Castor 1.4.0 - Reference documentation

Chapter 1. Castor XML - XML data binding

1.1. XML framework

1.1.1. Introduction

Castor XML is an XML data binding framework. Unlike the two main XML APIs, DOM (Document Object Model) and SAX (Simple API for XML) which deal with the structure of an XML document, Castor enables you to deal with the data defined in an XML document through an object model which represents that data.

Castor XML can marshal almost any "bean-like" Java Object to and from XML. In most cases the marshalling framework uses a set of ClassDescriptors and FieldDescriptors to describe how an Object should be marshalled and unmarshalled from XML.

For those not familiar with the terms "marshal" and "unmarshal", it's simply the act of converting a stream (sequence of bytes) of data to and from an Object. The act of "marshalling" consists of converting an Object to a stream, and "unmarshalling" from a stream to an Object.

1.1.2. Castor XML - The XML data binding framework

The XML data binding framework, as it's name implies, is responsible for doing the conversion between Java and XML. The framework consists of two worker classes, `org/exolab/castor/xml/Marshaller` and `org.exolab.castor.xml.Unmarshaller` respectively, and a bootstrap class `org.exolab.castor.xml.XMLContext` used for configuration of the XML data binding framework and instantiation of the two worker objects.

Lets walk through a very simple example. Assume we have a simple `Person` class as follows:

```
import java.util.Date;

/** An simple person class */
public class Person
implements java.io.Serializable {

/** The name of the person */
private String name = null;

/** The Date of birth
*/
private Date dob = null;

/** Creates a Person with no name */
public Person() {
super();
}

/** Creates a Person
with the given name */
public Person(String name) { this.name = name; }

/**
* @return date of birth of the person
*/
public Date getDateOfBirth() { return dob; }

/**
* @return name of the person
*/
public String getName() { return
name; }
```

```
/**
* Sets the date of birth of the person
* @param name the name of the person
*/
public void
setDateOfBirth(Date dob) { this.dob = dob; }

/**
* Sets the name of the person
* @param name the name of the
person
*/
public void setName(String name) { this.name = name; }
}
```

To (un-)marshal data to and from XML, Castor XML can be used in one of three modes:

- introspection mode
- mapping mode
- descriptor mode (aka generation mode)

The following sections discuss each of these modes at a high level.

1.1.2.1. Introspection mode

The *introspection mode* is the simplest mode to use from a user perspective, as it does not require any configuration from the user. As such, the user does not have to provide any mapping file(s), nor point Castor to any generated descriptor classes (as discussed in the 'descriptor mode' section).

In this mode, the user makes use of **static** methods on the `org.exolab.castor.xml.Marshaller` and `org.exolab.castor.xml.Unmarshaller` classes, providing all required data as parameters on these method calls.

To marshal an instance of the person class you simply call the `org.exolab.castor.xml.Marshaller` as follows:

```
// Create a new Person
Person person = new Person("Ryan 'Mad Dog'
Madden");
person.setDateOfBirth(new Date(1955, 8, 15));

// Create a File to marshal to
writer = new
FileWriter("test.xml");

// Marshal the person object
Marshaller.marshal(person, writer);
```

This produces the XML shown in Example 1.1, “XML produced in introspection mode”

Example 1.1. XML produced in introspection mode

```
XML to written
```

To unmarshal an instance of the person class from XML, you simply call the `org.exolab.castor.xml.Unmarshaller` as follows:

```
// Create a Reader to the file to unmarshal from
reader = new
FileReader("test.xml");

// Marshal the person object
Person person = (Person)
Unmarshaller.unmarshal(Person.class, reader);
```

Marshalling and unmarshalling is basically that simple.

Note

Note: The above example uses the ***static*** methods of the marshalling framework, and as such no Marshaller and/or Unmarshaller instances need to be created. A common mistake in this context when using a **mapping file** is to call the `org.exolab.castor.xml.Marshaller` or `org.exolab.castor.xml.Unmarshaller` as in the above example. This won't work, as the mapping will be ignored.

In *introspection mode* , Castor XML uses Java reflection to establish the binding between the Java classes (and their properties) and the XML, following a set of (default) naming rules. Whilst it is possible to change to a different set of naming rules, there's no way to override this (default) naming for individual artifacts. In such a case, a *mapping file* should be used.

1.1.2.2. Mapping mode

In *mapping mode* , the user provides Castor XML with a user-defined mapping (in form of a mapping file) that allows the (partial) definition of a customized mapping between Java classes (and their properties) and XML.

When you are using a mapping file, create an instance of the `org.exolab.castor.xml.XMLContext` class and use the `org.exolab.castor.xml.XMLContext.addMapping(Mapping)` method to provide Castor XML with one of more mapping files.

To start using Castor XML for marshalling and/or unmarshalling based upon your custom mapping, create instances of `org.exolab.castor.xml.Marshaller` and `org.exolab.castor.xml.Unmarshaller` as needed using one of the following methods:

Table 1.1. Methods on XMLContext to create Un-/Marshaller objects

Method name	Description
createMarshaller	Creates a Marshaller instance.
createUnmarshaller	Creates a Unmarshaller instance.

and call any of the **non-static** (un)marshal methods to trigger data binding in either way.

Below code shows a full example that demonstrates unmarshalling a `Person` instance from XML using a `org.exolab.castor.xml.Unmarshaller` instance as obtained from an XMLContext previously configured to your needs.

Example 1.2. Unmarshalling from XML using a mapping

```
import org.exolab.castor.xml.XMLContext; import
org.exolab.castor.mapping.Mapping; import
org.exolab.castor.xml.Unmarshaller;

// Load Mapping
Mapping
mapping = new Mapping();
mapping.loadMapping("mapping.xml");


// initialize and configure XMLContext

XMLContext context = new XMLContext();
context.addMapping(mapping);


// Create a Reader to the file to
unmarshal from

reader = new FileReader("test.xml");

// Create a new Unmarshaller
Unmarshaller unmarshaller =
context.createUnmarshaller();
unmarshaller.setClass(Person.class);

// Unmarshal the person object
Person
person = (Person)
unmarshaller.unmarshal(reader);
```

To marshal the very same `Person` instance to XML using a `org.exolab.castor.xml.Marshaller` obtained from the **same** `org.exolab.castor.xml.XMLContext` , use code as follows:

Example 1.3. Marshalling to XML using a mapping

```
import org.exolab.castor.xml.Marshaller;

// create a Writer to the
file to marshal to
Writer writer = new FileWriter("out.xml");

// create a new Marshaller
Marshaller
marshaller = context.createMarshaller();
marshaller.setWriter(writer);

// marshal the person object
marshaller.marshal(person);
```

Please have a look at XML Mapping for a detailed discussion of the mapping file and its structure.

For more information on how to effectively deal with loading mapping file(s) especially in multi-threaded environments, please check the best practice section.

1.1.2.3. Descriptor mode

TBD

1.1.3. Sources and destinations

Castor supports multiple sources and destinations from which objects can be marshalled and unmarshalled.

Table 1.2. Marshalling destinations.

Destination	Description
`marshal(java.io.Writer)`	The character stream.
`marshal(org.xml.sax.DocumentHandler)`	The SAX document handler.
`marshal(org.xml.sax.ContentHandler)`	The SAX content handler.
`marshal(org.w3c.dom.Node)`	The DOM node to marshall object into.
`marshal(javax.xml.stream.XMLStreamWriter)`	The STaX cursor API.
`marshal(javax.xml.stream.XMLEventWriter)`	The STaX iterator API.
`marshal(javax.xml.transform.Result)`	`javax.xml.transform.dom.DOMResult` , `javax.xml.transform.sax.SAXResult` and `javax.xml.transform.stream.StreamResult` are supported.

Table 1.3. Unmarshalling sources.

Source	Description
`unmarshal(java.io.Reader)`	A character stream.
`unmarshal(org.xml.sax.InputSource)`	A SAX input source.
`unmarshal(org.w3c.dom.Node)`	A W3C DOM node which will be used for unmarshalling.
`unmarshal(javax.xml.stream.XMLStreamReader)`	A StAX cursor.
`unmarshal(javax.xml.stream.XMLEventReader)`	A StAX iterator.
`unmarshal(javax.xml.transform.Source)`	Supports `javax.xml.transform.dom.DOMSource` , `javax.xml.transform.sax.SAXSource` and `javax.xml.transform.stream.StreamSource` .

Source	Description

Castor 1.3.2 and 1.3.3 introduced support for the STaX API for both for marshalling and unmarshalling. The framework fully supports the STaX cursor and iterator API.

An example of marshalling using STaX:

Example 1.4. Marshalling to a StAX java.xml.stream.XMLStreamWriter

```
// marshalling using STaX
StringWriter writer = new StringWriter();
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLStreamWriter xmlStreamWriter =
outputFactory.createXMLStreamWriter(writer);

marshaller.setXmlStreamWriter(xmlStreamWriter);
marshaller.marshal(object);
```

Also beginning from version 1.3.3, the framework has been modified to support Source and Result interfaces. Now it is possible to use SAXSource, DOMSource and StreamSource for unmarshalling and corresponding classes for marshalling.

Below an example of marshalling into Result:

Example 1.5. Marshalling to a `javax.xml.transform.dom.DOMResult`

```
// instance of object to be marshalled
Object obj = ...

// marshalling
into DOM node
XMLContext xmlContext = ... // creates the xml context

// creates marshaller
Marshaller
marshaller = xmlContext.createMarshaller();

// creates DOM factory
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

// creates
document
Document document = builder.newDocument();

// sets the DOM result for the marshaller
marshaller.setResult(new DOMResult(document));

// marshalls object
marshaller.marshall(obj);
```

Another example of unmarshalling from Source:

Example 1.6. Unmarshalling from a `javax.xml.transform.sax.SAXSource`

```
// unmarshalling from SAX InputSource
XMLContext xmlContext = ... //
creates the xml context

// creates unmarshaller
Unmarshaller unmarshaller = xmlContext.createUnmarshaller();

// creates SAX input source
InputSource inputSource = new InputSource(new StringReader(xml));

// creates
instance of SAXSource
SAXSource saxSource = new SAXSource(inputSource);

// unmarshalls object
Object result =
unmarshaller.unmarshal(saxSource);
```

1.1.4. XMLContext - A consolidated way to bootstrap Castor

With Castor 1.1.2, the `org.exolab.castor.xml.XMLContext` class has been added to the Castor marshalling framework. This new class provides a bootstrap mechanism for Castor XML, and allows easy (and efficient) instantiation of `org.exolab.castor.xml.Marshaller` and `org.exolab.castor.xml.Unmarshaller` instances as needed.

As shown above, the `org.exolab.castor.xml.XMLContext` class offers various factory methods to obtain a new `org.exolab.castor.xml.Marshaller` , `org.exolab.castor.xml.Unmarshaller` .

When you need more than one `org.exolab.castor.xml.Unmarshaller` instance in your application, please call `org.exolab.castor.xml.XMLContext.createUnmarshaller()` as required. As all `Unmarshaller` instances are created from the very same `XMLContext` instance, overhead will be minimal. Please note, though, that use of one `Unmarshaller` instance is not thread-safe.

1.1.5. Using existing Classes/Objects

Castor can marshal "almost" any arbitrary Object to and from XML. When descriptors are not available for a specfic Class, the marshalling framework uses reflection to gain information about the object.

> **Note**
> Actually an in memory set of descriptors are created for the object and we will soon have a way for saving these descriptors as Java source, so that they may be modified and compiled with little effort.

If a set of descriptors exist for the classes, then Castor will use those to gain information about how to handle the marshalling. See Section 1.1.6, “Class Descriptors” for more information.

There is one main restrictions to marshalling objects. These classes must have have a public default constructor (ie. a constructor with no arguments) and adequete "getter" and "setter" methods to be properly be marshalled and unmarshalled.

The example illustrated in the previous section Section 1.1.2, “Castor XML - The XML data binding framework” demonstrates how to use the framework with existing classes.

1.1.6. Class Descriptors

Class descriptors provide the "Castor Framework" with necessary information so that the Class can be marshalled properly. The class descriptors can be shared between the JDO and XML frameworks.

Class descriptors contain a set of ???

XML Class descriptors provide the marshalling framework with the information it needs about a class in order to be marshalled to and from XML. The XMLClassDescriptor `org.exolab.castor.xml.XMLClassDescriptor` .

XML Class Descriptors are created in four main ways. Two of these are basically run-time, and the other two are compile time.

1. Compile-Time Descriptors

To use "compile-time" class descriptors, one can either implement the `org.exolab.castor.xml.XMLClassDescriptor` interface for each class which needs to be "described", or have the Source Code Generator create the proper descriptors.

The main advantage of compile-time descriptors is that they are faster than the run-time approach.

2. Run-Time Descriptors

To use "run-time" class descriptors, one can either simply let Castor introspect the classes, a mapping file can be provided, or a combination of both "default introspection" and a specified mapping file may be used.

For "default introspection" to work the class being introspected must have adequete setter/getter methods for each field of the class that should be marshalled and unmarshalled. If no getter/setter methods exist, Castor can handle direct field access to public fields. It does not do both at the same time. So if the respective class has any getter/setter methods at all, then no direct field access will take place.

There is nothing to do to enable "default introspection". If a descriptor cannot be found for a class, introspection occurs automatically.

Some behavior of the introspector may be controlled by setting the appropriate properties in the *castor.properties* file. Such behavior consists of changing the naming conventions, and whether primitive types are treated as attributes or elements. See *castor.properties* file for more information.

A mapping file may also be used to "describe" the classes which are to be marshalled. The mapping is loaded before any marshalling/unmarshalling takes place. See `org.exolab.castor.mapping.Mapping`

The main advantage of run-time descriptors is that it takes very little effort to get something working.

1.2. XML Mapping

1.2.1. Introduction

Castor XML mapping is a way to simplify the binding of java classes to XML document. It allows to transform the data contained in a java object model into/from an XML document.

Although it is possible to rely on Castor's default behavior to marshal and unmarshal Java objects into an XML document, it might be necessary to have more control over this behavior. For example, if a Java object model already exists, Castor XML Mapping can be used as a bridge between the XML document and that Java object model.

Castor allows one to specify some of its marshalling/unmarshalling behavior using a mapping file. This file gives explicit information to Castor on how a given XML document and a given set of Java objects relate to each other.

A Castor mapping file is a good way to dissociate the changes in the structure of a Java object model from the changes in the corresponding XML document format.

1.2.2. Overview

The mapping information is specified by an XML document. This document is written from the point of view of the Java object and describes how the properties of the object have to be translated into XML. One constraint for the mapping file is that Castor should be able to infer unambiguously from it how a given XML element/attribute has to be translated into the object model during unmarshalling.

The mapping file describes for each object how each of its fields have to be mapped into XML. A field is an abstraction for a property of an object. It can correspond directly to a public class variable or indirectly to a property via some accessor methods (setters and getters).

It is possible to use the mapping and Castor default behavior in conjunction: when Castor has to handle an object or an XML data but can't find information about it in the mapping file, it will rely on its default behavior. Castor will use the Java Reflection API to introspect the Java objects to determine what to do.

Note: Castor can't handle all possible mappings. In some complex cases, it may be necessary to rely on an XSL transformation in conjunction with Castor to adapt the XML document to a more friendly format.

1.2.2.1. Marshalling Behavior

For Castor, a Java class has to map into an XML element. When Castor marshals an object, it will:

- use the mapping information, if any, to find the name of the element to create

or

- by default, create a name using the name of the class

It will then use the fields information from the mapping file to determine how a given property of the object has to be translated into one and only one of the following:

- an attribute
- an element
- text content
- nothing, as we can choose to ignore a particular field

This process will be recursive: if Castor finds a property that has a class type specified elsewhere in the mapping file, it will use this information to marshal the object.

By default, if Castor finds no information for a given class in the mapping file, it will introspect the class and apply a set of default rules to guess the fields and marshal them. The default rules are as follows:

- All primitive types, including the primitive type wrappers (Boolean, Short, etc...) are marshalled as attributes.
- All other objects are marshalled as elements with either text content or element content.

1.2.2.2. Unmarshalling Behavior

When Castor finds an element while unmarshalling a document, it will try to use the mapping information to determine which object to instantiate. If no mapping information is present, Castor will use the name of the element to try to guess the name of a class to instantiate (for example, for an element named 'test-element', Castor will try to instantiate a class named 'TestElement' if no information is given in the mapping file). Castor will then use the field information of the mapping file to handle the content of the element.

If the class is not described in the mapping file, Castor will instrospect the class using the Java Reflection API to determine if there is any function of the form getXxxYyy()/setXxxYyy(<type> x). This accessor will be associated with XML element/attribute named 'xxx-yyy'. In the future, we will provide a way to override this default behavior.

Castor will introspect object variables and use direct access _only_ if no get/set methods have been found in the class. In this case, Castor will look for public variables of the form:

```
public <type> xxxYYY;
```

and expect an element/attribute named 'xxx-yyy'. The only handled collections for <type> are java.lang.Vector and array. (up to version 0.8.10)

For primitive <type>, Castor will look for an attribute first and then an element. If <type> is not a primitive type, Castor will look for an element first and then an attribute.

1.2.3. The Mapping File

The following sections define the syntax for each of the mapping file artefacts and their semantical meaning.

1.2.3.1. Sample domain objects

This section defines a small domain model that will be referenced by various mapping file (fragments/samples) in the following sections. The model consists of two two classes `Order` and `OrderItem`, where an order holds a list of order items.

```
public class Order {

    private List orderItems;
    private String orderNumber;

    public List getOrderItems() {
        return orderItems;
    }
    public void setOrderItems(List orderItems) {
        this.orderItems = orderItems;
    }
    public String getOrderNumber() {
        return orderNumber;
    }
```

```
    public void setOrderNumber(String orderNumber) {
        this.orderNumber = orderNumber;
    }
}

public class OrderItem {

    private String id;
    private Integer orderQuantity;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public Integer getOrderQuantity() {
        return orderQuantity;
    }
    public void setOrderQuantity(Integer orderQuantity) {
        this.orderQuantity = orderQuantity;
    }
}
```

As shown above in bold, the `Order` instance has a (private) field `'orderItems'` to hold a collection of `OrderItem` instances. This field is publically exposed by corresponding getter and setter methods.

1.2.3.2. The <mapping> element

```
<!ELEMENT mapping ( description?, include*, field-handler*, class*, key-generator* )>
```

The <mapping> element is the root element of a mapping file. It contains:

- an optional description
- zero or more <include> which facilitates reusing mapping files
- zero of more <field-handler> defining custom, configurable field handlers
- zero or more <class> descriptions: one for each class we intend to give mapping information
- zero or more <key-generator>: not used for XML mapping

A mapping file look like this:

```
<?xml version="1.0"?>

<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
      castor.org
                         "http://castor.org/mapping.dtd">

<mapping>
        <description>Description of the mapping</description>

        <include href="other_mapping_file.xml"/>

        <!-- mapping for class 'A' -->
        <class name="A">
                .........
        </class>

        <!-- mapping for class 'B' -->
        <class name="B">
                .........
        </class>
```

```
</mapping>
```

1.2.3.3. The <class> element

```
<!ELEMENT class ( description?, cache-type?, map-to?, field+ )>
<!ATTLIST class
          name            ID       #REQUIRED
          extends         IDREF    #IMPLIED
          depends         IDREF    #IMPLIED
          auto-complete   ( true |false ) "false"
          identity        CDATA    #IMPLIED
          access          ( read-only | shared | exclusive | db-locked )  "shared"
          key-generator   IDREF    #IMPLIED >
```

The `<class>` element contains all the information used to map a Java class into an XML document. The content of `<class>` is mainly used to describe the fields that will be mapped.

Table 1.4. Description of the attributes

Name	Description
name	The fuly-qualified name of the Java class that we want to map to.
extends	The fully qualified name of a parent class. This attribute should be used **only** if this class extends another class for which a class mapping is provided. It should **not** be used if there's no class maping for the extended class.
depends	Used with Castor JDO only; for more information on this field, please see the JDO documentation.
auto-complete	If true, the class will be introspected to determine its field and the fields specified in the mapping file will be used to overide the fields found during the introspection.
identity	Used with Castor JDO only; for more information on this field, please see see the JDO documentation.
access	Used with Castor JDO only; for more information on this field, please see see the JDO documentation.
key-generator	Used with Castor JDO only; for more information on this field, please see see the JDO documentation.

The auto-complete attributes is interesting as it allow a fine degree of control of the introspector: it is possible to specifiy only the fields whose Castor default behavior does not suite our needs. These feature should simplify the handling of complexe class containing many fields. Please see below for an example usage of this attribute.

Table 1.5. Description of the content

Name	Description
description	An optional description.
cache-type	Used with Castor JDO only; for more information on this field, please see see the [JDO documentation].
map-to	Used if the name of the element is not the name of the class. By default, Castor will infer the name of the element to be mapped from the name of the class: a Java class named 'XxxYyy' will be transformed in 'xxx-yyy'. If you don't want Castor to generate the name, you need to use <map-to> to specify the name you want to use. <map-to> is only used for the root element.
field	Zero or more <field> elements, which are used to describe the properties of the Java class being mapped.

1.2.3.3.1. Sample <class> mappings

The following mapping fragment defines a class mapping for the `OrderItem` class:

```
<class name="mypackage.OrderItem>

   <map-to xml="item"/>

   <field name="id" type="string">
      <bind-xml name="identity" node="attribute"/>
   </field>

   </field name="orderQuantity" type="integer">
      <bind-xml name="quantity" node="element"/>
   </field>

</class>
```

When marshalling an `OrderItem` instance, this yields the following XML:

```
<?xml version="1.0" ?>
<item identity="12">
   <quantity>100</quantity>
</item>
```

The following mapping fragment defines a class mapping for the same class, where for all properties but `id` introspection should be used; the use of the `auto-complete` attribute instructs Castor XML to use introspection for all attributes other than `'id'`, where the given field mapping will be used.

```
<class name="mypackage.OrderItem auto-complete="true">

   <map-to xml="item"/>

   <field name="id" type="string">
      <bind-xml name="identity" node="attribute"/>
   </field>

</class>
```

When marshalling the very same `OrderItem` instance, this yields the following XML:

```
<?xml version="1.0" ?>
<item identity="12">
   <order-quantity>100</order-quantity>
</item>
```

By removing the <map-to> element from above class mapping, ...

```
<class name="mypackage.OrderItem auto-complete="true">

   <field name="id" type="string">
      <bind-xml name="identity" node="attribute"/>
   </field>

</class>
```

... Castor will use introspection to infer the element name from the Java class name (`OrderItem`), applying a default naming convention scheme.

