind wildcard schema component,” on page 98.
- **Substitution**
  Schema derived code generated by a binding compiler operating in strict conforming mode is not allowed to perform any group or type substitution when unmarshalling XML content or when updating the Java representation of XML content. One should assume that all type definitions and element declarations in the schema have an effective block value of “#all” imposed upon them by the JAXB binding compilation process. Note that this specification does not specify how these non-supported substitutions are handled.

APPENDIX F

RELATIONSHIP TO JAX-RPC BINDING

F.1 Overview

Several minor differences in binding from XML to Java representation have been identified between JAXB technology and the JAX-RPC 1.0 Specification [JAX-RPC]. JAXB binding customizations are provided below that enable JAXB technology to bind from XML to a Java representation as JAX-RPC technology does for these cases.

F.2 Mapping XML name to Java identifier

By default, when mapping an XML Names to a Java identifier, the JAXB technology treats '_" (underscore) as a punctuation character (i.e. a word separator). However, JAX-RPC technology treats underscore as a character within a word as specified Section 20.1 in [JAX-RPC]. See customization option specified in Section 6.5.3, "Underscore Handling" to enable JAX-RPC mapping of XML name to Java identifier.

```
Customization to enable JAX-RPC conforming binding:
underscoreBinding  = "asCharInWord"
```

F.3 Bind XML enum to a typesafe enumeration

The JAX-RPC specification specifies the binding of XML datatype to typesafe enumeration class. JAXB-specified default bindings are designed to be as similar as possible to JAXRPC-specified bindings. However, there are differences that are described here. Customization options allow the JAX-RPC style of binding to be generated.

F.3.1 Restriction Base Type

The default restriction base type which can be mapped to a typesafe enumeration is different. The allowed types are customized using the customization option `typesafeEnumBase` specified in Section 6.5.1, "Usage."

```
Customization to enable JAX-RPC conforming binding:
typesafeEnumBase = “xsd:string xsd:decimal xsd:float xsd:double”
JAXB default is typesafeEnumBase ="xsd:NCName"
```

Note that all XML Schema built-in datatypes listed in the above customization and all datatypes that derived by restriction from these listed basetypes are mapped to typesafe enum classes. Thus, not all JAX-RPC supported types must be listed, only the types at the base of the derivation by restriction type hierarchy.

F.3.2 Enumeration Name Handling

If a legal Java identifier cannot be generated from an XML enumeration value, then by default, an error must be reported. However, JAX-RPC will revert the identifiers to be default enumeration label names as specified in Section 4.2.4 "Enumeration" in [JAX-RPC]. The latter behavior can be obtained enabling the customization `typesafeEnumMemberName` specified in Section 6.5.1, "Usage." Section 5.2.3.4, "XML Enumvalue-to-Java Identifier Mapping," on page 63 describes the enumeration member names generated when `typesafeEnumMemberName` is set to *"generateName."*

```
Customization to enable JAX-RPC conforming binding:
typesafeEnumMemberName = "generateName"
JAXB default is typesafeEnumMemberName = "generateError"
```

APPENDIX G

CHANGE LOG

G.1 Changes for Final

- Added method `javax.xml.bind.Marshaller.getNode(Object)` which returns a DOM view of the Java content tree. See method's javadoc for details.

G.2 Changes for Proposed Final

- Added Chapter 7, “Compatibility.”
- Section 5.9.2, “General Content Property,” removed value content list since it would not be tractable to support when type and group substitution are supported by JAXB technology.
- Added the ability to associate implementation specific property/value pairs to the unmarshal, validation and JAXB instance creation. Changes impact Section 3.4 “Unmarshalling,” Section 3.5 “Validator” and the ObjectFactory description in Section 4.2 “Java Package.”
- Section 5.9.10.1, “Bind a Choice Group to a Content Interface” was updated to handle Collection properties occurring within a Choice Content interface.
- Section 5.9.11, “Model Group binding algorithm” changed step 4(a) to bind to choice content interface rather than choice content property.
- Section 4.5.2.2, “List Property and Section 4.5.4, “isSet Property Modifier” updated so one can discard set value for a List property via calling unset method.

