# The Java™ Architecture for XML Binding (JAXB)

*Working-Draft Specification*
*Version 0.21*
*30 May 2001*

Mark Reinhold, *editor*
Core Java Platform Group
Java Software

Comments to:
*jaxb-feedback@java.sun.com*

*Sun Microsystems, Inc.*
*901 San Antonio Road*
*Palo Alto, CA 94303 USA*

**Java$^{TM}$ Architecture for XML Binding (JAXB) Specification ("Specification")**
**Version: 1.0**
**Status: Pre-FCS**
**Release: May 30, 2001**

# Contents

**Preface** *1*

**1 Introduction** *3*
1.1 Data binding *3*
1.2 Goals *5*

**2 Architecture** *9*
2.1 Overview *9*
2.2 Varieties of validation *11*
2.3 An example *12*

**3 The binding framework** *15*
3.1 Validatable objects *15*
3.2 Elements, root elements, and identifiable elements *16*
3.3 Marshallable objects *17*
3.4 Marshalling *18*
3.5 Unmarshalling *18*
3.6 Dispatching *19*
3.7 Summary *20*
3.8 Exceptions *21*

**4 Names and identifiers** *23*
4.1 The algorithm *23*
4.2 Collisions and conflicts *25*

**5 Properties** *27*
5.1 Primitive properties *27*
5.2 Reference properties *28*
5.3 Collection properties *29*
5.4 Default values *32*

**6 Binding-schema syntax and semantics** *33*
6.1 Notation *34*
6.2 Limitations *34*
6.3 Root element and document type *35*
6.4 Options *35*
6.5 Element-class declarations *36*
6.6 General content-property declarations *39*
6.7 Conversion declarations *42*
6.8 Datatypes *46*
6.9 Element-value declarations *47*
6.10 Model-based content-property declarations *49*
6.11 Element-reference content-property declarations *51*
6.12 Choice, sequence, and rest content-property declarations *54*
6.13 Attribute-property declarations *55*
6.14 Default binding declarations *60*
6.15 Enumeration declarations *64*
6.16 Constructor declarations *67*
6.17 Interface declarations *68*

**A `javax.xml.bind` package specification** *71*
A.1 Interfaces *71*
A.2 Classes *72*
A.3 Checked exceptions *88*
A.4 Unchecked exceptions *93*

**B `javax.xml.marshal` package specification** *95*
B.1 Classes *95*
B.2 Checked exceptions *109*

**C DTD for the binding language** *111*

**D Examples** *115*
D.1 Simple stock trades *115*
D.2 Tree nodes *123*

# Preface

***Status of this document*** This is a working draft of the specification being developed within the Java Community Process for *JSR-31: XML Data Binding*. It is being published along with the first public early-access release of Sun's JAXB implementation in order to aid in the use and evaluation of that release.

This draft is known to be incomplete. It is not suitable for citation as an authoritative reference; it should only be cited as a work in progress.

> **The JSR-31 Expert Group reserves the right to make substantial changes to this draft in future revisions. The Expert Group makes no commitment as to whether the final specification will bear any resemblence to this draft. The Expert Group will not permit early applications or implementations of this draft to constrain the changes that it may make in future revisions.**

The publication of this draft should not be interpreted as an endorsement of this version of the specification by any particular member of the expert group. Some members of the expert group have chosen not to endorse this draft.

***How to comment on this document*** Please send comments via e-mail, in plain ASCII, to *jaxb-feedback@java.sun.com*. Please do not send an annotated version of this PDF file, as we do not have ready access to software for reading such annotations.

# 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. Schemas written in this language are by far the most common as of this writing. 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. These deficiencies have motivated proposals to add datatype information to DTDs, such as the DT4DTD datatype-annotation conventions. 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 aims to support both DTDs and a simple 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-level *binding* of the schema.

An *XML data-binding facility* therefore contains a *schema compiler* that binds components of an input schema to derived 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. 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 checks incoming XML documents for validity with respect to the schema. Similarly, the compiler generates code into the derived classes 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. The marshalling process operates only upon valid content trees, thereby ensuring that only valid documents are generated.

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

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 classes that handle all the details of marshalling and unmarshalling and also ensure that only valid documents will be produced and consumed. 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*. 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*. 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.

- Be *fast and lean*. It should be practical to use the facility for processing very large documents. It should also work well in resource-limited devices.

- *Support DTDs and W3C XML Schema*. The first version of this specification should support the DTD sublanguage of XML 1.0, the schema language that is most widely used today, along with the DT4DTD datatype conventions. It is highly desirable that it also support a simple subset of W3C XML Schema. Future versions may add more complete support for W3C XML Schema as well as support for other schema languages.

The derived classes produced by the schema 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 within the derived 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 make use of schema-derived classes.

- Be *extensible*. The customization mechanism most familiar to Java developers is that of extending a class by defining a subclass. It should be not only possible but easy to extend a derived class in order to add application-specific functionality.

- *Support serialization*. Some important applications of XML data binding, *e.g.*, the construction of Enterprise JavaBeans, make heavy use of the Java platform's object-serialization facility. Derived classes should make it easy to define serializable subclasses.

- *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 first marshalling the tree into XML.

- *Preserve equivalence* (round tripping). If an XML document can be unmarshalled into a content tree then marshalling that tree should produce an equivalent XML document. If a content tree can be marshalled into a document then unmarshalling that document should produce an equivalent tree.

***A non-goal*** It is not necessary for the facility described by this specification to implement every last feature of the schema languages that 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 non-goal does not imply that supporting document-oriented applications is something to be avoided; it merely recognizes 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.

***Goals abandoned*** Section 2.5 of JSR-31 describes a general-purpose *marshalling framework* that should

1. Not be specific to XML, if at all possible; and

2. Be usable for applications other than XML data binding, *e.g.*, for the XML-based archiving of arbitrary graphs of JavaBeans as envisioned in *JSR-57: Long-Term Persistence for JavaBeans*.

We have abandoned these goals in light of experience gained in the development of both this specification and a prototype implementation.

We abandoned (1) because defining a completely general marshalling framework is a very difficult problem. This goal was stated as a desideratum rather than a hard requirement in JSR-31, and achieving it is not necessary to the achievement of our other goals, therefore it was dropped in the interest of finishing this specification in a timely manner.

We partly abandoned (2) because it proved impractical to achieve completely without risking the achievement of several other goals. It is certainly possible to define an XML-oriented marshalling framework that is not specific to data binding, and indeed others have done so. A data-binding facility based upon such a framework would, however, have two significant drawbacks:

- *The runtime API would be more complex.* A general-purpose XML-based marshalling framework must be able to work with existing classes that cannot be modified, *i.e.*, classes to which self-contained **`marshal`** and **`unmarshal`** methods cannot be added. It must therefore support the definition of the required metadata in some other form.

- *Unmarshalling and marshalling would be less efficient.* In order to use a general-purpose framework it would be necessary to divorce the validation process from the unmarshalling and marshalling processes. After unmarshalling a content tree it would have to be traversed again in order to be validated. A tree would, similarly, have to be validated before being marshalled. Work on our prototype implementation has shown that tightly integrating these three processes allows validation to be performed or checked concurrently with both unmarshalling and marshalling.

This specification therefore describes a *binding framework* rather than a marshalling framework. In principle the binding framework as a whole may be used separately from the other components described in this specification, though we do not expect such usage to be common. The portion of the binding framework that deals exclusively with reading and writing XML documents does reside in a separate package, however, and we do expect that package to be more generally useful.

# 2 Architecture

## 2.1 Overview

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

- The *schema compiler* transforms, or *binds*, a *source schema* to a set of *derived classes*. As used in this specification, the term *schema* includes the document-type definition language of the XML 1.0 specification.

- The *binding framework* is a set of public interfaces and classes upon which derived classes rely to implement the operations of unmarshalling, marshalling, and validation.

- The *binding language* is an XML-based language that describes the binding of a source schema to a set of derived classes. A *binding schema* written in this language specifies the details of the classes derived from a particular source schema.

***Derived classes*** A *content class* is a class derived from a content-bearing component of the source schema. Most, but not all, schema-derived classes are content classes. An *element class* is the common case of a content class that corresponds directly to an element-type declaration in the source schema. A *root element class* is an element class that corresponds to an element type whose instances may serve as document roots. Instances of content classes are called *content objects*; instances of element classes are called *element objects*.

A coarse-grained schema component, such as an element-type declaration, is generally bound to a content class. A fine-grained component, such as an attribute declaration or a simple (*i.e.*, `#PCDATA`-only) element-type declaration, is bound directly to a *property* within a content class. A property is *realized* in a content class 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 object 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*.

***Operations*** The primary operations supported by a set of derived 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.

The unmarshalling process incorporates the validation process: If an unmarshalling operation is successful then 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. Only valid content trees may be marshalled, however; this guarantees that generated XML documents are always valid with respect to the source schema.

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 constructors and methods. Once created a content tree may be re-validated, either in whole or in part, at any time.

***Binding schemas*** A particular binding of a given source schema is defined by an auxiliary document called the *binding schema*. Binding schemas are written in the *binding language*, which is itself an application of XML. The set of classes that constitute a particular binding is the result of a process that evaluates a source schema together with a binding schema. The schema compiler hence actually requires two inputs, a source schema and a binding schema.

A binding schema defines those aspects of a binding that do not follow directly from the source schema. It defines the names of the derived classes as well as the names and types of the methods within those classes. It defines the binding of element-type declarations to element classes and allows components of an element type's content specification to be bound to specific properties in the corresponding class. It defines the binding of simple element-type declarations and attribute declarations to properties in the appropriate element classes. Finally, it provides for the definition of data conversions, constructors, interfaces, and typesafe-enumeration classes.

A binding schema need not define every last detail of a binding. The schema compiler assumes *default binding declarations* for those components of the source schema that are not mentioned explicitly in the binding schema. Default declarations both reduce the verbosity of a binding schema 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 schema at all.

## 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 given to attributes or used as the content of simple, **`#PCDATA`**-only elements.

- 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. Hence the schema compiler maps each constraint in the source schema into one or more Java language constructs or code fragments which enforce it.

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 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-compile 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 throws 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.*, an element-type declaration whose content specification is very complex. Suppose that an instance of the corresponding element class 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 element object 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 ensure that type constraints always hold, will be able to check structural constraints upon demand, and will require that all constraints hold before a content tree is marshalled.

## 2.3 An example

Throughout this specification we will use the following DTD, which describes a simple stock trade, as a running example to illustrate various concepts as they are defined.

```
<!ELEMENT trade ( symbol, quantity, limit?, stop?, date ) >
<!ATTLIST trade
          account CDATA #REQUIRED
          action ( buy | buy-to-cover | sell | sell-short ) #REQUIRED
          duration ( immediate | day | good-til-canceled ) "day" >

<!ELEMENT symbol (#PCDATA) >
<!ELEMENT quantity (#PCDATA) >
<!ELEMENT limit (#PCDATA) >
<!ELEMENT stop (#PCDATA) >
<!ELEMENT date (#PCDATA) >
```

The type constraints in this DTD are:

- The value of the **`action`** attribute must be either **`buy`**, **`buy-to-cover`**, **`sell`**, or **`sell-short`**, and
- The value of the **`duration`** attribute must be either **`immediate`**, **`day`**, or **`good-til-canceled`**.

The local structural constraints in this DTD are:

- A **`trade`** element must contain a **`symbol`** element, followed by a **`quantity`** element, optionally followed by a **`limit`** element or a **`stop`** element or both, followed by a **`date`** element, and
- The **`account`** and **`action`** attributes are required.

This DTD does not describe any global structural constraints.

Here is a document that is valid with respect to this DTD:

```
<trade account="2520034"
       action="sell"
       duration="good-til-canceled">
  <symbol>SUNW</symbol>
  <quantity>1000</quantity>
  <limit>35</limit>
  <date>2001-2-26</date>
</trade>
```

We will revisit this example periodically throughout this specification. Appendix D presents a complete binding schema as well as an implementation of the **`Trade`** and **`TradeBatch`** classes that might be generated by a schema compiler that implements this specification.

# 3 The binding framework

The *binding framework* defines the interfaces and abstract classes that are implemented and extended by the content classes derived from a source schema. It also defines the interfaces and classes that implement the validation, marshalling, and unmarshalling processes. The framework is presented here in overview; its full specification is in appendices A and B.

The binding framework resides in two packages. The **`javax.xml.marshal`** package defines lower-level classes that support the reading and writing of XML documents that are not necessarily valid or even well-formed. This package defines an **`XMLScanner`** class for scanning documents and an **`XMLWriter`** class for generating documents. Subclasses of these classes can read or write raw byte streams, consume or create SAX event streams, or traverse or create DOM parse trees. That these abstractions do not guarantee validity or well-formedness is intentional: By relying upon the unmarshalling and marshalling code in derived classes to enforce these properties it is possible to read and write XML documents much more quickly than with more conventional approaches.

The **`javax.xml.bind`** package defines higher-level classes and interfaces that are used directly by schema-derived content classes. The classes **`ValidatableObject`**, **`MarshallableObject`**, and **`MarshallableRootElement`** define the core unmarshalling, validation, and marshalling logic that is shared by all derived classes. An element class implements the **`Element`** interface, while a root element class implements the **`RootElement`** interface.

The **`javax.xml.bind`** package also defines *unmarshallers*, *validators*, and *marshallers*, which are auxiliary objects that govern the unmarshalling, validation, and marshalling processes, respectively. It defines *dispatchers*, which map element names to element classes, as well as classes for implementing collection properties and for representing parsed character data. Finally, it defines a rich hierarchy of exception classes for use when scanning errors occur, when constraints are violated, and when other types of errors are detected.

## 3.1 Validatable objects

Every schema-derived content class supports validation by indirectly extending the abstract class **`ValidatableObject`**. This class defines the public instance methods as well as the internal state required for content-tree validation:

```
public abstract class ValidatableObject {
    public void validateThis()
        throws LocalValidationException;
    public void validate(Validator v)
        throws StructureValidationException;
    public final void validate()
        throws StructureValidationException;
    public final void invalidate();
}
```

The **`validateThis`** method of a validatable object may be invoked at any time in order to check that the object does not violate any local structural constraints, throwing an instance of a **`LocalValidationException`** subclass upon failure.

A content tree may be validated by invoking the **`validate`** method of its root element object. This method creates a new *validator* by instantiating the **`Validator`** class. The validator governs the process of validating the content tree, serves as a registry for identifier references, and ensures that all local and global structural constraints are checked before the validation process is complete.

The validator invokes each object's **`validateThis`** method, if necessary, to ensure that all local structural constraints are satisfied; an invocation of one of these methods may throw a **`LocalValidationException`**.

The validator also invokes each object's **`validate(Validator)`** method, passing itself. This method recursively validates the object's children and updates the validator with any local data, such as identifier definitions or references, that are subject to global constraints. If a violation of a global structural constraint is detected then an instance of a **`GlobalValidationException`** subclass is thrown. The abstract superclass **`StructureValidationException`** unifies the **`LocalValidationException`** and **`GlobalValidationException`** classes so as to simplify code that handles these exceptions.

A validatable object's **`invalidate`** method is invoked in order to note that it has been changed in a way that may violate a structural constraint, and that it therefore requires re-validation. While this method is public, it is intended for use only by schema-derived code. The **`validate`** and **`invalidate`** methods are final in order to protect the integrity of the validation logic.

***Design note*** The approach taken here does not attempt to track whole-tree validity. Validation always checks all global constraints, and checks the local constraints of all content objects that have been marked invalid. If one object is invalid then global constraints may require checking, while if all objects are valid then it is known that all global constraints hold. A content object must therefore be invalidated if it is modified in a way that may affect either local or global validity. This allows the marshalling logic to detect when validation is required, in which case it throws a **`ValidationRequiredException`**.

The reason for not tracking whole-tree validity is that it would make it impossible to support natural and efficient manipulations of content trees. Every content object would have to contain a reference to its parent, or to the root of its tree. Splicing one part of a tree into another would hence require updating references in both trees, and an object could not be a member of more than one tree. The design described here avoids these limitations, allowing content objects to be used freely.

## 3.2 Elements, root elements, and identifiable elements

A content class derived directly from an element-type declaration in the source schema implements the **`Element`** interface. This interface specifies no methods beyond those already defined in the **ValidatableObject** class; it is simply a marker interface that allows element classes to be distinguished from other schema-derived content classes.

An element class for a root element type implements the **RootElement** class, which itself extends the **Element** interface. This interface is also a marker interface, used to distinguish root and non-root element classes.

If an element class contains a property derived from an identifier (ID) attribute then it also implements the **IdentifiableElement** interface. This interface specifies a single method that returns the value of the identifier property. It is used by the unmarshalling, marshalling, and validation processes to record and retrieve the identifier values of element objects.

## 3.3 Marshallable objects

A schema-derived content class extends, either directly or indirectly, the abstract class **MarshallableObject**, which itself extends **ValidatableObject**. This class defines the public instance methods that support both marshalling and unmarshalling:

```
public abstract class MarshallableObject
    extends ValidatableObject
{
    protected MarshallableObject();
    public void marshal(Marshaller m)
        throws IOException;
    public void unmarshal(Unmarshaller u)
        throws UnmarshalException;
}
```

The binding schema may instruct the schema compiler to generate content classes that can only be marshalled or can only be unmarshalled. If the method corresponding to an unsupported operation is invoked then an **UnsupportedOperationException** is thrown.

A marshallable root-element class extends the **MarshallableRootElement** abstract class, which itself extends **MarshallableObject**:

```
public abstract class MarshallableRootElement
    extends MarshallableObject
    implements RootElement
{
    protected MarshallableRootElement();
    public final void marshal(XMLWriter xw)
        throws IOException;
    public final void marshal(OutputStream out)
        throws IOException;
}
```

This class defines the primary marshalling method, which takes the **XMLWriter** with which it will generate the output document. It also defines a convenience method that accepts a byte-output stream rather than an XML writer. The **marshal** methods defined this class are final in order to protect the integrity of the marshalling logic.

## 3.4 Marshalling

A content tree may be marshalled by invoking one of the **marshal** methods of its root element object. This creates a new *marshaller* by instantiating the **Marshaller** class. The marshaller then proceeds to marshal the content tree by invoking the **marshal** method of each content object.

The marshalling process does not validate the content tree being marshalled, but it does require the tree to be valid. If an invalid object is detected during marshalling then a **ValidationRequiredException** is thrown and the marshalling operation is aborted.

## 3.5 Unmarshalling

The **MarshallableRootElement** class defines methods for marshalling but not for unmarshalling. The unmarshalling methods associated with a specific root element class must be able to create new instances of the class, hence they must be static methods. The Java programming language does not support abstract static methods, however, so the **MarshallableRootElement** class specifies, but does not define, the static unmarshalling methods which the schema compiler generates into each unmarshallable root element class. The primary unmarshalling method of a root element class *Foo* has the signature

```
public static Foo unmarshal(XMLScanner xs, Dispatcher d)
    throws UnmarshalException;
```

This method takes the **XMLScanner** with which it will scan the input document. It also takes the *dispatcher* with which it will map element names to element classes. (Dispatchers are described in more detail below.) The schema compiler also generates two static convenience methods:

```
public static Foo unmarshal(XMLScanner xs)
    throws UnmarshalException;
public static Foo unmarshal(InputStream in)
    throws UnmarshalException;
```

Both of these methods use a default dispatcher; the second method is a convenience method that creates a new **XMLScanner** from the given byte-input stream.

An XML document whose root element type is known may be unmarshalled into a content tree by invoking one of the static **unmarshal** methods of the corresponding root element class. This method creates a new *unmarshaller* by instantiating the **Unmarshaller** class. The unmarshaller governs the process of unmarshalling the input document into a newly-created content tree. To unmarshal an element it invokes the given dispatcher to map the element's name to a marshallable element class. The dispatcher, in turn, instantiates the class and invokes the newly-created element object's **`unmarshal(Unmarshaller)`** method in order to unmarshal the corresponding portion of the input document. This method will invoke the dispatcher recursively in order to unmarshal any subelements of the element being unmarshalled.

The unmarshaller validates the new content tree while constructing it. When initializing a property it checks the property's static and dynamic type constraints, throwing a **`TypeValidationException`** upon failure. It invokes each unmarshalled object's **`validateThis`** method to check that all local structural constraints are satisfied, throwing a **`LocalValidationException`** upon failure. Finally, it checks that all global structural constraints are satisfied before the unmarshalling operation completes, throwing a **`GlobalValidationException`** upon failure.

The abstract superclass **`UnmarshalException`** is defined as a superclass of these three exception classes, as well as of **`ScanException`**, so as to simplify code that handles these exceptions. Instances of **`ScanException`** subclasses are thrown by a scanner if the input document is not lexically well formed or if an I/O error occurs. Throwing any one of these exceptions aborts the unmarshalling process and closes the scanner.

## 3.6 Dispatching

An XML document whose root element type is not known may be unmarshalled by invoking one of the **`unmarshal`** methods of an appropriate *dispatcher*. Dispatchers are also used in a recursive fashion during the unmarshalling process to unmarshal subelements which may have arbitrary element names.

A dispatcher maps element names to class names using two maps: An *element-name map* maps element names to marshallable-object classes, and a *class map* maps schema-derived marshallable-object classes to arbitrary marshallable-object subclasses. The class map acts as the identity map on classes for which it contains no mapping.

This two-level mapping scheme supports the user subclassing of schema-derived marshallable-object subclasses. When asked to unmarshal an element, a dispatcher first applies the element-name map to the element name and then applies the class map to the result. When asked to unmarshal an instance of a specific marshallable-object class it first applies the class map to the class and then unmarshals the result.

Dispatchers may be constructed and initialized from scratch, but it will be more common for applications to use a *default dispatcher* obtained from the static **`newDispatcher`** method generated into every marshallable root element class by the schema compiler:

```
public static Dispatcher newDispatcher();
```

This method creates a new dispatcher and initializes it to map each element-type name defined in the source schema from which the root element class was derived to the corresponding marshallable root element class. After being initialized the element-name map is frozen so that it cannot be modified further. User-defined subclasses of schema-derived marshallable-object classes may then be registered if desired.

Dispatchers are intended eventually to support XML namespaces, but this functionality will not be complete until a future revision of this specification supports a namespace-capable schema language.

## 3.7 Summary

The interface and abstract classes that are implemented and extended by schema-derived content classes can be summarized pictorially as follows, with classes on the left and interfaces on the right:

```
ValidatableObject                                  Element
       |                                               ||
       +-- MarshallableObject            RootElement ==++
                  |                             ||
                  +-- MarshallableRootElement ==++
```

From the standpoint of a developer using the data-binding facility, the essential unmarshalling, validation, and marshalling methods of a marshallable root-element class *Foo* are:

```
public class Foo
    extends MarshallableRootElement
{
    public void validateThis()
        throws LocalValidationException;
    public final void validate()
        throws StructureValidationException;
    public final void marshal(XMLWriter xw)
        throws IOException;
    public final void marshal(OutputStream out)
        throws IOException;
    public static Foo unmarshal(XMLScanner xs, Dispatcher d)
        throws UnmarshalException;
    public static Foo unmarshal(XMLScanner xs)
        throws UnmarshalException;
    public static Foo unmarshal(InputStream in)
        throws UnmarshalException;
}
```

The `validate` method validates the entire content tree rooted at a particular instance of *Foo*. The `marshal(XMLWriter)` methods may be used to marshal the tree, and the `unmarshal` methods may be used to unmarshal a new tree from a given XML scanner or input stream.