When marshalling the very same `OrderItem` instance, this yields the following XML:

```
<?xml version="1.0" ?>
<order-item identity="12">
   <order-quantity>100</order-quantity>
</order-item>
```

1.2.3.4. The <map-to> element

```
<!ELEMENT map-to EMPTY>
<!ATTLIST map-to
          table                NMTOKEN  #IMPLIED
          xml                  NMTOKEN  #IMPLIED
          ns-uri               NMTOKEN  #IMPLIED
          ns-prefix            NMTOKEN  #IMPLIED
          ldap-dn              NMTOKEN  #IMPLIED
          element-definition   (true|false) "false"     NEW as of 1.0M3
          ldap-oc              NMTOKEN  #IMPLIED>
```

`<map-to>` is used to specify the name of the element that should be associated with the given class. `<map-to>` is only used for the root class. If this information is not present, Castor will:

- for marshalling, infer the name of the element to be mapped from the name of the class: a Java class named 'XxxYyy' will be transformed into 'xxx-yyy'.
- for unmarshalling, infer the name of the class from the name of the element: for an element named 'test-element' Castor will try to use a class named 'TestElement'

Please note that it is possible to change the naming scheme used by Castor to translate between the XML name and the Java class name in the `castor.properties` file.

Table 1.6. Description of attributes

xml	Name of the element that the class is associated to.

ns-uri	Namespace URI
ns-prefix	Desired namespace
element-definition	`True` if the descriptor as created from a schema definition that was of type element (as opposed to a <complexType> definition). This only is useful in the context of source code generation.
ldap-dn	Not used for Castor XML
ldap-oc	Not used for Castor XML

1.2.3.4.1. <map-to> samples

The following mapping fragment defines a <map-to element for the `OrderItem` class, manually setting the element name to a value of `'item'`.

```
<class name="myPackage.OrderItem">
   ...
   <map-to xml="item" />
   ...
</class>
```

The following mapping fragment instructs Castor to assign a namespace URI of `http://castor.org/sample/mapping/` to the <item> element, and use a namespace prefix of `'castor'` during un-/marshalling.

```
<class name="myPackage.OrderItem">
   ...
   <map-to xml="item" ns-uri="http://castor.org/sample/mapping/"
           ns-prefix="castor"/>
   ...
</class>
```

When marshalling an `OrderItem` instance, this will yield the following XML:

```
<?xml version="1.0" ?>
<castor:order-item xmlns:castor="http://castor.org/sample/mapping/" identity="12">
   <castor:order-quantity>100</castor:order-quantity>
</castor:order-item>
```

1.2.3.5. The <field> element

```
<!ELEMENT field ( description?, sql?, bind-xml?, ldap? )>
<!ATTLIST field
    name           NMTOKEN  #REQUIRED
    type           NMTOKEN  #IMPLIED
    handler        NMTOKEN  #IMPLIED
    required       ( true | false )  "false"
    direct         ( true | false )  "false"
    lazy           ( true | false )  "false"
    transient      ( true | false )  "false"
    nillable       ( true | false )  "false"
    container      ( true | false )  "false"
    get-method     NMTOKEN  #IMPLIED
    set-method     NMTOKEN  #IMPLIED
    create-method  NMTOKEN  #IMPLIED
    collection     ( array | vector | hashtable | collection | set | map )  #IMPLIED>
```

`<field>` is used to describe a property of a Java object we want to marshal/unmarshal. It gives:

- its identity ('name')
- its type (infered from 'type' and 'collection')
- its access method (infered from 'direct', 'get-method', 'set-method')

From this information, Castor is able to access a given property in the Java class.

In order to determine the signature that Castor expects, there are two easy rules to apply.

1. Determine <type>.

- **If there is no 'collection' attribute**, the <type> is just the Java type specified in <type_attribute> (the value of the 'type' attribute in the XML document). The value of <type_attribute> can be a fully qualified Java object like 'java.lang.String' or one of the allowed short name:

Table 1.7. Type shortnames

short name	Primitive type?	Java Class
other	N	java.lang.Object
string	N	java.lang.String
integer	Y	java.lang.Integer.TYPE
long	Y	java.lang.Long.TYPE
boolean	Y	java.lang.Boolean.TYPE
double	Y	java.lang.Double.TYPE
float	Y	java.lang.Float.TYPE
big-decimal	N	java.math.BigDecimal
byte	Y	java.lang.Byte.TYPE
date	N	java.util.Date
short	Y	java.lang.Short.TYPE
char	Y	java.lang.Character.TYPE
bytes	N	byte[]
chars	N	char[]
strings	N	String[]
locale	N	java.util.Locale

Castor will try to cast the data in the XML file in the proper Java type.

- **If there is a collection attribute** , you can use the following table:

Table 1.8. Type implementations

name	<type>	default implementation
array	<type_attribute>[]	<type_attribute>[]
arraylist	java.util.List	java.util.Arraylist
vector	java.util.Vector	java.util.Vector
hashtable	java.util.Hashtable	java.util.Hashtable
collection	java.util.Collection	java.util.Arraylist
set	java.util.Set	java.util.Hashset
map	java.util.Map	java.util.Hashmap
sortedset	java.util.SortedSet	java.util.TreeSet

The type of the object inside the collection is <type_attribute>. The 'default implementation' is the type used if the object holding the collection is found to be null and need to be instantiated.

For hashtable and maps (since 0.9.5.3), Castor will save both key and values. When marshalling output <key> and <value> elements. These names can be controlled by using a top-level or nested class mapping for the org.exolab.castor.mapping.MapItem class.

Note: for backward compatibility with prior versions of Castor, the *saveMapKeys* property can be set to false in the castor.properties file.

For versions prior to 0.9.5.3, hashtable and maps, Castor will save only the value during marshalling and during unmarshalling will add a map entry using the object as both the key and value, e.g. map.put(object, object).

It is necessary to use a collection when the content model of the element expects more than one element of the specified type.

Determine the signature of the function

- **If 'direct' is set to true**, Castor expects to find a class variable with the given signature:

```
public <type> <name>;
```

- **If 'direct' is set to false or omitted**, Castor will access the property though accessor methods. Castor determines the signature of the accessors as follow: If the 'get-method' or 'set-method' attributes are supplied, it will try to find a function with the following signature:

```
public <type> <get-method>();
```

or

```
public void <set-method>(<type> value);
```

If 'get-method' and 'set-method' attributes are not provided, Castor will try to find the following function:

```
public <type> get<capitalized-name>();
```

or

```
public void set<capitalized-name>(<type> value);
```

<capitalized-name> means that Castor takes the <name> attribute and put its first letter in uppercase without modifying the other letters.

The content of <field> will contain the information on how to map this given field to SQL, XML, ...

- **Exceptions concerning collection fields:**

  The default is to treat the 'get-method' as a simple getter returning the collection field, and the 'set-method' as a simple getter used to set a new instance on the collection field.

Table 1.9. Collection field access

Parameter	Description

'get-method'	If a 'get-method' is provided for a collection field, Castor - in adition to the default behaviour described above - will deviate from the standard case for the following special prefixes:
	`public Iterator iterate...();`
	A 'get-method' starting with the prefix ' `iterate` ' is treated as Iterator method for the given collection field.
	`public Enumeration enum...();`
	A 'get-method' starting with ' `enum` ' is treated as Enumeration method for the given collection field.

'set-method'	If 'set-method' is provided for a collection field, Castor - in addition to the default behaviour described above - will accept an 'add' prefix and expect the

Parameter	Description
	following signature: `public void add...(<type> value);` This method is called for each collection element while unmarshalling.

Table 1.10. Description of the attributes

Name	Description
name	The field 'name' is required even if no such field exists in the class. If 'direct' access is used, 'name' should be the name of a public instance member in the object to be mapped (the field must be public, not static and not transient). If no direct access and no 'get-/set-method' is specified, this name will be used to infer the name of the accessors methods.
type	The Java type of the field. It is used to access the field. Castor will use this information to cast the XML information (like string into integer). It is also used to define the signature of the accessor methods. If a collection is specified, this is used to specify the type of the objects held by the collection. See description above for more details.
required	A field can be optional or required.
nillable	A field can be of content '`nil`'.
transient	If true, this field will be ignored during the marshalling. This is usefull when used together with the auto-complete="true" option.
direct	If true, Castor will expect a public variable in the containing class and will access it directly (for both reading and writing).
container	Indicates whether the field should be treated as a container, i.e. only it's fields should be persisted, but not the containing class itself. In this case, the container attribute should be set to true (supported in Castor XML only).
collection	If a parent expects more than one occurrence of one of its element, it is necessary to specify which collection Castor will use to handle them. The type specified is used to define the type of the content inside the collection.

Name	Description
get-method	Optional name of the 'get method' Castor should use. If this attribute is not set and the set-method attribute is not set, then Castor will try to infer the name of this method with the algorithm described above.
set-method	Optional name of the 'set method' Castor should use. If this attribute is not set and the get-method attribute is not set, then Castor will try to infer the name of this method with the algorithm described above.
create-method	Optionally defines a factory method for the instantiation of a FieldHandler
handler	If present, specifies one of the following: • The fully-qualified class name of a custom field handler implementation, or • The (short) name of a configurable field handler definition.

1.2.3.6. Description of the content

In the case of XML mapping, the content of a field element should be one and only one **<bind-xml>** element describing how this given field will be mapped into the XML document.

1.2.3.6.1. Mapping constructor arguments (since 0.9.5)

Starting with release 0.9.5, for *attribute* mapped fields, support has been added to map a constructor field using the `set-method` attribute.

To specify that a field (mapped to an attribute) should be used as a constructor argument during object initialization, please specify a `set-method` attribute on the `<field>` mapping and use "%X" as the value of the `set-method` attribute, where `X` is a positive integer number, e.g. `%1` or `%21`.

For example:

```
<field name="foo" set-method="%1" get-method="getFoo" type="string">
   <bind-xml node="attribute"/>
</field>
```

Note that because the `set-method` is specified, the `get-method` also must be specified.

Tip: the XML HOW-TO section has a HOW-TO document for mapping constructor arguments, incl. a fully working mapping.

1.2.3.6.2. Sample 1: Defining a custom field handler

The following mapping fragment defines a `<field>` element for the `member` property of the `org.some.package.Root` class, specifying a custom `org.exolab.castor.mapping.FieldHandler` implementation.

```
<class name="org.some.package.Root">
   <field name="member" type="string" handler="org.some.package.CustomFieldHandlerImpl"/>
</class>
```

1.2.3.6.3. Sample 2: Defining a custom configurable field handler

The same custom field handler as in the previous sample can be defined with a separate configurable <field-handler> definition, where additional configuration can be provided.

```
<field-handler name="myHandler" class="org.some.package.CustomFieldHandlerImpl">
   <param name="date-format" value="yyyyMMddHHmmss"/>
</field-handler>
```

and subsequently be referred to by its **name** as shown in the following field mapping:

```
<class name="org.some.package.Root">
   <field name="member" type="string" handler="myHandler"/>
</class>
```

1.2.3.6.4. Sample 3: Using the container attribute

Assume you have a class mapping for a class `Order` which defines - amongst others - a field mapping as follows, where the field `item` refers to an instance of a class `Item`.

```
<class name="some.example.Order">

   ...
   <field name="item" type="some.example.Item" >
      <bind-xml> name="item" node="element" />
   </field>
   ...
</class>

<class name="some.example.Item">
   <field name="id" type="long" />
   <field name="description" type="string" />
</class>
```

Marshalling an instance of `Order` would produce XML as follows:

```
<order>
    ...
    <item>
        <id>100</id>
        <description>...</description>
    </item>
</order>
```

If you do not want the `Item` instance to be marshalled, but only its fields, change the field mapping for the `item` member to be as follows:

```
<field name="item" type="some.example.Item" container="false" >
   <bind-xml> name="item" node="element" />
</field>
```

The resulting XML would look as follows:

```
<order>
    ...
    <id>100</id>
    <description>...</description>
</order>
```

1.2.3.7. The <bind-xml> element

1.2.3.7.1. Grammar

```
<!ELEMENT bind-xml (class?, property*)>
<!ATTLIST bind-xml
          name     NMTOKEN     #IMPLIED
          type     NMTOKEN     #IMPLIED
          location CDATA       #IMPLIED
          matches  NMTOKENS    #IMPLIED
          QName-prefix NMTOKEN #IMPLIED
          reference   ( true | false ) "false"
          node        ( attribute | element | text )    #IMPLIED
          auto-naming ( deriveByClass | deriveByField ) #IMPLIED
          transient   ( true | false ) "false">
```

1.2.3.7.1.1. Definiton

The `<bind-xml>` element is used to describe how a given Java field should appear in an XML document. It is used both for marshalling and unmarshalling.

Table 1.11. Description of the attributes

name	The name of the element or attribute. **Note** The name is a QName, and a namespace prefix may be used to indicate the element or attribute belongs to a certain namespace. Note the prefix is not preserved or used during marshalling, it's simply used for qualification of which namespace the element or attribute belongs.
auto-naming	If no name is specified, this attribute controls how castor will automatically create a name for the field. Normally, the name is created using the field name, however many times it is necessary to create the name by using the class type instead (such as heterogenenous collections).
type	XML Schema type (of the value of this field) that requires specific handling in the Castor Marshalling Framework (such as 'QName' for instance).

location (since 0.9.4.4)	Allows the user to specify the "sub-path" for which the value should be marshalled to and from. This is useful for "wrapping" values in elements or for mapping values that appear on sub-elements to the current "element" represented by the class mapping. For more information, see the Location attribute below.
QName-prefix	When the field represents a QName value, a prefix can be provided that is used when marshalling value of type QName. More information on the use of 'QName-prefix' can be found in the SourceGenerator Documentation
reference	Indicates if this field has to be treated as a reference by the unmarshaller. In order to work properly, you must specify the node type to 'attribute' for both the 'id' and the 'reference'. In newer versions of Castor, 'element' node for reference is allowed. Remember to make sure that an *identity* field is specified on the `<class>` mapping for the object type being referenced so that Castor knows what the object's identity is.
matches	Allows overriding the matches rules for the name of the element. It is a standard regular expression and will be used instead of the 'name' field. A '*' will match any xml name, however it will only be matched if no other field exists that matches the xml name.
node	Indicates if the name corresponds to an attribute, an element, or text content. By default, primitive types are assumed to be an attribute, otherwise the node is assumed to be an elemen
transient	Allows for making this field transient for XML. The default value is inherited from the <field> element.

1.2.3.7.2. Nested class mapping

Since 0.9.5.3, the bind-xml element supports a nested class mapping, which is often useful when needing to specify more than one mapping for a particular class. A good example of this is when mapping Hashtable/HashMap/Map.

```
<bind-xml ...>
   <class name="org.exolab.castor.mapping.MapItem">
      <field name="key" type="java.lang.String">
        <bind-xml name="id"/>
      </field>
      <field name="value" type="com.acme.Foo"/>
   </class>
</bind-xml>
```

1.2.4. Usage Pattern

Here is an example of how Castor Mapping can be used. We want to map an XML document like the following one (called 'order.xml'). model.

```
<Order reference="12343-AHSHE-314159">
  <Client>
    <Name>Jean Smith</Name>
    <Address>2000, Alameda de las Pulgas, San Mateo, CA 94403</Address>
  </Client>

  <Item reference="RF-0001">
    <Description>Stuffed Penguin</Description>
    <Quantity>10</Quantity>
    <UnitPrice>8.95</UnitPrice>
  </Item>

  <Item reference="RF-0034">
    <Description>Chocolate</Description>
    <Quantity5</Quantity>
    <UnitPrice>28.50</UnitPrice>
  </Item>

  <Item reference="RF-3341">
     <Description>Cookie</Description>
     <Quantity>30</Quantity>
     <UnitPrice>0.85</UnitPrice>
  </Item>
</Order>
```

Into the following object model composed of 3 classes:

- **MyOrder:** represent an order
- **Client:** used to store information on the client
- **Item:** used to store item in an order

The sources of these classes follow.

```
import java.util.Vector;
import java.util.Enumeration;

public class MyOrder {

    private String _ref;
    private ClientData _client;
    private Vector _items;
    private float _total;

    public void setReference(String ref) {
        _ref = ref;
    }

    public String getReference() {
        return _ref;
    }

    public void setClientData(ClientData client) {
        _client = client;
    }

    public ClientData getClientData() {
        return _client;
    }

    public void setItemsList(Vector items) {
        _items = items;
    }

    public Vector getItemsList() {
```

```
        return _items;
    }


    public void setTotal(float total) {
        _total = total;
    }

    public float getTotal() {
        return _total;
    }

    // Do some processing on the data
    public float getTotalPrice() {
        float total = 0.0f;

        for (Enumeration e = _items.elements() ; e.hasMoreElements() ;) {
            Item item = (Item) e.nextElement();
            total += item._quantity * item._unitPrice;
        }

        return total;
    }
}
```

```
public class ClientData {

    private String _name;
    private String _address;

    public void setName(String name) {
        _name = name;
    }

    public String getName() {
        return _name;
    }

    public void setAddress(String address) {
        _address = address;
    }

    public String getAddress() {
        return _address;
    }
}
```

```
public class Item {
    public String _reference;
    public int    _quantity;
    public float  _unitPrice;
    public String _description;
}
```

The XML document and the java object model can be connected by using the following mapping file:

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                         "http://castor.org/mapping.dtd">

<mapping>
  <class name="MyOrder">
    <map-to xml="Order"/>

    <field name="Reference"
           type="java.lang.String">
      <bind-xml name="reference" node="attribute"/>
    </field>

    <field name="Total"
           type="float">
```

```
      <bind-xml name="total-price" node="attribute"/>
    </field>

    <field name="ClientData"
           type="ClientData">
      <bind-xml name="Client"/>
    </field>

    <field name="ItemsList"
           type="Item"
              collection="vector">
      <bind-xml name="Item"/>
    </field>
  </class>

  <class name="ClientData">
    <field name="Name"
           type="java.lang.String">
      <bind-xml name="Name" node="element"/>
    </field>

    <field name="Address"
           type="java.lang.String">
      <bind-xml name="Address" node="element"/>
    </field>
  </class>

  <class name="Item">
    <field name="_reference"
           type="java.lang.String"
           direct="true">
      <bind-xml name="reference" node="attribute"/>
    </field>

    <field name="_quantity"
           type="integer"
           direct="true">
      <bind-xml name="Quantity" node="element"/>
    </field>

    <field name="_unitPrice"
           type="float"
           direct="true">
      <bind-xml name="UnitPrice" node="element"/>
    </field>

    <field name="_description"
           type="string"
           direct="true">
      <bind-xml name="Description" node="element"/>
    </field>
  </class>

</mapping>
```

The following class is an example of how to use Castor XML Mapping to manipulate the file 'order.xml'. It unmarshals the document 'order.xml', computes the total price, sets the total price in the java object and marshals the object model back into XML with the calculated price.