- At end of Section 4, added an UML diagram of the JAXB Java representation of XML content.
- Updated default binding handling in Section 5.5, “Model Group Definition.” Specifically, content interfaces, element interfaces and typesafe enum classes derived from the content model of a model group definition are only bound once, not once per time the group is referenced.
- Change Section 5.9.5, “Bind wildcard schema component,” to bind to a JAXB property with a basetype of `java.lang.Object,` not `javax.xml.bind.Element.` Strict and lax wildcard validation processing allows for contents constrained only by `xsi:type` attribute. Current APIs should allow for future support of `xsi:type`.
- Simplify anonymous simple type definition binding to typesafe enum class. Replace incomplete approach to derive a name with the requirement that the @name attribute for element typesafeEnumClass is mandatory when associated with an anonymous simple type definition.
- Changed Section 5.5.4, “Deriving Class Names for Named Model Group Descendants” to state that all classes and interfaces generated for XML Schema component that directly compose the content model for a model group, that these classes/interfaces should be generated once as top-level interface/class in a package, not in every content model that references the model group.
- Current Section 6.5, “<globalBindings> Declaration”:
    - Replaced `modelGroupAsClass` with `bindingStyle`.
    - Specified schema types that cannot be listed in `typesafeEnumBase`.
- Section 6.8, “<property> Declaration:
    - Clarified the customization of model groups with respect to `choiceContentProperty, elementBinding and modelGroupBinding.` Dropped `choiceContentProperty` from the `<property>` declaration.
    - Added `<baseType>` element and clarified semantics.
    - Added support for customization of simple content.
    - Added customization of simple types at point of reference.
    - Clarified restrictions and relationships between different customizations.
- Section 6.9, “<javaType> Declaration”:

    - ❍ Added `javax.xml.bind.DatatypeConverterInterface` interface.
    - ❍ Added `java.xml.bind.DatatypeConverter` class for use by user specified parse and print methods.
    - ❍ Added `java.xml.namespace.NamespaceContext` class for processing of QNames.
    - ❍ Clarified print and parse method requirements.
    - ❍ Added narrowing and widening conversion requirements.
- Throughout Chapter 6, “Customization,” clarified the handling of invalid customizations.

G.3 Changes for Public Draft 2

Many changes were prompted by inconsistencies detected within the specification by the reference implementation effort. Change bars indicate what has changed since Public Draft.

- Section 4.5.4, “isSetProperty Modifier,” describes the customization required to enable its methods to he generated.
- Section 5.7.2, “Binding of an anonymous type definition,” clarifies the generation of content interfaces and typesafe enum classes from an anonymous type definition.
- Section 5.2.4, “List” Simple Type Definition and the handling of list members within a union were added since public draft.
- Clarification on typesafe enum global customization “generateName” in Section 5.2.3.4, “XML Enumvalue To Java Identifier Mapping.”
- Clarification of handling binding of wildcard content in Section 5.9.4.
- Chapter6, “Customization,” resolved binding declaration naming inconsistencies between specification and normative binding schema.
- removed `enableValidation` attribute (a duplicate of `enableFailFastCheck)` from `<globalBindings>` declaration.
- Added default values for `<globalBindings>` declaration attributes.

- Changed `typesafeEnumBase` to a list of QNames. Clarified the binding to typesafe enum class.
- Clarified the usage and support for `implClass` attribute in `<class>` declaration.
- Clarified the usage and support for `enableFailFastCheck` in the `<property>` declaration.
- Added `<javadoc>` to typesafe enum class, member and property declarations.
- Mention that embedded HTML tags in `<javadoc>` declaration must be escaped.
- Fixed mistakes in derived Java code throughout document.
- Added Section 7. Compatibility and updated Appendix E.2 “Non required XML Schema Concepts” accordingly.

G.4 Changes for Public Draft

- Section 5.9.10.2, “Bind choice group to a choice content property,” replaced overloading of choice content property setter method with a single setter method with a value parameter with the common type of all members of the choice. Since the resolution of overloaded method invocation is performed using compile-time typing, not runtime typing, this overloading was problematic. Same change was made to binding of union types.
- Added details on how to construct factory method signature for nested content and element interfaces.
- Section 3.3, default validation handler does not fail on first warning, only on first error or fatal error.
- Add ID/IDREF handling in section 5.
- Updated name mapping in appendix C.
- section 4.5.2.1 on page 43, added getIDLenth() to indexed property.
- Removed ObjectFactory.setImplementation method from Section 4.2, “Java Package,” on page 36. The negative impact on implementation provided to be greater than the benefit it provided the user.
- Introduced external binding declaration format.

- Introduced a method to introduce extension binding declarations.
- Added an appendix section describing JAXB custom bindings that align JAXB binding with JAX-RPC binding from XML to Java representation.
- Generate isID() accessor for boolean property.
- Section 6, Customization has been substantially rewritten.