## 3.8 Exceptions

Methods defined in both binding-framework and schema-derived classes can throw a variety of exceptions, all of which are defined in the binding framework.

The three primary constraint-exception classes reflect the categories of schema constraints discussed in §2.2:

- 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`**.

- An instance of a **`LocalValidationException`** subclass is thrown when a violation of a structural constraint is detected. Such exceptions are thrown by the **`validateThis`** methods defined in schema-derived classes.

- An instance of a **`GlobalValidationException`** subclass is thrown when a violation of a global structural constraint is detected. Such exceptions are thrown by the **`validate(Validator v)`** methods defined in schema-derived classes and by various other methods in the binding framework.

The **`LocalValidationException`** and **`GlobalValidationException`** exception classes are unified by their superclass **`StructureValidationException`**.

The unmarshalling and validation processes are slightly different: Unmarshalling may detect the violation of a type constraint, but validation need only check structural constraints since schema-derived classes guarantee that type constraints always hold. If a type-constraint exception is thrown during unmarshalling then it is wrapped into a checked **`TypeValidationException`**. The **`StructureValidationException`** and **`TypeValidationException`** classes are unified by **`ValidationException`**.

Another difference between unmarshalling and validation is that unmarshalling involves scanning an input document, during which a lexical well-formedness error may be detected or an I/O error may occur. Such errors cause an instance of a **`ScanException`** subclass to be thrown; in particular, if a scanner is scanning XML text from a byte-input stream then a **`java.io.IOException`** thrown by the input stream is wrapped in a **`ScanIOException`**.

A final essential difference between unmarshalling and validation is that unmarshalling may apply conversions specified in the binding schema. If a conversion's parse method throws an exception then it is wrapped in a new **`ConversionException`**. The **`ValidationException`**, **`ScanException`**, and **`ConversionException`** exception classes are unified by their superclass **`UnmarshalException`**.

# 4 Names and identifiers

XML schema languages use *XML names*, *i.e.*, strings that match production 5 of the XML 1.0 specification, to label schema components. This set of strings is much larger than the set of valid Java class, method, and constant identifiers. This chapter defines 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 source schema, and is unlikely to result in many collisions.

## 4.1 The 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 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, or are prefixed by punctuation characters, which is not usual in Java 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 *lower-case* 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:

| *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** |

(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 or, if no prefix is required, the first character is converted to lower case.
- 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.

***Examples***

| *XML name* | *Class name* | *Method name* | *Constant name* |
|---|---|---|---|
| **`mixedCaseName`** | **`MixedCaseName`** | **`mixedCaseName`** | **`MIXED_CASE_NAME`** |
| **`Answer42`** | **`Answer42`** | **`answer42`** | **`ANSWER_42`** |
| **`name-with-dashes`** | **`NameWithDashes`** | **`nameWithDashes`** | **`NAME_WITH_DASHES`** |
| **`other_punct•chars`** | **`OtherPunctChars`** | **`otherPunctChars`** | **`OTHER_PUNCT_CHARS`** |

## 4.2 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 schema compiler and are reported as errors; they may be repaired by revising the source schema or the binding schema.

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 **`ValidatableObject`**, **`MarshallableObject`**, **`MarshallableRootElement`**, or in the interface **`IdentifiableElement`**. Such conflicts are reported as errors and may be repaired by revising the appropriate schema.

***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 schema. 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. In a DTD, *e.g.*, 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.

# 5 Properties

The schema compiler binds schema components to *properties* within derived content classes. A property is defined by:

- A *name*, which is an XML name;
- A *base type*, which may be a primitive type (*e.g.*, **`int`**), an **`Element`** or **`MarshallableObject`** type, or some other reference type;
- A *predicate*, which is a code fragment that tests values of the base type for validity and throws a **`TypeConstraintException`** if a type constraint expressed in the source schema is violated;
- An optional *collection type*, which is used for properties whose values may be composed of more than one value; and
- An optional *default value*.

A property is *realized* by a set of *access methods*. The precise set of methods is determined by the property's base type and its collection type, if any.

A property's access methods are, by default, named in the standard JavaBeans style: The name-mapping algorithm is applied to the property name and then each method name is constructed by prepending the appropriate prefix verb (**`get`**, **`set`**, **`has`**, **`delete`**, *etc.*). The retrieval (**`get`**) and mutation (**`set`**) methods may instead be named in the C++ style, without the **`get`** and **`set`** prefixes, by declaring the appropriate option in the binding schema.

A property is said to have a *given value* if that value was explicitly assigned to it during unmarshalling, during construction, or by invoking a mutation method; if there is no such value then the given value is said to be undefined. The *current value* of a property is its given value, if defined; otherwise, it is the property's default value, if any; otherwise, it is undefined.

## 5.1 Primitive properties

A non-collection property named *prop* with a primitive base type *type* is realized by the four methods

```
public type getId();           or    public type id();
public void setId(type x);     or    public type id(type x);
public boolean hasId();
public void deleteId();
```

where *Id* and *id* are metavariables that represent the Java method identifiers computed by applying the name mapping algorithm to *prop*, the former with the first character in upper case and the latter with the first character in lower case. The retrieval- and mutation-method declarations on the right are used instead of those on the left when the C++ style is selected.

Each method examines or manipulates the property's given and default values in the content object upon which it is invoked:

- The **`get`** method returns the property's current value. If the property has no current value, *i.e.*, it has neither a given nor a default value, then a **`NoValueException`** is thrown.
- The **`set`** method defines the property's given value to be the argument value. The value is first validated by applying the property's predicate, which may throw a **`TypeConstraintException`**.
- The **`has`** method returns a boolean value indicating whether or not the property has a current value. This method may be invoked prior to invoking the **`get`** method in order to prevent a **`NoValueException`** from being thrown.
- The **`delete`** method discards the property's given value, if any.

The realizations of primitive properties contain **`has`** and **`delete`** methods because with primitive types there is no convenient value with which to denote the absence of a value.

***Example*** In our stock-trade example, the **`account`** attribute of the **`trade`** element type is declared:

```
<!ATTLIST trade
          account CDATA #REQUIRED >
```

With an appropriate binding schema, this attribute could be bound to a primitive **`int`** property realized by these four methods:

```
public int getAccount();
public void setAccount(int x);
public boolean hasAccount();
public void deleteAccount();
```

It is legal to invoke the **`deleteAccount`** method even though the attribute is **`#REQUIRED`** in the DTD. That the attribute actually have a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

## 5.2 Reference properties

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

```
public Type getId();          or    public Type id();
public void setId(Type x);    or    public void id(Type x);
```

where *Id* and *id* are defined as above and the declarations on the right are used instead of those on the left when the C++ style is selected.

- The **`get`** method returns the property's current value. If the property has no current value then the value **`null`** is returned.

- The **`set`** method defines the property's given value to be the argument value. If the value is not **`null`** then it is first validated by applying the property's predicate, which may throw a **`TypeConstraintException`**. If the value is **`null`** then the property's given value, if any, is discarded.

Reference properties do not have **`has`** or **`delete`** methods because the value **`null`** is used to denote the absence of a value. To test whether a reference property has a current value, invoke its **`get`** method and check that the result is not **`null`**. To discard a reference property's given value, invoke its **`set`** method with an argument value of **`null`**.

***Example*** In our stock-trade example, the **`action`** attribute of the **`trade`** element type is declared:

```
<!ATTLIST trade
          action ( buy | buy-to-cover | sell | sell-short )
                 #REQUIRED >
```

This attribute declaration could be bound to a reference property with the base type **`java.lang.String`**:

```
public String getAction();
public void setAction(String x);
```

The **`setAction`** method would apply a predicate to its argument to ensure that the new value is legal, *i.e.*, that it is one of the strings **`"buy"`**, **`"buy-to-cover"`**, **`"sell"`**, or **`"sell-short"`**.

It is legal to pass **`null`** to the **`setAction`** method even though the **`action`** attribute is **`#REQUIRED`** in the DTD. As with the **`account`** attribute, that the **`action`** attribute actually have a value is a local structural constraint rather than a type constraint, so it is checked during validation rather than during mutation.

## 5.3 Collection properties

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

A collection property may have a current value that is empty, *i.e.*, of length zero. A value of length zero is distinct from no value.

***Design note*** Non-sequential collection types such as sets are not supported because XML documents are inherently linear.

### 5.3.1 Array properties

An array property *prop* with base type *Type* is realized by the two methods

```
public Type[] getId();              or    public Type[] id();
public void setId(Type[] x);        or    public void id(Type[] x);
```

regardless of whether *Type* is a primitive type or a reference type. *Id* and *id* are defined as above, and the declarations on the right are used instead of those on the left when the C++ style is selected.

- The **`get`** method returns an array containing the property's current value. If the property has no current value then **`null`** is returned.
- The **`set`** method defines the property's given value. If the argument is not **`null`** then the sequence of values in the array are first validated by applying the property's predicate, which may throw a **`TypeConstraintException`**. A **`NullValueException`** is thrown if the property's base type is a reference type and one or more elements of the array are **`null`**. If the argument itself is **`null`** then the property's given value, if any, is discarded.

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

The realizations of array properties, like those of reference properties, do not contain **`has`** or **`delete`** methods. To test whether an array property has a current value, invoke its **`get`** method and check that the result is not **`null`**. To discard an array property's given value, invoke its **`set`** method with an argument of **`null`**.

***Example*** Suppose that we extend the stock-trade DTD with the **`trade-batch`** element type:

```
<!ELEMENT trade-batch ( trade+ ) >
```

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

```
public Trade[] getTrades();
public void setTrades(Trade[] x);
```

The **`setTrades`** method would ensure that its argument has no **`null`** elements. It would not, however, require that its argument have a non-zero length or not be **`null`**, since that is a local structural constraint rather than a type constraint.

### 5.3.2 List properties

A list property *prop* with base type *Type* is realized by the three methods

```
public List getId();    or    public List id();
public void deleteId();
public void emptyId();
```

where **List** is the interface **java.util.List**, *Id* and *id* are defined as above, and the declaration on the right is used instead of that on the left when the C++ style is selected.

- 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 current value. If the property has a default value but no given value then the collection contains the default value and is therefore immutable. If the property has no current value then **null** is returned.
- The **delete** method discards the property's given value, if any.
- The **empty** method discards the property's given value, if any, and then defines its given value to be a newly-allocated, empty, mutable list.

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. The list's mutation methods do not admit **null** values; an attempt to add a **null** element to the list or to replace an existing element's value with **null** will cause a **NullValueException** to be thrown. 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 **get** method always returns the same collection object between invocations of the **delete** and **empty** methods. Any attempt to use a collection previously returned by the **get** method after one of the latter two methods has been invoked will cause an **IllegalStateException** to be thrown.

***Design notes*** 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***>**.

It may seem odd that list properties have both **delete** and **empty** methods. A list can be empty, so why not indicate that there is no current value simply by returning an empty list? This approach is unworkable because emptiness is not equivalent to absence. An attribute whose value is represented by a list property, *e.g.*, will be marshalled even if the list is empty, but it will not be marshalled if the list is absent.

***Example*** The content specification of the **trade-batch** element type could alternatively be bound to a list property realized by these three methods:

```
public List getTrades();
public void deleteTrades();
public void emptyTrades();
```

The list returned by the **`getTrades`** method would be guaranteed only to contain instances of the **`Trade`** class. As before, its length or emptiness would only be checked during validation, since the requirement that there be at least one **`trade`** in a **`trade-batch`** is a structural constraint rather than a type constraint.

## 5.4 Default values

If a property *prop* has a default value then its realization contains the additional method

```
public boolean defaultedId();
```

where *Id* is defined as above.

The **`defaulted`** method returns **`true`** if, and only if, the property has a default value but not a given value. This method returns **`false`** even if the property has a given value that happens to be equal to its default value.

***Example*** The **`duration`** attribute of the **`trade`** element type is declared:

```
<!ATTLIST trade
          duration ( immediate | day | good-til-canceled )
                   "day" >
```

This attribute declaration, like that for the **`action`** attribute, could be bound to a reference property with the base type **`java.lang.String`**. The realization of this property would contain an additional **`defaultedDuration`** method:

```
public String getDuration();
public void setDuration(String x);
public boolean defaultedDuration();
```

If the property does not have a given value then an invocation of the **`defaultedDuration`** method would return **`true`** and an invocation of the **`getDuration`** method would return **`"day"`**, the property's default value. The **`setDuration`** method would apply a predicate to its argument to ensure that the new value is legal according to the attribute declaration.

***Design note*** Each sort of property is realized by a slightly different set of access methods. For uniformity we could define every property to be realized by **`get`**, **`set`**, **`has`**, and **`delete`** methods, but this would yield element classes that are less natural and more complex. The **`get`** methods of simple reference properties, *e.g.*, would throw a **`NoValueException`** rather than return **`null`**, and the corresponding **`set`** methods would similarly throw an exception when invoked with a **`null`** argument. The semantics of collection properties would become more complex because a **`set`** method would have to be supported and it would no longer be practical to provide fail-fast checking of property predicates. For these and other reasons we have chosen to take the above non-uniform approach.

# 6 Binding-schema syntax and semantics

The schema compiler interprets a source schema together with a binding schema in order to generate a set of Java classes and interfaces. The source schema is written in the source schema language. The binding schema is written in the *binding language*, the syntax and semantics of which are defined in this chapter.

The binding language is, in essence, a specialized declarative programming language with the following kinds of constructs:

- *Element declarations*, of which there are two kinds: An *element-class* declaration binds an element-type declaration in the source schema to a specific generated Java class, while an *element-value* declaration specifies that instances of an element type are to be treated as values.

- *Content-property declarations*, of which there are two kinds: A *general content property* declaration defines a single property that represents all of an element's content, while a *model-based content property* declaration binds some or all of the components of the associated element type's content model to specific properties.

- An *attribute-property declaration* binds an attribute declaration to a property that represents the attribute's value.

- A *conversion declaration* defines a named *conversion* that may be specified in an element-value or attribute declaration in order to convert element content or attribute values to values of types other than **`java.lang.String`**.

- An *enumeration declaration* defines an *enumeration class*, which represents a set of named values and a bijective mapping between strings and these values; it also defines an implicit conversion for that class.

- A *constructor declaration* defines a constructor in an element class that, when invoked, initializes one or more properties with its arguments.

- An *interface declaration* defines a Java interface that is implemented by one or more of the classes or interfaces defined elsewhere in the binding schema; the interface may also specify one or more of the properties defined by these classes and interfaces.

- An *options declaration* contains various simple declarations that govern the interpretation of both the source schema and the binding schema.

The syntax of the binding language is independent of any particular source schema language. Its semantics, however, are of necessity partly dependent upon that language. In this chapter we separate the independent and dependent parts insofar as possible, though for explanatory purposes we often use the terminology of DTDs when talking about concepts that are common to nearly all XML schema languages.

A binding schema is interpreted relative to both a source schema and a set of *default binding declarations* induced by that schema. A binding schema therefore need not contain a declaration for every element type, attribute, or content-model component declared in the source schema. Default declarations both reduce the verbosity of a binding schema and make it more robust to the evolution of the source schema.

## 6.1 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 that described by the DTD in appendix C.

For the sake of brevity we omit end tags in all example XML text, using indentation instead to indicate element nesting. Empty-element tags are, however, still used in order to highlight the fact that a particular element is empty. Thus the fragment

```
<parent>
  <child name="alpha">
    <grandchild name="alpha2"/>
  </child>
  <child name="beta"/>
</parent>
```

is abbreviated to

```
<parent>
  <child name="alpha">
    <grandchild name="alpha2"/>
  <child name="beta"/>
```

Metavariables are in *italics* and other metanotation is in gray. Optional attributes are enclosed in [**`square="brackets"`**], and the legal values for enumeration attributes are written as {**`a`**|**`b`**|**`c`**}. Text is also grayed when it is part of the context of the construct being described rather than part of the construct itself.

## 6.2 Limitations

The only source-schema language supported by this specification is the DTD sublanguage of XML 1.0. A future revision is expected to include support for a simple subset of the W3C XML Schema language.

The following features of DTDs are not supported:

- Internal subsets,
- **`NOTATION`**s, and
- the **`ENTITY`**, **`ENTITIES`**, and enumerated **`NOTATION`** attribute types.

These constructs are rarely used in data-oriented applications of DTDs; they are also difficult, if not impossible, to support in the context of data binding.

## 6.3 Root element and document type

The root element of a binding schema starts with the tag

```
<xml-java-binding-schema [version="1.0-ea"]>
```

and may have, as its content, any number of interface, conversion, enumeration, or element declarations; it may also have at most one options declaration. The immediate children of the root element constitute the *top level* of the binding schema.

The value of the **version** attribute identifies the version of the binding language in which the binding schema is written. Each distinct version of this specification will specify a distinct version string. An implementation of a particular version of this specification will accept binding schemas whose root elements specify the corresponding version string and interpret them according to that version of this specification; if the version attribute is not present then that version is assumed. An implementation may, optionally, support earlier versions of this specification, in which case the value of the version attribute determines the version to be used.

An XML 1.0 document-type definition (DTD) for the version of the binding language described by this specification will be available at the URI

```
http://java.sun.com/dtd/jaxb/1.0-ea/xjs.dtd
```

This URI is therefore suitable for use as a system identifier in the **DOCTYPE** declaration of a binding schema:

```
<!DOCTYPE xml-java-binding-schema
  SYSTEM "http://java.sun.com/dtd/jaxb/1.0-ea/xjs.dtd">
```

A binding schema is not required to have a **DOCTYPE** declaration, but it is often helpful to provide one for use by XML editors and other tools.

## 6.4 Options

An options declaration has the form

```
<options [package="package"]
         [default-reference-collection-type="{array|list}"]
         [property-get-set-prefixes="{true|false}"]
         [marshallable="{true|false}"]
         [unmarshallable="{true|false}"]/>
```

and may appear, at most once, only at the top level of the binding schema.

The options declaration serves as a container for simple declarations that govern the interpretation of the source schema and of the other declarations in the binding schema.

- The **`package`** option specifies the *target package* for the classes and interfaces that will be generated by element-class, interface, and enumeration declarations; its value must be a valid Java package name. If this option is not specified then the generated classes will be in the unnamed package.

- The **`default-reference-collection-type`** option specifies the collection type to be used for reference properties that may have compound values. If this option is not specified then its value is defaulted to **`list`**.

- The **`property-get-set-prefixes`** option specifies whether property access methods are to be named in the JavaBeans style, with **`get`** and **`set`** prefixes, or in the C++ style, without such prefixes. If this option is not specified then its value is defaulted to **`true`**.

- The **`marshallable`** option specifies whether or not the generated classes are to support marshalling. If the value **`false`** is given then the **`marshal`** methods defined by the generated element classes will always throw an **`UnsupportedOperationException`** when invoked. If this option is not specified then its value is defaulted to **`true`**.

- The **`unmarshallable`** option specifies whether or not the generated classes are to support unmarshalling. If the value **`false`** is given then the **`unmarshal`** methods defined by the generated element classes will always throw an **`UnsupportedOperationException`** when invoked. If this option is not specified then its value is defaulted to **`true`**.

## 6.5 Element-class declarations

An element-class declaration starts with the tag

```
<element name="elt" type="class"
         [class="Class"] [root="{true|false}"]>
```

and may appear only at the top level of the binding schema. Its content may be any number of attribute-property, constructor, enumeration, and conversion declarations; its content may also include at most one content-property declaration.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for an element named *elt*.

If the class name *Class* is specified in the element-class declaration then it must be a legal Java class name; it must not be prefixed with a package name.

***Result*** The schema compiler defines a new public marshallable-element class in the target package. An instance of this class will represent a single *elt* element in an input document.

The name of the class is *Class*, if the **`class`** attribute is present; otherwise it is constructed by applying the name-mapping algorithm to *elt*. The class name must be unique with respect to the interfaces, enumeration classes, and other element classes declared at the top level of the binding schema.

The new class is a root element class if, and only if, one of the following conditions is true:

- The source schema declares, either explicitly or by default, that an instance of the element type *elt* can be the root element of a valid document. In this case the **`root`** attribute, if given, must have the value **`true`**.

- The source schema does not declare, either explicitly or by default, whether or not such an instance can be a document root, and the **`root`** attribute is given the value **`true`**.

Some source schema languages, and in particular DTDs, have no way of expressing that an instance of an element type can be a document root; for these languages only the second case above applies.

If the new class is a root element class then it extends the binding-framework class **`MarshallableRootElement`** and implements the related interface **`RootElement`**; it also defines the static **`unmarshal`** methods and the static **`newDispatcher`** method that are specified in the **`MarshallableRootElement`** class. If the new class is not a root element class then it extends the binding-framework class **`MarshallableObject`** and implements the related interface **`Element`**.

The new class defines a zero-argument public constructor; instances created by invoking this constructor will initially be invalid.

The new class defines the following public final methods:

- The **`unmarshal(Unmarshaller u)`** method implements the specification of that method in the **`MarshallableObject`** class. It uses the given unmarshaller's scanner to scan a single *elt* element, processing the element's attributes and content according to the relevant declarations in both the source and binding schemas in order to initialize the element object's properties.

  If the scanner is not initially positioned at a start tag for an *elt* element then an **`InvalidContentException`** is thrown. If the content of the unmarshalled element violates a constraint expressed in the source schema then an appropriate **`ValidationException`** is thrown.

- The **`validateThis()`** method implements the specification of that method in the **`ValidatableObject`** class. If the content of this object's properties violates a local structural constraint expressed in the source schema then an appropriate **`LocalValidationException`** is thrown.

- The **`validate(Validator v)`** method implements the specification of that method in the **`ValidatableObject`** class. If this object's children violate a structural constraint expressed in the source schema then an appropriate **`StructureValidationException`** is thrown.

- The **`marshal(Marshaller m)`** method implements the specification of that method in the **`MarshallableObject`** class. It uses the given marshaller's writer to write a single *elt* element, converting the element's property values back into attributes and content according to the relevant declarations in both the source and binding schemas.

The unmarshalling and validation methods are specified to throw various checked exceptions when various error conditions are detected. This specification does not mandate the order in which such conditions are detected, and it therefore does not mandate that any particular exception be thrown before any other. One implementation of this specification may, *e.g.*, unmarshal all of an element's content before checking local structural constraints, while another may check such constraints as content is unmarshalled. These implementations could behave differently when unmarshalling an invalid document, but they would behave identically when unmarshalling any valid document.

The new class also defines three public, non-final methods:

- The **`equals`** method tests that the given value of each property defined in this element class is equal to the given value of that property in the object to which this object is being compared. This equality test uses the value-based **`equals`** method rather than the object-identity-based **`==`** operator, and is extended to pointwise equality for collection properties.