```
import org.exolab.castor.mapping.Mapping;
import org.exolab.castor.mapping.MappingException;

import org.exolab.castor.xml.Unmarshaller;
import org.exolab.castor.xml.Marshaller;

import java.io.IOException;
import java.io.FileReader;
import java.io.OutputStreamWriter;

import org.xml.sax.InputSource;

public class main {

    public static void main(String args[]) {
```

```
        Mapping      mapping = new Mapping();

        try {
            // 1. Load the mapping information from the file
            mapping.loadMapping( "mapping.xml" );

            // 2. Unmarshal the data
            Unmarshaller unmar = new Unmarshaller(mapping);
            MyOrder order = (MyOrder)unmar.unmarshal(new InputSource(new FileReader("order.xml")));

            // 3. Do some processing on the data
            float total = order.getTotalPrice();
            System.out.println("Order total price = " + total);
            order.setTotal(total);

            // 4. marshal the data with the total price back and print the XML in the console
            Marshaller marshaller = new Marshaller(new OutputStreamWriter(System.out));
            marshaller.setMapping(mapping);
            marshaller.marshal(order);

        } catch (Exception e) {
            System.out.println(e);
            return;
        }
    }
}
```

1.2.5. xsi:type

Ordinarily, a mapping will only reference types that are concrete classes (i.e. not interfaces nor abstract classes). The reason is that to unmarshal a type requires instantiating it and one cannot instantiate an interface. However, in many real situations, object models depend on the use of interfaces. Many class properties are defined to have interface types to support the ability to swap implementations. This is often the case in frameworks.

The problem is that a different mapping must be used each time the same model is to be used to marshal/unmarshal an implementation that uses different concrete types. This is not convenient. The mapping should represent the model and the specific concrete type used to unmarshal a document is a configuration parameter; it should be specified in the instance document to be unmarshalled, not the mapping.

For example, assume a very simple object model of an engine that has one property that is a processor:

```
public interface IProcessor {
    public void process();
}

public class Engine {
    private IProcessor processor;
    public IProcessor getProcessor() {
        return processor;
    }
    public void setProcessor(IProcessor processor) {
        this.processor = processor;
    }
}
```

A typical mapping file for such a design may be:

```
<mapping>
    <class name="Engine">
        <map-to xml="engine" />

        <field name="processor" type="IProcessor" required="true">
           <bind-xml name="processor" node="element" />
```

```
      </field>

   </class>
</mapping>
```

It is possible to use such a mapping and still have the marshal/unmarshal process work by specifying the concrete implementation of IProcessor in the document to be unmarshalled, using the xsi:type attribute, as follows:

```
<engine>
   <processor xsi:type="java:com.abc.MyProcessor" />
</engine>
```

In this manner, one is still able to maintain only a single mapping, but vary the manner in which the document is unmarshalled from one instance document to the next. This flexibility is powerful because it enables the support of polymorphism within the castor xml marshalling framework.

Suppose we wanted the following XML instead:

```
<engine>
   <myProcessor/>
</engine>
```

In the above output our XML name changed to match the type of the class used instead of relying on the xsi:type attribute. This can be achieved by modifying the mapping file as such:

```
<mapping>
   <class name="Engine">
      <map-to xml="engine" />
      <field name="processor" type="IProcessor" required="true">
         <bind-xml auto-naming="deriveByClass" node="element" />
      </field>
   </class>

   <class name="MyProcessor">
      <map-to xml="myProcessor" />
   </class>

</mapping>
```

1.2.6. Location attribute

Since 0.9.5

The location attribute allows the user to map fields from nested elements or specify a wrapper element for a given field. Wrapper elements are simply elements which appear in the XML instance, but do not have a direct mapping to an object or field within the object model.

For example to map an instance of the following class:

```
public class Foo {

    private Bar bar = null;

    public Foo();

    public getBar() {
        return bar;
```

```
    }

    public void setBar(Bar bar) {
        this.bar = bar;
    }
}
```

into the following XML instance:

```
<?xml version="1.0"?>
<foo>;
   <abc>
      <bar>...</bar>
   </abc>
</foo>
```

(notice that an 'abc' field doesn't exist in the Bar class) One would use the following mapping:

```
<?xml version="1.0"?>
   ...
   <class name="Foo">
      <field name="bar" type="Bar">
         <bind-xml name="bar" location="abc"/>
      </field>
   </class>
   ...
</mapping>
```

Note the "location" attribute. The value of this attribute is the name of the wrapper element. To use more than one wrapper element, the name is separated by a forward-slash as such:

```
<bind-xml name="bar" location="abc/xyz" />
```

Note that the name of the element is not part of the location itself and that the location is always relative to the class in which the field is being defined. This works for attributes also:

```
<bind-xml name="bar" location="abc" node="attribute" />
```

will produce the following:

```
<?xml version="1.0"?>
<foo>
   <abc bar="..."/>;
</foo>
```

1.2.7. Tips

Some helpful hints...

1.2.7.1. Automatically create a mapping file

Castor comes with a tool that can automatically create a mapping from class files. Please see the XML FAQ for more information.

1.2.7.2. Create your own FieldHandler

Sometimes to handle complex situations you'll need to create your own FieldHandler. Normally a FieldHandler deals with a specific class and field, however generic, reusable FieldHandlers can also be created by extending org.exolab.castor.mapping.GeneralizedFieldHandler or org.exolab.castor.mapping.AbstractFieldHandler. The FieldHandler can be specified on the <field> element.

For more information on writing a custom FieldHandler please see the following: XML FieldHandlers.

1.2.7.3. Mapping constructor arguments (since 0.9.5)

You may map any attributes to constructor arguments. For more information on how to map constructor arguments see the information available in the section on set-method above.

Please note that mapping **elements** to constructor arguments is not yet supported.

Tip: the XML HOW-TO section has a HOW-TO document for mapping constructor arguments.

1.2.7.4. Preventing Castor from checking for a default constructor (since 0.9.5)

Sometimes it's useful to prevent Castor from checking for a default constructor, such as when trying to write a mapping for an interface or type-safe enum. You can use the "undocumented" verify-constructable="false" attribute on the <class> element to prevent Castor from looking for the default constructor.

1.2.7.5. Type safe enumeration mapping (since 0.9.5)

While you can always use your own custom FieldHandler for handling type-safe enumeration classes, Castor does have a built-in approach to dealing with these types of classes. If the type-safe enum class has a **public static <type> valueOf(String)** method Castor will call that method so that the proper instance of the enumeration is returned. Note: You'll also need to disable the default constructor check in the mapping file (see section 7.4 above to see more on this).

1.3. Configuring Castor XML (Un)Marshaller

1.3.1. Introduction

To be defined ...

1.3.2. Configuring the Marshaller

Before using the `Marshaller` class for marshalling Java objects to XML, the `Marshaller` can be fine-tuned according to your needs by calling a variety of set-methods on this class. This section enlists the available properties and provides you with information about their meaning, possible values and the default value.

Table 1.12. Marshaller properties

Name	Description	Values	Default	Since
suppressNamespaces		`true` or `false`	`false`	-

1.3.3. Configuring the Unmarshaller

Before using the `Unmarshaller` class for unmarshalling Java objects from XML, the `Unmarshaller` can be fine-tuned according to your needs by calling a variety of set-methods on this class. This section enlists the available properties and provides you with information about their meaning, possible values and the default value.

Table 1.13. Unmarshaller properties

Name	Description	Values	Default	Since
rootObject		A Class instance identifying the root class to use for unmarshalling.	-	-

1.4. Usage of Castor and XML parsers

1.4.1. SAX/DOM

Being an **XML data binding framework** by definition, Castor XML relies on the availability of an XML parser at run-time. In Java, an XML parser is by default accessed though either the DOM or the SAX APIs: that implies that the XML Parser used needs to comply with either (or both) of these APIs.

With the creation of the JAXP API (and its addition to the Java language definition as of Java 5.0), Castor internally has been enabled to allow usage of the JAXP interfaces to interface to XML parsers. As such, Castor XML allows the use of a JAXP-compliant XML parser as well.

By default, Castor ships with Apache Xerces 2.6.2. You may, of course, upgrade to a newer version of Apache Xerces at your convenience, or switch to any other XML parser as long as it is JAXP compliant or implements a particular SAX interface. Please note that users of Java 5.0 and above do not need to have Xerces available at run-time, as JAXP and Xerces have both been integrated into the run-time library of Java.

For marshalling, Castor XML can equally use any JAXP complaint XML parser (or interact with an XML parser that implements the SAX API), with the exception of the following special case: when using 'pretty printing' during marshalling (by setting the corresponding property in `castor.properties` to `true`) with Java 1.4 or below, Apache Xerces has to be on the classpath, as Castor XML internally uses Xerces' `XMLSerializer` to implement this feature.

The following table enlists the requirements relative to the Java version used in your environment.

Table 1.14. XML APIs on various Java versions

Java 1.4 and below	Java 5.0 and above
Xerces 2.6.2	-
XML APIs	-

1.4.2. StAX

As of Castor 1.3.2, Castor XML can be used with a StAX-compliant parser to unmarshal from XML. Please see Example 1.1, “XML produced in introspection mode” for StAX-specific `unmarshal` methods added to `org.exolab.castor.xml.Unmarshaller`.

1.5. XML configuration file

1.5.1. News

- Added a section on how to access the properties as defined in the Castor properties file from within code.
- **Release 1.2.1:** : Added new `org.exolab.castor.xml.lenient.integer.validation` property to allow configuration of leniency for validation for Java properties generated from `<xs:integer>` types during code generation.
- **Release 1.2:** : Access to the `org.exolab.castor.util.LocalConfiguration` class has been removed completely. To access the properties as used by Castor from code, please refer to the below section.
- **Release 1.1.3:** Added special processing of proxied classes. The property org.exolab.castor.xml.proxyInterfaces allows you to specify a list of interfaces that such proxied objects implement. If your object implements one of these interfaces Castor will not use the class itself but its superclass at introspection or to find class mappings and ClassDescriptors.
- **Release 0.9.7:** Added new org.exolab.castor.persist.useProxies property to allow configuration of JDBC proxy classes. If enabled, JDBC proxy classes will be used to wrap `java.sql.Connection` and `java.sql.PreparedStatement` instances, to allow for more detailed and complete JDBC statements to be output during logging. When turned off, no logging statements will be generated at all.

1.5.2. Introduction

Castor uses a configuration file for environmental properties that are shared across all the Castor sub systems. The configuration file is specified as a Java properties file with the name `castor.properties`.

By definition, a default configuration file is included with the Castor XML JAR. Custom properties can be supplied using one of the following methods. Please note that the custom properties specified will **override** the default configuration.

- Place a file named `castor.properties` anywhere on the classpath of your application.
- Place a file named `castor.properties` in the working directory of your application.
- Use the system property `org.castor.user.properties.location` to specify the location of your custom properties.

Please note that Castor XML - upon startup - will try the methods given above in exactly the sequence as stated above; if it managed to find a custom property file using any of the given methods, it will cancel its search.

When running the provided examples, Castor will use the configuration file located in the examples directory which specifies additional debugging information as well as pretty printing of all produced XML documents.

The following properties are currently supported in the configuration file:

Table 1.15.

Name	Description	Values	Default	Since
org.exolab.castor.xml.introspector.primitive.nodetype	Property specifying the type of XML node to use for primitive values, either `element` or `attribute`	`element` or `attribute`	`attribute`	-
org.exolab.castor.parser	Property specifying the class name of the SAX XML parser to use.	-	-	-
org.exolab.castor.parser.validation	Specifies whether to perform XML document validation by default.	`true` and `false`	`false`	-
org.exolab.castor.parser.namespaces	Specifies whether to support XML namespaces by default.	`false` and `true`	`false`	-
org.exolab.castor.xml.nspackages	Specifies a list of XML namespace to Java package mappings.	-	-	-
org.exolab.castor.xml.naming	Property specifying the 'type' of the XML naming conventions to use. Values of this property must be either `mixed` , `lower` , or the name of a class which extends `org.exolab.castor.xml.XMLNaming` .	`mixed` , `lower` , or the name of a class which extends `org.exolab.castor.xml.XMLNaming`	`lower`	-
org.castor.xml.java.naming	Property specifying the 'type' of the Java naming conventions to use. Values of this property must be either `null` or the name of a class which extends `link org.castor.xml.JavaNaming`.	`null` or the name of a class which extends `link org.castor.xml.JavaNaming`.	`null`	-

Name	Description	Values	Default	Since
org.exolab.castor.marshalling.validation	Specifies whether to use validation during marshalling.	`false` or `true`	`true`	-
org.exolab.castor.indent	Specifies whether XML documents (as generated at marshalling) should use indentation or not.	`false` or `true`	`false`	-
org.exolab.castor.sax.features	Specifies additional features for the XML parser.	A comma separated list of SAX (parser) features (that might or might not be supported by the specified SAX parser).	`http://apache.org/xml/features/disallow-doctyp`	-
org.exolab.castor.sax.features-to-disable	Specifies features to be disbaled on the underlying SAX parser.	A comma separated list of SAX (parser) features to be disabled.	`http://xml.org/sax/features/external-general-en` `http://xml.org/sax/features/external-parameter` `http://apache.org/xml/features/nonvalidating/l`	**1.0.4**
org.exolab.castor.regexp	Specifies the regular expression validator to use.	A class that implements `org.exolab.castor.util.RegExpValidator`.	-	-
org.exolab.castor.xml.strictelements	Specifies whether to apply strictness to elements when unmarshalling. When enabled, the existence of elements in the XML document, which cannot be mapped to a class, causes a {@link SAXException} to be thrown. If set to false, these 'unknown' elements are ignored.	`false` or `true`	`false`	-
org.exolab.castor.xml.loadPackageMappings	Specifies whether the ClassDescriptorResolver should (automatically) search for and consult with package mapping	`false` or `true`	`true`	**1.0.2**

Name	Description	Values	Default	Since
	files (`.castor.xml`) to retrieve class descriptor information			
org.exolab.castor.xml.serializer.factory	Specifying what XML serializers factory to use.	A class name	org.exolab.castor.xml.XercesXMLSerializerFactory	**1.0**
org.exolab.castor.xml.lenient.sequence.order	Specifies whether sequence order validation should be lenient.	`false` or `true`	`false`	**1.1**
org.exolab.castor.xml.lenient.id.validation	Specifies whether id/href validation should be lenient.	`false` or `true`	`false`	**1.1**
org.exolab.castor.xml.proxyInterfaces	Specifies whether or not to search for an proxy interface at marshalling. If property is not empty the objects to be marshalled will be searched if they implement one of the given interface names. If the interface is implemented, the superclass will be marshalled instead of the class itself.	A list of proxy interfaces	-	**1.1.3**
org.exolab.castor.xml.lenient.integer.validation	Specifies whether validation for Java properties generated from <xs:integer> should be lenient, i.e. allow for `int` s as well.	`false` or `true`	`false`	**1.2.1**
org.exolab.castor.xml.version	Specifies the XML document version number to be used during marshalling; defaults to 1.0.	`1.0` or `1.1`	`1.0`	**1.3.2**

Note

As of Castor 1.3.3, the default values for `org.exolab.castor.sax.features` and `org.exolab.castor.sax.features-to-disable` have changed to include/disable selected features.

1.5.3. Accessing the properties from within code

As of Castor 1.1, it is possible to read and set the value of properties programmatically using the `getProperty(String)` and `setProperty(String,String)` on the following classes:

- `org.exolab.castor.xml.Unmarshaller`
- `org.exolab.castor.xml.Marshaller`
- `org.exolab.castor.xml.XMLContext`

Whilst using the setter methods on the first two classes will change the settings of the respective instances only, using the `setProperty()` method on the `org.exolab.castor.xml.XMLContext` class will change the configuration globally, and affect all `org.exolab.castor.xml.Unmarshaller` and `org.exolab.castor.xml.Marshaller` instances created thereafter using the `org.exolab.castor.xml.XMLContext.createUnmarshaller()` and `org.exolab.castor.xml.XMLContext.createMarshaller()` methods.

1.6. Castor XML - Tips & Tricks

1.6.1. Logging and Tracing

When developing using Castor, we recommend that you use the various `setLogWriter` methods to get detailed information and error messages.

Using a logger with `org.exolab.castor.mapping.Mapping` will provide detailed information about mapping decisions made by Castor and will show the SQL statements being used.

Using a logger with `org.exolab.castor.jdo.JDO` will provide trace messages that show when Castor is loading, storing, creating and deleting objects. All database operations will appear in the log; if an object is retrieved from the cache or is not modified, there will be no trace of load/store operations.

Using a logger with `org.exolab.castor.xml.Unmarshaller` will provide trace messages that show conflicts between the XML document and loaded objects.

A simple trace logger can be obtained from `org.exolab.castor.util.Logger`. This logger uses the standard output stream, but prefixes each line with a short message that indicates who generated it. It can also print the time and date of each message. Since logging is used for warning messages and simple tracing, Castor does not require a sophisticated logging mechanism.

Interested in integratating Castor's logging with Log4J? Then see this question in the JDO FAQ.

1.6.2. Indentation

By default the marshaler writes XML documents without indentation. When developing using Castor or when debugging an application that uses Castor, it might be desireable to use indentation to make the XML documents human-readable. To turn indentation on, modify the Castor properties file, or create a new properties file in the classpath (named `castor.properties`) with the following content:

```
org.exolab.castor.indent=true
```

Indentation inflates the size of the generated XML documents, and also consumes more CPU. It is recommended not to use indentation in a production environment.

1.6.3. XML:Marshal validation

It is possible to disable the validation in the marshaling framework by modifying the Castor properties file or by creating a new properties file in the classpath (named `castor.properties`) with the following content:

```
org.exolab.castor.marshalling.validation=false
```

1.6.4. NoClassDefFoundError

Check your CLASSPATH, check it often, there is no reason not to!

1.6.5. Mapping: auto-complete

Note

This only works with Castor-XML.

To save time when writing your mappings, try using the *auto-complete* attribute of *class*. When using auto-complete, Castor will introspect your class and automatically fill in any missing fields.

Example:

```
<class name="com.acme.Foo" auto-complete="true"/>
```

This is also compatible with generated descriptor files. You can use a mapping file to override some of the behavior of a compiled descriptor by using auto-complete.

Note

Be careful to make sure you use the exact field name as specified in the generated descriptor file in order to modify the behavior of the field descriptor! Otherwise, you'll probably end up with two fields being marshaled!

1.6.6. Create method

Castor requires that classes have a public, no-argument constructor in order to provide the ability to marshal & unmarshal objects of that type.

create-method is an optional attribute to the `<field>` mapping element that can be used to overcome this restriction in cases where you have an existing object model that consists of, say, singleton classes where public, no-argument constructors must not be present by definition.

Assume for example that a class "`A`" that you want to be able to unmarshal uses a singleton class as one of its properties. When attempting to unmarshal class "`A`", you should get an exception because the singleton property has no public no-arg constructor. Assuming that a reference to the singleton can be obtained via a static getInstance() method, you can add a "create method" to class `A` like this:

```
public MySingleton getSingletonProperty() {
   return MySingleton.getInstance();
}
```

and in the mapping file for class `A`, you can define the singleton property like this:

```
<field name="mySingletonProperty"
      type="com.u2d.MySingleton"
      create-method="getSingletonProperty">
   <bind-xml name="my-singleton-property" node="element" />
</field>
```

This illustrates how the create-method attribute is quite a useful mechanism for dealing with exceptional situations where you might want to take advantage of marshaling even when some classes do not have no-argument public constructors.

Note

As of this writing, the specified create-method must exist as a method in the current class (i.e. the class being described by the current `<class>` element). In the future it may be possible to use external static factory methods.

1.6.7. MarshalListener and UnmarshalListener

Castor allows control on the object being marshaled or unmarshaled by a set of two listener interfaces: MarshalListener and UnmarshalListener.

The MarshalListener interface located in `org.exolab.castor.xml` listens to two different events that are intercepted by the following methods:

- preMarshal: this method is called before an object gets marshaled.
- postMarshal: this method is called once an object has been marshaled.

The UnmarshalListener located also in `org.castor.xml` listens to four different events that are intercepted by the following methods:

- initialized: this method is called once an object has been instantiated.
- attributesProcessed: this method is called when the attributes have just been read and processed.
- fieldAdded: this method is called when an object is added to a parent.
- unmarshalled: this method is called when an object has been **fully** unmarshaled

Note: The `UnmarshalListener` had been part of `org.exolab.castor.xml` but as an extention of this interface had been required a new interface in `org.castor.xml` was introduced. Currently the `org.exolab.castor.xml.UnmarshalListener` interface can still be used but is deprecated.

1.7. Castor XML: Writing Custom FieldHandlers

1.7.1. Introduction

Sometimes we need to deal with a data format that Castor doesn't support out-of-the-box, such as an unsupported Date/Time representation, or we want to wrap and unwrap fields in Wrapper objects to get the desired XML output without changing our object model. To handle these cases Castor allows specifying a custom `org.exolab.castor.mapping.FieldHandler` which can do these varying conversions during calls to the fields setter and getter methods.

Note

The *FieldHandler* is the basic interface used by the Castor Framework when accessing field values or setting them. By specifying a custom *FieldHandler* in the mapping file we can basically intercept the calls to retrieve or set a field's value and do whatever conversions are necessary.

1.7.2. Writing a simple FieldHandler

When a writing a FieldHandler handler we need to provide implementations of the various methods specified in the FieldHandler interface. The main two methods are the *getValue* and *setValue* methods which will basically handle all our conversion code. The other methods provide ways to create a new instance of the field's value or reset the field value.

Tip

It's actually even easier to write custom field handlers if we use a GeneralizedFieldHandler. See more details in Section 1.7.3, “Writing a GeneralizedFieldHandler”

Let's take a look at how to convert a date in the format YYYY-MM-DD using a custom FieldHandler. We want to marshal the following XML input file `text.xml`:

```
<?xml version="1.0"?>
<root>2004-05-10</root>
```

The class we'll be marshalling from and unmarshalling to looks as follows:

```
import java.util.Date;

public class Root {

    private Date _date;

    public Root() {
        super();
    }

    public Date getDate() {
        return _date;
    }


    public void setDate(final Date date) {
        _date = date;
    }
```

So we need to write a custom FieldHandler that takes the input String and converts it into the proper java.util.Date instance:

```
import org.exolab.castor.mapping.FieldHandler;
import org.exolab.castor.mapping.FieldDescriptor;
import org.exolab.castor.mapping.ValidityException;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * The FieldHandler for the Date class
 *
 */
public class MyDateHandler implements FieldHandler
{

    private static final String FORMAT = "yyyy-MM-dd";

    /**
     * Creates a new MyDateHandler instance
     */
    public MyDateHandler() {
        super();
    }

    /**
     * Returns the value of the field from the object.
     *
     * @param object The object
     * @return The value of the field
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     */
    public Object getValue(final Object object) throws IllegalStateException {
        Root root = (Root)object;
        Date value = root.getDate();
        if (value == null) return null;
        SimpleDateFormat formatter = new SimpleDateFormat(FORMAT);
        Date date = (Date)value;
        return formatter.format(date);
    }


    /**
     * Sets the value of the field on the object.
     *
     * @param object The object
     * @param value The new value
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     * @throws IllegalArgumentException The value passed is not of
     *  a supported type
     */
    public void setValue(Object object, Object value)
       throws IllegalStateException, IllegalArgumentException {

        Root root = (Root)object;
        SimpleDateFormat formatter = new SimpleDateFormat(FORMAT);
        Date date = null;
        try {
            date = formatter.parse((String)value);
        }
        catch(ParseException px) {
            throw new IllegalArgumentException(px.getMessage());
        }
        root.setDate(date);
    }


    /**
     * Creates a new instance of the object described by this field.
     *
     * @param parent The object for which the field is created
```

```
     * @return A new instance of the field's value
     * @throws IllegalStateException This field is a simple type and
     *  cannot be instantiated
     */
    public Object newInstance(Object parent) throws IllegalStateException {
        //-- Since it's marked as a string...just return null,
        //-- it's not needed.
        return null;
    }


    /**
     * Sets the value of the field to a default value.
     *
     * Reference fields are set to null, primitive fields are set to
     * their default value, collection fields are emptied of all
     * elements.
     *
     * @param object The object
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     */
    public void resetValue(Object object) throws IllegalStateException, IllegalArgumentException {
        ((Root)object).setDate(null);
    }
}
```

Tip

The *newInstance* method should return null for immutable types.

Note

There is also an `org.exolab.castor.mapping.AbstractFieldHandler` that we can extend instead of implementing FieldHandler directly. Not only do we not have to implement deprecated methods, but we can also gain access to the *FieldDescriptor* used by Castor.

In order to tell Castor that we want to use our Custom FieldHandler we must specify it in the mapping file `mapping.xml`:

```
<?xml version="1.0"?>
<mapping>
  <class name="Root">
     <field name="date" type="string" handler="MyDateHandler">
        <bind-xml node="text"/>
     </field>
  </class>
</mapping>
```

We can now use a simple Test class to unmarshal our XML document:

```
import java.io.*;
import org.exolab.castor.xml.*;
import org.exolab.castor.mapping.*;

public class Test {

    public static void main(String[] args) {
            try {

               //--load mapping
               Mapping mapping = new Mapping();
               mapping.loadMapping("mapping.xml");
```

```
            System.out.println("unmarshalling root instance:");
            System.out.println();

            Reader reader = new FileReader("test.xml");
            Unmarshaller unmarshaller = new Unmarshaller(Root.class);
            unmarshaller.setMapping(mapping);
            Root root = (Root) unmarshaller.unmarshal(reader);
            reader.close();

            System.out.println("Root#getDate : " + root.getDate());
             }
             catch (Exception e) {
                 e.printStackTrace();
             }
    }
}
```

Now simply compile the code and run!