- The **`hashCode`** method takes into account each value of primitive type, and the hash code of each object, that is the value of a property or a component of the value of a property defined in this class. The precise manner in which it does so is not specified.

- The **`toString`** method produces a string representation of the content of the object upon which it is invoked. The precise format of this string is not specified, but it should contain sufficient information to be useful in debugging.

The content- and attribute-property declarations within this element-class declaration will place further requirements upon all of the above methods. These declarations will also define public non-final methods to implement the declared properties.

The code that implements the mutation of any property's value is required to invalidate the containing element object if a mutation operation could cause any local or global structural constraint to be violated. In other words, the schema compiler must generate code that invalidates the element object whenever a property is mutated unless it can be determined, either at compile time or at run time, that no structural constraints would be violated by the property's new value.

### 6.5.1 DTD-specific semantics

If the element-type declaration for *elt* has an attribute of type **`ID`** then the derived class also implements the **`IdentifiableElement`** interface.

### 6.5.2 Example

We can begin a binding schema for the stock-trade DTD with an element-class declaration for the **`trade`** element type:

```
<element name="trade" type="class" root="true">  ...
```

This declaration defines a class with the (partial) signature:

```
public class Trade
    extends MarshallableRootElement
    implements RootElement
{
    public final void validateThis();
    public final void validate(Validator v);
    public final void unmarshal(Unmarshaller u)
        throws UnmarshalException;
    public final void marshal(Marshaller m) throws IOException;
    public static Trade unmarshal(XMLScanner xs, Dispatcher d)
        throws UnmarshalException;
    public static Trade unmarshal(XMLScanner xs)
        throws UnmarshalException;
    public static Trade unmarshal(InputStream in)
        throws UnmarshalException;
}
```

Over the course of this chapter we will gradually complete the binding schema for the stock-trade DTD.

## 6.6 General content-property declarations

The declaration of a general content property has the form

```
<element name="elt" type="class" ...>
  <content property="prop"
           [collection="{array|list}"]
           [supertype="type"]/>
```

and may appear only within an element-class declaration.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for the element *elt* named by the enclosing element-class declaration. An element class must be declared in the binding schema for each element type that is referenced in the element-type declaration's content specification.

The property name *prop* must be a valid XML name.

If the supertype *type* is specified then it must be the name of a class or interface declared in the binding schema. That type must also be a supertype of each of the element classes to which the element types referenced in the content specification are bound.

***Result*** The schema compiler defines, in the class being generated for the containing element-class declaration, a property to represent the element's content:

- The property's name is *prop*. The method identifiers used in the realization of this property must be unique among the methods defined in the containing element class.
- The property's base type is *type*, if specified; otherwise it defaults to the binding-framework class **`MarshallableObject`**.
- The property's predicate only admits values of the base type, otherwise throwing an **`InvalidContentObjectException`**.
- The property's collection type is taken to be the value of the **`collection`** attribute, if given; otherwise it is the default reference-collection type if the content specification allows more than one element; otherwise the property is a reference property.

The content-property declaration imposes further requirements upon the methods defined in the enclosing element class:

- The **`unmarshal(Unmarshaller u)`** method, after unmarshalling any attributes, unmarshals the input element's content by invoking the given unmarshaller's **`unmarshal()`** or **`unmarshal(Class)`** methods, as appropriate. It unmarshals element content until it reaches the end tag for the input element. This produces a list of marshallable-element objects, including instances of the **`PCData`** class for character data, that is the initial value of the content property.
- The **`validateThis()`** method validates the value of the content property against the local structural constraints expressed in the source schema.
- The **`validate(Validator v)`** method validates the element's children by invoking the given validator's **`validate(ValidatableObject)`** method upon each object in the content property.
- The **`marshal(Marshaller m)`** method, after marshalling any attributes and before writing the end tag, marshals the element's children by invoking the

given marshaller's **`marshal(MarshallableObject)`** method upon each of the objects in the content property, in the order in which they appear.

### 6.6.1 DTD-specific semantics

***Local structural constraints*** The content property's value must satisfy the content specification given in the corresponding element-type declaration of the source DTD.

If the element-type declaration's content specification is an element-content model then the content property's value must be a valid representation of a string of element-type names that is in the language generated by the regular expression described by the model.

An **`EMPTY`** content specification requires the content property's value, if defined, to be of length zero. Mixed and **`ANY`** content specifications do not impose any local structural constraints.

### 6.6.2 Example

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; model-based content properties, by contrast, require content specifications to have a specific structure. General content properties are therefore the most robust type with respect to schema evolution.

Given that the **`trade`** element type is declared

```
<!ELEMENT trade ( symbol, quantity, limit?, stop?, date ) >
```

we can handle its content by declaring a general content property in its element-class declaration:

```
<element name="trade" type="class" root="true">
  <content property="content"/>
```

This declaration defines three methods in the **`Trade`** class:

```
public class Trade ... {
    public List getContent();
    public void deleteContent();
    public void emptyContent();
}
```

The list returned by the **`getContent`** method will be constrained locally to contain one instance each of the **`Symbol`**, **`Quantity`**, **`Limit`**, **`Stop`**, and **`Date`** element classes, in that order, with all but the **`Limit`** and **`Stop`** instances being required.

Element classes for these other element types can be declared in a similar manner. The **`symbol`** element type, *e.g.*, is declared

```
<!ELEMENT symbol (#PCDATA) >
```

which in combination with the element-class declaration

```
<element name="symbol" type="class">
    <content property="content"/>
```

will define a **`Symbol`** class with the (partial) signature

```
public class Symbol
    extends MarshallableObject
    implements Element
{
    public MarshallableObject getContent();
    public void setContent(MarshallableObject x);
}
```

The character content of a **`symbol`** element will be represented as an instance of the binding-framework class **`PCData`**. Non-root element classes for the other **`#PCDATA`** element types **`quantity`**, **`limit`**, **`stop`**, and **`date`** can be declared in a similar manner. Binding such simple element types in this very general way is rather awkward; such types are better handled with element-value declarations, as will be seen below.

## 6.7 Conversion declarations

A conversion declaration has the form

```
<conversion name="name" [type="type"]
            [parse="parse"] [print="print"]
            [whitespace="{preserve|normalize|collapse}"]/>
```

and may appear either at the top level of the binding schema or within an element-class declaration.

***Source constraints*** The conversion *name* must be a valid XML name. It must be unique with respect to the other conversions declared at the same lexical level of the binding schema.

The *type*, if specified, is the target type of the conversion and therefore must be a legal Java type name. The target type may include a package prefix; if it does not then the target package is assumed. If the target type is not specified then it defaults to *name*, which is then further constrained to be a legal Java type name.

The *parse* method, if specified, describes how to convert a string into a value of the target type. It must be the token **`new`** or be of the form *ClassName*`.`*methodName*, where *ClassName* is a legal Java class name, possibly including a package prefix, and *methodName* is a legal Java method name.

The *print* method, if specified, describes how to convert a value of the target type back into a string. It must be of the form *methodName* or of the form *ClassName*`.`*methodName*, where *ClassName* and *methodName* are constrained as for the *parse* method.

***Result*** The schema compiler defines a *binding conversion* with the specified *name*. A binding conversion consists of a target type, a parse method, and a print method.

When a binding conversion is specified in an element-value declaration or in an attribute-property declaration then the base type of the declaration is taken to be the conversion's target type.

The parse method of a binding conversion is applied during unmarshalling in order to convert a string from the input document into a value of the target type. Before invoking the parse method, whitespace in the input string is processed according to the value of the **`whitespace`** attribute, which defaults to **`collapse`**:

- **`preserve`**: Nothing is done.
- **`normalize`**: Each occurrence of a **`\u0009`** (TAB), **`\u000A`** (LINE FEED), or **`\u000D`** (CARRIAGE RETURN) character is replaced by a space character (**`\u0020`**, SPACE).
- **`collapse`**: The processing of the previous step is performed, after which contiguous sequences of spaces are collapsed to single spaces and leading and trailing spaces are removed.

Normalizing or collapsing whitespace may cause round-tripping to fail if strict, character-for-character document equivalence is required, but collapsing whitespace is almost always acceptable in data-oriented applications of XML.

After whitespace processing the parse method is invoked as follows:

- If the parse method is specified as **`new`**, or if it is not specified, then the compiler assumes that the target type is a class that defines a constructor that takes a single **`String`** argument. To apply the conversion to a string it generates code that invokes this constructor, passing it the input string.
- If the parse method is specified in the form *ClassName*`.`*methodName* then the compiler assumes that the class *ClassName* exists and that it defines a static method named *methodName* that takes a single **`String`** argument and returns a value of the target type. To apply the conversion to a string it generates code that invokes this method, passing it the input string.

The print method of a binding conversion is applied during marshalling in order to convert a value of the target type into a string that will be written to the output document:

- If the print method is not specified then it defaults to **`toString`** and is handled according to the next case.

- If the print method is specified in the form *methodName* then the compiler assumes 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 generates code to invoke this method upon an instance of the target type.

- If the print method is specified in the form *ClassName*`.`*methodName* then the compiler assumes that the class *ClassName* exists and that it defines a static method named *methodName* that takes a single argument of the target type and returns a **`String`**. To apply the conversion to a string it generates code that invokes this method, passing it a value of the target type.

The parse and print methods together define a bidirectional mapping between values of the target type and a set of strings. This mapping should always be value-preserving with respect to the target type. In other words, the print method should be able to convert every value $x$ of the target type into a string representation that, when processed by the parse method, returns the original value $x$. This mapping will often, though not always, be value-preserving with respect to the set of strings upon which it is defined; a particular floating-point number, *e.g.*, might be represented by more than one string.

The print method is assumed to work on every value of the target type and therefore never to fail. A parse method may fail, however, since it is only defined on those strings that are valid representations of target-type values and it will be applied to arbitrary strings. A parse method indicates failure by throwing an exception of whatever type is appropriate, though it should never throw a **`TypeConstraintException`**. The schema compiler ensures that an exception thrown by a parse method is caught and wrapped within a new **`ConversionException`** that is then immediately thrown. For the rare case in which a print method fails, it similarly ensures that an exception thrown by a print method is caught and wrapped within a new **`ConversionFailureException`** that is then immediately thrown.

***Built-in conversions*** A built-in binding conversion is defined for the following Java primitive types. In each case the name of the conversion is the name of the type.

- For **`boolean`**, the built-in conversion maps the string **`"true"`** to **`true`** and the string **`"false"`** to **`false`**; for all other input strings it is undefined.

- For each of the numeric types **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, and **`double`**, the built-in conversion uses the static **`parse`***Type***`(String)`** method of the corresponding **`java.lang`** wrapper class to convert strings to values, and the static **`toString(`***type***`)`** method of the same class to convert values to strings.

***Scope*** Conversion declarations nest lexically within the binding schema. If a conversion declaration appears at the top level of the binding schema and has the same name as a built-in conversion then the declaration takes precedence. If a conversion declaration appears within an element-class declaration and has the same name as a conversion declared at the top level, or the same name as a built-in conversion, then the innermost declaration takes precedence.

### 6.7.1 Example

The **`stop`** and **`limit`** element types declared in the stock-trade DTD are meant to contain prices expressed as decimal fractions of dollars. It would not be a good idea to use the built-in conversions for the **`float`** or **`double`** primitive numeric types, since these types are inherently approximate. Hence we define a new **`price`** conversion that represents prices as instances of the class **`java.math.BigDecimal`**, which can represent decimal fractions precisely:

```
<conversion name="price" type="java.math.BigDecimal"/>
```

With this conversion, a string will be converted into a **`BigDecimal`** instance during unmarshalling by invoking the **`BigDecimal(String)`** constructor; such an instance will be converted back into a string during marshalling by invoking the **`toString`** method of that class. This conversion will be used later when we give element-value declarations for the **`stop`** and **`limit`** element types.

If we define the auxiliary class

```
public class Cnv {
    private static java.text.SimpleDateFormat df
        = new java.text.SimpleDateFormat("yyyy-MM-dd");
    public static java.util.Date parseDate(String s)
        throws java.text.ParseException
    {
        return df.parse(s);
    }
    public static String printDate(java.util.Date d) {
        return df.format(d);
    }
}
```

then we can similarly define a **`date`** conversion that converts strings representing dates into instances of the class **`java.util.Date`**:

```
<conversion name="date" type="java.util.Date"
            parse="Cnv.parseDate" print="Cnv.printDate"/>
```

The **`parseDate`** method may throw a **`ParseException`**; if this happens then the unmarshalling code will catch the exception, wrap it in a **`ConversionException`**, and then throw this new exception.

## 6.8 Datatypes

A source schema associates a *datatype* with every attribute declaration and with every declaration of a simple element type, *i.e.*, an element type whose content specification allows character but not element content. The datatype may be specified explicitly in the source schema or it may be implied by the semantics of the source schema language.

A datatype is either *atomic* or *compound*. An atomic datatype describes values that are not subject to further decomposition; *e.g.*, strings, floating-point numbers in a certain range, or tokens in a certain set. A compound datatype describes values that are sequences of atomic values; *e.g.*, arrays of floating-point numbers or lists of tokens in a certain set. A source-schema component with a compound datatype can only be bound to a collection property.

Whether atomic or compound, a source-schema datatype has three components:

- A *representation type*, which is the type used in generated code to represent atomic values of the datatype.
- An optional *datatype conversion*, which is much like a binding conversion in that it consists of a target type, a parse method, and a print method. The target type is the datatype's representation type. The parse method indicates failure by throwing a **`TypeConstraintException`**.
- A *predicate*, which captures the type constraints of the datatype that are not implied by the representation type and so must be checked dynamically.

***Datatypes vs. binding conversions*** Datatype conversions are distinct from binding conversions.

Datatype conversions are associated with specific datatypes that are either defined in the source schema or implicit in the source schema language itself. If the parse method of such a conversion fails during unmarshalling then the input document is invalid, which is why it is required to throw a **`TypeConstraintException`**.

Binding conversions, by contrast, are declared and named in the binding schema, and are provided as a means of augmenting the source schema with additional application-specific conversions. A binding conversion's parse method is simply a convenience; as such its failure during unmarshalling does not imply that the input document is invalid, and so it is not permitted to throw a **`TypeConstraintException`**.

If a source-schema component declares a datatype, and that datatype defines a conversion, then a binding conversion may not be applied to that component.

### 6.8.1 DTD-specific semantics

The datatyping facilities of DTDs are very weak. The content of instances of simple **`#PCDATA`**-only element types cannot be constrained at all. Attribute values can be constrained in a few ways, for example to be tokens in a certain set, or lists of tokens, but these are still inadequate for typical data-oriented applications of XML. Many such applications have therefore adopted the "DT4DTD" datatype-annotation conventions defined in the W3C note *Datatypes for DTDs* [*http://www.w3.org/TR/dt4dtd*]. This specification supports those conventions.

> ***Note*** The precise set of datatypes to be supported via these conventions has yet to be determined.

## 6.9 Element-value declarations

An element-value declaration has the form

```
<element name="elt" type="value" [convert="cnv"]/>
```

and may appear only at the top level of the binding schema.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for the element type *elt*. The element type must be *simple*, *i.e.*, its content specification must allow character content but no element content (*e.g.*, **`#PCDATA`** or its equivalent), and it must not have any attributes, except for special attributes used only to convey datatype information.

If a binding-conversion name *cnv* is specified then it must name a conversion defined at the top level of the binding schema or defined as a built-in conversion. A binding conversion may not be specified if the element type's datatype defines a datatype conversion.

***Result*** The schema compiler generates code that treats an *elt* element as a value representing the element's content rather than as an instance of an element class. The value's type is the target type of the binding conversion, if any, otherwise it is the representation type of the element type's datatype.

An element value is bound to a property by an element-reference content-property declaration, defined below in §6.11. An element-value declaration imposes the following requirements upon the methods defined in the containing element class:

- The **`unmarshal(Unmarshaller u)`** method unmarshals an *elt* element in the input document directly, by scanning its start tag, its character data, and its end tag. If the element type's datatype is compound then the character data is broken into non-whitespace tokens. If the datatype defines a conversion, or if a binding conversion is specified, then that conversion is applied to the character data or to each token, as appropriate. The result is taken to be the value of the element.

If the scanner is not initially positioned at a start tag for an *elt* element then an **`InvalidContentException`** is thrown. If an attribute is scanned then an **`InvalidAttributeException`** is thrown. If the content of the unmarshalled element violates a type constraint expressed in the source schema then a **`TypeConstraintException`** is thrown; if the content violates any other constraint expressed in the source schema then an appropriate **`ValidationException`** is thrown.

- The **`validateThis()`** method checks any local structural constraints that are expressed in the source schema.
- The **`marshal(Marshaller m)`** method marshals an *elt* element directly, by writing its start tag, its character content, and its end tag. If a binding or datatype conversion is defined then it is first applied to the element's value in order to compute the characters to be written. If the datatype is compound then the property's values are converted individually, if necessary, and then written with adjacent values separated by single space characters.

### 6.9.1 DTD-specific semantics

***Datatypes and type constraints*** If the source DTD declares an attribute **`e-dtype`** for the element type *elt* then it is interpreted in accordance with the DT4DTD conventions. The attribute's type must be **`CDATA`** and its default declaration must be **`#FIXED`**. The element's datatype is named by the default value of the attribute.

If no datatype is declared then the element type's datatype is taken to be an atomic datatype with the representation type **`java.lang.String`**, no conversion, and a predicate that admits any value.

### 6.9.2 Example

In §6.6.2 we saw that we could declare element classes for the **`symbol`**, **`quantity`**, **`limit`**, **`stop`**, and **`date`** element types and treat the content of a **`trade`** element as a general collection property. This is more convenient than using the SAX or DOM APIs, but it is still quite awkward. We can instead declare that instances of these element types should be treated as values rather than as objects, and even convert these values to more appropriate types than **`String`**:

```
<element name="symbol" type="value"/>
<element name="quantity" type="value" convert="int"/>
<element name="limit" type="value" convert="price"/>
<element name="stop" type="value" convert="price"/>
<element name="date" type="value" convert="date"/>
```

These declarations will be used below when we give a model-based content-property declaration for the **`trade`** element type's content specification.

## 6.10 Model-based content-property declarations

A model-based content-property declaration has the form

```
<element name="elt" type="class">
  <content>
    ...
    <element-ref name="ref-elt" [property="prop"] [collection="coll"]/>
    <choice property="prop" [collection="coll"] [supertype="type"]/>
    <sequence property="prop" [collection="coll"] [supertype="type"]/>
    ...
    <rest property="prop" [collection="coll"] [supertype="type"]/>
```

and may appear only within an element-class declaration. Its content may be any number of element-reference-, choice-, or sequence-property declarations, in any order, possibly ending with a single rest-property declaration. These subsidiary declarations serve to bind the components of the content specification's content model to specific properties, hence they are together referred to as the *property model* of the containing content-property declaration.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for the element named by the enclosing element-class declaration. The content specification of this element-type declaration must be an element-content model, or its equivalent; *i.e.*, it must be a description of the sequences of elements that are valid as the content of an *elt* element.

The property model in the binding schema must *cover* the content model in the source schema. This constraint ensures that every component of the content model is related to a specific declaration in the property model, and that each declaration binds zero or more components to a content property.

To define this relation more precisely we assume that a content model can be viewed, abstractly, as a tree of *content particles* consisting of element-type references, choice groups of particles, and sequence groups of particles. Each particle carries a *repeat specification* whose syntax and semantics are defined by the source schema language. If the root of the tree is not a sequence group then we enclose it in a sequence group with a repeat specification that requires exactly one occurrence and then take that group to be the root. The covering relation is then defined as follows:

- An element-reference content-property declaration covers an element-type reference in the content model.
- A choice declaration covers a choice group in the content model.
- A sequence declaration covers a sequence group in the content model.
- A property model covers a content model if, and only if,

- The repeat specification of the root sequence group requires exactly one occurrence, and
- There is an initial subsequence of the root sequence group such that each declaration in the property model covers the corresponding child of the subsequence.

If the property model ends with a rest-property declaration then the remaining components of the root sequence group, if any, are considered to be covered by the rest-property declaration. If a rest-property declaration is not given then any remaining components must be element-type references; they are covered by default element-reference property declarations as described below in §6.14.

***Example*** The content model of the DTD element-type declaration

```
<!ELEMENT x ( a, b*, ( c | d )+, ( e, f )+, g, h ) >
```

is covered by each of the following property models:

```
<content>
  <element-ref name="a"/>
  <element-ref name="b"/>
  <rest property="content"/>

<content>
  <element-ref name="a"/>
  <element-ref name="b"/>
  <choice property="c-or-d"/>
  <sequence property="e-and-f"/>
  <element-ref name="g"/>
  <element-ref name="h"/>
  <rest property="extras"/>

<content>
  <element-ref name="a"/>
  <element-ref name="b"/>
  <choice property="c-or-d"/>
  <sequence property="e-and-f"/>
```

In the first case the rest-property declaration covers all of the content model except for `a` and `b*`, while in the second case it covers no content-model components; these are discussed further in §6.12. The last case is interpreted as though it ends with the declarations `<element-ref name="g"/>` and `<element-ref name="h"/>`, which are appended as defaults as explained in §6.14.

***Result*** The schema compiler defines, in the class being generated for the containing element-class declaration, the set of properties specified by the declarations in the property model. The meanings of these declarations are defined in the next two sections of this chapter.

The content-property declaration imposes exactly those requirements upon the methods defined in the enclosing element class that are imposed by the subsidiary element-reference-, choice-, sequence-, and rest-property declarations.

Examples of the various kinds of property-model declarations are given later in this chapter, after the requisite constructs are defined.

### 6.10.1 DTD-specific semantics

The element-content model in an element-type declaration is *flattened* as it is read by the compiler: Each choice or sequence group without an explicit repeat specification is replaced by its content. Thus the element-type declaration

```
<!ELEMENT y ( a, b, ( c, d, e ), (f | g | (h | i)) ) >
```

is effectively simplified to:

```
<!ELEMENT y ( a, b, c, d, e, (f | g | h | i) ) >
```

This removes the redundant choice and sequence groups that often arise when parameter entities are used to build up complex content models. The structure of a model-based content-property declaration must match that of the flattened form of the content model; *i.e.*, it must not contain a choice- or sequence-property declaration that attempts to bind a choice or sequence group that has no explicit repeat specification.

## 6.11 Element-reference content-property declarations

This type of declaration has the form

```
<element name="elt" type="class">
  <content>
    ...
    <element-ref name="ref-elt" [property="prop"] [collection="coll"]/>
    ...
```

and may appear only within a model-based content-property declaration.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for the element *elt* named by the enclosing element-class declaration. The element-reference property declaration must cover a reference in the content specification of that element-type declaration to an element type named *ref-elt*. The reference may have any repeat specification that is valid in the source schema language, with one exception: If the element type *ref-elt* has a compound datatype then the repeat specification must not allow more than one occurrence.

***Result*** The schema compiler defines, in the class being generated for the enclosing element-class declaration, a property to represent the content that is matched by the element-type reference. The name of the property is *prop*, if specified; otherwise it is taken to be *ref-elt*. The method identifiers used in the realization of this property must be unique among the methods defined in the containing element class.

Beyond this, the meaning of the declaration depends upon whether the element type *ref-elt* has an element-class or an element-value declaration in the binding schema.

### 6.11.1 Element classes

If an element class $C$ is declared in the binding schema for the referenced element type *ref-elt* then the resulting property is a special case of a general-content property:

- The property's base type is $C$.
- The property's predicate only accepts instances of $C$, otherwise throwing an `InvalidContentObjectException`.
- The property's collection type is taken to be the value of the `collection` attribute, if given; otherwise it is the default reference-collection type if the repeat specification of the element-type reference allows more than one element; otherwise the property is a reference property.

The further requirements imposed upon methods defined in the enclosing element class, and the type and local constraints that must be enforced by those methods, are the same as for a general-content property.

### 6.11.2 Element values

If the referenced element type *ref-elt* instead has an element-value declaration in the binding schema then the resulting property is defined as follows:

- The property's base type is the target type of the binding conversion, if any; otherwise it is the representation type of the element type's datatype.
- The property's predicate enforces any type constraints that are expressed in the source schema; in particular, this predicate incorporates the predicate of the element type's datatype.
- The property's collection type is taken to be the value of the `collection` attribute, if present. If the base type is a primitive type then only the `array` collection type may be specified.

  If no collection type is specified then the property's collection type is a function of the base type, of the element type's datatype, and of the repeat specification of the element-type reference:

– If the repeat specification allows at most one occurrence and the datatype is atomic then the property is a primitive property, if the base type is a primitive type, otherwise it is a reference property.

– If the repeat specification allows more than one occurrence, or if the datatype is compound, then the property is an array property, if the base type is a primitive type, otherwise it is of the default reference-collection type.

The further requirements imposed upon methods defined in the enclosing element class, and the type and local constraints that must be enforced by those methods, are those induced by the element-value declaration.

### 6.11.3 DTD-specific semantics

***Local structural constraints*** The element-type reference in the source DTD that is covered by the element-reference property declaration may have a repeat specification of `?`, `*`, or `+`, which should be interpreted as required by the XML 1.0 specification.

### 6.11.4 Example

We can now give a model-based content-property declaration for the `trade` element type's content model:

```
<element name="trade" type="class" root="true">
  <content>
    <element-ref name="symbol"/>
    <element-ref name="quantity"/>
    <element-ref name="limit" property="limit-price"/>
    <element-ref name="stop" property="stop-price"/>
    <element-ref name="date"/>
```

In combination with the element-value declarations given earlier, these declarations will define the following property-access methods in the `Trade` class:

```
public class Trade ... {
    public String getSymbol();
    public void setSymbol(String x);
    public int getQuantity();
    public void setQuantity(int x);
    public boolean hasQuantity();
    public void deleteQuantity();
    public java.math.BigDecimal getLimitPrice();
    public void setLimitPrice(java.math.BigDecimal x);
    public java.math.BigDecimal getStopPrice();
```

```
        public void setStopPrice(java.math.BigDecimal x);
        public java.util.Date getDate();
        public void setDate(java.util.Date x);
        ...
    }
```

The **`symbol`**, **`quantity`**, and **`date`** elements are required by the **`trade`** element type's content model, hence the validation logic in the **`Trade`** class will ensure that the corresponding properties are given values.

## 6.12 Choice, sequence, and rest content-property declarations

These types of declarations have the form

```
<element name="elt" type="class">
  <content>
    ...
    <choice property="prop" [collection="coll"] [supertype="type"]/>
    <sequence property="prop" [collection="coll"] [supertype="type"]/>
    ...
    <rest property="prop" [collection="coll"] [supertype="type"]/>
```

and may appear only within a model-based content-property declaration.

***Source constraints*** The source schema must contain an element-type declaration, or its equivalent, for the element *elt* named by the enclosing element-class declaration.

A choice- or sequence-property declaration must cover a corresponding component of the element-type declaration's content model. A rest-property declaration is considered to cover those components, if any, of the content model's root sequence group that are not otherwise covered.

If a choice-, sequence-, or rest-property declaration specifies a supertype *type* then that type must be a class or interface declared in the binding schema. That type must also be a supertype of each of the element classes to which the element types referenced by the covered component(s) of the content model are bound.

***Result*** For each instance of one of these three types of property declarations the schema compiler defines, in the class being generated for the containing element-class declaration, a property to represent the content matched by the covered content-model component(s).

Each such property is defined exactly as if it were a general-content property, with the following exceptions:

- The property's content is constrained by the covered component(s) rather than the entire content specification.
- A rest-property declaration always defines a collection property, either by specifying the collection type explicitly or by using the default reference-collection type.
- If a rest-property declaration does not cover any components, as in the second example on p. 50, then the property's predicate accepts any value of the base type but the property is given a local structural constraint that permits no values.

The further requirements imposed upon methods defined in the enclosing element class, and the type and local constraints that must be enforced by those methods, are otherwise the same as for a general-content property.

A content-property declaration of the form

```
<content>
  <rest property="prop">
```

is in fact equivalent to the general content-property declaration

```
<content property="prop" collection="list"/>
```

A complete example of a rest-property declaration is given below, in §6.14.1.

## 6.13 Attribute-property declarations

An attribute-property declaration has the form

```
<attribute name="attr" [convert="cnv"]
           [property="prop"] [collection="{array|list}"]/>
```

and may appear only within an element-class declaration.

***Source constraints*** The source schema must contain an attribute declaration, or its equivalent, for an attribute named *attr* on the element type named by the enclosing element-class declaration.

If a binding-conversion name *cnv* is specified then it must name a conversion defined within the enclosing element-class declaration, defined at the top level of the binding schema, or defined as a built-in conversion. A binding conversion may not be specified if the attribute' datatype defines a datatype conversion, or if it is otherwise forbidden below.

The property name *prop*, if given, must be a valid XML name.

***Result*** The schema compiler defines, in the class being generated for the containing element-class declaration, a property to represent the value of the named attribute:

- The property's name is *prop*, if specified; otherwise it is taken to be *attr*. The method identifiers used in the realization of this property must be unique among the methods defined in the containing element class.

- The property's base type is the target type of the binding conversion, if any; otherwise it is the representation type of the attribute's datatype.

- The property's predicate enforces any type constraints that are expressed in the source schema; in particular, this predicate incorporates the predicate of the element type's datatype.

- The property's collection type is taken to be the value of the **`collection`** attribute, if present. If the base type is a primitive type then only the **`array`** collection type may be specified.

  If no collection type is specified then the property's collection type is a function of the base type and of the attribute's datatype:

  – If the attribute's datatype is atomic (*e.g.*, **`CDATA`** or **`NMTOKEN`**) then the property is a primitive property, if the base type is a primitive type, otherwise it is a reference property.

  – If the attribute's datatype is compound (*e.g.*, **`IDREFS`** or **`NMTOKENS`**) then the property is an array property, if the base type is a primitive type, otherwise it is a collection property of the default reference-collection type.

The attribute-property declaration imposes further requirements upon the methods defined in the enclosing element class:

- The **`unmarshal(Unmarshaller u)`** method unmarshals an attribute by reading its character data. If the attribute's datatype is compound then the character data is broken into non-whitespace tokens. If the datatype defines a conversion, or if a binding conversion is specified, then that conversion is applied to the character data or to each token, as appropriate. The result is taken to be the value of the attribute property.

  If the attribute's value violates a type constraint expressed in the source schema then a **`TypeConstraintException`** is thrown; if its value violates any other constraint expressed in the source schema then an appropriate **`ValidationException`** is thrown.

- The **`validateThis()`** method validates the value of the attribute property against the local structural constraints expressed in the source schema.

- The **`marshal(Marshaller m)`** method marshals an attribute by writing its name and its value. If a binding or datatype conversion is specified then it is first applied to the components of the property's value in order to compute the characters to be written. If the datatype is compound then the property's values are converted individually, if necessary, and then written with adjacent values separated by single space characters.

### 6.13.1 DTD-specific semantics

If an attribute declaration specifies a default value then the property's default value is computed as if by unmarshalling the declaration's default value.

DTDs do not permit duplicate attributes, so if two attributes with the same name are scanned then a **`DuplicateAttributeException`** is thrown.

Attribute properties may not be declared for attributes named **`e-dtype`** and **`a-dtype`**, which are interpreted according to the DT4DTD conventions.

Attributes of type **`ID`**, **`IDREF`**, and **`IDREFS`** are given special treatment beyond the computation of default values and the enforcement of constraints:

- An attribute of type **`ID`** is mapped to a string property as specified below. An additional final string method, **`id()`**, is defined that also returns the property's value, thereby completing the implementation of the **`IdentifiableElement`** interface by the enclosing element class. This method is invoked by the binding framework during the unmarshalling and validation processes.

- An attribute of type **`IDREF`** or **`IDREFS`** is mapped to a property with the base type **`IdentifiableElement`**. The unmarshalling process resolves identifier references in the input document into the identifiable element objects that represent the referent elements.

Binding conversions may not be applied to attributes of type **`IDREF`** and **`IDREFS`**.

***Datatypes and type constraints*** Attribute types in the source DTD are mapped to datatypes with representation types, predicates, and predicate-failure exceptions as follows:

- The **`CDATA`** attribute type maps to an atomic datatype with the representation type **`String`** and a predicate that accepts any value, unless a more specific datatype is specified via the DT4DTD conventions. In the latter case the attribute's element type must declare a **`#FIXED CDATA`** attribute **`a-dtype`** whose default value is a sequence of pairs of attribute names and datatype names. This attribute's datatype is named by the second component of the pair whose first component is this attribute's name.

- The **`ID`** attribute type maps to an atomic datatype with the representation type **`String`** and a predicate that requires its argument to be a string that is a valid XML name, throwing an **`InvalidNameException`** upon failure.

- The **`IDREF`** and **`IDREFS`** types map to atomic and compound datatypes, respectively, with the representation type **`IdentifiableElement`** and a predicate that accepts any value.

- The **`NMTOKEN`** and **`NMTOKENS`** types map atomic and compound datatypes, respectively, with the representation type **`String`** and a predicate that requires its argument to be a string that is a valid XML name token, throwing an **`InvalidNameTokenException`** upon failure.

- An enumeration attribute type maps to an atomic datatype with the representation type **`String`** and a predicate that requires its argument to be a string that is a member of the list of XML name tokens given in the attribute-type declaration, throwing an **`InvalidEnumerationValueException`** upon failure.

If the attribute declaration's default component is **`#FIXED`** then the attribute property's predicate further requires its argument to be the specified fixed value, throwing a **`FixedValueException`** upon failure.

***Local structural constraints*** An attribute declaration in the source DTD may impose the following kinds of local structural constraints upon an instance of the enclosing element class:

- If the attribute declaration's type is either **`NMTOKENS`** or **`IDREFS`** then if the property has a value its length must be nonzero. If this constraint does not hold then an **`EmptyAttributeValueException`** is thrown.

- If the attribute declaration's default component is **`#REQUIRED`** then the property must have a given value. It may be an empty value, *e.g.*, the empty string, but it must be present. If this constraint does not hold then a **`MissingAttributeException`** is thrown.

Attributes that are not explicitly declared in the source DTD must not appear in input elements. If this constraint does not hold then an **`InvalidAttributeException`** is thrown during unmarshalling.

***Global structural constraints*** An attribute declaration in the source DTD may impose the following kinds of global structural constraints upon an input document, during unmarshalling, or upon a content tree, during validation:

- If the attribute declaration's type is **`ID`** then the property's value, if any, must be unique in the input document or in the content tree. If this constraint is violated then a **`DuplicateIdentifierException`** is thrown. This constraint is automatically enforced by the binding framework during the unmarshalling and validation processes.

- If the attribute declaration's type is **`IDREF`** or **`IDREFS`** then during unmarshalling the attribute's value(s), if any, in the input document must be the identifier(s) of elements in that document. If this constraint is violated then an **`UndefinedIdentifierException`** is thrown.

  This constraint is enforced by the binding framework during unmarshalling by a mechanism that also serves to resolve identifier references into **`IdentifiableElement`** objects in the resulting content tree. For this to work requires that the **`unmarshal`** method of the enclosing element class register each referenced identifier by invoking the **`reference(String, Validator.Patcher)`** method of the given umarshaller's validator. The patcher object passed to this method defines a **`patch`** method that takes a single **`IdentifiableElement`** object. The validator will invoke the patcher, passing the resolved referent, some time before the unmarshalling process is complete. The patcher should, when invoked, save the referent as the value of the original reference. An example of how this is done is shown in appendix D.2.

- If the attribute declaration's type is **`IDREF`** or **`IDREFS`** then the property's values, if any, are one or more marshallable-element objects that implement the **`IdentifiableElement`** interface.

  The validation process must ensure that each such object has an identifier, *i.e.*, that the object's **`id`** method returns a non-**`null`** value. If this constraint is violated then a **`MissingAttributeException`** is thrown.

  The validation process must also ensure that each such object is a member of the same content tree as the object containing this property. If this constraint is violated then an **`ExternalReferenceException`** is thrown.

  Both of these constraints are enforced by the binding framework during validation so long as the **`validate`** method of the enclosing element class invokes the given validator's **`reference(IdentifiableElement)`** method upon each referenced object.

### 6.13.2 Example

We can finally complete the element-class declaration for the **trade** element type by adding attribute-property declarations. The attributes of the **trade** element type are declared:

```
<!ATTLIST trade
          account CDATA #REQUIRED
          action ( buy | buy-to-cover | sell | sell-short )
                 #REQUIRED
          duration ( immediate | day | good-til-canceled )
                   "day" >
```

The corresponding attribute-property declarations

```
<element name="trade" type="class" root="true">
  <content>
    <element-ref name="symbol"/>
    ...
  <attribute name="account" convert="int"/>
  <attribute name="action"/>
  <attribute name="duration"/>
```

define the following methods in the **Trade** class:

```
public class Trade ... {
    public int getAccount();
    public void setAccount(int x);
    public boolean hasAccount();
    public void deleteAccount();
    public String getAction();
    public void setAction(String x);
    public String getDuration();
    public void setDuration(String x);
    ...
}
```

The value of an **account** attribute will be converted to an integer value by the built-in **int** conversion. The **setAction** and **setDuration** methods will only permit their arguments to have the values specified in the DTD.

## 6.14 Default binding declarations

Having defined element-class and -value declarations, content-property declarations, and attribute-property declarations, we can now define the default binding declarations assumed by the schema compiler when a construct in the source schema does not have a corresponding declaration in the binding schema.

Default binding declarations often yield a valid binding schema, but this is not guaranteed. Every default declaration is subject to all of the constraints that apply to a non-default declaration of the same type. It is the responsibility of the binding-schema author to correct the binding schema so that the explicit and default declarations, taken together, are valid.

***Element-type references in property models*** Suppose that a model-based content-property declaration does not end with a rest-property declaration and does not cover the entire content model of the corresponding element-type declaration. If the remaining components of the content model are references to the element types $elt_1, elt_2, \ldots$, then an element-reference property declaration is assumed at the end of the property model for each of the $elt_i$, in the order in which they appear in the content model:

```
<element name="elt" ...>
  <content>
    ...
    <element-ref name="elt1"/>
    <element-ref name="elt2"/>
    ...
```

***Content properties*** Suppose that an element-class declaration, whether specified explicitly or by default, does not contain a content-property declaration. If the corresponding element type's content specification is a sequence group with a repeat specification that requires exactly one value, and if that group consists of one or more element-type references, then the model-based content-property declaration

```
<element name="elt" type="class">
  ...
  <content/>
  ...
```

is assumed, and subsidiary element-reference property declarations are assumed as above.

Otherwise, if the element type's content specification is non-empty, *i.e.*, it allows, but does not necessarily require, at least some character or element content, then the general content-property declaration

```
<element name="elt" type="class">
  ...
  <content property="content"/>
  ...
```

is assumed.

***Element classes and values*** Suppose that an element type *elt* is declared in the source schema but there is neither an element-class declaration nor an element-value declaration for *elt* in the binding schema. If *elt* is simple, *i.e.*, if its content specification allows character content but no element content, and if it is referenced either explicitly or implicitly by at least one model-based content property, then the element-value declaration

```
<element name="elt" type="value"/>
```

is assumed. Otherwise, the element-class declaration

```
<element name="elt" type="class"/>
```

is assumed.

***Attributes*** Suppose that an element-class declaration, whether specified explicitly or by default, does not contain an attribute-property declaration for an attribute *attr* that is declared in the source schema. In this case the following attribute-content property is assumed:

```
<element name="elt" type="class">
  ...
  <attribute name="attr"/>
```

### 6.14.1 Example: Rest properties

In combination with default element-class declarations, rest-content properties provide a very general way to handle the evolution of source schemas. Suppose that we append a rest-content property to the property model of the `trade` element-class declaration:

```
<element name="trade" type="class" root="true">
  <content>
    ...
    <element-ref name="date"/>
    <rest property="rest"/>
```

This will define a list property in the `Trade` class:

```
public class Trade ...{
    ...
    public List getRest();
    public void deleteRest();
    public void emptyRest();
    ...
}
```

With the stock-trade DTD as it stands, this property will be constrained locally to be either absent or empty. The DTD can later be evolved by extending the content model of the `trade` element type. So long as the original content model is a prefix of the new model, the binding schema will continue to be valid. Recompiling the new DTD with the original binding schema will generate a `Trade` class with exactly the same signature, but the **`rest`** property will be allowed to contain element objects in accordance with the new components of the **`trade`** element type's content model. If the new components of the content model refer to element types for which element classes have not been declared then default element-class declarations will be assumed as described above.

### 6.14.2 Example: Default element-reference properties

Another way to handle the extension of content models over time is to take advantage of the default insertion of element-reference declarations into property models when a rest-property declaration is not provided. Suppose, *e.g.*, that the stock-trade DTD is extended with **`filled`**, **`price`**, and **`exchange`** element types declared

```
<!ELEMENT filled ( date, price, exchange ) >
<!ELEMENT price ( #PCDATA ) >
<!ELEMENT exchange ( #PCDATA ) >
```

and that the content model of the **`trade`** element type is extended to

```
<!ELEMENT trade
          ( symbol, quantity, limit?, stop?, date, filled? ) >
```

so that a **`filled`** element records the date on which a trade is executed, the actual price at which the shares were traded, and the name of the exchange upon which they were traded.

If the source DTD is changed in this way then the binding schema need not change at all, since the default element-reference declaration

```
<element name="trade" type="class" root="true">
  <content>
    ...
    <element-ref name="date"/>
    <element-ref name="filled"/>
```

will be assumed in the property model of the **`trade`** element-class declaration and the default element-class and -value declarations

```
<element name="filled" type="class">
  <content/>
<element name="price" type="value"/>
<element name="exchange" type="value"/>
```

will be assumed for the new **`filled`**, **`price`**, and **`exchange`** element types. Recompiling the new DTD with the original binding schema will generate a **`Trade`** class that contains two additional methods:

```
public class Trade ...{
    ...
    public Filled getFilled();
```

```
        public void setFilled(Filled x);
        ...
    }
```

The new binding will also define an element class for the **filled** element type:

```
    public class Filled
        extends MarshallableObject
        implements Element
    {
        ...
        public java.util.Date getDate();
        public void setDate(java.util.Date x);
        public String getPrice();
        public void setPrice(String x);
        public String getExchange();
        public void setExchange(String x);
        ...
    }
```

This new class uses the existing conversion for the **date** element type but treats the new **price** and **exchange** elements as simple strings.

### 6.14.3 Example: Compact binding schemas

Default binding declarations make it possible to write very short binding schemas. The minimal complete binding schema for the stock-trade DTD, *e.g.*, is

```
    <xml-java-binding-schema>
      <element name="trade" type="class" root="true"/>
```

This much is required only because DTDs do not provide a means for identifying root element types; the minimal binding schema for a source schema that identifies root element types is completely empty.

The quality of the binding defined by the minimal binding schema depends upon the amount of datatype information in the source schema. For a plain DTD such as the stock-trade example it produces a simple but usable binding that has no conversions but at least binds simple element types directly to string properties and, if nothing else, can serve as a guide for writing a more complete binding schema by overriding the default binding declarations. For a source schema that specifies datatypes the binding defined by the minimal binding schema is likely to be more directly usable, though an explicit binding schema will still be required to specify additional conversions or to define better bindings of complex content models.

## 6.15 Enumeration declarations

An enumeration declaration has the form

**<enumeration name="***name***" members="**$m_1 \ m_2 \ \ldots \ m_n$**"/>**

and may appear either at the top level of the binding schema or within an element-class declaration.

***Source constraints*** The enumeration *name* must be a valid Java class name; it must not have a package prefix. It must be unique with respect to the names of other enumerations, and of all binding conversions, defined at the same lexical level of the binding schema. If the enumeration is declared at the top level of the binding schema then *name* must also be unique with respect to the interfaces and element classes declared at that level.

Each member name $m_i$ must be a valid XML name.

***Result*** The schema compiler defines a public final *enumeration class* of the specified name. If the declaration appears at the top level of the binding schema then the class is an ordinary class; if the declaration appears within an element-class declaration then the new class is a static inner class of the containing element class.

The enumeration class defines a set of named values together with a bijective mapping between strings and those values. Each value is represented by a unique instance of the class and is named by a static constant defined in the class. An enumeration class *Enum* therefore contains one public, static, and final constant of type *Enum* for each enumeration member $m_i$. The name of each constant is computed by applying the name-mapping algorithm to the corresponding member name. The resulting collection of constant names must not contain any duplicates.

The bijective mapping defined by an enumeration is implemented by the **parse** and **toString** methods; an enumeration class also defines appropriate **equals** and **hashCode** methods:

```
public static Enum parse(String str);
public String toString();
public boolean equals(Object ob);
public int hashCode();
```

- The static **parse** method attempts to map its string argument to one of the enumeration's values. If a match is found then the appropriate value object is returned; otherwise, an **IllegalEnumerationValueException** is thrown.

- The **toString** method returns the string associated with the instance upon which it is invoked.

- The **equals** method returns **true** if, and only if, its argument is the object upon which it is invoked.

- The **hashCode** method returns the identity hash code of the object upon which it is invoked.

An implicit binding conversion is defined for every enumeration. The conversion's name is the enumeration's name, its parse method is the static **`parse`** method of the enumeration class, and its print method is the **`toString`** instance method of the enumeration class.

***Scope*** Enumeration declarations nest lexically within the binding schema. If an enumeration declaration appears within an element-class declaration and has the same name as an enumeration declared at the top level then the innermost declaration takes precedence.

### 6.15.1 DTD-specific semantics

If the implicit binding conversion defined by an enumeration class is specified in an attribute-property declaration for an enumeration attribute then the set of member names specified in the enumeration declaration must be identical to the set of name tokens specified in the attribute declaration.