```
% java Test
unmarshalling root instance:

Root#getDate : Mon May 10 00:00:00 CDT 2004
```

After running our test program we can see that Castor invoked our custom FieldHandler and we got our properly formatted date in our Root.class.

1.7.3. Writing a GeneralizedFieldHandler

A `org.exolab.castor.mapping.GeneralizedFieldHandler` is an extension of FieldHandler interface where we simply write the conversion methods and Castor will automatically handle the underlying get/set operations. This allows us to re-use the same FieldHandler for fields from different classes that require the same conversion.

Note

Note: Currently the GeneralizedFieldHandler cannot be used from a *binding-file* for use with the SourceGenerator, an enhancement patch will be checked into SVN for this feature, shortly after 0.9.6 final is released.

The same FieldHandler we used above can be written as a GeneralizedFieldHandler as such:

```
import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.FieldDescriptor;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * The FieldHandler for the Date class
 *
 */
public class MyDateHandler extends GeneralizedFieldHandler {

    private static final String FORMAT = "yyyy-MM-dd";

    /**
     * Creates a new MyDateHandler instance
     */
    public MyDateHandler() {
```

```
        super();
    }

    /**
     * This method is used to convert the value when the
     * getValue method is called. The getValue method will
     * obtain the actual field value from given 'parent' object.
     * This convert method is then invoked with the field's
     * value. The value returned from this method will be
     * the actual value returned by getValue method.
     *
     * @param value the object value to convert after
     *  performing a get operation
     * @return the converted value.
     */
    public Object convertUponGet(Object value) {
        if (value == null) return null;
        SimpleDateFormat formatter = new SimpleDateFormat(FORMAT);
        Date date = (Date)value;
        return formatter.format(date);
    }


    /**
     * This method is used to convert the value when the
     * setValue method is called. The setValue method will
     * call this method to obtain the converted value.
     * The converted value will then be used as the value to
     * set for the field.
     *
     * @param value the object value to convert before
     *  performing a set operation
     * @return the converted value.
     */
    public Object convertUponSet(Object value) {
        SimpleDateFormat formatter = new SimpleDateFormat(FORMAT);
        Date date = null;
        try {
            date = formatter.parse((String)value);
        }
        catch(ParseException px) {
            throw new IllegalArgumentException(px.getMessage());
        }
        return date;
    }

    /**
     * Returns the class type for the field that this
     * GeneralizedFieldHandler converts to and from. This
     * should be the type that is used in the
     * object model.
     *
     * @return the class type of of the field
     */
    public Class getFieldType() {
        return Date.class;
    }

    /**
     * Creates a new instance of the object described by
     * this field.
     *
     * @param parent The object for which the field is created
     * @return A new instance of the field's value
     * @throws IllegalStateException This field is a simple
     *  type and cannot be instantiated
     */
    public Object newInstance(Object parent) throws IllegalStateException
    {
        //-- Since it's marked as a string...just return null,
        //-- it's not needed.
        return null;
    }

}
```

Everything else is the same. So we can re-run our test case using this GeneralizedFieldHandler and we'll get the same result. The main difference is that we implement the *convertUponGet* and *convertUponSet* methods.

Notice that we never reference the *Root* class in our `GeneralizedFieldHandler`. This allows us to use the same exact `FieldHandler` for any field that requires this type of conversion.

1.7.4. Use ConfigurableFieldHandler for more flexibility

In some situations, the `GeneralizedFieldHandler` might not provide sufficient flexibility. Suppose your XML document uses more than one date format. You could solve this by creating a `GeneralizedFieldHandler` subclass per date format, but that would lead to code duplication, which in itself is not desirable.

A `ConfigurableFieldHandler` is a `FieldHandler` that can be configured in the mapping file with any kind and any number of parameters. You can simply configure two (or more) instances of the same `ConfigurableFieldHandler` class with different date format patterns. Here's a mapping file that uses a `ConfigurableFieldHandler` to marshal and unmarshal the date field, similar to the preceding examples:

```
<?xml version="1.0"?>
<mapping>

   <field-handler name="myHandler" class="FieldHandlerImpl">
      <param name="date-format" value="yyyyMMddHHmmss"/>
   </field-handler>

   <class name="Root">
      <field name="date" type="string" handler="myHandler"/>
   </class>

</mapping>
```

The *field-handler* element defines the `ConfigurableFieldHandler`. The class must be an implementation of the *org.exolab.castor.mapping.ConfigurableFieldHandler* interface. This instance is configured with a date format. However, each implementation can decide which, and how many parameters to use.

The field handler instance is referenced by the *field* element, using the *handler* attribute.

Here's the ConfigurableFieldHandler implementation:

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Date;
import java.util.Properties;

import org.exolab.castor.mapping.ConfigurableFieldHandler;
import org.exolab.castor.mapping.FieldHandler;
import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.ValidityException;

public class FieldHandlerImpl implements FieldHandler, ConfigurableFieldHandler {

    private DateFormat formatter;

    public void setConfiguration(final Properties config) throws ValidityException {
            String pattern = config.getProperty("date-format");
            if (pattern == null) {
                    throw new ValidityException("Required parameter \"date-format\" is missing for FieldHandlerI
            }
            try {
                    formatter = new SimpleDateFormat(pattern);
            } catch (IllegalArgumentException e) {
```

```
                    throw new ValidityException("Pattern \""+pattern+"\" is not a valid date format.");
            }
    }

    /**
     * Returns the value of the field from the object.
     *
     * @param object The object
     * @return The value of the field
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     */
    public Object getValue(Object object) throws IllegalStateException {
        Root root = (Root)object;
        Date value = root.getDate();
        if (value == null) return null;
        return formatter.format(value);
    }

    /**
     * Sets the value of the field on the object.
     *
     * @param object The object
     * @param value The new value
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     * @throws IllegalArgumentException The value passed is not of
     *  a supported type
     */
    public void setValue(Object object, Object value)
        throws IllegalStateException, IllegalArgumentException {
        Root root = (Root)object;
        Date date = null;
        try {
            date = formatter.parse((String)value);
        }
        catch(ParseException px) {
            throw new IllegalArgumentException(px.getMessage());
        }
        root.setDate(date);
    }

    /**
     * Creates a new instance of the object described by this field.
     *
     * @param parent The object for which the field is created
     * @return A new instance of the field's value
     * @throws IllegalStateException This field is a simple type and
     *  cannot be instantiated
     */
    public Object newInstance(Object parent)
        throws IllegalStateException
    {
        //-- Since it's marked as a string...just return null,
        //-- it's not needed.
        return null;
    }

    /**
     * Sets the value of the field to a default value.
     *
     * Reference fields are set to null, primitive fields are set to
     * their default value, collection fields are emptied of all
     * elements.
     *
     * @param object The object
     * @throws IllegalStateException The Java object has changed and
     *  is no longer supported by this handler, or the handler is not
     *  compatible with the Java object
     */
    public void resetValue(Object object)
        throws IllegalStateException, IllegalArgumentException {
        ((Root)object).setDate(null);
    }

}
```

This implementation is similar to the first *MyDateHandler* example on this page, except that is adds a *setConfiguration* method as specified by the *ConfigurableFieldHandler* interface. All parameters that are configured in the mapping file will be passed in as a *Properties* object. The implementing method is responsible for processing the configuration data.

As a convenience, *org.exolab.castor.mapping.AbstractFieldHandler* already implements *ConfigurableFieldHandler*. However, the *setConfiguration* method is not doing anything. Any subclass of *AbstractFieldHandler* only has to override this method to leverage the configuration capabilities. Since *AbstractFieldHandler* and its subclass *GeneralizedFieldHandler* are useful abstract classes, you'd probably want to use them anyway. It eliminates the need to implement the *ConfigurableFieldHandler* interface yourself.

1.7.5. Reuse a `ConfigurableFieldHandler` for more than one field definition

Imagine a scenario where you want to use above `ConfigurableFieldHandler` instance for more than one field - a valid use case as it promotes reuse.

```
<?xml version="1.0"?>
<mapping>

   <field-handler name="myFirstHandler" class="FieldHandlerImpl">
      <param name="date-format" value="yyyyMMddHHmmss"/>
   </field-handler>

   <field-handler name="mySecondHandler" class="FieldHandlerImpl">
      <param name="date-format" value="yyyy-MM-ddHH:mm:ss"/>
   </field-handler>

   <class name="Root">
      <field name="firstDate" type="string" handler="myFirstHandler"/>
      <field name="secondDate" type="string" handler="myFirstHandler"/>
      <field name="thirdDate" type="string" handler="mySecondHandler"/>
   </class>

</mapping>
```

For this to work, there's one more thing you will have to do: your `ConfigurableFieldHandler` implementation has to implement the `ClonableFieldHandlerMarker` interface and implement the `copyFieldHandler()` method. As indicated by the name, please return a clone/copy of your `FieldHandler` instance ... and you are all set.

A simplified sample implementation could look as follows, extending the `FieldHandlerImpl` class from the previous section:

```
public class FieldHandlerImpl implements FieldHandler, ConfigurableFieldHandler, ClonableFieldHandlerMarker {

    private DateFormat format;

    ...

    public void setFormat(DateFormat format) {
        this.format = format;
    }

    @Override
    public FieldHandler copyFieldHandler() {
        FieldHandlerImpl handler = new FieldHandlerImpl();
        handler.setFormat(this.getFormat());
        return handler;
    }

}
```

1.7.6. No Constructor, No Problem!

A number of classes such as type-safe enum style classes have no constructor, but instead have some sort of static factory method used for converting a string value into an instance of the class. With a custom FieldHandler we can allow Castor to work nicely with these types of classes.

Tip

Castor XML automatically supports these types of classes if they have a specific method:

```
public static {Type} valueOf(String)
```

Note

We're working on the same support for Castor JDO

Even though Castor XML supports the "valueOf" method type-safe enum style classes, we'll show you how to write a custom handler for these classes anyway since it's useful for any type of class regardless of the name of the factory method.

Let's look at how to write a handler for the following type-safe enum style class, which was actually generated by Castor XML (javadoc removed for brevity):

```
import java.io.Serializable;
import java.util.Enumeration;
import java.util.Hashtable;

public class Color implements java.io.Serializable {

    public static final int RED_TYPE = 0;

    public static final Color RED = new Color(RED_TYPE, "red");

    public static final int GREEN_TYPE = 1;

    public static final Color GREEN = new Color(GREEN_TYPE, "green");

    public static final int BLUE_TYPE = 2;

    public static final Color BLUE = new Color(BLUE_TYPE, "blue");

    private static java.util.Hashtable _memberTable = init();

    private int type = -1;

    private java.lang.String stringValue = null;


    private Color(int type, java.lang.String value) {
        super();
        this.type = type;
        this.stringValue = value;
    } //-- test.types.Color(int, java.lang.String)


    public static java.util.Enumeration enumerate()
    {
        return _memberTable.elements();
    } //-- java.util.Enumeration enumerate()

    public int getType()
```

```
    {
        return this.type;
    } //-- int getType()

    private static java.util.Hashtable init()
    {
        Hashtable members = new Hashtable();
        members.put("red", RED);
        members.put("green", GREEN);
        members.put("blue", BLUE);
        return members;
    } //-- java.util.Hashtable init()

    public java.lang.String toString()
    {
        return this.stringValue;
    } //-- java.lang.String toString()

    public static Color valueOf(java.lang.String string)
    {
        Object obj = null;
        if (string != null) obj = _memberTable.get(string);
        if (obj == null) {
            String err = "'" + string + "' is not a valid Color";
            throw new IllegalArgumentException(err);
        }
        return (Color) obj;
    } //-- test.types.Color valueOf(java.lang.String)

}
```

The *GeneralizedFieldHandler* for the above *Color* class is as follows (javadoc removed for brevity):

```

import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.FieldDescriptor;

/**
 * The FieldHandler for the Color class
**/
public class ColorHandler
    extends GeneralizedFieldHandler
{

    public ColorHandler() {
        super();
    }

    public Object convertUponGet(Object value) {
        if (value == null) return null;
        Color color = (Color)value;
        return color.toString();
    }


    public Object convertUponSet(Object value) {
        return Color.valueOf((String)value);
    }

    public Class getFieldType() {
        return Color.class;
    }

    public Object newInstance( Object parent )
        throws IllegalStateException
    {
        //-- Since it's marked as a string...just return null,
        //-- it's not needed.
        return null;
    }

}
```

That's all there really is to it. Now we just need to hook this up to our mapping file and run a sample test.

If we have a root class *Foo* as such:

```
public class Foo {

    private Color _color = null;
    private int _size = 0;
    private String _name = null;

    public Foo() {
        super();
    }

    public Color getColor() {
        return _color;
    }

    public String getName() {
        return _name;
    }

    public int getSize() {
        return _size;
    }

    public void setColor(Color color) {
        _color = color;
    }

    public void setName(String name) {
        _name = name;
    }

    public void setSize(int size) {
        _size = size;
    }

}
```

Our mapping file would be the following:

```
<?xml version="1.0"?>
<mapping>
  <class name="Foo">
     <field name="size" type="integer">
        <bind-xml node="element"/>
     </field>
     <field name="name" type="string"/>
     <field name="color" type="string" handler="ColorHandler"/>
  </class>
</mapping>
```

We can now use our custom FieldHandler to unmarshal the following xml input:

```
<?xml version="1.0"?>
<foo>
   <name>MyFoo</name>
   <size>345</size>
   <color>blue</color>
</foo>
```

A sample test class is as follows:

```
import java.io.*;
import org.exolab.castor.xml.*;
import org.exolab.castor.mapping.*;
```

```
public class Test {

    public static void main(String[] args) {
            try {

                //--load mapping
                Mapping mapping = new Mapping();
                mapping.loadMapping("mapping.xml");

            System.out.println("unmarshalling Foo:");
            System.out.println();

            Reader reader = new FileReader("test.xml");
            Unmarshaller unmarshaller = new Unmarshaller(Foo.class);
            unmarshaller.setMapping(mapping);
            Foo foo = (Foo) unmarshaller.unmarshal(reader);
            reader.close();

            System.out.println("Foo#size : " + foo.getSize());
            System.out.print("Foo#color: ");
            if (foo.getColor() == null) {
                System.out.println("null");
            }
            else {
                System.out.println(foo.getColor().toString());
            }

            PrintWriter pw = new PrintWriter(System.out);
            Marshaller marshaller = new Marshaller(pw);
            marshaller.setMapping(mapping);
            marshaller.marshal(foo);
            pw.flush();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
    }
}
```

1.7.7. Collections and FieldHandlers

Note

With Castor 0.9.6 and later, the *GeneralizedFieldHandler* automatically supports iterating over the items of a collection and passing them one-by-one to the *convertUponGet*.

For backward compatibility or to handle the collection iteration yourself, simply add the following to the constructor of your `GeneralizedFieldHandler` implementation:

```
setCollectionIteration(false);
```

If you're going to be using custom field handlers for collection fields with a `GeneralizedFieldHandler` using versions of Castor prior to 0.9.6, then you'll need to handle the collection iteration yourself in the *convertUponGet* method.

If you're not using a `GeneralizedFieldHandler`, then you'll need to handle the collection iteration yourself in the *FieldHandler#getValue()* method.

Tip

Since Castor incrementally adds items to collection fields, there usually is no need to handle collections directly in the *convertUponSet* method (or the *setValue()* for those not using `GeneralizedFieldHandler`).

1.8. Best practice

There are many users of Castor XML who (want to) use Castor XML in in high-volume applications. To fine-tune Castor for such an environment, it is necessary to understand many of the product features in detail and to be able to balance their use according to the application needs. Even though many of these features are documented in various places, people frequently asked for a 'best practices' document, a document that brings together these technical topics in one place and that presents them as a set of easy-to-use recipes.

Please be aware that this document is *under construction*. But still we believe that this document -- even when in its conception phase -- provides valuable information to users of Castor XML.

1.8.1. General

1.8.1.1. Source Generator

It is not generally recommended to generate code into the default package, especially since code in the default package cannot be referenced from code in any other package.

Additionally, we recommend that generated code go into a different package then the code that makes use of the generated code. For example, if your application uses Castor to process an XML configuration file that is used by code in the package `org.example.userdialog` then we do not recommend that the generated code also go into that package. However, it would be reasonable to generate source to process this XML configuration file into the package `org.example.userdialog.xmlconfig`.

1.8.2. Performance Considerations

1.8.2.1. General

Creating instances of `org.exolab.castor.xml.Marshaller` and `org.exolab.castor.xml.Unmarshaller` for the purpose of XML data binding is easy to achieve at the API usage level. However, details of API use have an impact on application performance; each instance creation involves setup operations.

This is generally not an issue for one-off invocations; however, in a multi-threaded, high volume use scenario this can be become a serious issue. Internally, Castor uses a collection of *Descriptor* classes to keep information about the Java entities to be marshaled and unmarshaled. With each instance creation of (Un)Marshaller, this collection will be built from scratch (again and again).

To avoid this initial configuration 'penalty', Castor allows you to cache these Descriptor classes through its `org.exolab.castor.xml.ClassDescriptorResolver` component. This cache allows reuse of these Descriptor instances between (Un)Marshaller invocations.

1.8.2.2. Use of XMLContext - With and without a mapping file

With the introduction of the new `org.exolab.castor.xml.XMLContext` class, the use of a `ClassDescriptorResolver` has been greatly simplified in that such an instance is managed by the XMLContext per default. As such, there's no need to pass a `ClassDescriptorResolver` instance to `Marshaller`/`Unmarshaller` instances anymore, as this is done automatically when such instances are created through

- `org.exolab.castor.xml.XMLContext.createMarshaller()`
- `org.exolab.castor.xml.XMLContext.createUnmarshaller()`

For example, to create a `Marshaller` instance that is pre-configured with an instance of `ClassDescriptorResolver`, use the following code fragment:

```
Mapping mapping = new Mapping();
mapping.loadMapping(new InputSource(...));

XMLContext context = new XMLContext();
context.addMapping(mapping);

Marshaller marshaller = context.createMarshaller();
```

In the case where no mapping file is used, it is still possible to instruct the `org.exolab.castor.xml.XMLContext` to *pre-load* class descriptors for a given package via the methods enlisted below.

As above, create an instance of `org.exolab.castor.xml.XMLContext` and configure it according to your needs as shown below:

```
XMLContext context = new XMLContext();
context.addPackage("your.package.name");

Marshaller marshaller = context.createMarshaller();
```

The `org.exolab.castor.xml.XMLContext` class provides for various methods to load class descriptors for individual classes and/or packages.

Table 1.16. Methods on XMLContext to create Un-/Marshaller objects

Method	Description	`.castor.cdr`
`addClass(Class)` on `org.exolab.castor.xml.XMLContext`	Loads the class descriptor for one class.	n/a
`addClass(Class[])` on `org.exolab.castor.xml.XMLContext`	Loads the class descriptors for a collection of classes.	n/a
`addPackage(String)` on `org.exolab.castor.xml.XMLContext`	Loads the class descriptor for all classes in the defined package.	Required
`addPackages(String[])` on `org.exolab.castor.xml.XMLContext`	Loads the class descriptor for all classes in the defined packages.	Required

Note

For some of the methods, pre-loading class descriptords will only work if you provide the `.castor.cdr` file with your generated classes (as generated by the XML code generator). If no such file is shipped, Castor will not be able to pre-load the descriptors, and will fall back to its default descriptor loading mechanism.

1.8.2.3. Use of Marshaller/Unmarshaller

1.8.2.3.1. Use of ClassDescriptorResolver

When you do not use the `XMLContext` class, you will have to manually manage your `org.exolab.castor.xml.XMLClassDescriptorResolver`. To do so, first create an instance of `org.exolab.castor.xml.XMLClassDescriptorResolver` using the following code fragment:

```
XMLClassDescriptorResolver classDescriptorResolver =
   (XMLClassDescriptorResolver) ClassDescriptorResolverFactory.createClassDescriptorResolver(BindingType.XML);
MappingUnmarshaller mappingUnmarshaller = new MappingUnmarshaller();
MappingLoader mappingLoader =
   mappingUnmarshaller.getMappingLoader(mapping, BindingType.XML);
classDescriptorResolver.setMappingLoader(mappingLoader);
```

and then reuse this instance as shown below:

```
   Unmarshaller unmarshaller = new Unmarshaller();
   unmarshaller.setResolver(classDescriptorResolver);
   unmarshaller.unmarshal(...);
```

1.8.2.3.2. Use of ClassDescriptorResolver for pre-loading compiled descriptors

When you are not using a mapping file, but you have generated Java classes and their corresponding descriptor classes using the Castor XML code generator, you might want to instruct the `org.exolab.castor.xml.XMLClassDescriptorResolver` to *pre-load* class descriptors (as enumerated explicitly or for a given package) using various `add*` methods.