### 6.15.2 Example

Enumeration classes, while perhaps unfamiliar to the average programmer, leverage the Java programming language's type system to statically enforce the constraint that the value of a property must be a member of a specific set. With DTDs they are most often used to define conversions for enumeration attributes, though they can also be applied to character content via element-value declarations.

We can, *e.g.*, declare an enumeration class for the **`action`** attribute of the **`trade`** element type:

```
<enumeration name="Action"
             members="buy buy-to-cover sell sell-short"/>
```

This declaration would define an enumeration class with this signature:

```
public final class Action {
    public static final Action BUY;
    public static final Action BUY_TO_COVER;
    public static final Action SELL;
    public static final Action SELL_SHORT;
    public static Action parse(String x);
    public String toString();
    public boolean equals(Object ob);
    public int hashCode();
}
```

The implicitly-defined conversion **`Action`** may be specified in the declaration of the **`action`** attribute property:

```
<element name="trade" type="class" root="true">
  ...
  <attribute name="action" convert="Action"/>
```

This would then define a reference property of type **`Action`** in the **`Trade`** class:

```
public class Trade ... {
    ...
    public Action getAction();
    public void setAction(Action x);
    ...
}
```

Similar declarations may be used to define a **`Duration`** enumeration class for use with the **`duration`** attribute.

## 6.16 Constructor declarations

A constructor declaration has the form

**`<constructor properties="`**$p_1\ p_2\ \ldots\ p_n$**`"/>`**

and may appear only within an element-class declaration.

***Source constraints*** The property names $p_i$, of which there must be at least one, must be valid XML names and must be unique. Each property name must be identical to the name of a property defined in the enclosing element class.

***Result*** The schema compiler defines, in the class being generated for the containing element-class declaration, a constructor that takes a parameter of the appropriate type for each of the specified properties. When invoked, the constructor initializes these properties with the values passed in as the corresponding arguments. The constructor enforces type constraints in the same manner as they would be enforced by a property mutation method. Instances created by invoking this constructor will initially be invalid.

### 6.16.1 Example

The constructor declaration

```
<element name="trade" type="class" root="true">
  ...
  <constructor properties="account action symbol quantity"/>
```

defines a constructor in the **`Trade`** class for all of a **`trade`** element's required attributes and content except for **`date`**, which we assume will be filled in later:

```
public class Trade ... {
    Trade(int account, Action action,
          String symbol, int quantity);
    ...
}
```

## 6.17 Interface declarations

An interface declaration has the form

```
<interface name="name"
           members="C1 C2 ... Cn"
           [properties="p1 p2 ... pm"]/>
```

and may appear only at the top level of the binding schema.

***Source constraints*** The interface *name* must be a valid Java class name; it must not have a package prefix. It must be unique with respect to the element classes, enumeration classes, and other interfaces declared at the top level of the binding schema.

The member names $C_i$ must be unique. Each member name must be a valid Java class name and must be the name of an element class or an interface declared in the binding schema.

The property names must be valid XML names and must be unique. For each property name $p_i$, each class or interface $C_i$ must define a property of that name. These properties must have the same base type and collection type across these classes and interfaces, but they may have different predicates and default values.

***Result*** The schema compiler defines a public interface of the specified name. Each member class $C_i$ is declared to implement this interface. The interface declares the methods that realize each of the specified properties.

### 6.17.1 Example

Interface declarations provide a way to define a type more specific than the default type **`MarshallableObject`** for a property to which a complex content content specification is bound. This partly mimics the use of parameter entities in DTDs to encapsulate choices in element-content models.

Suppose, *e.g.*, that we extend the stock-trade DTD so that we can also describe transfers. The **`transfer`** element type is similar to **`trade`** except that it has an additional required attribute, **`to-account`**, to represent the destination account, and it does not allow **`limit`** or **`stop`** elements in its content:

```
<!ELEMENT transfer ( symbol, quantity, date ) >
<!ATTLIST transfer
          account CDATA #REQUIRED
          to-account CDATA #REQUIRED >
```

The element-class declaration

```
<element name="transfer" type="class" root="true">
  <attribute name="account" convert="int"/>
  <attribute name="to-account" convert="int/>
  <content>
```

```
        <element-ref name="symbol"/>
        <element-ref name="quantity"/>
        <element-ref name="date"/>
```

will define a **Transfer** class whose signature will exactly match that of the **Trade** class for each of the shared properties **account**, **symbol**, **quantity**, and **date**.

Suppose that we further extend the stock-trade DTD so that we can describe batches of transfers and trades. If the **transaction-batch** element type is declared

```
<!ELEMENT transaction-batch ( trade | transfer )+ >
```

then the element-class declaration

```
<element name="transaction-batch" type="class" root="true">
  <content>
    <choice property="transactions" collection="array"/>
```

defines a class with the (partial) signature

```
public class TransactionBatch
    extends MarshallableRootElement
    implements RootElement
{
    public MarshallableObject[] getTransactions();
    public void setTransactions(MarshallableObject[] x);
}
```

The **TransactionBatch** class is somewhat clumsy to use as it stands: The array returned by the **getTransactions** method is of type **MarshallableObject[]**, so we must perform **instanceof** tests and insert type casts in order to make use of its content. We can improve the situation by declaring an interface that is implemented by both the **Trade** and **Transfer** classes and that declares the methods used in the realizations of their common properties. The interface declaration

```
<interface name="Transaction"
           members="Trade Transfer"
           properties="account symbol quantity date"/>
```

defines the interface

```
public interface Transaction {
    public int getAccount();
    public void setAccount(int x);
    public boolean hasAccount();
    public void deleteAccount();
    public String getSymbol();
    public void setSymbol(String x);
    public int getQuantity();
    public void setQuantity(int x);
    public boolean hasQuantity();
```

```
        public void deleteQuantity();
        public java.util.Date getDate();
        public void setDate(java.util.Date x);
    }
```

which in turn can be specified as the supertype of the **transactions** property of the element-class declaration for the **transaction-batch** element type:

```
    <element name="transaction-batch" type="class" root="true">
      <content>
        <choice property="transactions" collection="array"
                    supertype="Transaction"/>
```

These changes yield a **TransactionBatch** class with the signature

```
    public class TransactionBatch
        extends MarshallableRootElement
        implements RootElement
    {
        public Transaction[] getTransactions();
        public void setTransactions(Transaction[] x);
    }
```

which obviates the need for **instanceof** tests and type casting insofar as trades and transactions can be treated uniformly.

# A `javax.xml.bind` package specification

**A.1 Interfaces** *71*
*`Element`* *71*
*`IdentifiableElement`* *72*
*`PredicatedLists.Predicate`* *72*
*`RootElement`* *72*
**A.2 Classes** *72*
`Dispatcher` *73*
`MarshallableObject` *76*
`MarshallableRootElement` *77*
`Marshaller` *79*
`PCData` *80*
`PredicatedLists` *81*
`Unmarshaller` *82*
`ValidatableObject` *84*
`Validator` *85*
`Validator.Patcher` *88*
**A.3 Checked exceptions** *88*
`ConversionException` *88*
`GlobalValidationException` *89*
`InvalidContentException` *89*
`LocalValidationException` *89*
`MissingContentException` *90*
`MissingIdentifierException` *90*
`StructureValidationException` *90*
`TypeValidationException` *90*
`UnmarshalException` *91*
`UnrecognizedElementNameException` *92*
`ValidationException` *92*
**A.4 Unchecked exceptions** *93*
`TypeConstraintException` *93*

## A.1 Interfaces

### *`Element`* `javax.xml.bind`

```
 interface Element {
     void  validateThis() throws LocalValidationException;
}
```

Interface implemented by all element classes, that is, classes derived directly from element-type declarations.

For convenience, this interface redeclares the **`validateThis()`** method defined in the **`ValidatableObject`** class. The full specification of this method may be found in that class.

**`void validateThis() throws LocalValidationException;`**

Ensures that this object does not violate any local structural constraints. This method is exactly the **`validateThis()`** method defined in the **`ValidatableObject`** class.

***throws*** **`InvalidContentException`** If the children of this object violate the content specification of the schema component from which this object's class was derived

***throws*** If a property derived from a required attribute has no value

***throws*** **`MissingContentException`** If a property derived from a required content component has no value

### *IdentifiableElement* javax.xml.bind

```
 interface IdentifiableElement
    extends Element
{
     String  id();
}
```

Interface implemented by element classes derived from element types containing an ID attribute.

**String id();**

Returns this element object’s identifier value, or **null** if its identifier property has not been given a value.
***returns*** This element’s ID value, or **null** if it has none

### *PredicatedLists.Predicate* javax.xml.bind

```
static interface PredicatedLists.Predicate {
     void  check(Object ob);
}
```

A predicate for a predicated list.

**void check(Object ob);**

Checks that this predicate holds for the given object.
***throws*** **RuntimeException** If the given object violates this predicate

### *RootElement* javax.xml.bind

```
 interface RootElement
    extends Element
{
     void  validate() throws StructureValidationException;
}
```

Interface implemented by all root element classes.

For convenience, this interface redeclares the public root validation method defined in the **ValidatableObject** class. The full specification of this method may be found in that class.

**void validate() throws StructureValidationException;**

Validates the content tree rooted at this object. This method is exactly the **validate()** method defined in the **ValidatableObject** class.
***throws*** **StructureValidationException** If any structural constraints are violated

## A.2 Classes

## Dispatcher javax.xml.bind

```
final class Dispatcher {
                  Dispatcher();
            void  freezeClassMap();
            void  freezeElementNameMap();
           Class  lookup(Class mobClass);
           Class  lookup(String elementName)
                      throws UnrecognizedElementNameException;
            void  register(Class mobClass, Class userClass);
            void  register(String elementName, Class elementClass);
     RootElement  unmarshal(java.io.InputStream in) throws UnmarshalException;
     RootElement  unmarshal(javax.xml.marshal.XMLScanner xs)
                      throws UnmarshalException;
     RootElement  unmarshal(javax.xml.marshal.XMLScanner xs, Class rootClass)
                      throws UnmarshalException;
}
```

A *dispatcher* is used to map element names to class names, and to initiate the unmarshalling process.

A dispatcher contains two maps, an *element-name map* and a *class map*. The element-name map maps element names to marshallable-object classes; the class map maps marshallable-object classes to user-defined subclasses.

New mappings may be *registered* in each map. Each map may also be *frozen* in order to guard against unwanted concurrent modification. The unmarshalling process always freezes both maps of the dispatcher that it uses.

### Dispatcher();

Constructs a new, empty dispatcher.

### void freezeClassMap();

Freezes this dispatcher's class map. Further attempts to register user subclasses of schema-derived marshallable-object classes will result in an **IllegalStateException** being thrown.

### void freezeElementNameMap();

Freezes this dispatcher's element-name map. Further attempts to register element names will result in an **IllegalStateException** being thrown.

### Class lookup(Class mobClass);

Applies the class map to the given marshallable-object class.

***returns*** The user class to which the given class is mapped, or the given class if it has not been mapped

**`Class lookup(String elementName)`**
**`    throws UnrecognizedElementNameException;`**

Determines the class to be used when unmarshalling the given element name. The class to be used is computed by applying the element-name map to the given name and then applying the class map to the resulting class.

***param*** **`elementName`** The non-colonized element name being looked up; must not be the empty string

***returns*** The class to which the given element name is mapped

***throws*** **`UnrecognizedElementNameException`** If no class for the given element name can be found

**`void register(Class mobClass, Class userClass);`**

Registers the given user class so that it will be used in place of the given schema-derived marshallable-object class during unmarshalling. Requires that this dispatcher's class map is not frozen.

This method is provided primarily so that user subclasses of schema-derived element classes may be defined and used.

***param*** **`mobClass`** A schema-derived subclass of **`MarshallableObject`** for which no class mapping has yet been defined

***param*** **`userClass`** The user-defined subclass of **`mobClass`**, for which no class mapping has been defined, to be used in place of that class during unmarshalling

***throws*** **`IllegalStateException`** If this dispatcher's class map is frozen

**`void register(String elementName, Class elementClass);`**

Registers the given element name so that it will be unmarshalled into an instance of the given element class. Requires that this dispatcher's element-name map is not frozen.

***param*** **`elementName`** The non-colonized element name being registered; must not be the empty string

***param*** **`elementClass`** The class to be used to unmarshal elements with the given name; must extend **`MarshallableObject`** and implement the **`Element`** interface

***throws*** If the element name has already been registered

***throws*** **`IllegalStateException`** If this dispatcher's element-name map is frozen

***`RootElement`*** **`unmarshal(java.io.InputStream in)`**
**`    throws UnmarshalException;`**

Unmarshals and validates a content tree from the given input stream. An invocation of this convenience method behaves in the same manner as the expression

```
unmarshal(javax.xml.marshal.XMLScanner.open(in))
```

***param*** **`in`** The input stream from which data will be unmarshalled

***returns*** The root element object of a valid, unmarshalled content tree

***throws*** **`javax.xml.marshal.ScanIOException`** If an I/O error occurs

***throws*** **`javax.xml.marshal.ScanException`** If the input document is not well-formed

***throws*** **`ValidationException`** If the input document violates the constraints expressed in the source schema, or some other validation error is detected

```
RootElement unmarshal(javax.xml.marshal.XMLScanner xs)
    throws UnmarshalException;
```

Unmarshals and validates a content tree using the given scanner. After freezing both the element-name and class maps, this method examines the scanner's current start tag, maps it to a marshallable-object class via the **`lookup`** method, and then invokes the **`unmarshal(javax.xml.marshal.XMLScanner, Class)`** method, passing the scanner and the class.

***param*** **`xs`** The scanner from which data will be unmarshalled

***returns*** The root element object of a valid, unmarshalled content tree

***throws*** **`InvalidContentException`** If the scanner is not currently positioned at a start tag, or if some other invalid content is later encountered

***throws*** **`UnrecognizedElementNameException`** If the element name in the current start tag is not registered, or if some other unrecognized element name is encountered

***throws*** If an attribute that is not permitted for the current element is scanned

***throws*** **`javax.xml.marshal.ScanIOException`** If an I/O error occurs

***throws*** **`javax.xml.marshal.ScanException`** If the input document is not well-formed

***throws*** **`ValidationException`** If the input document violates the constraints expressed in the source schema, or some other validation error is detected

```
RootElement unmarshal(javax.xml.marshal.XMLScanner xs,
                      Class rootClass)
    throws UnmarshalException;
```

Unmarshals and validates a content tree using the given scanner and root element class. After freezing both the element-name and class maps, this method instantiates a new **`Unmarshaller`** with this dispatcher and the given scanner. It then uses the unmarshaller to unmarshal an instance of the given class and validate the resulting content tree. This method closes the given scanner when it terminates, whether normally or by throwing an exception.

***param*** **`xs`** The scanner from which data will be unmarshalled

***param*** **`rootClass`** The root marshallable-object class to be instantiated

***returns*** The root element object of a valid, unmarshalled content tree

***throws*** **`InvalidContentException`** If the scanner is not currently positioned at a start tag, or if some other invalid content is later encountered

***throws*** **`UnrecognizedElementNameException`** If the element name in the current start tag is not the element name handled by the given class, or if some other unrecognized element name is later encountered

***throws*** If an attribute that is not permitted for the current element is scanned

***throws*** **`javax.xml.marshal.ScanIOException`** If an I/O error occurs

***throws*** **`javax.xml.marshal.ScanException`** If the input document is not well-formed

***throws*** **`ValidationException`** If the input document violates the constraints expressed in the source schema, or some other validation error is detected

## MarshallableObject javax.xml.bind

```
abstract class MarshallableObject
    extends ValidatableObject
{
        protected  MarshallableObject();
             void  marshal(Marshaller m) throws java.io.IOException;
    abstract void  unmarshal(Unmarshaller u) throws UnmarshalException;
}
```

Abstract base class for objects that can be marshalled and unmarshalled. The methods in this class, in conjunction with those in the **`Marshaller`**, **`Dispatcher`**, and **`Unmarshaller`** classes, define the processes of marshalling content trees into XML documents and vice-versa.

This class does not implement the **`Element`** interface because some schema-derived classes will be marshallable but will not be element classes.

Both marshalling and unmarshalling are optional operations because in some applications only one or the other operation is of interest. If a particular operation is not supported then an **`UnsupportedOperationException`** is thrown when the corresponding method is invoked.

### protected MarshallableObject();

Initializes a new marshallable object.

### void marshal(Marshaller m) throws java.io.IOException;

Marshals the content of this object, and its children, using the given marshaller *(optional operation)*. Requires that the object be valid.

This method must use the given marshaller's **`XMLWriter`** to write the content of this object, and it must invoke the given marshaller's **`marshal(MarshallableObject)`** method to marshal any children.

As defined in this class, this method throws an **`UnsupportedOperationException`**; it should be overridden only by schema-derived classes. This method should only be invoked by the marshalling process; its behavior when invoked in any other manner is unspecified.

***param*** **`m`** The marshaller to be used

***throws*** **`java.io.IOException`** If an I/O error occurs

***throws*** **`UnsupportedOperationException`** If this object does not support marshalling

**abstract void unmarshal(Unmarshaller u) throws UnmarshalException;**

Unmarshals an element or some character data into this object, converting data as necessary, checking type constraints and local structural constraints, and recording the information required to check global structural constraints.

This method must use the given unmarshaller's **XMLScanner** to parse the elements and attributes represented by this object. It must recursively invoke the given unmarshaller's **unmarshal()** and **unmarshal(Class)** methods to unmarshal any subelements. Finally, it must invoke the **reference(String, Validator.Patcher)** method of the marshaller's validator in order to arrange for the target of each identifier reference to be installed once the reference is resolved.

As defined in this class, this method throws an **UnsupportedOperationException**; it should be overridden only by schema-derived classes. This method should only be invoked by the unmarshalling process; its behavior when invoked in any other manner is unspecified.

| | | |
|---|---|---|
| ***param*** | **u** | The unmarshaller to be used |
| ***throws*** | **javax.xml.marshal.ScanIOException** | If an I/O error occurs |
| ***throws*** | **javax.xml.marshal.ScanException** | If the source document is not well-formed |
| ***throws*** | **InvalidContentException** | If the unmarshaller's scanner is not initially positioned at an appropriate start tag or at some character data, according to whether this object is to represent an element or character data, respectively; or if this object is to represent an element and the element's content specification as described in the source schema is violated |
| ***throws*** | **ValidationException** | If a type or structural constraint is violated by the input document |

## MarshallableRootElement javax.xml.bind

```
abstract class MarshallableRootElement
    extends MarshallableObject // ValidatableObject
    implements RootElement
{
     protected  MarshallableRootElement();
    final void  marshal(java.io.OutputStream out) throws java.io.IOException;
    final void  marshal(javax.xml.marshal.XMLWriter xw)
                    throws java.io.IOException;
}
```

Abstract base class for root element objects that can be marshalled and unmarshalled. The marshalling methods in this class, in conjunction with the unmarshalling methods defined in the **Dispatcher** class and in every schema-derived marshallable root element class, are the primary entry points for the marshalling and unmarshalling processes.

Both marshalling and unmarshalling are optional operations because in some applications only one or the other operation is of interest. If a particular operation is not supported then an **UnsupportedOperationException** is thrown when the corresponding method is invoked.

***Static methods in marshallable root element classes*** This class only defines marshalling methods. The unmarshalling methods associated with a specific root class, as well as the method for creating a default dispatcher, must be static methods. The Java programming language does not support abstract static methods, however, so the schema compiler generates four static methods in each marshallable root element class. For a root element class ***Foo*** the primary unmarshalling method has the signature

```
public static Foo unmarshal(javax.xml.marshal.XMLScanner xs,
                               Dispatcher d)
    throws UnmarshalException;
```

and behaves exactly the same as the expression

```
(Foo)(d.unmarshal(xs, Foo.class))
```

The other two static unmarshalling methods are simply convenience methods that use a dispatcher obtained from the static **newDispatcher** method and, in the last case, also create a new scanner from the given input stream:

```
public static Foo unmarshal(javax.xml.marshal.XMLScanner xs)
    throws UnmarshalException;
public static Foo unmarshal(java.io.InputStream in)
    throws UnmarshalException;
```

Finally, the static method for creating a new default dispatcher has the signature

```
public static Dispatcher newDispatcher();
```

This method creates a new dispatcher and initializes it to map each element name defined in the source schema from which the root element class was derived to the corresponding marshallable root element class. After being initialized the element-name map is frozen. User subclasses of schema-derived marshallable-object classes may then be registered if desired.

**protected**
**MarshallableRootElement();**

Initializes a new marshallable root element object.

**final void marshal(java.io.OutputStream out)**
**throws java.io.IOException;**

Marshals the content tree rooted at this object to the given output stream *(optional operation)*. This convenience method behaves in exactly the same way as the expression **marshal(new XMLWriter(out))**.

***param*** **xw** The XML writer to be used

***throws*** **java.io.IOException** If an I/O error occurs

***throws*** If the content tree requires validation

***throws*** **UnsupportedOperationException** If some object in the content tree does not support marshalling

**`final void marshal(javax.xml.marshal.XMLWriter xw)`**
**`    throws java.io.IOException;`**

Marshals the content tree rooted at this object using the given XML writer *(optional operation)*.

This method creates a new **`Marshaller`** based upon the given writer, invokes its **`marshal(MarshallableObject)`** method, passing this object, and then flushes the writer.

***param*** **`xw`** The XML writer to be used
***throws*** **`java.io.IOException`** If an I/O error occurs
***throws*** If the content tree requires validation
***throws*** **`UnsupportedOperationException`** If some object in the content tree does not support marshalling

## Marshaller javax.xml.bind

```
final class Marshaller {
     void  marshal(MarshallableObject mob) throws java.io.IOException;
     javax.xml.marshal.XMLWriter
           writer();
}
```

A *marshaller* governs the process of marshalling a valid content tree into an XML document. It encapsulates the writer used during a marshalling operation, and provides the basic marshalling method.

The process of marshalling a content tree begins when one of the **`marshal`** methods of the tree's root element object is invoked. This method creates a new unmarshaller from the given writer and invokes its **`marshal`** method, passing the root object, and then flushes the writer.

The marshalling process does not validate the content tree being marshalled, but it does require the tree to be valid. If an invalid object is detected during marshalling then a **`ValidationRequiredException`** will be thrown and the marshalling operation aborted. This may cause an incomplete XML document to be written; to avoid that, ensure that the content tree is valid before invoking a **`marshal`** method.

**`void marshal(MarshallableObject mob) throws java.io.IOException;`**

Marshals the given marshallable object using this marshaller's writer.

This method invokes the given object's **`marshal(Marshaller)`** method, passing this marshaller.

***throws*** **`java.io.IOException`** If an I/O error occurs
***throws*** If the given object requires validation

**javax.xml.marshal.XMLWriter**
    **writer();**

Returns the writer to be used during this marshalling operation.

***returns*** The writer

## PCData javax.xml.bind

```
final class PCData
    extends MarshallableObject // ValidatableObject
    implements Comparable
{
              PCData();
              PCData(String chars);
       String chars();
         void chars(String chars);
          int compareTo(Object ob);
      boolean equals(Object ob);
       String getChars();
          int hashCode();
         void marshal(Marshaller m) throws java.io.IOException;
         void setChars(String chars);
         void unmarshal(Unmarshaller um) throws UnmarshalException;
         void validateThis() throws MissingContentException;
}
```

Marshallable-object class for parsed character data.