As above, create an instance of `org.exolab.castor.xml.XMLClassDescriptorResolver">XMLClassDescriptorResolver` using the following code fragment:

```
   XMLClassDescriptorResolver classDescriptorResolver = (XMlClassDescriptorResolver)
      ClassDescriptorResolverFactory.createClassDescriptorResolver(BindingType.XML);
   classDescriptorResolver.setClassLoader(...);
   classDescriptorResolver.addClass("your.package.name.A");
   classDescriptorResolver.addClass("your.package.name.B");
   classDescriptorResolver.addClass("your.package.name.C");
```

and then reuse this instance as shown above. Alternatively, add complete packages to the resolver configuration as follows:

```
   XMLClassDescriptorResolver classDescriptorResolver = (XMlClassDescriptorResolver)
      ClassDescriptorResolverFactory.createClassDescriptorResolver(BindingType.XML);
   classDescriptorResolver.setClassLoader(...);
   classDescriptorResolver.addPackage("your.package.name");
```

The `org.exolab.castor.xml.XMLClassDescriptorResolver` interface provides various other methods to load class descriptors for individual classes and/or packages.

Table 1.17. blah

Method	**Description**	**Requires `.castor.cdr`**
`addClass(String)`	Loads the class descriptor for one class.	No
`addClass(String[])`	Loads the class descriptors for a collection of classes.	No
`addPackage(String)`	Loads the class descriptors for all classes in the package defined.	Yes
`addPackages(String[])`	Loads the class descriptors for all classes in the package defined.	Yes

Note

For some of the methods, pre-loading class descriptords will only work if you provide the `.castor.cdr` file with your generated classes (as generated by the XML code generator). If no such file is shipped, Castor will not be able to pre-load the descriptors, and will fall back to its default descriptor loading mechanism.

1.9. Castor XML - HOW-TO's

1.9.1. Introduction

This is a collection of HOW-TOs. The Castor project is actively seeking additional HOW-TO contributors to expand this collection. For information on how to do that, please see 'How to write a How-to'.

1.9.2. Documentation

- How to Author a How-To **(Author wanted!)**
- How to Author an FAQ **(Author wanted!)**
- How to Author a Code Snippet **(Author wanted!)**
- How to Author Core Documentation **(Author wanted!)**

1.9.3. Contribution

- How to submit an XML-specific bug report
- How to prepare a patch
- How to Contribute a Patch via Jira **(Author wanted!)**
- How to run Castor XML's test suite

1.9.4. Mapping

- How to use XMLContext for un-/marshalling
- How to map a collection of elements
- How to map a map/hashtable of elements
- How to map a list of elements at the root
- How to map constructor arguments
- How to map an inner class
- How to Unmarshal raw XML segments into arbitrary types
- How to use references in XML and Castor
- How to wrap a collection with a wrapper element
- How to prevent a collection from being exposed
- How to write a configurable field handler
- How to map text content
- How to work with wrapper elements around collections
- How to work marshal XML documents with version 1.1

1.9.5. Validation

- How to use XML validation

1.9.6. Source generation

- How to use a binding file with source generation

1.9.7. Others

- How to implement a custom serializer
- How to fetch DTDs and XML Schemas from JAR files
- How to marshal Hibernate proxies

1.10. XML FAQ

This section provides answers to frequently answered questions, i.e. questions that have been asked repeatedly on one of the mailing lists. Please check with these F.A.Q.s frequently, as addressing questions that have been answered in the past already again and again places an unnecessary burden on the committers/contributors.

This section is structured along the lines of the following areas ...

- Section 1.10.1, “General”
- Section 1.10.2, “Introspection”
- Section 1.10.3, “Mapping”
- Section 1.10.4, “Marshalling”
- Section 1.10.5, “Source code generation”
- Section 1.10.6, “Miscellaneous”
- Section 1.10.7, “Serialization”

1.10.1. General

1.10.1.1. How do I set the encoding?

Create a new instance of the `Marshaller` class and use the `setEncoding` method. You'll also need to make sure the encoding for the Writer is set properly as well:

```
...
String encoding = "ISO-8859-1";
FileOutputStream fos = new FileOutputStream("result.xml");
OutputStreamWriter osw = new OuputStreamWriter(fos, encoding);
Marshaller marshaller = new Marshaller(osw);
marshaller.setEncoding(encoding);
...
```

1.10.1.2. I'm getting an error about 'xml' prefix already declared?

> **Note**
>
> For Castor 0.9.5.2 only

The issue occurs with newer versions of Xerces than the version 1.4 that ships with Castor. The older version works OK. For some reason, when the newer version of Xerces encounters an "xml" prefixed attribute, such as "xml:lang", it tries to automatically start a prefix mapping for "xml". Which, in my opinion, is technically incorrect. They shouldn't be doing that. According to the w3c, the "xml" prefix should never be declared.

The reason it started appearing in the new Castor (0.9.5.2), is because of a switch to SAX 2 by default during unmarshaling.

Solution: A built in work-around has been checked into the Castor SVN and will automatically exist in any post 0.9.5.2 releases. For those who are using 0.9.5.2 and can't upgrade, I found a simple workaround (tested with Xerces 2.5). At first I thought about disabling namespace processing in Xerces, but then realized that it's already disabled by default by Castor ... so I have no idea why they call #startPrefixMapping when namespace processing has been disabled. But in any event... explicitly enabling namespace processing seems to fix the problem:

in the `castor.properties` file, change the following line:

```
org.exolab.castor.parser.namespaces=false
```

to:

```
org.exolab.castor.parser.namespaces=true
```

Note

This work-around has only been tested with Xerces 2.5 and above.

1.10.1.3. Why is my 'get' method called twice?

The get method will be called a second time during the validation process. To prevent this from happening, simply disable validation on the Marshaller or Unmarshaller.

1.10.1.4. How can I speed up marshalling/unmarshalling performance?

- Cache the descriptors!

```
import org.exolab.castor.xml.ClassDescriptorResolver;
import org.exolab.castor.xml.Unmarshaller;
import org.exolab.castor.xml.util.ClassDescriptorResolverImpl;
...
ClassDescriptorResolver cdr = new ClassDescriptorResovlerImpl();
...
Unmarshaller unm = new Unmarshaller(...);
unm.setResolver(cdr);
```

  By reusing the same `ClassDescriptorResolver` any time you create an Unmarshaller instance, you will be reusing the existing class descriptors previously loaded.

- Disable validation

```
unm.setValidation(false);
```

- Reuse objects

  To cut down on object creation, you can reuse an existing object model, but be careful because this is an experimental feature. Create an Unmarshaller with your existing root object and set object reuse to true...

```
Unmarshaller unm = new
Unmarshaller(myObjectRoot);
```

```
```

```
unm.setReuseObjects(true);
```

- If you have enabled pretty-printing (indenting), then disable it. The Xerces Serializer is much slower with indenting enabled.
- Try changing parsers to something other than Xerces.

There are probably other approaches you can use as well, but those seem to be the most popular ones. Let us know if you have a solution that you think we should add here.

1.10.1.5. How do I ignore elements during unmarshalling?

- Use the `Unmarshaller#setIgnoreExtraElements()` method:

  ```
  Unmarshaller unm = new Unmarshaller(...);
  unm.setIgnoreExtraElements(true);
  ```

  If any elements appear in the XML instance that Castor cannot find mappings for, they will be skipped.

- You can also set the `org.exolab.castor.xml.strictelements` property in the `castor.properties` file:

  ```
  org.exolab.castor.xml.strictelements=true
  ```

1.10.1.6. Where does Castor search for the castor.properties file?

Castor loads the castor.properties in the following order:

- From classpath (usually from the jar file)
- From {java.home}/lib (if present)
- From the local working directory

Each properties file overrides the previous. So you don't have to come up with a properties file with all the properties and values, just the ones you want to change. This also means you don't have to touch the properties file found in the jar file.

> **Note**
>
> Note: You can also use `LocalConfiguration.getInstance().getProperties()` to change the properties values programatically.

1.10.1.7. Can I programmatically change the properties found in the castor.properties file?

Yes, many of these properties can be set directly on the Marshaller or Unmarshaller, but you can also use `LocalConfiguration.getInstance().getProperties()` to change the properties values programatically.

1.10.2. Introspection

1.10.2.1. Can private methods be introspected?

Castor does not currently support introspection of private methods. Please make sure proper public accesssor methods are available for all fields that you wish to be handled by the Marshalling Framework.

1.10.3. Mapping

1.10.3.1. My mapping file seems to have no effect!

Make sure you are not using one of the *static* methods on the Marshaller/Unmarshaller. Any configuration changes that you make to the Marshaller or Unmarshaller are not available from the static methods.

1.10.3.2. Are there any tools to automatically create a mapping file?

Yes! We provide one such tool, see `org.exolab.castor.tools.MappingTool` . There are some 3rd party tools as well.

1.10.3.3. How do I specify a namespace in the mapping file?

For a specific field you can use a QName for the value of the bind-xml name attribute as such:

```
<bind-xml name="foo:bar" xmlns:foo="http://www.acme.com/foo"/>
```

Note: The namespace prefix is only used for qualification during the loading of the mapping, it is not used during Marshaling. To map namespace prefixes during marshaling you currently need to set these via the Marshaler directly.

For a class mapping, use the <map-to> element. For more information see the XML Mapping documentation .

1.10.3.4. How do I prevent a field from being marshaled?

Set the **transient** attribute on the <bind-xml> element to true:

```
<bind-xml transient="true"/>
```

Note: You can also set transient="true" on the <field> element.

1.10.4. Marshalling

1.10.4.1. The XML is marshalled on one line, how do I force line-breaks?

For all versions of Castor:

To enable pretty-printing (indenting, line-breaks) just modify the ***castor.properties*** file and uncomment the following:

```
# True if all documents should be indented on output by default
#
#org.exolab.castor.indent=true
```

Note: This will slow down the marshalling process

1.10.4.2. What is the order of the marshalled XML elements?

If you are using Castor's default introspection to automatically map the objects into XML, then there is no guarantee on the order. It simply depends on the order in which the fields are returned to Castor using the Java reflection API.

Note: If you use a mapping file Castor will generate the XML in the order in which the mapping file is specified.

1.10.5. Source code generation

1.10.5.1. Can I use a DTD with the source generator?

Not directly, however you can convert your DTD to an XML Schema fairly easily. We provide a tool (`org.exolab.castor.xml.dtd.Converter`) to do this. You can also use any number of 3rd-party tools such as XML Spy or XML Authority.

1.10.5.2. My XML output looks incorrect, what could be wrong?

Also: I used the source code generator, but all my xml element names are getting marshaled as lowercase with hyphens, what's up with that?

Solution: Are the generated class descriptors compiled? Make sure they get compiled along with the source code for the object model.

1.10.5.3. The generated source code has incorrect or missing imports for imported schema types

Example: Castor generates the following:

```
import types.Foo;
```

instead of:

```
import com.acme.types.Foo;
```

This usually happens when the namespaces for the imported schemas have not been mapped to appropriate java packages in the *castorbuilder.properties* file.

Solution:

- Make sure the `castorbuilder.properties` is in your classpath when you run the SourceGenerator.

- Uncomment and edit the `org.exolab.castor.builder.nspackages` property. Make sure to copy the value of the imported namespace exactly as it's referred to in the schema (i.e. trailing slashes and case-sensitivity matter!).

For those using 0.9.5.1, you'll need to upgrade due to a bug that is fixed in later releases.

1.10.5.4. How can I make the generated source code more JDO friendly?

For Castor 0.9.4 and above:

Castor JDO requires a reference to the actual collection to be returned from the get-method. By default the source generator does not provide such a method. To enable such methods to be created, simple add the following line to your `castorbuilder.properties` file:

```
org.exolab.castor.builder.extraCollectionMethods=true
```

Note: The default `castorbuilder.properties` file has this line commented out. Simply uncomment it.

Your mapping file will also need to be updated to include the proper set/get method names.

1.10.6. Miscellaneous

1.10.6.1. Is there a way to automatically create an XML Schema from an XML instance?

Yes! We provide such a tool. Please see `org.exolab.castor.xml.schema.util.XMLInstance2Schema` . It's not 100% perfect, but it does a reasonable job.

1.10.6.2. How to enable XML validation with Castor XML

To enable XML validation at the parser level, please add properties to your `castor.properties` file as follows:

```
org.exolab.castor.parser.namespaces=true
org.exolab.castor.sax.features=http://xml.org/sax/features/validation,\
http://apache.org/xml/features/validation/schema,\
http://apache.org/xml/features/validation/schema-full-checking
```

Please note that the example given relies on the use of Apache Xerces, hence the `apache.org` properties; similar options should exist for other parsers.

1.10.6.3. Why is mapping ignored when using a FieldHandlerFactory

When using a custom FieldHandlerFactory as in the following example

```
Mapping mapping = ... ;
FieldHandlerFactoyt factory = ...;
Marshaller m = new Marshaller(writer);
ClassDescriptorResolverImpl cdr = new ClassDescriptorResolverImpl();
cdr.getIntrospector().addFieldHandlerFactory(factory);
m.setResolver(cdr);
marshaller.setMapping(mapping);
```

please make sure that you set the mapping file **after** you set the ClassDescriptorResolver. You will note the following in the Javadoc for `org.exolab.castor.xml.Marshaller.html#setResolver(org.exolab.castor.xml.ClassDescriptorResolver)` :

Note

Note: This method will nullify any Mapping currently being used by this Marshaller

1.10.7. Serialization

1.10.7.1. Is it true that the use of Castor XML mandates Apache Xerces as XML parser?

Yes and no. It actually depends. When requiring *pretty printing* during marshalling, Castor internally relies on Apache's Xerces to implement this feature. As such, when not using this feature, Xerces is not a requirement, and any JAXP-compliant XML parser can be used (for unmarshalling).

In other words, with the latter use case, you do **not** have to download (and use) Xerces separetely.

1.10.7.2. Do I still have to download Xerces when using Castor XML with Java 5.0?

No. Starting with release 1.1, we have added support for using the Xerces instance as shipped with the JRE/JDK for serialization. As such, for Java 5.0 users, this removes the requirement to download Xerces separately when wanting to use 'pretty printing' with Castor XML during marshalling.

To enable this feature, please change the following properties in your **local** `castor.properties` file (thus redefining the default value) as shown below:

```
# Defines the XML parser to be used by Castor.
# The parser must implement org.xml.sax.Parser.
org.exolab.castor.parser=org.xml.sax.helpers.XMLReaderAdapter

# Defines the (default) XML serializer factory to use by Castor, which must
# implement org.exolab.castor.xml.SerializerFactory; default is
# org.exolab.castor.xml.XercesXMLSerializerFactory
org.exolab.castor.xml.serializer.factory=org.exolab.castor.xml.XercesJDK5XMLSerializerFactory

# Defines the default XML parser to be used by Castor.
org.exolab.castor.parser=com.sun.org.apache.xerces.internal.parsers.SAXParser
```

Chapter 2. XML code generation

2.1. Why Castor XML code generator - Motivation

tbd

2.2. Introduction

2.2.1. News

2.2.1.1. Source generation & Java field naming conventions

Starting with **release 1.3.3**, the Castor source generator supports a new naming scheme for Java field names, which will be enabled by default. As such, Java field names as generated will follow more closely the standard Java property naming conventions. Should there be a need to keep using the old naming schema, please amend the following property in your custom `castorbuilder.properties` file:

```
#
# Property specifying whether for Java field names the old naming conventions
# should be used.
#
# Possible values:
# - true
# - false (default)
#
# <pre>
# org.exolab.castor.builder.field-naming.old = false
# </pre>
#
org.exolab.castor.builder.field-naming.old=true
```

2.2.1.2. Source generation & Java 5.0

1. Since **release 1.0.2**, the Castor source generator supports the optional the generation of Java 5.0 compliant code.
2. With **release 1.3**, the XML code generator will generate Java 5.0 compliant code by default.

With support for Java 5.0 enabled, the generated code will support the following Java 5.0-specific artifacts:

- Use of parameterized collections, e.g. `ArrayList<String>`.
- Use of `@Override` annotations with the generated methods that require it.
- Use of `@SuppressWarnings` with "unused" method parameters on the generated methods that needed it.
- Added "enum" to the list of reserved keywords.

To disable this feature (on by default), please amend the following property in your custom `castorbuilder.properties` file:

```
# Specifies whether the sources generated should be source compatible with
# Java 1.4 or Java 5.0. Legal values are "1.4" and "5.0".  When "5.0" is
# selected, generated source will use Java 5 features such as generics and
# annotations.
# Defaults to "5.0".
#
org.exolab.castor.builder.javaVersion=5.0
```

2.2.2. Introduction

Castor's Source Code Generator creates a set of Java classes which represent an object model for an XML Schema (W3C XML Schema 1.0 Second Edition, Recommendation), as well as the necessary Class Descriptors used by the marshaling framework to obtain information about the generated classes.

> **Note**
>
> The generated source files will need to be compiled. A later release may add an Ant taskdef to handle this automatically.

2.2.3. Invoking the XML code generator

The XML code generator can be invoked in many ways, including by command line, via an Ant task and via Maven. Please follow the below links for detailed instructions on each invocation mode.

- Section 2.5.3, “Command line”
- Section 2.5.1, “Ant task definition”
- Maven plugin for Castor XML

2.2.4. XML Schema

The input file for the source code generator is an XML schema[1]footnote>. The currently supported version is the **W3C XML Schema 1.0, Second Edition** [2]. For more information about XML schema support, check Section 2.6, “XML schema support”.

2.3. Properties

2.3.1. Overview

Please find below a list of properties that can be configured through the builder configuration properties, as defined in either the default or a custom XML code generator configuration file. These properties allow you to control various advanced options of the XML source generator.

[1]XML Schema is a W3C Recommendation

[2]Castor supports the XML Schema 1.0 Second Edition

Table 2.1. <column> - Definitions

Option	Description	Values	Default	Since version
org.exolab.castor.builder.javaVersion	Compliance with Java version	`1.4/5.0`	1.4	**1.0.2**
org.exolab.castor.builder.forceJava4Enums	Forces the code generator to create 'old' Java 1.4 enumeration classes even in Java 5 mode.	`true/false`	false	**1.1.3**
org.exolab.castor.builder.boundproperties	Generation of bound properties	`true/false`	`false`	0.8.9
org.exolab.castor.builder.javaclassmapping	Class generation mode	`element/type`	`element`	0.9.1
org.exolab.castor.builder.superclass	Global super class (for all classes generated)	Any valid class name	-	0.8.9
org.exolab.castor.builder.nspackages	XML namespace to package name mapping	A series of mappings	-	0.8.9
org.exolab.castor.builder.equalsmethod	Generation of `equals/hashCode()` method	`true/false`	`false`	0.9.1
org.exolab.castor.builder.useCycleBreaker	Use of cycle breaker code in generated `equals/hashCode()` method	`true/false`	`true`	1.3.2
org.exolab.castor.builder.primitivetowrapper	Generation of Object wrappers instead of primitives	`true/false`	`false`	0.9.4
exolab.castor.builder.automaticConflictResolution	Specifies whether **automatic class name conflict resolution** should be used or not	`true/false`	`false`	**1.1.1**
g.exolab.castor.builder.extraCollectionMethods	Specifies whether **extra** (additional) methods should be created for collection-style fields. Set this to true if you want your code to be	`true/false`	`false`	0.9.1

Option	Description	Values	Default	Since version
	more compatible with Castor JDO or other persistence frameworks.			
org.exolab.castor.builder.jclassPrinterFactories	Enlists the available modes for (J)Class *printing* during XML code generation.	`org.exolab.castor.builder.printing.WriterJClassPrinterFactory/ org.exolab.castor.builder.printing.TemplateJClassPrinterFactory`	n/a	**1.2.1**
org.exolab.castor.builder.extraDocumentationMethods	specifying whether extra members/methods for extracting XML schema documentation should be made available.	`true/false`	`false`	**1.2**

2.3.2. Customization - Lookup mechanism

By default, the Castor XML code generator will look for such a property file in the following places:

1. If no custom property file is specified, the Castor XML code generator will use the default builder configuration properties at `org/exolab/castor/builder/castorbuilder.properties` as shipped as part of the XML code generator JAR.
2. If a file named `castorbuilder.properties` is available on the CLASSPATH, the Castor XML code generator will use each of the defined property values to override the default value as defined in the default builder configuration properties. This file is commonly referred to as a **custom** builder configuration file.

2.3.3. Detailed descriptions

2.3.3.1. Source generation & Java 5.0

As of **Castor 1.0.2**, the Castor source generator now supports the generation of Java 5.0 compliant code. The generated code - with the new feature enabled - will make use of the following Java 5.0-specific artifacts:

- Use of parameterized collections, e.g. ArrayList<String>.
- Use of @Override annotations with the generated methods that require it.
- Use of @SupressWarnings with "unused" method parameters on the generated methods that needed it.
- Added "enum" to the list of reserved keywords.

To enable this feature (off by default), please uncomment the following property in your custom `castorbuilder.properties` file:

```
# This property specifies whether the sources generated
# should comply with java 1.4 or 5.0; defaults to 1.4
org.exolab.castor.builder.javaVersion=5.0
```

2.3.3.2. SimpleType Enumerations

In previous versions, castor only supported (un)marshalling of "simple" java5 enums, meaning enums where all facet values are valid java identifiers. In these cases, every enum constant name can be mapped directly to the xml value. See the following example:

```
<xs:simpleType name="AlphabeticalType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="A"/>
    <xs:enumeration value="B"/>
    <xs:enumeration value="C"/>
  </xs:restriction>
</xs:simpleType>
```

```
public enum AlphabeticalType {
    A, B, C
}
```

```
<root>
  <AlphabeticalType>A</AlphabeticalType>
</root>
```

So if there is at least ONE facet that cannot be mapped directly to a valid java identifier, we need to extend the enum pattern. Examples for these cases are value="5" or value="-s". Castor now introduces an extended pattern, similar to the jaxb2 enum handling. The actual value of the enumeration facet is stored in a private String property, the name of the enum constant is translated into a valid identifier. Additionally, some convenience methods are introduced, details about these methods are described after the following example:

```
<xs:simpleType name="CompositeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="5"/>
    <xs:enumeration value="10"/>
  </xs:restriction>
</xs:simpleType>
```

```
public enum CompositeType {
    VALUE_5("5"),
    VALUE_10("10");

    private final java.lang.String value;

    private CompositeType(final java.lang.String value) {
        this.value = value;
    }

    public static CompositeType fromValue(final java.lang.String value) {
        for (CompositeType c: CompositeType.values()) {
            if (c.value.equals(value)) {
                return c;
            }
        }
        throw new IllegalArgumentException(value);
```

```
    }

    public java.lang.String value() {
        return this.value;
    }

    public java.lang.String toString() {
        return this.value;
    }
}
```

```
<root>
  <CompositeType>5</CompositeType>
</root>
```

2.3.3.2.1. Unmarshalling of complex enums

Castor uses the static void `fromValue(String value)` method to retrieve the correct instance from the value in the XML input file. In our example, the input is "5", fromValue returns `CompositeType.VALUE_5`.