Instances of this class are used to represent character data occurring in elements with simple character content or mixed content.

An instance of this class must contain character data in order to be valid.

### PCData();

Constructs a new instance of this class with no character data.

### PCData(String chars);

Constructs a new instance of this class with the given character data.

***param*** **chars** The character data for this instance

### String chars();

Returns the character data in this instance.

***returns*** The character data

### void chars(String chars);

Changes the character data in this instance.

***param*** **chars** The new character data for this instance; may not be **null**

**`int compareTo(Object ob);`**

Compares this object to another object. Two instances of this class are compared by comparing their character data in the manner of the **`String.compareTo`** method.

|  |  |
|---|---|
| ***param*** | **`ob`** The object with which this object is to be compared |
| ***returns*** | A negative integer, a positive integer, or zero as this object is less than, equal to, or greater than the given object |
| ***throws*** | **`ClassCastException`** If the given object is not an instance of this class |

**`boolean equals(Object ob);`**

Tells whether or not this object is equal to another. Two instances of this class are equal if, and only if, their character data are equal.

|  |  |
|---|---|
| ***param*** | **`ob`** The object with which this object is to be compared |
| ***returns*** | **`true`** if, and only if, this object is equal to the given object |

**`String getChars();`**

Returns the character data in this instance.

|  |  |
|---|---|
| ***returns*** | The character data |

**`int hashCode();`**

Returns a hash-code value for this object.

|  |  |
|---|---|
| ***returns*** | A hash-code value |

**`void marshal(Marshaller m) throws java.io.IOException;`**

**`void setChars(String chars);`**

Changes the character data in this instance.

|  |  |
|---|---|
| ***param*** | **`chars`** The new character data for this instance |

**`void unmarshal(Unmarshaller um) throws UnmarshalException;`**

**`void validateThis() throws MissingContentException;`**

## PredicatedLists javax.xml.bind

```
final class PredicatedLists {
    static java.util.List  createInvalidating(MarshallableObject mob,
                                              PredicatedLists.Predicate pred,
                                              java.util.List list);
    static java.util.List  create(MarshallableObject mob,
                                  PredicatedLists.Predicate pred,
                                  java.util.List list);
}
```

Lists for implementing schema-derived list properties.

A predicated list enforces a *predicate* represented by an object that implements the **`PredicatedLists.Predicate`** interface. If an attempt is made to add an object that does not satisfy the predicate then an appropriate runtime exception is thrown.

A predicated list optionally invalidates the marshallable object with which it is associated after the list is modified in any way. This ensures that both the marshallable object and the list are revalidated during the next validation operation.

```
static java.util.List
    createInvalidating(MarshallableObject mob,
                       PredicatedLists.Predicate pred,
                       java.util.List list);
```

Constructs a new invalidating predicated list.

The given marshallable object will be invalidated each time the list is modified.

| | | |
|---|---|---|
| ***param*** | **`mob`** | The marshallable object with which the new predicated list is to be associated |
| ***param*** | **`pred`** | The predicate to be enforced upon list elements |
| ***param*** | **`list`** | The list that will back the new predicated list |
| ***returns*** | The new predicated list | |

```
static java.util.List create(MarshallableObject mob,
                             PredicatedLists.Predicate pred,
                             java.util.List list);
```

Constructs a new predicated list.

| | | |
|---|---|---|
| ***param*** | **`mob`** | The marshallable object with which the new predicated list is to be associated |
| ***param*** | **`pred`** | The predicate to be enforced upon list elements |
| ***param*** | **`list`** | The list that will back the new predicated list |
| ***returns*** | The new predicated list | |

## Unmarshaller javax.xml.bind

```
final class Unmarshaller {
    javax.xml.marshal.XMLScanner
                        scanner();
    MarshallableObject  unmarshal() throws UnmarshalException;
    MarshallableObject  unmarshal(Class mobClass) throws UnmarshalException;
             Validator  validator();
}
```

An *unmarshaller* governs the process of unmarshalling an XML document into a newly-created content tree, validating the tree as it is constructed. It encapsulates the dispatcher, scanner, and validator objects used during an unmarshalling operation, and it provides the basic unmarshalling methods.

The process of unmarshalling a content tree begins when one of the unmarshal methods of a **`Dispatcher`** object is invoked. (An unmarshalling operation may also be initiated by invoking one of the static methods in each marshallable root element class generated by the schema compiler.) These methods examine the given scanner's current state, if necessary, in order to locate the element class to be unmarshalled. They then create a new unmarshaller from the dispatcher, the given scanner, and a new validator. Finally, the new unmarshaller's **`unmarshal(Class)`** method is invoked, passing the element class. After this method returns the validator is used to resolve any pending identifier references and check any global constraints.

The unmarshalling process makes the same guarantees as the validation process. If unmarshalling completes successfully then the validation propositions are guaranteed to hold for every member of the resulting content tree. It is also guaranteed that the target of every identifier reference exists.

This class does not have any public or protected constructors; it is intended only for use by the unmarshalling process.

```
javax.xml.marshal.XMLScanner
    scanner();
```

Returns the scanner to be used during this unmarshalling operation.
***returns*** The scanner

```
MarshallableObject unmarshal() throws UnmarshalException;
```

Unmarshals the next element in this unmarshaller's scanner and validates the resulting instance subtree.

This method examines the scanner's current start tag, uses the dispatcher to map it to a marshallable-object class, and then passes the resulting class to the **`unmarshal(Class)`** method.

***returns*** A valid, unmarshalled object
***throws*** **`InvalidContentException`** If the scanner is not currently positioned at a start tag, or if some other invalid content is later encountered
***throws*** **`UnrecognizedElementNameException`** If the element name in the current start tag is not registered in this unmarshaller's dispatcher, or if some other unrecognized element name is later encountered
***throws*** **`javax.xml.marshal.ScanIOException`** If an I/O error occurs
***throws*** **`javax.xml.marshal.ScanException`** If the input document is not well-formed
***throws*** **`ValidationException`** If the input document violates the constraints expressed in the source schema, or some other validation error is detected

```
MarshallableObject unmarshal(Class mobClass)
    throws UnmarshalException;
```

Unmarshals the next element in this unmarshaller's scanner, using the given schema-derived marshallable-object class, and validates the resulting subtree.

This method applies the dispatcher's class map to the given marshallable class and creates an instance of the resulting class. It then invokes the **`unmarshal(Unmarshaller)`** method of the new instance, passing this unmarshaller in order to unmarshal recursively the element and its subelements, if any. Once the new instance has been unmarshalled it is locally validated by invoking its **`validateThis()`** method. If the instance implements the **`IdentifiableElement`** interface and has an identifier then the validator is updated with that identifier value.

***returns*** A valid, unmarshalled object

***throws*** If the instantiated class modified the instance in any way during its initialization

***throws*** **`InvalidContentException`** If the scanner is not currently positioned at a start tag, or if some other invalid content is later encountered

***throws*** **`UnrecognizedElementNameException`** If the element name in the current start tag is not the element name handled by the given class, or if some other unrecognized element name is later encountered

***throws*** **`javax.xml.marshal.ScanIOException`** If an I/O error occurs

***throws*** **`javax.xml.marshal.ScanException`** If the input document is not well-formed

***throws*** **`ValidationException`** If the input document violates the constraints expressed in the source schema, or some other validation error is detected

**`Validator validator();`**

Returns the validator to be used during this unmarshalling operation.

***returns*** The validator

## ValidatableObject javax.xml.bind

```
abstract class ValidatableObject {
    final void  invalidate();
    final void  validate() throws StructureValidationException;
          void  validateThis() throws LocalValidationException;
          void  validate(Validator vd) throws StructureValidationException;
}
```

Abstract base class for all validatable objects. The methods in this class, in conjunction with those in the **`Validator`** class, define the content-tree validation process.

**`final void invalidate();`**

Marks this object invalid. An invalid object will be revalidated the next time its content tree is validated. An invalid object will also prevent its content tree from being marshalled.

This method is intended to be invoked only by the property-mutation methods defined in schema-derived classes. It should not be invoked during the unmarshalling process.

**`final void validate() throws StructureValidationException;`**

Validates the content tree rooted at this object, which must be a root element.

This method creates a new **`Validator`** object and then proceeds to validate this content tree by invoking the validator's **`validate(ValidatableObject)`** method, passing this object. If this recursive process completes successfully then the validation propositions are known to hold for every object in the content tree. This method then checks that the target of every identifier reference has an identifier and is a member of the content tree.

***throws*** **`StructureValidationException`** If any structural constraints are violated

***throws*** **`UnsupportedOperationException`** If this object is not an instance of the **`RootElement`** interface

**`void validateThis() throws LocalValidationException;`**

Ensures that this object does not violate any local structural constraints. In particular, this method ensures that any required properties have been given values and that the object's children in the content tree, if any, match the content specification of the schema component from which this object's class was derived. This method does not recursively validate this object's children.

The default implementation of this method does nothing; it should be overridden only by schema-derived classes. This method may be invoked outside of the normal validation process.

***throws*** If a collection property derived from a compound attribute is empty, but the source schema requires at least one value

***throws*** **`InvalidContentException`** If the children of this object violate the content specification of the schema component from which this object's class was derived

***throws*** If a property derived from a required attribute has no value

***throws*** **`MissingContentException`** If a property derived from a required content component has no value

**`void validate(Validator vd) throws StructureValidationException;`**

Recursively validates this object's children using the given validator and updates the validator with any local information that is subject to global constraints.

This method must invoke the validator's **`validate(ValidatableObject)`** method upon the children of this object in the content tree. It must also invoke the validator's **`reference(IdentifiableElement)`** method upon the target of each identifier-reference property in this object.

The default implementation of this method does nothing; it should be overridden only by schema-derived classes. This method may only be invoked by the validation process; its behavior when invoked in any other manner is unspecified.

***param*** **`vd`** The validator to be applied to this object's children

***throws*** **`StructureValidationException`** If any structural constraints are violated

## Validator javax.xml.bind

```
final class Validator {
     void  reference(IdentifiableElement elt) throws MissingIdentifierException;
     void  reference(String id, Validator.Patcher p);
     void  validate(ValidatableObject vob) throws StructureValidationException;
}
```

A *validator* governs the process of validating a content tree. It supplies the central validation method, **`validate`**, methods for recording and resolving identifier references, and the internal logic required to validate global structural constraints.

The process of validating a content tree begins when the **`validate()`** method of the tree's root element object is invoked. This method creates a new validator and then proceeds to validate the content tree by invoking the validator's **`validate`** method, passing the root object.

The validator's **`validate`** method ensures that the following *validation propositions* are true of its argument object:

1. The object has not yet been visited during this validation, *i.e.*, that it is not part of a cycle;

2. The object is locally valid, *i.e.*, that its **`validateThis()`** method would not throw a **`LocalValidationException`** if invoked; and that

3. If the object implements the **`IdentifiableElement`** interface then its identifier, if not **`null`**, is unique among the identifiers so far encountered during this validation.

The validator's **`validate`** method also invokes its argument's **`validate(Validator)`** method, passing the validator itself, in order to:

- Ensure that the above propositions are true of its argument's children in the content tree, if any; and to

- Arrange to ensure before the end of the validation process that the targets of any identifier-reference properties in the object have identifiers and are members of the content tree being validated.

Each validatable object's **`validate(Validator)`** method recursively validates the object's children, if any, by invoking the validator's **`validate`** method upon them. It also invokes the validator's **`reference(IdentifiableElement)`** method upon the target of each identifier-reference property in the object.

If this recursive process completes successfully then the propositions defined above are known to hold for every object in the content tree. The root element object's **`validate()`** method then checks that the target of every identifier reference has an identifier and is a member of the content tree.

Validators are also used during the unmarshalling process to validate an content tree as it is being unmarshalled. The validation process is slightly different when carried out during unmarshalling, although the guarantees that it makes are identical. There is no need to check for cycles, in particular, since that proposition is guaranteed by the linear nature of the unmarshalling process. It is necessary, however, to arrange that identifiers referring to elements that have not yet been unmarshalled be resolved before the unmarshalling process is complete. This requirement is supported by the validator's

**reference(String, Validator.Patcher)** method, which takes a **patcher** object that represents the unresolved reference. The unmarshalling process ensures that all references are resolved before it completes. It also ensures that all patchers are invoked with the targets of their references so that appropriate links can be installed in the objects referring to those targets.

This class does not have any public or protected constructors; it is intended only for use by the validation and unmarshalling processes.

**void reference(*IdentifiableElement* elt)**
**throws MissingIdentifierException;**

Records the fact that a reference has been made to the given identifiable element object. This method arranges to ensure before the end of the validation process that the given object is a member of the content tree being validated.

***param*** **elt** The referenced element object

***throws*** **MissingIdentifierException** If the argument does not have an identifier

**void reference(String id, Validator.Patcher p);**

Records the fact that a reference has been made to an element object with the given identifier, and registers a patcher for later invocation. This method arranges to ensure before the end of the unmarshalling process that the given identifier is resolved and that the patcher is invoked with the target of the reference.

***param*** **id** The identifier value being referenced

***param*** **patcher** The patcher to be invoked after the identifier is resolved

**void validate(ValidatableObject vob)**
**throws StructureValidationException;**

Validates the given object.

This method ensures that the validation propositions hold for the given object. It also invokes the given object's **validate(Validator)** method, passing this validator, in order to ensure that the propositions are true of the given object's children and to arrange to ensure before the end of the validation process that the targets of any identifier-reference properties in this object have identifiers and are members of the content tree being validated.

This method is not intended to be invoked outside of the validation process; its behavior when invoked in any other manner is unspecified.

***param*** **vob** The object to be validated

***throws*** If the object has already been visited during this validation, in which case there is a cycle in this content tree

***throws*** If the object has an identifier that is the same as the identifier of some other object in this content tree

***throws*** **StructureValidationException** If some other validation failure is detected

### Validator.Patcher javax.xml.bind

```
static abstract class Validator.Patcher {
                   Validator.Patcher();
    abstract void  patch(IdentifiableElement target);
}
```

A *patcher* represents an unresolved identifier reference.

When unmarshalling a content tree, patchers are used to install references to element objects that may have yet to be unmarshalled. A patcher has a single method, **patch**, which is guaranteed to be invoked some time after the reference is resolved but before the unmarshalling process is complete. When the method is invoked, the target element of the reference is passed as its single argument.

In typical use an anonymous inner subclass of this abstract class is created and passed to the **reference(String, Validator.Patcher)** method:

```
vd.reference(id, new Patcher()
    public void patch(IdentifiableElement target)
        link = (Type)target;
);
```

where **vd** is the validator being used, **id** is the identifier being referenced, and ***Type*** is the expected type of the referent.

This class is intended to be instantiated only by schema-derived unmarshalling code.

**Validator.Patcher();**

**abstract void patch(*IdentifiableElement* target);**

Patch method, guaranteed to be invoked some time after the reference represented by this patcher is resolved.

***param*** **target** The target of the reference

## A.3 Checked exceptions

### ConversionException javax.xml.bind

```
 class ConversionException
    extends UnmarshalException // Exception
{
               ConversionException(String desc);
               ConversionException(String desc, Throwable x);
     Throwable getCause();
          void printStackTrace(java.io.PrintStream out);
}
```

Checked exception thrown by an XML scanner when an I/O error occurs. The I/O exception that is thrown is wrapped within this exception.

**`ConversionException(String desc);`**

Constructs a new instance of this class.
***param*** **`desc`** A description of component being converted

**`ConversionException(String desc, Throwable x);`**

Constructs a new instance of this class.
***param*** **`desc`** A description of component being converted
***param*** **`cause`** The exception that was thrown during conversion

**`Throwable getCause();`**

Retrieves the **`Throwable`** that caused this exception.
***returns*** The throwable

**`void printStackTrace(java.io.PrintStream out);`**

## GlobalValidationException javax.xml.bind

```
abstract class GlobalValidationException
    extends StructureValidationException // ValidationException
{
}
```

## InvalidContentException javax.xml.bind

```
 class InvalidContentException
    extends LocalValidationException // StructureValidationException
{
             InvalidContentException(String );
     String  getMessage();
}
```

**`InvalidContentException(String );`**

**`String getMessage();`**

## LocalValidationException javax.xml.bind

```
abstract class LocalValidationException
    extends StructureValidationException // ValidationException
{
}
```

## MissingContentException javax.xml.bind

```
 class MissingContentException
    extends LocalValidationException // StructureValidationException
{
             MissingContentException(String );
     String  getMessage();
}
```

**MissingContentException(String );**

**String getMessage();**

## MissingIdentifierException javax.xml.bind

```
 class MissingIdentifierException
    extends GlobalValidationException // StructureValidationException
{
      MissingIdentifierException();
}
```

**MissingIdentifierException();**

## StructureValidationException javax.xml.bind

```
abstract class StructureValidationException
    extends ValidationException // UnmarshalException
{
}
```

## TypeValidationException javax.xml.bind

```
 class TypeValidationException
    extends ValidationException // UnmarshalException
{
                              TypeValidationException(TypeConstraintException tcx);
     TypeConstraintException  getException();
}
```

Checked exception thrown when a **TypeConstraintException** is thrown during unmarshalling. That exception is wrapped within this exception.

**TypeValidationException(TypeConstraintException tcx);**

Constructs an instance wrapping the given **TypeConstraintException**.
***param*** **tcx** The **TypeConstraintException** to be wrapped

**TypeConstraintException**
    **getException();**

Returns the **TypeConstraintException** wrapped within this exception.
***returns*** The **TypeConstraintException** wrapped within this exception

## UnmarshalException javax.xml.bind

```
abstract class UnmarshalException
    extends Exception
{
     protected  UnmarshalException();
     protected  UnmarshalException(javax.xml.marshal.ScanPosition pos);
     protected  UnmarshalException(javax.xml.marshal.ScanPosition pos,
                                   String desc);
     protected  UnmarshalException(String desc);
    final javax.xml.marshal.ScanPosition
                getPosition();
    final void  initPosition(javax.xml.marshal.ScanPosition sp);
}
```

Abstract checked-exception class for exceptions thrown when errors occur during unmarshalling.

Unmarshalling errors fall into three categories, each of which has a specific subclass of this class:

- A **javax.xml.marshal.ScanException** is thrown when the input document is not well-formed or when an I/O error occurs.
- A **ValidationException** is thrown when a violation of a constraint expressed in the source schema is detected.
- A **ConversionException** is thrown when a binding conversion's parse method throws an exception.

Validation exceptions are further subdivided as described in the specification of the **ValidationException** class.

**protected UnmarshalException();**

Initializes a new instance of this class.

**protected UnmarshalException(javax.xml.marshal.ScanPosition pos);**

Initializes a new instance of this class with the given scan position.
***param*** **pos** The scan position

```
protected UnmarshalException(javax.xml.marshal.ScanPosition pos,
                             String desc);
```

Initializes a new instance of this class with the given scan position and description.

***param*** **`pos`** The scan position

***param*** **`desc`** The description

```
protected UnmarshalException(String desc);
```

Initializes a new instance of this class with the given description and no scan position.

***param*** **`desc`** The description

```
final javax.xml.marshal.ScanPosition
    getPosition();
```

Returns this exception's scan position, or **`null`** if the scan position is undefined.

***returns*** This exception's scan position

```
final void initPosition(javax.xml.marshal.ScanPosition sp);
```

Sets this exception's scan position. Once set, the scan position may not be set again.

***param*** **`sp`** The scan position

***throws*** **`IllegalStateException`** If the scan position has already been set

## UnrecognizedElementNameException javax.xml.bind

```
 class UnrecognizedElementNameException
    extends InvalidContentException // LocalValidationException
{
             UnrecognizedElementNameException(String );
     String  getElementName();
     String  getMessage();
}
```

```
    UnrecognizedElementNameException(String );
```

```
String getElementName();
```

```
String getMessage();
```

## ValidationException javax.xml.bind

```
abstract class ValidationException
    extends UnmarshalException // Exception
{
}
```

## A.4 Unchecked exceptions

### TypeConstraintException javax.xml.bind

```
 class TypeConstraintException
    extends RuntimeException
{
}
```

# B `javax.xml.marshal` package specification

| | |
|---|---|
| **B.1 Classes** | *95* |
| `DocumentScanPosition` | *95* |
| `ScanPosition` | *95* |
| `StreamScanPosition` | *96* |
| `XMLScanner` | *98* |
| `XMLWriter` | *106* |
| **B.2 Checked exceptions** | *109* |
| `EndOfDocumentException` | *109* |
| `ScanException` | *109* |
| `ScanIOException` | *110* |

## B.1 Classes

### DocumentScanPosition — javax.xml.marshal

```
final class DocumentScanPosition
    extends ScanPosition
{
                       DocumentScanPosition(org.w3c.dom.Node node);
     org.w3c.dom.Node  node();
}
```

An object describing a scanner's position in a DOM tree.

A document-scan position contains a reference to the DOM node at which the scanner is currently positioned.

**`DocumentScanPosition(org.w3c.dom.`*`Node`*` node);`**

Constructs a new scan position with the given node.
***param*** **`node`** The node; must not be **`null`**

**`org.w3c.dom.`*`Node`*` node();`**

Returns this scan position's node
***returns*** This position's node

### ScanPosition — javax.xml.marshal

```
abstract class ScanPosition {
}
```

A description of a scanner's position. Scan positions are used to report the locations of both well-formedness and validity errors encountered during unmarshalling. Different types of scan positions are defined by the subclasses of this class according to the different types of input that can be scanned: **`StreamScanPosition`** objects are created by stream scanners, and **`DocumentScanPosition`** objects are created by DOM-tree scanners.

## StreamScanPosition javax.xml.marshal

```
final class StreamScanPosition
    extends ScanPosition
{
             StreamScanPosition(int line);
             StreamScanPosition(int line, int col);
             StreamScanPosition(int line, int col, String uri);
        int  column();
        int  line();
     String  toString();
     String  uriString();
}
```

An object describing a scanner’s position in an input stream.

A stream-scan position contains a line number and an optional column number and source URI.

Both line and column numbers start at one. Lines are terminated by a LINE FEED character (**’\u000A’**), a CARRIAGE RETURN character (**’\u000D’**), or a CARRIAGE-RETURN character followed immediately by a LINE FEED character. Columns are counted in terms of sixteen-bit Unicode characters, in UTF-16, rather than bytes in any particular encoding.

### `StreamScanPosition(int line);`

Constructs a new scan position with the given line, column, and source URI.

***param*** **`line`** The line number, a positive integer

### `StreamScanPosition(int line, int col);`

Constructs a new scan position with the given line, column, and source URI.

***param*** **`line`** The line number, a positive integer
***param*** **`col`** The column number, or zero if the column number is not known

### `StreamScanPosition(int line, int col, String uri);`

Constructs a new scan position with the given line, column, and source URI.