2.3.3.2.2. Marshalling of complex enums

Currently, we have to distinguish between enums with a class descriptor and the ones without. If you are using class descriptors, the EnumerationHandler uses the value() method to write the xml output.

If no descriptor classes are available, castor uses per default the `toString()` method to marshall the value. In this case, the override of the `java.lang.Enum.toString()` method is mandatory, because `java.lang.Enum.toString()` returns the NAME of the facet instead of the VALUE. So in our example, `VALUE_10` would be returned instead of "10". To avoid this, castor expects an implementation of `toString()` that returns `this.value`.

2.3.3.2.3. Source Generation of complex enums

If the java version is set to "5.0", the new default behavior of castor is to generate complex java5 enums for simpleType enumerations, as described above. In java 1.4 mode, nothing has changed and the old style enumeration classes using a HashMap are created.

Users, who are in java5 mode and still want to use the old style java 1.4 classes, can force this by setting the new `org.exolab.castor.builder.forceJava4Enums` property to true as follows:

```
 # Forces the code generator to create 'old' Java 1.4 enumeration classes instead
 # of Java 5 enums for xs:simpleType enumerations, even in Java 5 mode.
 #
 # Possible values:
 # - false (default)
 # - true
org.exolab.castor.builder.forceJava4Enums=false
```

2.3.3.3. Bound Properties

Bound properties are "properties" of a class, which when updated the class will send out a `java.beans.PropertyChangeEvent` to all registered `java.beans.PropertyChangeListeners`.

To enable bound properties, please add a property definition to your custom builder configuration file as follows:

```
# To enable bound properties uncomment the following line. Please
```

```
# note that currently *all* fields will be treated as bound properties
# when enabled. This will change in the future when we introduce
# fine grained control over each class and it's properties.
#
org.exolab.castor.builder.boundproperties=true
```

When enabled, **all** properties will be treated as bound properties. For each class that is generated a `setPropertyChangeListener` method is created as follows:

```
/**
 * Registers a PropertyChangeListener with this class.
 * @param pcl The PropertyChangeListener to register.
 **/

public void addPropertyChangeListener (java.beans.PropertyChangeListener pcl)
{
   propertyChangeListeners.addElement(pcl);
} //-- void addPropertyChangeListener
```

Whenever a property of the class is changed, a `java.beans.PropertyChangeEvent` will be sent to all registered listeners. The property name, the old value and the new value will be set in the **java.beans.PropertyChangeEvent**.

Note

To prevent unnecessary overhead, if the property is a collection, the old value will be *null*.

2.3.3.4. Class Creation/Mapping

The source generator can treat the XML Schema structures such as `<complexType>` and `<element>` in two main ways. The first, and current default method is called the "element" method. The other is called the "type" method.

Table 2.2. <column> - Definitions

Method	Explanation
'element'	The "element" method creates classes for all elements whose type is a <complexType>. Abstract classes are created for all top-level <complexType>s. Any elements whose type is a top-level type will have a new class create that extends the abstract class which was generated for that top-level complexType. Classes are not created for elements whose type is a <simpleType>.
'type'	The "type" method creates classes for all top-level <complexType>s, or elements that contain an "anonymous" (in-lined) <complexType>. Classes will not be generated for elements whose type is a top-level type.

To change the "method" of class creation, please add the following property definition to your custom builder configuration file:

```
# Java class mapping of <xsd:element>'s and <xsd:complexType>'s
#
org.exolab.castor.builder.javaclassmapping=type
```

Please note that setting this property will not affect class creation when the `defaultBindingType` is explicitely used in a binding file. In that case, the value set there will take precedence.

2.3.3.5. Setting a super class

The source generator enables the user to set a super class to **all** the generated classes (of course, class descriptors are not affected by this option). Please note that, though the binding file, it is possible to define a super class for individual classes

To set the global super class, please add the following property definition to your custom builder configuration file:

```
# This property allows one to specify the super class of *all*
# generated classes
#
org.exolab.castor.builder.superclass=com.xyz.BaseObject
```

2.3.3.6. Mapping XML namespaces to Java packages

An XML Schema instance is identified by a namespace. For data-binding purposes, especially code generation it may be necessary to map namespaces to Java packages.

This is needed for imported schema in order for Castor to generate the correct imports during code generation for the primary schema.

To allow the mapping between namespaces and Java packages , edit the castorbuilder.properties file :

```
# XML namespace mapping to Java packages
#
#org.exolab.castor.builder.nspackages=\
   http://www.xyz.com/schemas/project=com.xyz.schemas.project,\
   http://www.xyz.com/schemas/person=com.xyz.schemas.person
```

2.3.3.7. Generate equals()/hashCode() method

Since version: 0.9.1

The Source Generator can override the `equals()` and `hashCode()` method for the generated objects.

To have `equals()` and `hashCode()` methods generated, override the following property in your custom castorbuilder.properties file:

```
# Set to true if you want to have an equals() and
# hashCode() method generated for each generated class;
# false by default
org.exolab.castor.builder.equalsmethod=true
```

2.3.3.8. Use CycleBreaker for generation of equals()/hashcode() methods.

Since version: 1.3.2

Specifies whether cycle breaker code should be added to generated methods `equals()` and `hashcode()`.

```
# Property specifying whether cycle breaker code should be added
# to generated methods 'equals' and 'hashcode'.
#
# Possible values:
# - true (default)
# - false
#
# <pre>
# org.exolab.castor.builder.useCycleBreaker
# </pre>
org.exolab.castor.builder.useCycleBreaker=true
```

2.3.3.9. Maps java primitive types to wrapper object

Since version 0.9.4

It may be convenient to use java objects instead of primitives, the Source Generator provides a way to do it. Thus the following mapping can be used:

- boolean to java.lang.Boolean
- byte to java.lang.Byte
- double to java.lang.Double
- float to java.lang.Float
- int and integer to java.lang.Integer
- long to java.lang.Long
- short to java.lang.Short

To enable this property, edit the castor builder.properties file:

```
# Set to true if you want to use Object Wrappers instead
# of primitives (e.g Float instead of float).
# false by default.
#org.exolab.castor.builder.primitivetowrapper=false
```

2.3.3.10. Automatic class name conflict resolution

Since version 1.1.1

With this property enabled, the XML code generator will use a new automatic class name resolution mode that has special logic implemented to automatically resolve class name conflicts.

This new mode deals with various class name conflicts where previously a binding file had to be used to resolve these conflicts manually.

To enable this feature (turned off by default), please add the following property definitio to your custom `castorbuilder.properties` file:

```
# Specifies whether automatic class name conflict resolution
# should be used or not; defaults to false.
#
org.exolab.castor.builder.automaticConflictResolution=true
```

2.3.3.11. Extra collection methods

Specifies whether **extra** (additional) methods should be created for collection-style fields. Set this to `true` if you want your code to be more compatible with Castor JDO (or other persistence frameworks in general).

By setting this property to `true`, additional getter/setter methods for the field in question, such as get/set by reference and set as copy methods, will be added. In order to have these additional methods generated, please override the following code generator property in a custom `castorbuilder.properties` as shown:

```
# Enables generation of extra methods for collection fields, such as get/set by
# reference and set as copy.  Extra methods are in addition to the usual
# collection get/set methods.  Set this to true if you want your code to be
# more compatible  with Castor JDO.
#
# Possible values:
# - false (default)
# - true
org.exolab.castor.builder.extraCollectionMethods=true
```

2.3.3.12. Class printing

As of release 1.2, Castor supports the use of Velocity-based code templates for code generation. For the time being, Castor will support two modes for code generation, i.e. the new Velocity-based and an old legacy mode. **Default** will be the *legacy* mode; this will be changed with a later release of Castor.

In order to use the new Velocity-based code generation, please call the method `setJClassPrinterType(String)` on `org.exolab.castor.builder.SourceGenerator` with a value of `velocity`.

As we consider the code stable enough for a major release, we do encourage users to use the new Velocity-based mode and to provide us with (valuable) feedback.

Please note that we have changed the mechanics of changing the JClass printing type between releases 1.2 and 1.2.1.

2.3.3.13. Extra documentation methods

As of release 1.2, the Castor XML code generator - if configured as shown below - now supports generation of additional methods to allow programmatic access to <xs:documentation> elements for top-level type/element definitions as follows:

```
public java.lang.String getXmlSchemaDocumentation(final java.lang.String source);
public java.util.Map getXmlSchemaDocumentations();
```

In order to have these additional methods generated as shown above, please override the following code generator property in a custom `castorbuilder.properties` as shown:

```
# Property specifying whether extra members/methods for extracting XML schema
# documentation should be made available; defaults to false
org.exolab.castor.builder.extraDocumentationMethods=true
```

2.4. Custom bindings

This section defines the Castor XML binding file and describes - based upon the use of examples - how to use it.

The default binding used to generate the Java Object Model from an XML schema may not meet your expectations. For instance, the default binding doesn't deal with naming collisions that can appear because XML Schema allows an element declaration and a complexType definition to use the same name. The source generator will attempt to create two Java classes with the same qualified name. However, the latter class generated will simply overwrite the first one.

Another example of where the default source generator binding may not meet your expectations is when you want to change the default datatype binding provided by Castor or when you want to add validation rules by implementing your own validator and passing it to the Source Generator.

2.4.1. Binding File

The binding declaration is an XML-based language that allows the user to control and tweak details about source generation for the generated classes. The aim of this section is to provide an overview of the binding file and a definition of the several XML components used to define this binding file.

A more in-depth presentation will be available soon in the [Source Generator User Document (PDF).]()

2.4.1.1. <binding> element

```
<binding
    defaultBindingType = (element|type)>
    (include*,
     package*,
     namingXML?,
     elementBinding*,
     attributeBinding,
     complexTypeBinding,
     groupBinding)
</binding>
```

The binding element is the root element and contains the binding information.

Table 2.3. <column> - Definitions

Name	Description	Default	Required ?
defaultBindingType	Controls the class creation mode for details on the available modes. Please note that the mode specified in this attribute	`element`	No

Name	Description	Default	Required ?
	will override the binding type specified in the `castorbuilder.properties` file.		

2.4.1.2. <include> element

```
<include
    URI = xsd:anyURI/>
```

This element allows you to include a binding declaration defined in another file. This allows reuse of binding files defined for various XML schemas.

Attributes of <include>

URI:
The URI of the binding file to include.

2.4.1.3. <package> element

```
<package>
    name = xsd:string
    (namespace|schemaLocation) = xsd:string>
</package>
```

Table 2.4. <package> - Definitions

Name	Description
name	A fully qualified java package name.
namespace	An XML namespace that will be mapped to the package name defined by the *name* element.
schemaLocation	A URL that locates the schema to be mapped to the package name defined by the *name* element.

The `targetNamespace` attribute of an XML schema identifies the namespace in which the XML schema elements are defined. This language namespace is defined in the generated Java source as a package declaration. The `<package/>` element allows you to define the mapping between an XML namespace and a Java package.

Moreover, XML schema allows you to factor the definition of an XML schema identified by a unique namespace by including several XML schemas instances to build one XML schema using the `<xsd:include/>` element. Please make sure you understand the difference between `<xsd:include/>` and `<xsd:import/>`. `<xsd:include/>` # relies on the URI of the included XML schema. This element allows you to keep the structure hierarchy defined in XML schema in a single generated Java package. Thus the binding file allows you to define the mapping between a `schemaLocation` attribute and a Java package.

2.4.1.4. <namingXML> element

```
<namingXML>
   (elementName,complexTypeName,modelGroupName)
</namingXML>

<elementName|complexTypeName|modelGroupName>
    (prefix?, suffix?) = xsd:string
</elementName|complexTypeName|modelGroupName>
```

Table 2.5. <namingXML> - Definitions

Name	Description
prefix	The prefix to add to the names of the generated classes.
suffix	The suffix to append to the the names of the generated classes.

One of the aims of the binding file is to avoid naming collisions. Indeed, XML schema allows <element>s and <complexType>s to share the same name, resulting in name collisions when generating sources. Defining a binding for each element and complexType that share the same name is not always a convenient solution (for instance the BPML XML schema and the UDDI v2.0 XML schema use the same names for top-level complexTypes and top-level elements).

The main aim of the `<namingXML/>` element is to define default prefices and suffices for the names of the classes generated for an <element>, a <complexType> or a model group definition.

Note

It is not possible to control the names of the classes generated to represent nested model groups (all, choice, and sequence).

2.4.1.5. <componentBinding> element

```
<elementBinding|attributeBinding|complexTypeBinding|groupBinding
    name = xsd:string>
   ((java-class|interface|member|contentMember),
     elementBinding*,
     attributeBinding*,
     complexTypeBinding*,
     groupBinding*)
</elementBinding|attributeBinding|complexTypeBinding|groupBinding>
```

Table 2.6. <componentBinding> - Definitions

Name	Description
name	The name of the XML schema component for which we are defining a binding.

These elements are the tenets of the binding file since they contain the binding definition for an XML schema element, attribute, complex type and model group definition. The first child element (`<java-class/>`, `<interface>`, `<member>` or `<contentMember/>`) will determine the type of binding one is defining. Please note that defining a `<java-class>` binding on an XML schema attribute will have absolutely no effect.

The binding file is written from an XML schema point of view; there are two distinct ways to define the XML schema component for which we are defining a binding.

1. (XPath-style) name
2. Embedded definitions

2.4.1.5.1. Name

First we can define it through the `name` attribute.

The value of the name attribute uniquely identifies the XML schema component. It can refer to the top-level component using the NCName of that component or it can use a location language based on [XPath](). The grammar of that language can be defined by the following [BNF]():

```
[1]Path         ::= '/'LocationPath('/'LocationPath)*
[2]LocationPath ::= (Complex|ModelGroup|Attribute|Element|Enumeration)
[3]Complex      ::= 'complexType:'(NCName)
[4]ModelGroup   ::= 'group:'NCName
[5]Attribute    ::= '@'NCName
[6]Element      ::= NCName
[7]Enumeration  ::= 'enumType':(NCName)
```

Please note that all values for the `name` attribute have to start with a `'/'`.

2.4.1.5.2. Embedded definitions

The second option to identify an XML schema component is to embed its binding definition inside its parent binding definition.

Considering below XML schema fragment ...

```
<complexType name="fooType">
    <sequence>
        <element name="foo" type="string" />
    </sequence>
</complexType>
```

the following binding definitions are equivalent and identify the <element> `foo` defined in the top-level <complexType> `fooType`.

```
<elementBinding name="/complexType:fooType/foo>
   <member name="MyFoo" handler="mypackage.myHandler"/>
</elementBinding>

<complexTypeBinding name="/fooType">
   <elementBinding name="/foo>
      <member name="MyFoo" handler="mypackage.myHandler"/>
   </elementBinding>
<complexTypeBinding>
```

2.4.1.6. <java-class>

```
<java-class
    name? = xsd:string
    package? = xsd:string
    final? = xsd:boolean
    abstract? = xsd:boolean
    equals? = xsd:boolean
    bound? = xsd:boolean
    (implements*,extends?)
</java-class>
```

This element defines all the options for the class to be generated, including common properties such as class name, package name, and so on.

Attributes of <java-class>

name:
The name of the class that will be generated.

package:
The package of the class to be generated. if set, this option overrides the mapping defined in the `<package/>` element.

final:
If true, the generated class will be final.

abstract:
If true, the generated class will be abstract.

equals:
If true, the generated class will implement the `equals()` and `hashCode()` method.

bound:
If true, the generated class will implement bound properties, allowing property change notification.

For instance, the following binding definition instructs the source generator to generate a class `CustomTest` for a global element named 'test', replacing the default class name `Test` with `CustomTest`.

```
<elementBinding name="/test">
   <java-class name="CustomTest" final="true"/>
</elementBinding>
```

In addition to the properties listed above, it is possible to define that the class generated will extend a class given and/or implement one or more interfaces.

For instance, the following binding definition instructs the source generator to generate a class `TestWithInterface` that implements the interface `org.castor.sample.SomeInterface` in addition to `java.io.Serializable`.

```
<elementBinding name="/test">
   <java-class name="TestWithInterface">
      <implements>org.castor.sample.SomeInterface</implements>
   </java-class>
</elementBinding>
```

The subsequent binding definition instructs the source generator to generate a class `TestWithExtendsAndInterface` that implements the interface `org.castor.sample.SomeInterface` in addition to `java.io.Serializable`, and extends from a (probably abstract) base class `SomeAbstractBaseClass`.

```
<elementBinding name="/test">
   <java-class name="TestWithExtendsAndInterface">
      <extends>org.castor.sample.SomeAbstractBaseClass</extends>
      <implements>org.castor.sample.SomeInterface</implements>
   </java-class>
</elementBinding>
```

The generated class `SomeAbstractBaseClass` will have a class signature as shown below:

```
...

public class TestWithExtendsAndInterface
   extends SomeAbstractBaseClass
   implements SomeInterface, java.io.Serializable {
   ...
```

2.4.1.7. <member> element

```
 <member
  name? = xsd:string
  java-type? = xsd:string
  wrapper? = xsd:boolean
  handler? = xsd:string
  visibility? = (public|protected|private)
  collection? = (array|vector|arraylist|hashtable|collection|odmg|set|map|sortedset)
  validator? = xsd:string/>
```

This element represents the binding for class member. It allows the definition of its name and java type as well as a custom implementation of FieldHandler to help the Marshaling framework in handling that member. Defining a validator is also possible. The names given for the validator and the fieldHandler must be fully qualified.

Table 2.7. <member> - Definitions

Name	Description
name	The name of the class member that will be generated.
java-type	Fully qualified name of the java type.
wrapper	If true, a wrapper object will be generated in case the Java type is a java primitive.
handler	Fully qualified name of the custom FieldHandler to use.
collection	If the schema component can occur more than once then this attribute allows specifying the collection to use to represent the component in Java.
validator	Fully qualified name of the FieldValidator to use.

Name	Description
visibility	A custom visibility of the content class member generated, with the default being `public`.

For instance, the following binding definition:

```
<elementBinding name="/root/members">
   <member collection="set"/>
</elementBinding>
```

instructs the source generator to generate -- within a class `Root` -- a Java member named `members` using the collection type `java.util.Set` instead of the default `java.util.List`:

```
         public class Root {

            private java.util.Set members;

            ...

         }
```

The following (slightly amended) binding element:

```
<elementBinding name="/root/members">
   <member name="memberSet" collection="set"/>
</elementBinding>
```

instructs the source generator to generate -- again within a class `Root` -- a Java member named `memberSet` (of the same collection type as in the previous example), overriding the name of the member as specified in the XML schema:

```
public class Root {

   private java.util.Set memberSet;

   ...

}
```

2.4.1.8. <contentMember> element

```
          <contentMember
           name? = xsd:string
           visiblity? = (public|protected|private)
```

This element represents the binding for *content* class member generated as a result of a mixed mode declaration of a complex type definition. It allows the definition of its name and its visibility

name:
The name of the class member that will be generated, overriding the default name of `_content`.

visibility:

A custom visibility of the content class member generated, with the default being `public`.

For a complex type definition declared to be *mixed* such as follows ...

```
<complexType name="RootType" mixed="true">
   <sequence>
      ...
   >/sequence>
>/complexType>
```

... the following binding definition ...

```
<elementBinding name="/complexType:RootType">
   <contentMember name="customContentMember"/>
</elementBinding>
```

instructs the source generator to generate -- within a class `RootType` -- a Java member named `customContentMember` of type `java.lang.String`:

```
public class RootType {

   private java.util.String customContentMember;

   ...
}
```

2.4.1.9. <enumBinding> element

```
<enumBinding>
   (enumDef)
</enumBinding>

<enumDef>
   (enumClassName = xsd:string, enumMember*)
</enumDef>

<enumMember>
   (name = xsd:string, value = xsd:string)
</enumMember>
```

The `<enumBinding>` element allows more control on the code generated for type-safe enumerations, which are used to represent an XML Schema `<simpleType>` enumeration.

For instance, given the following XML schema enumeration definition:

```
<xs:simpleType name="durationUnitType">
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Y' />
    <xs:enumeration value='M' />
    <xs:enumeration value='D' />
    <xs:enumeration value='h' />
    <xs:enumeration value='m' />
    <xs:enumeration value='s' />
  </xs:restriction>
</simpleType>
```

the Castor code generator would generate code where the default naming convention used during the generation would overwrite the first constant definition for value '`M`' with the one generated for value '`m`'.

The following binding definition defines -- through the means of an `<enumMember>` definition for the enumeration value 'M' -- a special binding for this value:

```
<enumBinding name="/enumType:durationUnitType">
  <enum-def>
    <enumMember>
      <value>M</value>
      <javaName>CUSTOM_M</javaName>
    </enumMember>
  </enum-def>
</enumBinding>
```

and instructs the source generator to generate -- within a class `DurationUnitType` -- a constant definition named `CUSTOM_M` for the enumeration value `M`.

2.4.1.10. Not implemented yet

2.4.1.10.1. <javadoc>

The `<javadoc>` element allows one to enter the necessary JavaDoc representing the generated classes or members.

2.4.1.10.2. <interface> element

```
<interface>
  name = xsd:string
</interface>
```

- **name:**The name of the interface to generate.

This element specifies the name of the interface to be generated for an XML schema component.

2.4.2. Class generation conflicts

As mentioned previously, you use a binding file for two main reasons:

- To customize the Java code generated
- To avoid class generation conflicts.