***param*** **`line`** The line number, a positive integer
***param*** **`col`** The column number, or zero if the column number is not known
***param*** **`uri`** The source URI string, or **`null`** if the source URI is not known

### `int column();`

Returns this position’s column number.

***returns*** This position’s column number, a positive integer, or zero if it is not known

### `int line();`

Returns this position’s line number.

***returns*** This position’s line number, a positive integer

**`String toString();`**

Returns a string describing this position.

The returned string has the format *uri* `:  line` *i* `, column` *j*, where *uri* is the URI, *i* is the line number, and *j* is the column number. If the URI is not known then the URI and the colon and space characters that follow it are omitted; if the column number is not known then the column number and the comma and space characters that precede it are omitted.

***returns*** A string describing this position

**`String uriString();`**

Returns this position’s source URI.

***returns*** This position’s source URI, or `null` if it is not known

## XMLScanner javax.xml.marshal

```
abstract class XMLScanner {
         static final int  WS_COLLAPSE;
         static final int  WS_NORMALIZE;
         static final int  WS_PRESERVE;
                           XMLScanner();
         abstract boolean  atAttribute();
         abstract boolean  atAttributeValue();
         abstract boolean  atAttributeValueToken();
         abstract boolean  atChars(int whitespace) throws ScanException;
         abstract boolean  atEnd() throws ScanException;
         abstract boolean  atEndOfDocument()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
         abstract boolean  atEnd(String name) throws ScanException;
         abstract boolean  atStart() throws ScanException;
         abstract boolean  atStart(String name) throws ScanException;
            abstract void  close() throws ScanIOException;
        static XMLScanner  open(java.io.InputStream in) throws ScanException;
        static XMLScanner  open(org.w3c.dom.Document doc) throws ScanException;
          abstract String  peekStart()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
    abstract ScanPosition  position();
          abstract String  takeAttributeName()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
             final String  takeAttributeValue()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
          abstract String  takeAttributeValue(int whitespace)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
          abstract String  takeAttributeValueToken()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
          abstract String  takeChars(int whitespace)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
               final void  takeEmpty(String name)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
          abstract String  takeEnd()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
            abstract void  takeEndOfDocument()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
            abstract void  takeEnd(String name)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
             final String  takeLeaf(String name, int whitespace)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
          abstract String  takeStart()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
            abstract void  takeStart(String name)
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
            abstract void  tokenizeAttributeValue()
                               throws javax.xml.bind.InvalidContentException,
                                   ScanException;
}
```

A scanner of XML input streams or data structures.

When unmarshalling XML into a content tree it is not necessary to use a full-fledged XML parser because schema-derived classes enforce all validity constraints as well as the following non-local well-formedness constraints of the XML 1.0 specification :

- *Element type match:* The name in an element's end-tag must match the element type in its start-tag.
- *Unique Att Spec:* No attribute name may appear more than once in the same start-tag or empty-element tag.

An XML scanner therefore enforces only the remaining lexical well-formedness constraints of XML 1.0.

An XML scanner is in one of the following states:

- *Start:* The scanner is positioned at a start tag. This state may be followed by the *AttributeName*, *Chars*, *Start*, or *End* states. An empty tag, that is, a tag of the form **`<foo/>`**, will yield the *Start* state followed by the *End* state, possibly with some intervening attribute states.
- *AttributeName:* The scanner is positioned at an attribute name. This state may be followed only by the *AttributeValue* state. This state will be entered exactly once for each attribute that is read. Attributes are read in the order in which they appear in the input document.
- *AttributeValue:* The scanner is positioned at an attribute value. This state may be followed by the *AttributeName*, *Chars*, *Start*, or *End* states. If the **`tokenizeAttributeValue`** method is invoked then this state may also be followed by the *AttributeValueToken* state.
- *AttributeValueToken:* The scanner is positioned at one of the tokens of a tokenized attribute value. This state may be followed by the *AttributeValueToken*, *AttributeName*, *Chars*, *Start*, or *End* states.
- *Chars:* The scanner is positioned at some character content. This state may be followed by the *Start* or *End* states.
- *End:* The scanner is positioned at an end tag. This state may be followed by the *Chars*, *Start*, *End*, or *EndOfDocument* states.
- *EndOfDocument:* The scanner has reached the end of the input document, at which point it closes itself. The state of the scanner will not change after it reaches this state.

For each state *Foo* there is at least one of each of the following kinds of methods:

- Methods named **`at`***Foo* return a boolean value indicating whether the scanner is in the *Foo* state and possibly whether some other condition holds.

- Methods named **`take`** *Foo* check that the scanner is in the *Foo* state; if so, a relevant value is returned and the scanner is (in most cases) advanced to the next state. A **`javax.xml.bind.InvalidContentException`** is thrown if the scanner is not in the ***`Foo`*** state.

The methods for reading attribute values and character data take a **`whitespace`** parameter, one of the constants **`WS_COLLAPSE`**, **`WS_NORMALIZE`**, or **`WS_PRESERVE`**, indicating how whitespace is to be processed. Whitespace is defined here exactly as in the XML 1.0 specification: A whitespace character is one of TAB (**`'\u0009'`**), LINE FEED (**`'\u000A'`**), CARRIAGE RETURN (**`'\u000D'`**), or SPACE (**`'\u0020'`**).

This class also defines a method for retrieving the scanner's position and factory methods for creating scanners that read byte-input streams and scanners that read DOM trees.

**`XMLScanner();`**

**`abstract boolean atAttribute();`**

Tests whether the scanner is positioned at an attribute name.
***returns*** **`true`** if the scanner's state is *AttributeName*

**`abstract boolean atAttributeValue();`**

Tests whether the scanner is positioned at an attribute value.
***returns*** **`true`** if the scanner's state is *AttributeValue*

**`abstract boolean`**
**`atAttributeValueToken();`**

Tests whether the scanner is positioned at an attribute-value token.
***returns*** **`true`** if the scanner's state is **`AttributeValueToken`**

**`abstract boolean atChars(int whitespace) throws ScanException;`**

Tests whether the scanner is positioned at some character data.

If the value of the **`whitespace`** parameter is **`WS_COLLAPSE`** then any initial whitespace is first skipped.

***param*** **`whitespace`** Determines how whitespace in the character data will be handled; must be one of **`WS_COLLAPSE`**, **`WS_NORMALIZE`**, or **`WS_PRESERVE`**
***returns*** **`true`** if the scanner's state is *Chars*
***throws*** **`IllegalStateException`** If this method has already been invoked for the current state but with a different value for the **`whitespace`** parameter
***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

**`abstract boolean atEnd() throws ScanException;`**

Skips whitespace, if any, and then tests whether the scanner is positioned at an end tag.

***returns*** **`true`** if, after skipping whitespace, the scanner's state is *End*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract boolean
    atEndOfDocument()
        throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, and then tests whether the scanner has reached the end of the input document.

***returns*** **`true`** if, after skipping whitespace, the scanner's state is *End*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

**`abstract boolean atEnd(String name) throws ScanException;`**

Skips whitespace, if any, and then tests whether the scanner is positioned at an end tag with the given name.

***param*** **`name`** The element name to be tested

***returns*** **`true`** if, after skipping whitespace, the scanner's state is *End* and the name in the tag is equal to **`name`**

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

**`abstract boolean atStart() throws ScanException;`**

Skips whitespace, if any, and then tests whether the scanner is positioned at a start tag.

***returns*** **`true`** if, after skipping whitespace, the scanner's state is *Start*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

**`abstract boolean atStart(String name) throws ScanException;`**

Skips whitespace, if any, and then tests whether the scanner is positioned at a start tag with the given name.

***param*** **`name`** The element name to be tested

***returns*** **`true`** if, after skipping whitespace, the scanner's state is *Start* and the name in the tag is equal to **`name`**

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

**`abstract void close() throws ScanIOException;`**

Closes this scanner.

If this scanner was created from a byte-input stream then the input stream is closed.

***throws*** **`ScanIOException`** If an I/O error occurs

```
static XMLScanner open(java.io.InputStream in)
    throws ScanException;
```

Creates a new scanner that reads an XML document from the given input stream.

***param*** **`in`** The input stream to be scanned

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
static XMLScanner open(org.w3c.dom.Document doc)
    throws ScanException;
```

Creates a new scanner that scans the given DOM tree.

***param*** **`doc`** The document to be scanned

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String peekStart()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, and then reads the current start tag.

***returns*** The name in the current start tag

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *Start*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract ScanPosition position();
```

Returns a new scan-position object reporting the scanner's current position.

***returns*** A scan-position object

```
abstract String takeAttributeName()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current attribute name and then advances the scanner to the next state.

***returns*** The current attribute name

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *AttributeName*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
final String takeAttributeValue()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current attribute value, collapsing whitespace, and then advances the scanner to the next state.

An invocation of this method behaves in exactly the same way as an invocation of the **`takeAttributeValue(int)`** method, passing **`WS_COLLAPSE`** for the **`whitespace`** argument.

***returns*** The current attribute value

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *AttributeValue*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String takeAttributeValue(int whitespace)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current attribute value and then advances the scanner to the next state.

***param*** **`whitespace`** Determines how whitespace in the attribute value will be handled; must be one of **`WS_COLLAPSE`**, **`WS_NORMALIZE`**, or **`WS_PRESERVE`**

***returns*** The current attribute value

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *AttributeValue*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String
    takeAttributeValueToken()
        throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current attribute-value token and then advances the scanner to the next state.

***returns*** The current attribute-value token

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *AttributeValueToken*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String takeChars(int whitespace)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current character data and then advances the scanner to the next state.

***param*** **`whitespace`** Determines how whitespace in the character data will be handled; must be one of **`WS_COLLAPSE`**, **`WS_NORMALIZE`**, or **`WS_PRESERVE`**

***returns*** The current character data

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *Chars*

***throws*** **`IllegalStateException`** If the **`atChars`** method has already been invoked for the current state but with a different value for the **`whitespace`** parameter

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
final void takeEmpty(String name)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Takes an empty tag.

This method takes a start tag with the given name and then takes an end tag with the same name.

***param*** **`name`** The element name of the expected start and end tags

***throws*** **`javax.xml.bind.InvalidContentException`** If the expected tags cannot be scanned

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String takeEnd()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, reads the current end tag, and then advances the scanner to the next state.

***returns*** The name in the current end tag

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *End*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract void takeEndOfDocument()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, and then checks that the scanner has reached the end of the input document.

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *End*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract void takeEnd(String name)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, checks that the current end tag's name is equal to the given name, and then advances the scanner to the next state.

***param*** **`name`** The element name to be scanned

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *End* and the name in the tag is not equal to **`name`**

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
final String takeLeaf(String name, int whitespace)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Takes a simple leaf element.

This method takes start tag with the given name, takes a sequence of characters, if present, takes an end tag with the given name, and then returns the character data, if any.

***param*** **`name`** The element name of the expected start and end tags

***param*** **`whitespace`** Determines how whitespace in the character data will be handled; must be one of **`WS_COLLAPSE`**, **`WS_NORMALIZE`**, or **`WS_PRESERVE`**

***returns*** The character data, or **`null`** if no character data was scanned

***throws*** **`javax.xml.bind.InvalidContentException`** If the expected tags and the character data cannot be scanned

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract String takeStart()
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, reads the current start tag, and then advances the scanner to the next state.

***returns*** The name in the current start tag

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *Start*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract void takeStart(String name)
    throws javax.xml.bind.InvalidContentException, ScanException;
```

Skips whitespace, if any, checks that the current start tag's name is equal to the given name, and then advances the scanner to the next state.

***param*** **`name`** The element name to be scanned

***returns*** The name in the current start tag

***throws*** **`javax.xml.bind.InvalidContentException`** If, after skipping whitespace, the scanner's state is not *Start* and the name in the tag is not equal to **`name`**

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

```
abstract void
    tokenizeAttributeValue()
        throws javax.xml.bind.InvalidContentException, ScanException;
```

Reads the current attribute's value as a sequence of non-whitespace tokens, returning them in succeeding *AttributeValueToken* states. If the current attribute's value is only whitespace then the next state will not be *AttributeValueToken*.

***throws*** **`javax.xml.bind.InvalidContentException`** If the scanner's state is not *AttributeValue*

***throws*** **`ScanException`** If input that is not lexically well-formed is scanned, or if an I/O error occurs

## XMLWriter javax.xml.marshal

```
 class XMLWriter {
          XMLWriter(java.io.OutputStream out) throws java.io.IOException;
          XMLWriter(java.io.OutputStream out, String enc)
              throws java.io.UnsupportedEncodingException, java.io.IOException;
          XMLWriter(java.io.OutputStream out, String enc, boolean declare)
              throws java.io.UnsupportedEncodingException, java.io.IOException;
    void  attributeName(String name) throws java.io.IOException;
    void  attribute(String name, String value) throws java.io.IOException;
    void  attributeValue(String value) throws java.io.IOException;
    void  attributeValueToken(String token) throws java.io.IOException;
    void  chars(String chars) throws java.io.IOException;
    void  close() throws java.io.IOException;
    void  doctype(String root, String dtd) throws java.io.IOException;
    void  end(String name) throws java.io.IOException;
    void  flush() throws java.io.IOException;
    void  inlineLeaf(String name) throws java.io.IOException;
    void  inlineLeaf(String name, String chars) throws java.io.IOException;
    void  leaf(String name) throws java.io.IOException;
    void  leaf(String name, String chars) throws java.io.IOException;
    void  setQuote(char quote);
    void  start(String name) throws java.io.IOException;
}
```

A writer of XML output streams.

An XML writer knows hardly anything about XML document well-formedness, to say nothing of validity. It relies upon the invoker to ensure that the generated document is well-formed and, if required, valid.

Note: This class is incomplete. In the next draft it will be extended to more closely mimic **XMLScanner**, and to support output to both SAX streams and DOM documents.

### XMLWriter(java.io.OutputStream out) throws java.io.IOException;

Creates a new writer that will write to the given byte-output stream using the UTF-8 encoding. An initial XML declaration will be written to the stream.

| | | |
|---|---|---|
| ***param*** | **out** | The target byte-output stream |
| ***throws*** | **java.io.IOException** | If an I/O error occurs |

### XMLWriter(java.io.OutputStream out, String enc) throws java.io.UnsupportedEncodingException, java.io.IOException;

Creates a new writer that will write to the given byte-output stream using the given encoding. An initial XML declaration will be written to the stream.

| | | |
|---|---|---|
| ***param*** | **out** | The target byte-output stream |
| ***param*** | **enc** | The character encoding to be used |
| ***throws*** | **java.io.IOException** | If an I/O error occurs |
| ***throws*** | **java.io.UnsupportedEncodingException** | If the named encoding is not supported |

```
XMLWriter(java.io.OutputStream out, String enc, boolean declare)
    throws java.io.UnsupportedEncodingException,
        java.io.IOException;
```

Creates a new writer that will write to the given byte-output stream using the given encoding. An initial XML declaration will optionally be written to the stream.

***param*** **`out`** The target byte-output stream

***param*** **`enc`** The character encoding to be used

***param*** **`declare`** If **`true`**, write the XML declaration to the output stream

***throws*** **`java.io.IOException`** If an I/O error occurs

***throws*** **`java.io.UnsupportedEncodingException`** If the named encoding is not supported

```
void attributeName(String name) throws java.io.IOException;
```

Writes an attribute name for the current element. After invoking this method, invoke the **`attributeValue`** method to write the attribute value, or invoke the **`attributeValueToken`** method to write one or more space-separated value tokens.

***param*** **`name`** The attribute's name

***throws*** **`IllegalStateException`** If the previous method invoked upon this object was neither **`start`** nor **`attribute`**

```
void attribute(String name, String value)
    throws java.io.IOException;
```

Writes an attribute for the current element.

***param*** **`name`** The attribute's name

***param*** **`value`** The attribute's value

***throws*** **`IllegalStateException`** If the previous method invoked upon this object was neither **`start`** nor **`attribute`**

***throws*** **`java.io.IOException`** If an I/O error occurs

```
void attributeValue(String value) throws java.io.IOException;
```

Writes a value for the current attribute.

***param*** **`value`** The attribute's value

***throws*** **`IllegalStateException`** If the previous method invoked upon this object was not **`attributeName`**

```
void attributeValueToken(String token) throws java.io.IOException;
```

Writes one token of the current attribute's value. Adjacent tokens will be separated by single space characters.

***param*** **`token`** The token to be written

***throws*** **`IllegalStateException`** If the previous method invoked upon this object was neither **`attributeName`** nor **`attributeValueToken`**

**`void chars(String chars) throws java.io.IOException;`**

Writes some character data.

***param*** **`chars`** The character data to be written

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void close() throws java.io.IOException;`**

Flushes the writer and closes the underlying byte-output stream.

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void doctype(String root, String dtd) throws java.io.IOException;`**

Writes a DOCTYPE declaration.

***param*** **`root`** The name of the root element

***param*** **`dtd`** The URI of the document-type definition

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void end(String name) throws java.io.IOException;`**

Writes an end tag for the named element.

***param*** **`name`** The name to be used in the end tag

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void flush() throws java.io.IOException;`**

Flushes the writer.

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void inlineLeaf(String name) throws java.io.IOException;`**

```
void inlineLeaf(String name, String chars)
    throws java.io.IOException;
```

**`void leaf(String name) throws java.io.IOException;`**

Writes an empty leaf element.

***param*** **`The`** name to be used in the empty-element tag

**`void leaf(String name, String chars) throws java.io.IOException;`**

Writes a leaf element with the given character content.

***param*** **`name`** The name to be used in the start and end tags

***param*** **`chars`** The character data to be written

This method writes a start tag with the given name, followed by the given character data, followed by an end tag. If the **`chars`** parameter is **`null`** or the empty string then an empty tag is written.

***throws*** **`java.io.IOException`** If an I/O error occurs

**`void setQuote(char quote);`**

Sets the quote character to be used by this writer when writing attribute values.

***param*** **`quote`** The new quote character, either a QUOTATION MARK (**`'\u0022'`**), or an APOSTROPHE-QUOTE (**`'\u0027'`**)

***throws*** **`IllegalArgumentException`** If the argument is neither of the above characters

**`void start(String name) throws java.io.IOException;`**

Writes a start tag for the named element.

***param*** **`name`** The name to be used in the start tag

***throws*** **`java.io.IOException`** If an I/O error occurs

## B.2 Checked exceptions

### EndOfDocumentException javax.xml.marshal

```
 class EndOfDocumentException
    extends ScanException // javax.xml.bind.UnmarshalException
{
      EndOfDocumentException(ScanPosition pos);
}
```

Checked exception thrown by an XML scanner when it unexpectedly reaches the end of an input document.

**`EndOfDocumentException(ScanPosition pos);`**

Constructs an instance of this class.

***param*** **`pos`** The position at which the error was detected

### ScanException javax.xml.marshal

```
 class ScanException
    extends javax.xml.bind.UnmarshalException // Exception
{
    protected  ScanException(ScanPosition pos);
               ScanException(ScanPosition pos, String desc);
       String  toString();
}
```

Checked exception thrown by an XML scanner when it encounters input that is not lexically well-formed or when an I/O error occurs.

**protected ScanException(ScanPosition pos);**

Initializes a new instance of this class.
***param*** **pos** The position at which the error was detected

**ScanException(ScanPosition pos, String desc);**

Constructs a new instance of this class.
***param*** **pos** The position at which the error was detected
***param*** **desc** A string describing the error

**String toString();**

Returns a string describing this exception that includes the position at which the error was detected, if available.
***returns*** A string describing this exception

## ScanIOException javax.xml.marshal

```
 class ScanIOException
    extends ScanException // javax.xml.bind.UnmarshalException
{
                          ScanIOException(ScanPosition pos,
                                          java.io.IOException iox);
     java.io.IOException  getIOException();
                    void  printStackTrace(java.io.PrintStream out);
}
```

Checked exception thrown by an XML scanner when an I/O error occurs. The I/O exception that is thrown is wrapped within this exception.

**ScanIOException(ScanPosition pos, java.io.IOException iox);**

Constructs a new instance of this class.
***param*** **pos** The position at which the I/O error occurred
***param*** **iox** The **java.io.IOException** that was thrown

**java.io.IOException**
**getIOException();**

Retrieves the **java.io.IOException** that caused this exception.
***returns*** The I/O exception

**void printStackTrace(java.io.PrintStream out);**

# C DTD for the binding language

```
<?xml version="1.0" encoding="US-ASCII"?>

<!--
  JAXB XML/Java binding-schema DTD
  URI: http://java.sun.com/dtd/jaxb/1.0-ea/xjs.dtd
  @(#)xjs.dtd   1.11 01/05/31

  Copyright 2000-2001 by Sun Microsystems, Inc.,
  901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
  All rights reserved.

  This software is the confidential and proprietary information
  of Sun Microsystems, Inc. ("Confidential Information").  You
  shall not disclose such Confidential Information and shall use
  it only in accordance with the terms of the license agreement
  you entered into with Sun.
 !-->

<!ENTITY % top-level-decl "element | interface | enumeration | conversion" >
<!ENTITY % internal-decl "constructor | enumeration | conversion" >

<!ELEMENT xml-java-binding-schema ( options?, ( %top-level-decl; )* ) >
<!ATTLIST xml-java-binding-schema
          version CDATA #FIXED "1.0ea" >

<!ENTITY % boolean "( true | false )" >
<!ENTITY % collection "( array | list )" >

<!ELEMENT options EMPTY >
<!ATTLIST options
          package NMTOKEN #IMPLIED
          default-reference-collection-type %collection; "list"
          property-get-set-prefixes %boolean; "true"
          marshallable %boolean; "true"
          unmarshallable %boolean; "true" >

<!ENTITY % attribute-or-internal-decl "attribute | %internal-decl;" >

<!ELEMENT element ( ( %attribute-or-internal-decl; )*,
                    content?,
                    ( %attribute-or-internal-decl; )* ) >
<!ATTLIST element
          name ID #REQUIRED
          type ( value | class ) #REQUIRED
          convert NMTOKEN #IMPLIED
          class NMTOKEN #IMPLIED
          root %boolean; #IMPLIED >

<!--
  Further constraints: @convert requires @type="value"
                       @class, @root require @type="class"
                       ./* requires @type="class"
  <element name={elt} type="value" [convert={cnv}]
  <element name={elt} type="class" [class={class}] [root={root}]>
 !-->

<!ELEMENT attribute EMPTY >
<!ATTLIST attribute
          name NMTOKEN #REQUIRED
          convert NMTOKEN #IMPLIED
          property NMTOKEN #IMPLIED
          collection %collection; #IMPLIED >

<!--
  <attribute name={attr} [convert={cnv}]
             [property={prop}] [collection={coll}]
 !-->

<!ELEMENT content ( ( element-ref | choice | sequence )*, rest? ) >
<!ATTLIST content
          property NMTOKEN #IMPLIED
          collection %collection; #IMPLIED
          supertype NMTOKEN #IMPLIED >

<!--
  Further constraints: @property, @collection, @supertype forbid content

  <content property={prop} [collection={coll}] [supertype={tau}]/>
  <content>
    <element-ref name={elt} [property={prop}] [collection={coll}]/>
    <choice property={prop} [collection={coll}] [supertype={tau}]/>
    <sequence property={prop} [collection={coll}] [supertype={tau}]/>
    <rest property={prop} [collection={coll}] [supertype={tau}]/>
 !-->

<!ELEMENT element-ref EMPTY >
<!ATTLIST element-ref
          name NMTOKEN #REQUIRED
          property NMTOKEN #IMPLIED
          collection %collection; #IMPLIED >

<!ELEMENT choice EMPTY >
<!ATTLIST choice
          property NMTOKEN #REQUIRED
          collection %collection; #IMPLIED
          supertype NMTOKEN #IMPLIED >

<!ELEMENT sequence EMPTY >
<!ATTLIST sequence
          property NMTOKEN #IMPLIED
          collection %collection; #IMPLIED
          supertype NMTOKEN #IMPLIED >

<!ELEMENT rest EMPTY >
<!ATTLIST rest
          property NMTOKEN #REQUIRED
          collection %collection; #IMPLIED
          supertype NMTOKEN #IMPLIED >
```

```
<!-- Declarations -->

<!ELEMENT constructor EMPTY >
<!ATTLIST constructor
          properties NMTOKENS #REQUIRED >

<!ELEMENT interface EMPTY >
<!ATTLIST interface
          name NMTOKEN #REQUIRED
          members NMTOKENS #REQUIRED
          properties NMTOKENS #IMPLIED >

<!ELEMENT enumeration EMPTY >
<!ATTLIST enumeration
          name CDATA #REQUIRED
          members NMTOKENS #REQUIRED >

<!ELEMENT conversion EMPTY>
<!ATTLIST conversion
          name NMTOKEN #REQUIRED
          type NMTOKEN #IMPLIED
          parse NMTOKEN #IMPLIED
          print NMTOKEN #IMPLIED >

<!--
  @parse = "new", to invoke a constructor that takes a string, or
           "ClassName.staticParseMethod"
  @print = "ClassName.staticPrintMethod" or "instancePrintMethod"
  In both cases, ClassName may have a package prefix
 !-->
```

# D Examples

The first example is the stock-trade example that is discussed throughout this specification. The second example, on p. 123, illustrates how collection properties and **`ID`**/**`IDREF`** properties can be implemented.

## D.1 Simple stock trades

### D.1.1 DTD

```
<!ELEMENT trade ( symbol, quantity, limit?, stop?, date ) >

<!ATTLIST trade
          account CDATA #REQUIRED
          action ( buy | buy-to-cover | sell | sell-short ) #REQUIRED
          duration ( immediate | day | good-til-canceled ) "day" >

<!ELEMENT symbol (#PCDATA) >
<!ELEMENT quantity (#PCDATA) >
<!ELEMENT limit (#PCDATA) >
<!ELEMENT stop (#PCDATA) >
<!ELEMENT date (#PCDATA) >
```

### D.1.2 Binding schema

```
<xml-java-binding-schema version="1.0-ea">

  <conversion name="price" type="java.math.BigDecimal"/>
  <conversion name="date" type="java.util.Date"
              parse="Cnv.parseDate" print="Cnv.printDate"/>

  <element name="symbol" type="value"/>
  <element name="quantity" type="value" convert="int"/>
  <element name="limit" type="value" convert="price"/>
  <element name="stop" type="value" convert="price"/>
  <element name="date" type="value" convert="date"/>

  <element name="trade" type="class" root="true">
    <content>
      <element-ref name="symbol"/>
      <element-ref name="quantity"/>
      <element-ref name="limit" property="limit-price"/>
      <element-ref name="stop" property="stop-price"/>
      <element-ref name="date"/>
    </content>

  <!-- Constructor declarations not yet implemented
    <constructor properties="account action symbol quantity"/>
    -->
    <enumeration name="Action"
                 members="buy buy-to-cover sell sell-short"/>
    <enumeration name="Duration"
                 members="immediate day good-til-canceled"/>

    <attribute name="account" convert="int"/>
    <attribute name="action"/>
    <attribute name="duration"/>
  </element>
</xml-java-binding-schema>
```

### D.1.3 The `Trade` class

The following code could be generated by one particular implementation of this specification; other implementations may produce somewhat different code.

```
import java.io.IOException;
import java.io.InputStream;
import java.math.BigDecimal;
import java.text.ParseException;
import java.util.Date;
import javax.xml.bind.ConversionException;
import javax.xml.bind.Dispatcher;
import javax.xml.bind.DuplicateAttributeException;
import javax.xml.bind.IllegalEnumerationValueException;
import javax.xml.bind.InvalidAttributeException;
import javax.xml.bind.LocalValidationException;
import javax.xml.bind.MarshallableRootElement;
import javax.xml.bind.Marshaller;
import javax.xml.bind.MissingAttributeException;
import javax.xml.bind.MissingContentException;
import javax.xml.bind.NoValueException;
import javax.xml.bind.RootElement;
import javax.xml.bind.StructureValidationException;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.Validator;
import javax.xml.marshal.XMLScanner;
import javax.xml.marshal.XMLWriter;

public class Trade
    extends MarshallableRootElement
    implements RootElement
{

    private boolean has_Account = false;
    private int _Account;
    private Action _Action;
    private final static Duration DEFAULTED_DURATION = Duration.parse("day");
    private Duration _Duration;
    private String _Symbol;
    private boolean has_Quantity = false;
    private int _Quantity;
    private BigDecimal _LimitPrice;
    private BigDecimal _StopPrice;
    private Date _Date;
    public Trade() { }

    public Trade(int _Account, Action _Action, String _Symbol, int _Quantity) {
        setAccount(_Account);
        setAction(_Action);
        setSymbol(_Symbol);
        setQuantity(_Quantity);
    }

    public int getAccount() {
        if (has_Account) return _Account;
        throw new NoValueException("account");
    }

    public void setAccount(int _Account) {
        this._Account = _Account;
        has_Account = true;
    }

    public boolean hasAccount() { return has_Account; }

    public void deleteAccount() {
        has_Account = false;
        invalidate();
    }

    public Action getAction() { return _Action; }

    public void setAction(Action _Action) {
        this._Action = _Action;
        if (_Action == null) invalidate();
    }

    public Duration getDuration() {
        if (_Duration == null) return DEFAULTED_DURATION;
        return _Duration;
    }

    public void setDuration(Duration _Duration) {
        this._Duration = _Duration;
        if (_Duration == null) invalidate();
    }

    public boolean defaultedDuration() { return _Duration != null; }

    public String getSymbol() { return _Symbol; }

    public void setSymbol(String _Symbol) {
        this._Symbol = _Symbol;
        if (_Symbol == null) invalidate();
    }

    public int getQuantity() {
        if (has_Quantity) return _Quantity;
        throw new NoValueException("quantity");
    }
    public void setQuantity(int _Quantity) {
        this._Quantity = _Quantity;
        has_Quantity = true;
    }

    public boolean hasQuantity() { return has_Quantity; }

    public void deleteQuantity() {
        has_Quantity = false;
        invalidate();
    }

    public BigDecimal getLimitPrice() { return _LimitPrice; }

    public void setLimitPrice(BigDecimal _LimitPrice) {
        this._LimitPrice = _LimitPrice;
        invalidate();
    }

    public BigDecimal getStopPrice() { return _StopPrice; }

    public void setStopPrice(BigDecimal _StopPrice) {
        this._StopPrice = _StopPrice;
        invalidate();
    }

    public Date getDate() { return _Date; }

    public void setDate(Date _Date) {
        this._Date = _Date;
        if (_Date == null) invalidate();
    }

    // Declared in javax.xml.bind.ValidatableObject
    public void validateThis()
        throws LocalValidationException
    {
        if (!has_Account) throw new MissingAttributeException("account");
        if (_Action == null) throw new MissingAttributeException("action");
        if (_Symbol == null) throw new MissingContentException("symbol");
        if (!has_Quantity) throw new MissingContentException("quantity");
        if (_Date == null) throw new MissingContentException("date");
    }

    public void validate(Validator v)
        throws StructureValidationException
    {
        // The DTD does not describe any global structural constraints
    }

    // Declared in javax.xml.bind.MarshallableRootElement
    public void marshal(Marshaller m)
        throws IOException
    {
        XMLWriter w = m.writer();
        w.start("trade");
        w.attribute("account", Integer.toString(_Account));
        if (_Action != null) w.attribute("action", _Action.toString());
        if (_Duration != null) w.attribute("duration", _Duration.toString());
        w.leaf("symbol", _Symbol.toString());
        w.leaf("quantity", Integer.toString(_Quantity));
        if (_LimitPrice != null) w.leaf("limit", _LimitPrice.toString());
        if (_StopPrice != null) w.leaf("stop", _StopPrice.toString());
        w.leaf("date", Cnv.printDate(_Date));
        w.end("trade");
    }

    public void unmarshal(Unmarshaller u)
        throws UnmarshalException
    {
        XMLScanner xs = u.scanner();
        Validator v = u.validator();
        xs.takeStart("trade");
        while (xs.atAttribute()) {
            String an = xs.takeAttributeName();
            String av = xs.takeAttributeValue(XMLScanner.WS_COLLAPSE);
            if (an.equals("account")) {
                if (has_Account)
                    throw new DuplicateAttributeException(an);
                try {
                    _Account = Integer.parseInt(av);
                } catch (Exception x) {
                    throw new ConversionException("account", x);
                }
                has_Account = true;
                continue;
            }
            if (an.equals("action")) {
                if (_Action != null)
                    throw new DuplicateAttributeException(an);
                try {
                    _Action = Action.parse(av);
                } catch (Exception x) {
                    throw new ConversionException("action", x);
                }
                continue;
            }
            if (an.equals("duration")) {
                if (_Duration != null)
                    throw new DuplicateAttributeException(an);
                try {
                    _Duration = Duration.parse(av);
                } catch (Exception x) {
                    throw new ConversionException("duration", x);
                }
                continue;
            }
            throw new InvalidAttributeException(an);
        }
        _Symbol = xs.takeLeaf("symbol", XMLScanner.WS_COLLAPSE);
        String s = xs.takeLeaf("quantity", XMLScanner.WS_COLLAPSE);
        try {
            _Quantity = Integer.parseInt(s);
        } catch (Exception x) {
            throw new ConversionException("quantity", x);
        }
        has_Quantity = true;
        if (xs.atStart("limit")) {
            s = xs.takeLeaf("limit", XMLScanner.WS_COLLAPSE);
            try {
                _LimitPrice = new BigDecimal(s);
            } catch (Exception x) {
                throw new ConversionException("limit", x);
            }
        }
        if (xs.atStart("stop")) {
            s = xs.takeLeaf("stop", XMLScanner.WS_COLLAPSE);
            try {
                _StopPrice = new BigDecimal(s);
            } catch (Exception x) {
                throw new ConversionException("stop", x);
            }
        }
        s = xs.takeLeaf("date", XMLScanner.WS_COLLAPSE);
        try {
            _Date = Cnv.parseDate(s);
        } catch (Exception x) {
            throw new ConversionException("date", x);
        }
        xs.takeEnd("trade");
    }

    // Specified in javax.xml.bind.MarshallableObject
    public static Trade unmarshal(InputStream in)
        throws UnmarshalException
    {
        return unmarshal(XMLScanner.open(in));
    }

    public static Trade unmarshal(XMLScanner xs)
        throws UnmarshalException
    {
        return unmarshal(xs, newDispatcher());
    }

    public static Trade unmarshal(XMLScanner xs, Dispatcher d)
        throws UnmarshalException
    {
        return ((Trade)d.unmarshal(xs, Trade.class));
    }

    public static Dispatcher newDispatcher() {
        Dispatcher d = new Dispatcher();
        d.register("trade", Trade.class);
        d.freezeElementNameMap();
        return d;
    }

    public boolean equals(Object ob) {
        if (this == ob) return true;
        if (!(ob instanceof Trade)) return false;
        Trade tob = (Trade)ob;
        if (has_Account) {
            if (!tob.has_Account) return false;
            if (_Account != tob._Account) return false;
        } else if (tob.has_Account) return false;
        if (_Action != tob._Action) return false;
        if (_Duration != tob._Duration) return false;
        if (_Symbol != null) {
            if (tob._Symbol == null) return false;
            if (!_Symbol.equals(tob._Symbol)) return false;
        } else if (tob._Symbol != null) return false;
        if (has_Quantity) {
            if (!tob.has_Quantity) return false;
            if (_Quantity != tob._Quantity) return false;
        } else if (tob.has_Quantity) return false;
        if (_LimitPrice != null) {
            if (tob._LimitPrice == null) return false;
            if (!_LimitPrice.equals(tob._LimitPrice)) return false;
        } else if (tob._LimitPrice != null) return false;
        if (_StopPrice != null) {
            if (tob._StopPrice == null) return false;
            if (!_StopPrice.equals(tob._StopPrice)) return false;
        } else if (tob._StopPrice != null) return false;
        if (_Date != null) {
            if (tob._Date == null) return false;
            if (!_Date.equals(tob._Date)) return false;
        } else if (tob._Date != null) return false;
        return true;
    }

    public int hashCode() {
        int h = 0;
        h = (31 * h) + ((has_Account) ? _Account: 0);
        h = (127 * h) + ((_Action != null) ? _Action.hashCode() : 0);
        h = (127 * h) + ((_Duration != null) ? _Duration.hashCode() : 0);
        h = (127 * h) + ((_Symbol != null) ? _Symbol.hashCode() : 0);
        h = (31 * h) + ((has_Quantity) ? _Quantity : 0);
        h = (127 * h) + ((_LimitPrice != null) ? _LimitPrice.hashCode() : 0);
        h = (127 * h) + ((_StopPrice != null) ? _StopPrice.hashCode() : 0);
        h = (127 * h) + ((_Date != null) ? _Date.hashCode() : 0);
        return h;
    }

    public String toString() {
        StringBuffer sb = new StringBuffer("<<trade");
        if (has_Account) {
            sb.append(" account=");
            sb.append(Integer.toString(_Account));
        }
        if (_Action != null) {
            sb.append(" action=");
            sb.append(_Action.toString());
        }
        sb.append(" duration=");
        sb.append(_Duration.toString());
        if (_Symbol != null) {
            sb.append(" symbol=");
            sb.append(_Symbol.toString());
        }
        if (has_Quantity) {
            sb.append(" quantity=");
            sb.append(_Quantity);
        }
        if (_LimitPrice != null) {
            sb.append(" limit=");
            sb.append(_LimitPrice.toString());
        }
        if (_StopPrice != null){
            sb.append(" stop=");
            sb.append(_LimitPrice.toString());
        }
        if (_Date != null) {
            sb.append(" date=");
            sb.append(Cnv.printDate(_Date));
        }
        sb.append(">>");
        return sb.toString();
    }

    public final static class Action {

        private String _Action;
        public final static Action BUY = new Action("buy");
        public final static Action BUY_TO_COVER = new Action("buy-to-cover");
        public final static Action SELL = new Action("sell");
        public final static Action SELL_SHORT = new Action("sell-short");

        private Action(String _Action) { this._Action = _Action; }

        public static Action parse(String _Action) {
            if (_Action.equals("buy")) return BUY;
            if (_Action.equals("buy-to-cover")) return BUY_TO_COVER;
            if (_Action.equals("sell")) return SELL;
            if (_Action.equals("sell-short")) return SELL_SHORT;
            throw new IllegalEnumerationValueException(_Action);
        }

        public String toString() { return _Action; }

        public boolean equals(Object ob) { return ob == this; }

        public int hashCode() { return _Action.hashCode(); }

    }

    public final static class Duration {

        private String _Duration;
        public final static Duration DAY = new Duration("day");
        public final static Duration GOOD_TIL_CANCELED
            = new Duration("good-til-canceled");

        private Duration(String _Duration) { this._Duration = _Duration; }

        public static Duration parse(String _Duration) {
            if (_Duration.equals("day")) return DAY;
            if (_Duration.equals("good-til-canceled"))
                return GOOD_TIL_CANCELED;
            throw new IllegalEnumerationValueException(_Duration);
        }

        public String toString() { return _Duration; }

        public boolean equals(Object ob) { return ob == this; };

        public int hashCode() { return _Duration.hashCode(); }

    }

}
```

## D.2 Tree nodes

### D.2.1 DTD

```
<!ELEMENT node ( node* ) >
<!ATTLIST node
          name ID #REQUIRED
          ref IDREF #IMPLIED >
```

### D.2.2 Binding schema

```
<xml-java-binding-schema version="1.0-ea">

  <options property-get-set-prefixes="false"/>

  <element name="node" type="class" root="true">
    <content>
      <element-ref name="node" property="children"/>
    </content>

  <!-- Constructor declarations not yet implemented
    <constructor properties="children"/>
    -->

  </element>

</xml-java-binding-schema>
```

### D.2.3 Sample valid document

```
<node name="root">
  <node name="usr">
    <node name="usr-bin"/>
    <node name="usr-lib">
      <node name="usr-lib-libc.so" ref="lib-libc.so.6.2"/>
      <node name="usr-lib-uucp"/>
    </node>
    <node name="usr-man"/>
  </node>
  <node name="etc">
```

```
    <node name="etc-rc.d"/>
    <node name="etc-inetd.conf"/>
  </node>
  <node name="lib">
    <node name="lib-libc.so.6.2"/>
  </node>
</node>
```

### D.2.4 The `Node` class

The following code could be generated by one particular implementation of this specification; other implementations may produce somewhat different code.

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.List;
import javax.xml.bind.Dispatcher;
import javax.xml.bind.IdentifiableElement;
import javax.xml.bind.InvalidAttributeException;
import javax.xml.bind.InvalidContentObjectException;
import javax.xml.bind.LocalValidationException;
import javax.xml.bind.MarshallableObject;
import javax.xml.bind.MarshallableRootElement;
import javax.xml.bind.Marshaller;
import javax.xml.bind.MissingAttributeException;
import javax.xml.bind.PredicatedLists;
import javax.xml.bind.RootElement;
import javax.xml.bind.StructureValidationException;
import javax.xml.bind.UnmarshalException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.ValidatableObject;
import javax.xml.bind.Validator;
import javax.xml.marshal.XMLScanner;
import javax.xml.marshal.XMLWriter;

public class Node
    extends MarshallableRootElement
    implements RootElement, IdentifiableElement
{

    private String _Name;
    private IdentifiableElement _Ref;

    private PredicatedLists.Predicate pred_Children
        = new PredicatedLists.Predicate() {
                public void check(Object ob) {
                    if (!(ob instanceof Node))
                        throw new InvalidContentObjectException();
                }
            };

    private List _Children = PredicatedLists.create(this, pred_Children);

    public Node() { }
    public Node(List _Children) {
        this._Children.add(_Children);
    }

    public String name() { return _Name; }

    public void name(String _Name) { this._Name = _Name; invalidate(); }

    // Specified in javax.xml.bind.IdentifiableElement
    public String id() { return _Name; }

    public IdentifiableElement ref() { return _Ref; }

    public void ref(IdentifiableElement _Ref) {
        this._Ref = _Ref;
        invalidate();
    }

    public List children() {
        return _Children;
    }

    public void emptyChildren() {
        _Children = PredicatedLists.create(this, pred_Children);
    }

    public void deleteChildren() {
        _Children = null;
        invalidate();
    }

    // Declared in javax.xml.bind.ValidatableObject
    public void validateThis()
        throws LocalValidationException
    {
        if (_Name == null) throw new MissingAttributeException("name");
    }

    public void validate(Validator v)
        throws StructureValidationException
    {
        if (_Ref != null) v.reference(_Ref);
        if (_Children != null)
            for (Iterator i = _Children.iterator(); i.hasNext();)
                v.reference((IdentifiableElement)i.next());
    }

    // Specified in javax.xml.bind.MarshallableRootElement
    public void marshal(Marshaller m)
        throws IOException
    {
        XMLWriter w = m.writer();
        w.start("node");
        if (_Name != null) w.attribute("name", _Name);
        if (_Ref != null) w.attribute("ref", _Ref.id());
        if (_Children != null)
            for (Iterator i = _Children.iterator(); i.hasNext();)
                m.marshal((MarshallableObject)i.next());
        w.end("node");
    }

    public void unmarshal(Unmarshaller u)
        throws UnmarshalException
    {
        XMLScanner xs = u.scanner();
        Validator v = u.validator();
        xs.takeStart("node");
        while (xs.atAttribute()) {
            String an = xs.takeAttributeName();
            String av = xs.takeAttributeValue();
            if (an.equals("name")) {
                if (_Name != null)
                    throw new DuplicateAttributeException;
                _Name = av;
                continue;
            }
            if (an.equals("ref")) {
                Validator.Patcher p
                    = new Validator.Patcher() {
                            public void patch(IdentifiableElement target) {
                                _Ref = (Node)target;
                            }};
                v.reference(av, p);
                continue;
            }
            throw new InvalidAttributeException(an);
        }
        while (xs.atStart("node"))
            _Children.add(u.unmarshal(Node.class));
        xs.takeEnd("node");
    }

    // Declared in javax.xml.bind.MarshallableObject
    public static Node unmarshal(InputStream in)
        throws UnmarshalException
    {
        return unmarshal(XMLScanner.open(in));
    }

    public static Node unmarshal(XMLScanner xs)
        throws UnmarshalException
    {
        return unmarshal(xs, newDispatcher());
    }

    public static Node unmarshal(XMLScanner xs, Dispatcher d)
        throws UnmarshalException
    {
        return ((Node) d.unmarshal(xs, Node.class));
    }

    public static Dispatcher newDispatcher() {
        Dispatcher d = new Dispatcher();
        d.register("node", Node.class);
        d.freezeElementNameMap();
```

```
        return d;
    }

    public boolean equals(Object ob) {
        if (this == ob) return true;
        if (!(ob instanceof Node)) return false;
        Node tob = (Node)ob;
        if (_Name != null) {
            if (tob._Name == null) return false;
            if (!_Name.equals(tob._Name)) return false;
        } else if (tob._Name != null) return false;
        if (_Ref != null) {
            if (tob._Ref == null) return false;
            if (!_Ref.equals(tob._Ref)) return false;
        } else if (tob._Ref != null) return false;
        if (_Children != null) {
            if (tob._Children == null) return false;
            if (!_Children.equals(tob._Children)) return false;
        } else if (tob._Children != null) return false;
        return true;
    }

    public int hashCode() {
        int h = 0;
        h = (127 * h) + ((_Name != null) ? _Name.hashCode() : 0);
        h = (127 * h) + ((_Ref != null) ? _Ref.hashCode() : 0);
        h = (127 * h) + ((_Children != null) ? _Children.hashCode() : 0);
        return h;
    }

    public String toString() {
        StringBuffer sb = new StringBuffer("<<node");
        if (_Name != null) {
            sb.append(" name=");
            sb.append(_Name.toString());
        }
        if (_Ref != null) {
            sb.append(" ref=");
            sb.append(_Ref.id());
        }
        if (_Children != null) {
            sb.append(" children=");
            sb.append(_Children.toString());
        }
        sb.append(">>");
        return sb.toString();
    }

}
```