For the latter case, you'll (often) notice such collisions by looking at generated Java code that frequently does not compile. Whilst this is relatively easy for small(ish) XML schema(s), this task gets tedious for more elaborate XML schemas. To ease your life in the context of this 'collision detection', the Castor XML code generator provides you with a few advanced features. The following sections cover these features in detail.

2.4.2.1. Collision reporting

During code generation, the Castor XML code generator will run into situations where a class (about to be generated, and as such about to be written to the file system) will overwrite an already existing class. This, for example, is the case if within one XML schema there's two (local) element definitions within separate complex type definitions with the same name. In such a case, Castor will emit warning messages that inform the user that a class will be overwritten.

As of release 1.1, the Castor XML code generator supports two *reporting modes* that allow different levels of control in the event of such collisions, `warnViaConsoleDialog` and `informViaLog` mode.

Table 2.8. <column> - Definitions

Mode	Description	Since
`warnViaConsoleDialog`	Emits warning messages to `stdout` and ask the users whether to continue.	0.9
`informViaLog`	Emits warning messages only via the standard logger.	1.1

Please select the reporting mode of your choice according to your needs, the default being `warnViaConsoleDialog`. Please note that the `informViaLog` reporting mode should be the preferred choice when using the XML code generator in an automated environment.

In general, the warning messages produced are very useful in assisting you in your creation of the binding file, as shown in below example for the `warnViaConsoleDialog` mode:

```
Warning: A class name generation conflict has occurred between element
        '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
Warning: A class name generation conflict has occurred between element
        '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
Warning: A class name generation conflict has occurred between element
        '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
Warning: A class name generation conflict has occurred between element
        'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
Warning: A class name generation conflict has occurred between element
        'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
Warning: A class name generation conflict has occurred between element
        'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
        Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
```

2.4.2.1.1. Reporting mode 'warnViaConsoleDialog'

As already mentioned, this mode emits warning messages to `stdout`, and asks you whether you want to continue with the code generation or not. This allows for very fine grained control over the extent of the code generation.

Please note that there is several *setter* methods on the `org.exolab.castor.builder.SourceGenerator` that allow you to fine-tune various settings for this reporting mode. Genuinely, we believe that for automated code generation through either Ant or Maven, the new `informViaLog` is better suited for these needs.

2.4.2.2. Automatic collision resolution

As of Castor 1.1.1, support has been added to the Castor XML code generator for a (nearly) automatic conflict resolution. To enable this new mode, please override the following property in your custom property file as shown below:

```
# Specifies whether automatic class name conflict resolution
# should be used or not; defaults to false.
#
org.exolab.castor.builder.automaticConflictResolution=true
```

As a result of enabling automatic conflict resolution, Castor will try to resolve such name collisions automatically, using one of the following two strategies:

Table 2.9. <column> - Definitions

Name	Description	Since	Default
`xpath`	Prepends an XPATH fragment to make the suggested Java name unique.	1.1.1	Yes
`type`	Appends type information to the suggested Java name.	1.1.1	No

2.4.2.2.1. Selecting the strategy

For selecting one of the two strategies during XML code generation, please see the documentation for the following code artifacts:

- `setClassNameConflictResolver` on `org.exolab.castor.builder.SourceGenerator`
- `org.exolab.castor.builder.SourceGeneratorMain"`
- Ant task definition
- Maven plugin for Castor XML

In order to explain the *modus operandi* of these two modes, please assume two complex type definitions `AType` and `BType` in an XML schema, with both of them defining a local element named `c`.

```
<xs:complexType name="AType">
    <xs:sequence>
        <xs:element name="c" type="CType1" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="BType">
    <xs:sequence>
        <xs:element name="c" type="CType2" />
    </xs:sequence>
</xs:complexType>
```

Without automatic collision resolution enabled, Castor will create identically named classes `C.java` for both members, and one will overwrite the other. Please note the different types for the two `c` element definitions, which requires two class files to be generated in order not to lose this information.

2.4.2.2.2. 'XPATH' strategy

This strategy will prepend an XPATH fragment to the default Java name as derived during code generation, the default name (frequently) being the name of the XML schema artifact, e.g. the element name of the complex type name. The XPATH fragment being prepended is minimal in the sense that the resulting rooted XPATH is unique for the XML schema artifact being processed.

With automatic collision resolution enabled and the strategy 'XPATH' selected, Castor will create the following two classes, simply prepending the name of the complex type to the default element name:

- ATypeC.java
- BTypeC.java

2.4.2.2.3. 'TYPE' strategy

This strategy will append 'type' information to the default Java name as derived during code generation, the default name (frequently) being the name of the XML schema artifact, e.g. the element name of the complex type name.

With automatic collision resolution enabled and the strategy 'TYPE' selected, Castor will create the following two classes, simply appending the name of the complex type to the default element name (with a default '`By`' inserted):

- CByCType1.java
- CByCType2.java

To override the default '`By`' inserted between the default element name and the type information, please override the following property in your custom property file as shown below:

```
# Property specifying the 'string' used in type strategy to be inserted
# between the actual element name and the type name (during automatic class name
# conflict resolution); defaults to 'By'.
org.exolab.castor.builder.automaticConflictResolutionTypeSuffix=ByBy
```

2.4.2.2.4. Conflicts covered

The Castor XML code generator, with automatic collision resolution enabled, is capable of resolving the following collisions automatically:

- Name of local element definition same as name of a global element
- Name of local element definition same as name of another local element definition.

> **Note**
>
> Please note that *collision resolution* for a local to local collision will only take place for the second local element definition encountered (and subsequent ones).

2.5. Invoking the XML code generator

2.5.1. Ant task

An alternative to using the command line as shown in the previous section, the Castor Source Generator Ant Task can be used to call the source generator for class generation. The only requirement is that the castor-<version>-codegen-antask.jar must additionally be on the CLASSPATH.

2.5.1.1. Specifying the source for generation

As shown in the subsequent table, there's multiple ways of specifying the input for the Castor code generator. **At least one** input source has to be specified.

Table 2.10. <column> - Definitions

Attribute	Description	Required	Since
file	The XML schema, to be used as input for the source code generator.	No.	-
dir	Sets a directory such that all XML schemas in this directory will have code generated for them.	No	-
schemaURL	URL to an XML schema, to be used as input for the source code generator.	No.	**1.2**

In addition, a nested **<fileset>** can be specified as the source of input. Please refer to the samples shown below.

2.5.1.2. Parameters

Please find below the complete list of parameters that can be set on the Castor source generator to fine-tune the execution behavior.

Table 2.11. Ant task properties

Attribute	Description	Required	Since
package	The default package to be used during source code generation.	No; if not given, all classes will be placed in the root package.	-
todir	The destination directory to be used during source code generation. In this directory all generated Java classes will be placed.	No	-
bindingfile	A Castor source generator binding file.	No	-
lineseparator	Defines whether to use	No; if not set, system	-

Attribute	Description	Required	Since
	Unix- or Windows- or Mac-style line separators during source code generation. Possible values are: 'unix', 'win' or 'mac'.	property 'line.separator' is used instead.	
types	Defines what collection types to use (Java 1 vs. Java 2). Possible values: 'vector', 'arraylist' (aka 'j2') or 'odmg'.	No; if not set, the default collection used will be Java 1 type	-
verbose	Whether to output any logging messages as emitted by the source generator	No	-
warnings	Whether to suppress any warnings as otherwise emitted by the source generator	No	-
nodesc	If used, instructs the source generator not to generate *Descriptor classes.	No	-
generateMapping	If used, instructs the source generator to (additionally) generate a mapping file.	No	-
nomarshal	If specified, instructs the source generator not to create (un)marshalling methods within the Java classes generated.	No	-
caseInsensitive	If used, instructs the source generator to generate code for enumerated type lookup in a case insensitive manner.	No	-
sax1	If used, instructs the source generator to generate SAX-1 compliant code.	No	-
generateImportedSchemas	If used, instructs the source generator to generate code for imported schemas as	No	-

Attribute	Description	Required	Since
	well.		
nameConflictStrategy	If used, sets the name conflict strategy to use during XML code generation; possible values are '`warnViaConsoleDialog`' and '`informViaLog`'.	No	-
properties	Location of file defining a set of properties to be used during source code generation. This overrides the default mechanisms of configuring the source generator through a `castorbuilder.properties` (that has to be placed on the CLASSPATH)	No	-
automaticConflictStrategy	If used, sets the name conflict resolution strategy used during XML code generation; possible values are '`type`' and '`xpath`' (default being '`xpath`').	No	-
jClassPrinterType	Sets the mode for printing JClass instances during XML code generation; possible values are '`standard`' and '`velocity`' (default being '`standard`').	No	**1.2.1**
generateJdoDescriptors	If used, instructs the source generator to generate JDO class descriptors as well; default is false.	No	**1.3**
resourceDestination	Sets the destination directory for (generated) resources, e.g. `.castor.cdr` files.	No	**1.3.1**

2.5.1.3. Examples

2.5.1.3.1. Using a file

Below is an example of how to use this task from within an Ant target definition named 'castor:gen:src':

```
<target name="castor:gen:src" depends="init"
        description="Generate Java source files from XSD.">

    <taskdef name="castor-srcgen"
            classname="org.castor.anttask.CastorCodeGenTask"
            classpathref="castor.class.path" />
    <mkdir dir="generated" />
    <castor-srcgen file="src/schema/sample.xsd"
                   todir="generated-source"
                   package="org.castor.example.schema"
                   types="j2"
                   warnings="true" />
</target>
```

2.5.1.3.2. Using an URL

Below is the same sample as above, this time using the **url** attribute as the source of input instead:

```
<target name="castor:gen:src" depends="init"
        description="Generate Java source files from XSD.">

    <taskdef name="castor-srcgen"
            classname="org.castor.anttask.CastorCodeGenTask"
            classpathref="castor.class.path" />
    <mkdir dir="generated" />
    <castor-srcgen schemaURL="http://some.domain/some/path/sample.xsd"
                   todir="generated-source"
                   package="org.castor.example.schema"
                   types="j2"
                   warnings="true" />
</target>
```

2.5.1.3.3. Using a nested <fileset>

Below is the same sample as above, this time using the **url** attribute as the source of input instead:

```
<target name="castor:gen:src" depends="init"
        description="Generate Java source files from XSD.">

    <taskdef name="castor-srcgen"
            classname="org.castor.anttask.CastorCodeGenTask"
            classpathref="castor.class.path" />
    <mkdir dir="generated" />
    <castor-srcgen todir="generated-source"
                   package="org.castor.example.schema"
                   types="j2"
                   warnings="true" >
       <fileset dir="${basedir}/src/schema">
          <include name="**/*.xsd"/>
       </fileset>
    </castor-srcgen>
</target>
```

2.5.2. Maven 2 plugin

For those of you working with Maven 2 instead of Ant, the Maven 2 plugin for Castor can be used to integrate source code generation from XML schemas with the Castor XML code generator as part of the standard Maven build life-cycle. The following sections show how to configure the Maven 2 Castor plugin and hwo to instruct

Maven 2 to generate sources from your XML schemas.

2.5.2.1. Configuration

To be able to start source code generation from XML schema from within Maven, you will have to configure the Maven 2 Castor plugin as follows:

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>castor-maven-plugin</artifactId>
   <version>2.0</version>
</plugin>
```

Above configuration will trigger source generation using the default values as explained at the [Castor plugin page](), assuming that the XML schema(s) are located at `src/main/castor`, and code will be saved at `target/generated-sources/castor`. When generating sources for multiple schemas at the same time, you can put namespace to package mappings into `src/main/castor/castorbuilder.properties`.

To e.g. change some of these default locations, please add a <configuration> section to the plugin configuration as follows:

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>castor-maven-plugin</artifactId>
   <version>2.0</version>
   <configuration>
      <schema>src/main/resources/org/exolab/castor/builder/binding/binding.xsd</schema>
      <packaging>org.exolab.castor.builder.binding</packaging>
      <properties>src/main/resources/org/exolab/castor/builder/binding.generation.properties</properties>
   </configuration>
 </plugin>
```

Details on the available configuration properties can be found [here]().

By default, the Maven Castor plugin has been built and tested against a particular version of Castor. To switch to a newer version of Castor (not the plugin itself), please use a <dependencies> section as shown below to point the plugin to e.g. a newer version of Castor:

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>castor-maven-plugin</artifactId>
   <version>2.0</version>
   <dependencies>
     <dependency>
       <groupId>org.codehaus.castor</groupId>
       <artifactId>castor</artifactId>
       <version>1.3.1-SNAPSHOT</version>
     </dependency>
   </dependencies>
 </plugin>
```

2.5.2.2. Integration into build life-cycle

To integrate source code generation from XML schema into your standard build life-cycle, you will have to add an <executions> section to your standard plugin configuration as follows:

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
```

```
    <artifactId>castor-maven-plugin</artifactId>
    <version>2.0</version>
    <executions>
       <execution>
          <goals>
             <goal>generate</goal>
          </goals>
       </execution>
    </executions>
</plugin>
```

2.5.2.3. Example

Below command shows how to instruct Maven (manually) to generate Java sources from the XML schemas as configured above.

```
> mvn castor:generate
```

2.5.3. Command line

2.5.3.1. First steps

```
java org.exolab.castor.builder.SourceGeneratorMain -i foo-schema.xsd \
    -package com.xyz
```

This will generate a set of source files from the the XML Schema `foo-schema.xsd` and place them in the package `com.xyz`.

To compile the generated classes, simply run **javac** or your favorite compiler:

```
javac com/xyz/*.java
```

Created class will have `marshal` and `unmarshal` methods which are used to go back and forth between XML and an Object instance.

2.5.3.2. Source Generator - command line options

The source code generator has a number of different options which may be set. Some of these are done using the command line and others are done using a properties file located by default at `org/exolab/castor/builder/castorbuilder.properties`.

2.5.3.2.1. Specifying the input source

There's more than one way of specifying the input for the Castor code generator. **At least one** input source must be specified.

Table 2.12. Input sources

Option	Args	Description	Version
i	*filename*	The input XML Schema file	-

Option	Args	Description	Version
is	*URL*	URL of an XML Schema	**1.2 and newer**

2.5.3.2.2. Other command Line Options

Table 2.13. Other command line options

Option	Arguments	Description	Optional?
`-package`	package-name	The package for the generated source.	Optional
`-dest`	path	The destination directory in which to create the generated source	Optional
`-line-separator`	unix | mac | win	Sets the line separator style for the desired platform. This is useful if you are generating source on one platform, but will be compiling/modifying on another platform.	Optional
`-types`	type-factory	Sets which type factory to use. This is useful if you want JDK 1.2 collections instead of JDK 1.1 or if you want to pass in your own FieldInfoFactory (see Section 2.5.3.2.2.1, “Collection Types”).	Optional
`-h`		Shows the help/usage information.	Optional
`-f`		Forces the source generator to suppress all non-fatal errors, such as overwriting pre-existing files.	Optional
`-nodesc`		Do not generate the class descriptors	Optional
`-gen-mapping`		(Additionally) Generate a mapping file.	Optional
`-nomarshall`		Do not generate the marshaling framework methods (marshal, unmarshal, validate)	Optional
`-testable`		Generate the extra methods used by the CTF	Optional

Option	Arguments	Description	Optional?
		(Castor Testing Framework)	
`-sax1`		Generate marshaling methods that use the SAX1 framework (default is false).	Optional
`-binding-file`	<<binding file name>>.	Configures the use of a Binding File to allow finely-grained control of the generated classes	Optional
`-generateImportedSchemas`		Generates sources for imported XML Schemas in addition to the schema provided on the command line (default is false).	Optional
`-case-insensitive`		The generated classes will use a case insensitive method for looking up enumerated type values.	Optional
`-verbose`		Enables extra diagnostic output from the source generator	Optional
`-nameConflictStrategy`	<<conflict strategy name>>	Sets the name conflict strategy to use during XML code generation	Optional
`-fail`		Instructs the source generator to fail on the first error. When you are trying to figure out what is failing during source generation, this option will help.	Optional
`-classPrinter`	<<JClass printing mode>>.	Specifies the JClass printing mode to use during XML code generation; possible values are`standard` (default) and `velocity`; if no value is specified, the default mode is `standard`.	Optional
`-gen-jdo-desc`		(Additionally) generate JDO class descriptors.	Optional
`-resourcesDestination`	<destination>	An (optional) destination for (generated) resources	Optional

2.5.3.2.2.1. Collection Types

The source code generator has the ability to use the following types of collections when generating source code, using the `-type` option:

Table 2.14. Collection types

Option value	Type	Default
`-types j1`	Java 1.1	`java.util.Vector`
`-type j2`	Java 1.2	`java.util.Collection`
`-types odmg`	ODMG 3.0	`odmg.DArray`

The Java class name shown in above table indicates the default collection type that will be emitted during generation.

You can also write your own FieldInfoFactory to handle specific collection types. All you have to do is to pass in the fully qualified name of that FieldInfoFactory as follows:

```
-types com.personal.MyCoolFactory
```

Tip

For additional information about the Source Generator and its options, you can download the Source Generator User Document (PDF). Please note that the use of a binding file is not dicussed in that document.

2.6. XML schema support

Castor XML supports the W3C XML Schema 1.0 Second Edition Recommendation document (10/28/2004) The Schema Object Model (located in the package org.exolab.castor.xml.schema) provides an in-memory representation of a given XML schema whereas the XML code generator provides a binding between XML schema data types and structures into the corresponding ones in Java.

The Castor Schema Object Model can read (org.exolab.castor.xml.schema.reader) and write (org.exolab.castor.xml.schema.writer) an XML Schema as defined by the W3C recommandation. It allows you to create and manipulate an in-memory view of an XML Schema.

The Castor Schema Object Model supports the W3C XML Schema recommendation with no limitation. However the Source Generator does currently not offer a one to one mapping from an XML Schema component to a Java component for every XML Schema components; some limitations exist. The aim of the following sections is to provide a list of supported features in the Source Generator. Please keep in mind that the Castor Schema Object Model again can handle any XML Schema without limitations.

Some Schema types do not have a corresponding type in Java. Thus the Source Generator uses Castor implementation of these specific types (located in the org.exolab.castor.types package). For instance the `duration` type is implemented directly in Castor. Remember that the representation of XML Schema datatypes does not try to fit the W3C XML Schema specifications exactly. The aim is to map an XML Schema type to the Java type that is the best fit to the XML Schema type.

You will find next a list of the supported XML Schema data types and structures in the Source Code Generator. For a more detailed support of XML Schema structure and more information on the Schema Object Model, please refer to Source Generator User Document (PDF).

2.6.1. Supported XML Schema Built-in Datatypes

The following is a list of the supported datatypes with the corresponding facets and the Java mapping type.

2.6.1.1. Primitive Datatypes

Table 2.15. Supported primitive data types

XML Schema Type	Supported Facets	Java mapping type
anyURI	enumeration	`java.lang.String`
base64Binary		`byte[]`
boolean	pattern	`boolean` or `java.lang.Boolean`[a]
date	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.Date`
dateTime	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`java.util.Date`
decimal	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`java.math.BigDecimal`
double	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`double` or `java.lang.Double`[c]
duration	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.Duration`
float	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`float` or `java.lang.Float`[c]
gDay	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.GDay`
gMonth	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.GMonth`
gMonthDay	enumeration, maxInclusive, maxExclusive, minInclusive,	`org.exolab.castor.types.GMonthDay`

XML Schema Type	Supported Facets	Java mapping type
	minExclusive, pattern, whitespace[b]	
gYear	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.GYear`
gYearMonth	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.GYearMonth`
hexBinary		`byte[]`
QName	length, minLength, maxLength, pattern, enumeration	`java.lang.String`
string	length, minLength, maxLength, pattern, enumeration, whiteSpace	`java.lang.String`
time	enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, pattern, whitespace[b]	`org.exolab.castor.types.Time`

[a]For the various numerical types, the default behavior is to generate primitive types. However, if the use of wrappers is enabled by the following line in the `castorbuilder.properties` file: `org.exolab.castor.builder.primitivetowrapper=true` then the `java.lang.*` wrapper objects (as specified above) will be used instead.

[b] For the date/time and numeric types, the only supported value for whitespace is "collapse".

2.6.1.2. Derived Datatypes

Table 2.16. Supported derived data types

Type	Supported Facets	Java mapping type
byte	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`byte`/`java.lang.Byte` [c]
ENTITY		**Not implemented**
ENTITIES		**Not implemented**
ID	enumeration	`java.lang.String`
IDREF		`java.lang.Object`
IDREFS		`java.util.Vector` of `java.lang.Object`
int	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`int`/`java.lang.Integer`[c]
integer	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive,	`long`/`java.lang.Long`[c]

Type	Supported Facets	Java mapping type
	minExclusive, whitespace[b]	
language	length, minLength, maxLength, pattern, enumeration, whiteSpace	treated as a `xsd:string`[d]
long	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`long/java.lang.Long`[c]
Name		**Not implemented**
NCName	enumeration	`java.lang.String`
negativeInteger	totalDigits, fractionDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`long/java.lang.Long`[c]
NMTOKEN	enumeration, length, maxlength, minlength	`java.lang.String`
NMTOKENS		`java.util.Vector` of `java.lang.String`
NOTATION		**Not implemented**
nonNegativeInteger	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`long/java.lang.Long`[c]
nonPositiveInteger	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`long/java.lang.Long`[c]
normalizedString	enumeration, length, minLength, maxLength, pattern	`java.lang.String`
positiveInteger	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`long/java.lang.Long`[c]
short	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`short/java.lang.Short`[c]
token	length, minLength, maxLength, pattern, enumeration, whiteSpace	treated as a `xsd:string`[d],
unsignedByte	totalDigits, fractionDigits[a], maxExclusive, minExclusive, maxInclusive, minInclusive, pattern, whitespace[b]	`short/java.lang.Short`[c]

Type	Supported Facets	Java mapping type
unsignedInt	totalDigits, fractionDigits[a], maxExclusive, minExclusive, maxInclusive, minInclusive, pattern, whitespace[b]	`long`/`java.lang.Long`[c]
unsignedLong	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`java.math.BigInteger`
unsignedShort	totalDigits, fractionDigits[a], pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, whitespace[b]	`int` or `java.lang.Integer`[c]

[a]For the integral types, the only allowed value for fractionDigits is 0.

[b] For the date/time and numeric types, the only supported value for whitespace is "collapse".

[c]For the various numerical types, the default behavior is to generate primitive types. However, if the use of wrappers is enabled by the following line in the `castorbuilder.properties` file: `org.exolab.castor.builder.primitivetowrapper=true` then the `java.lang.*` wrapper objects (as specified above) will be generated instead.

[d] Currently, `<xsd:language>` and `<xsd:token>` are treated as if they were `<xsd:string>`.

2.6.2. Supported XML Schema Structures

Supporting XML schema structures is a constant work. The main structures are already supported with some limitations. The following will give you a rough list of the supported structures. For a more detailed support of XML Schema structure in the Source Generator or in the Schema Object Model, please refer to Source Generator User Document (PDF).

Supported schema components:

- Attribute declaration (`<attribute>`)
- Element declaration (`<element>`)
- Complex type definition (`<complexType>`)
- Attribute group definition (`<attributeGroup>`)
- Model group definition (`<group>`)
- Model group (`<all>`, `<choice>` and `<sequence>`)
- Annotation (`<annotation>`)
- Wildcard (`<any>`)
- Simple type definition (`<simpleType>`)

2.6.2.1. Groups

Grouping support covers both **model group definitions** (`<group>`) and **model groups** (`<all>`, `<choice>` and `<sequence>`). In this section we will label as a 'nested group' any model group whose first parent is another model group.

- For each top-level model group definition, a class is generated either when using the 'element' mapping property or the 'type' one.
- If a group -- nested or not -- appears to have `maxOccurs > 1`, then a class is generated to represent the items contained in the group.
- For each nested group, a class is generated. The name of the generated class will follow this naming convention: `Name,Compositor+,Counter?` where
  - 'Name' is name of the top-level component (element, complexType or group).
  - 'Compositor' is the compositor of the nested group. For instance, if a 'choice' is nested inside a sequence, the value of Compositor will be `SequenceChoice` ('Sequence'+'Choice'). Note: if the 'choice' is inside a Model Group and that Model Group **parent** is a Model Group Definition or a complexType then the value of'Compositor' will be only 'Choice'.
  - 'Counter' is a number that prevents naming collision.

2.6.2.2. Wildcard

`<any>` is supported and will be mapped to an `AnyNode` instance. However, full namespace validation is not yet implemented, even though an `AnyNode` structure is fully namespace aware.

`<anyAttribute>` is currently not supported. It is a work in progress.

2.7. Examples

In this section we illustrate the use of the XML code generator by discussing the classes generated from given XML schemas. The XML code generator is going to be used with the “java class mapping” property set to *element* (default value).

2.7.1. The invoice XML schema

2.7.1.1. The schema file

The input file is the schema file given with the XML code generator example in the distribution of Castor (under /src/examples/SourceGenerator/invoice.xsd).

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://castor.exolab.org/Test/Invoice">

    <xsd:annotation>
        <xsd:documentation>
            This is a test XML Schema for Castor XML.
        </xsd:documentation>
    </xsd:annotation>

    <!--
        A simple representation of an invoice. This is simply an example
        and not meant to be an exact or even complete representation of an invoice.
    -->
    <xsd:element name="invoice">
        <xsd:annotation>
            <xsd:documentation>
```

```
                A simple representation of an invoice
            </xsd:documentation>
        </xsd:annotation>

        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="ship-to">
                    <xsd:complexType>
                        <xsd:group ref="customer" />
                    </xsd:complexType>
                </xsd:element>
                <xsd:element ref="item"
                    maxOccurs="unbounded" minOccurs="1" />
                <xsd:element ref="shipping-method" />
                <xsd:element ref="shipping-date" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Description of a customer -->
    <xsd:group name="customer">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string" />
            <xsd:element ref="address" />
            <xsd:element name="phone"
                type="TelephoneNumberType" />
        </xsd:sequence>
    </xsd:group>

    <!-- Description of an item -->
    <xsd:element name="item">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Quantity"
                    type="xsd:integer" minOccurs="1" maxOccurs="1" />
                <xsd:element name="Price" type="PriceType"
                    minOccurs="1" maxOccurs="1" />
            </xsd:sequence>
            <xsd:attributeGroup ref="ItemAttributes" />
        </xsd:complexType>
    </xsd:element>

    <!-- Shipping Method -->
    <xsd:element name="shipping-method">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="carrier"
                    type="xsd:string" />
                <xsd:element name="option"
                    type="xsd:string" />
                <xsd:element name="estimated-delivery"
                    type="xsd:duration" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Shipping date -->
    <xsd:element name="shipping-date">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="date" type="xsd:date" />
                <xsd:element name="time" type="xsd:time" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- A simple U.S. based Address structure -->
    <xsd:element name="address">
        <xsd:annotation>
            <xsd:documentation>
                Represents a U.S. Address
            </xsd:documentation>
        </xsd:annotation>

        <xsd:complexType>
            <xsd:sequence>
                <!-- street address 1 -->
                <xsd:element name="street1"
                    type="xsd:string" />
```

```
                <!-- optional street address 2 -->
                <xsd:element name="street2"
                    type="xsd:string" minOccurs="0" />
                <!-- city-->
                <xsd:element name="city" type="xsd:string" />
                <!-- state code -->
                <xsd:element name="state"
                    type="stateCodeType" />
                <!-- zip-code -->
                <xsd:element ref="zip-code" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- A U.S. Zip Code -->
    <xsd:element name="zip-code">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:pattern value="[0-9]{5}(-[0-9]{4})?" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>

    <!-- State Code
        obviously not a valid state code....but this is just
        an example and I don't feel like creating all the valid
        ones.
    -->
    <xsd:simpleType name="stateCodeType">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z]{2}" />
        </xsd:restriction>
    </xsd:simpleType>

    <!-- Telephone Number -->
    <xsd:simpleType name="TelephoneNumberType">
        <xsd:restriction base="xsd:string">
            <xsd:length value="12" />
            <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}" />
        </xsd:restriction>
    </xsd:simpleType>

    <!-- Cool price type -->
    <xsd:simpleType name="PriceType">
        <xsd:restriction base="xsd:decimal">
            <xsd:fractionDigits value="2" />
            <xsd:totalDigits value="5" />
            <xsd:minInclusive value="1" />
            <xsd:maxInclusive value="100" />
        </xsd:restriction>
    </xsd:simpleType>

    <!-- The attributes for an Item -->
    <xsd:attributeGroup name="ItemAttributes">
        <xsd:attribute name="Id" type="xsd:ID" minOccurs="1"
            maxOccurs="1" />
        <xsd:attribute name="InStock" type="xsd:boolean"
            default="false" />
        <xsd:attribute name="Category" type="xsd:string"
            use="required" />
    </xsd:attributeGroup>
</xsd:schema>
```

The structure of this schema is simple: it is composed of a top-level element which is a complexType with references to other elements inside. This schema represents a simple invoice: an invoice is a customer (`customer` top-level group), an article (`item` element), a shipping method (`shipping-method` element) and a shipping date (`shipping-date` element). Notice that the `ship-to` element uses a reference to an `address` element. This `address` element is a top-level element that contains a reference to a non-top-level element (the `zip-cod` element). At the end of the schema we have two simpleTypes for representing a telephone number and a price. The Source Generator is used with the `element` property set for class creation so a class is going to be generated for all top-level elements. No classes are going to be generated for complexTypes and simpleTypes since the simpleType is not an enumeration.

To summarize, we can expect 7 classes : `Invoice`, `Customer`, `Address`, `Item`, `ShipTo`, `ShippingMethod` and `ShippingDate` and the 7 corresponding class descriptors. Note that a class is generated for the top-level group `customer`

2.7.1.2. Running the XML code generator

To run the source generator and create the source from the `invoice.xsd` file in a package `test`, we just call in the command line:

```
java -cp %CP% org.exolab.castor.builder.SourceGeneratorMain -i invoice.xsd -package test
```

2.7.1.3. The generated code

2.7.1.3.1. The Item.java class

To simplify this example we now focus on the `item` element.

```
<!-- Description of an item -->
<xsd:element name="item">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Quantity" type="xsd:integer"
                   minOccurs="1" maxOccurs="1" />
      <xsd:element name="Price" type="PriceType"
                   minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
    <xsd:attributeGroup ref="ItemAttributes" />
  </xsd:complexType>
</xsd:element>

<!-- Cool price type -->
<xsd:simpleType name="PriceType">
  <xsd:restriction base="xsd:decimal">
    <xsd:fractionDigits value="2" />
    <xsd:totalDigits value="5" />
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="100" />
  </xsd:restriction>
</xsd:simpleType>

<!-- The attributes for an Item -->
<xsd:attributeGroup name="ItemAttributes">
  <xsd:attribute name="Id" type="xsd:ID" minOccurs="1" maxOccurs="1" />
  <xsd:attribute name="InStock" type="xsd:boolean" default="false" />
  <xsd:attribute name="Category" type="xsd:string" use="required" />
</xsd:attributeGroup>
```

To represent an `Item` object, we need to know its `Id`, the `Quantity` ordered and the `Price` for one item. So we can expect to find a least three private variables: a string for the `Id` element, an `int` for the `quantity` element (see the section on XML Schema support if you want to see the mapping between a W3C XML Schema type and a java type), but what type for the `Price` element?

While processing the `Price` element, Castor is going to process the type of `Price` i.e. the simpleType `PriceType` which base is `decimal`. Since derived types are automatically mapped to parent types and W3C XML Schema `decimal` type is mapped to a `java.math.BigDecimal`, the price element will be a `java.math.BigDecimal`. Another private variable is created for `quantity`: quantity is mapped to a primitive java type, so a boolean `has_quantity` is created for monitoring the state of the quantity variable. The rest of the code is the *getter/setter* methods and the Marshalling framework specific methods. Please find below the complete `Item` class (with Javadoc comments stripped off):

```
/**
 * This class was automatically generated with
 * Castor 1.0.4,
 * using an XML Schema.
 */

package test;

public class Item implements java.io.Serializable {

   //--------------------------/
   //- Class/Member Variables -/
   //--------------------------/

   private java.lang.String _id;

   private int _quantity;

   /**
    * keeps track of state for field: _quantity
    */
   private boolean _has_quantity;

   private java.math.BigDecimal _price;

   //----------------/
   //- Constructors -/
   //----------------/

   public Item() {
      super();
   } //-- test.Item()


   //-----------/
   //- Methods -/
   //-----------/

   public java.lang.String getId() {
      return this._id; $
   } //-- java.lang.String getId()

   public java.math.BigDecimal getPrice() {
      return this._price;
   } //-- java.math.BigDecimal getPrice()

   public int getQuantity() {
      return this._quantity;
   } //-- int getQuantity()

   public boolean hasQuantity() {
      return this._has_quantity;
   } //-- boolean hasQuantity()

   public boolean isValid() {
      try {
         validate();
      } catch (org.exolab.castor.xml.ValidationException vex) {
         return false;
      }
      return true;
   } //-- boolean isValid()

   public void marshal(java.io.Writer out)
   throws org.exolab.castor.xml.MarshalException,org.exolab.castor.xml.ValidationException {
      Marshaller.marshal(this, out);
   } //-- void marshal(java.io.Writer)

   public void marshal(org.xml.sax.DocumentHandler handler)
   throws org.exolab.castor.xml.MarshalException, org.exolab.castor.xml.ValidationException {
      Marshaller.marshal(this, handler);
   } //-- void marshal(org.xml.sax.DocumentHandler)

   public void setId(java.lang.String _id) {
      this._id = _id;
   } //-- void setId(java.lang.String)
```

```
   public void setPrice(java.math.BigDecimal _price) {
      this._price = _price;
   } //-- void setPrice(java.math.BigDecimal)

   public void setQuantity(int _quantity) {
      this._quantity = _quantity;
      this._has_quantity = true;
   } //-- void setQuantity(int)

   public static test.Item unmarshal(java.io.Reader reader)
   throws org.exolab.castor.xml.MarshalException,org.exolab.castor.xml.ValidationException {
      return (test.Item) Unmarshaller.unmarshal(test.Item.class, reader);
   } //-- test.Item unmarshal(java.io.Reader)

   public void validate()
   throws org.exolab.castor.xml.ValidationException {
      org.exolab.castor.xml.Validator.validate(this, null);
   } //-- void validate()

}
```

The ItemDescriptor class is a bit more complex. This class is containing inner classes which are the XML field descriptors for the different components of an ‘Item’ element i.e. id, quantity and price.

2.7.1.3.2. The PriceType.java class

TODO ...

2.7.1.3.3. The Invoice.java class

In this section, we focus on the 'invoice' element as shown again below:

```
<xsd:element name="invoice">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="ship-to">
            <xsd:complexType>
               <xsd:group ref="customer" />
            </xsd:complexType>
         </xsd:element>
         <xsd:element ref="item"    minOccurs="1" maxOccurs="unbounded" />
         <xsd:element ref="shipping-method" />
         <xsd:element ref="shipping-date" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Amongst other things, an `<invoice>` is made up of at least one, but potentially many `<item>` elements. The Castor XML code generator creates a Java collection named 'itemList' for this unbounded element declaration, of type `java.util.List` if the scode generator is used with the '`arraylist`' field factory.

```
    private java.util.List _itemList;
```

If the '`j1`' field factory is used, this will be replaced with ...

```
    private java.util.Vector _itemList;
```

The complete class as generated (with irrelevant code parts removed) in '`j2`' (aka '`arraylist`') mode is shown below:

```
public class Invoice implements java.io.Serializable {
```

```
...

private java.util.List _itemList;

...

public Invoice()
 {
    super();
    this._itemList = new java.util.ArrayList();
} //-- xml.c1677.invoice.generated.Invoice()

...

public void addItem(xml.c1677.invoice.generated.Item vItem)
    throws java.lang.IndexOutOfBoundsException
{
    this._itemList.add(vItem);
} //-- void addItem(xml.c1677.invoice.generated.Item)

public void addItem(int index, xml.c1677.invoice.generated.Item vItem)
    throws java.lang.IndexOutOfBoundsException
{
    this._itemList.add(index, vItem);
} //-- void addItem(int, xml.c1677.invoice.generated.Item)

public java.util.Enumeration enumerateItem()
{
    return java.util.Collections.enumeration(this._itemList);
} //-- java.util.Enumeration enumerateItem()

public xml.c1677.invoice.generated.Item getItem(int index)
    throws java.lang.IndexOutOfBoundsException
{
    // check bounds for index
    if (index < 0 || index >= this._itemList.size()) {
        throw new IndexOutOfBoundsException("getItem: Index value '" + index
           + "' not in range [0.." + (this._itemList.size() - 1) + "]");
    }

    return (xml.c1677.invoice.generated.Item) _itemList.get(index);
} //-- xml.c1677.invoice.generated.Item getItem(int)

public xml.c1677.invoice.generated.Item[] getItem()
{
    int size = this._itemList.size();
    xml.c1677.invoice.generated.Item[] array = new xml.c1677.invoice.generated.Item[size];
    for (int index = 0; index < size; index++){
        array[index] = (xml.c1677.invoice.generated.Item) _itemList.get(index);
    }

    return array;
} //-- xml.c1677.invoice.generated.Item[] getItem()

public int getItemCount()
{
    return this._itemList.size();
} //-- int getItemCount()

public java.util.Iterator iterateItem()
{
    return this._itemList.iterator();
} //-- java.util.Iterator iterateItem()

public void removeAllItem()
{
    this._itemList.clear();
} //-- void removeAllItem()

public boolean removeItem(xml.c1677.invoice.generated.Item vItem)
{
    boolean removed = _itemList.remove(vItem);
    return removed;
} //-- boolean removeItem(xml.c1677.invoice.generated.Item)

public xml.c1677.invoice.generated.Item removeItemAt(int index)
{
```

```
        Object obj = this._itemList.remove(index);
        return (xml.c1677.invoice.generated.Item) obj;
    } //-- xml.c1677.invoice.generated.Item removeItemAt(int)

    public void setItem(int index, xml.c1677.invoice.generated.Item vItem)
        throws java.lang.IndexOutOfBoundsException
    {
        // check bounds for index
        if (index < 0 || index >= this._itemList.size()) {
            throw new IndexOutOfBoundsException("setItem: Index value '"
               + index + "' not in range [0.." + (this._itemList.size() - 1) + "]");
        }

        this._itemList.set(index, vItem);
    } //-- void setItem(int, xml.c1677.invoice.generated.Item)

    public void setItem(xml.c1677.invoice.generated.Item[] vItemArray)
    {
        //-- copy array
        _itemList.clear();

        for (int i = 0; i < vItemArray.length; i++) {
                this._itemList.add(vItemArray[i]);
        }
    } //-- void setItem(xml.c1677.invoice.generated.Item)

}
```

2.7.2. Non-trivial real world example

Two companies wish to trade with each other using a Supply Chain messaging system. This system sends and receives Purchase Orders and Order Receipt messages. After many months of discussion they have finally decided upon the structure of the Version 1.0 of their message XSD and both are presently developing solutions for it. One of the companies decides to use Java and Castor XML support for (un)marshaling and Castor's code generator to accelerate their development process.

2.7.2.1. The Supply Chain XSD

```
            <title>supplyChainV1.0.xsd</title>
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

    <xs:element name="Data">
      <xs:annotation>
        <xs:documentation>
          This section contains the supply chain message data
        </xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:choice>
          <xs:element name="PurchaseOrder">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"/>
              </xs:sequence>
              <xs:attribute name="OrderNumber" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
          <xs:element name="OrderReceipt">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="LineItem" type="ReceiptLineItemType" maxOccurs="unbounded"/>
              </xs:sequence>
              <xs:attribute name="OrderNumber" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
```

```
    </xs:element>

    <xs:complexType name="SkuType">
      <xs:annotation>
        <xs:documentation>Contains Product Identifier</xs:documentation>
      </xs:annotation>
      <xs:sequence>
        <xs:element name="Number" type="xs:integer"/>
        <xs:element name="ID" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ReceiptSkuType">
      <xs:annotation>
        <xs:documentation>Contains Product Identifier</xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="SkuType">
          <xs:sequence>
            <xs:element name="InternalID" type="xs:string"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="LineItemType">
      <xs:sequence>
        <xs:element name="Sku" type="SkuType"/>
        <xs:element name="Value" type="xs:double"/>
        <xs:element name="BillingInstructions" type="xs:string"/>
        <xs:element name="DeliveryDate" type="xs:date"/>
        <xs:element name="Number" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ReceiptLineItemType">
      <xs:sequence>
        <xs:element name="Sku" type="ReceiptSkuType"/>
        <xs:element name="Value" type="xs:double"/>
        <xs:element name="PackingDescription" type="xs:string"/>
        <xs:element name="ShipDate" type="xs:dateTime"/>
        <xs:element name="Number" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

2.7.2.2. Binding file? -- IT IS REQUIRED!

If you run the Castor CodeGenerator on the above XSD you end up with the following set of classes. (You also get lots of warning messages with the present version.)

```
Data.java
DataDescriptor.java
LineItem.java
LineItemDescriptor.java
LineItemType.java
LineItemTypeDescriptor.java
OrderReceipt.java
OrderReceiptDescriptor.java
PurchaseOrder.java
PurchaseOrderDescriptor.java
ReceiptLineItemType.java
ReceiptLineItemTypeDescriptor.java
ReceiptSkuType.java
ReceiptSkuTypeDescriptor.java
Sku.java
SkuDescriptor.java
SkuType.java
SkuTypeDescriptor.java
```

The problem here is that there are two different elements with the same name in different locations in the XSD.

This causes a Java code generation conflict. By default, Castor uses the element name as the name of the class. So the second class generated for the LineItem definition, which is different than the first, overwrites the first class generated.

A binding file is therefore necessary to help the Castor code generator differentiate between these generated classes and as such avoid such generation conflicts. That is, you can 'bind' an element in the XML schema to a differently named class file that you want to generate. This keeps different elements separate and ensures that source is properly generated for each XML Schema object.

Tip

The warning messages for Castor 0.99+ are very useful in assisting you in your creation of the binding file. For the example the warning messages for the example are:

```
    Warning: A class name generation conflict has occurred between element
            '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
    Warning: A class name generation conflict has occurred between element
            '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
    Warning: A class name generation conflict has occurred between element
            '/Data/OrderReceipt/LineItem' and element '/Data/PurchaseOrder/LineItem'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
    Warning: A class name generation conflict has occurred between element
            'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
    Warning: A class name generation conflict has occurred between element
            'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
    Warning: A class name generation conflict has occurred between element
            'complexType:ReceiptLineItemType/Sku' and element 'complexType:LineItemType/Sku'.
            Please use a Binding file to solve this problem.Continue anyway [not recommended] (y|n|?)y
```

The following binding file definition will overcome the naming issues for the generated classes:

```
<binding xmlns="http://www.castor.org/SourceGenerator/Binding"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.castor.org/SourceGenerator/Binding C:\\Castor\\xsd\\binding.xsd"
         defaultBinding="element">

   <elementBinding name="/Data/PurchaseOrder/LineItem">
      <java-class name="PurchaseOrderLineItem"/>
   </elementBinding>

   <elementBinding name="/Data/OrderReceipt/LineItem">
      <java-class name="OrderReceiptLineItem"/>
   </elementBinding>

   <elementBinding name="/complexType:ReceiptLineItemType/Sku">
      <java-class name="OrderReceiptSku"/>
   </elementBinding>

   <elementBinding name="/complexType:LineItemType/Sku">
      <java-class name="PurchaseOrderSku"/>
   </elementBinding>

</binding>
```

One thing to notice in the above `binding.xml` file is that the name path used is relative to the root of the XSD **and not** the root of the target XML. Also notice that the two complex types have the "complexType:" prefix to identify them followed by the name path relative to the root of the XSD.

The new list of generated classes is:

```
Data.java
DataDescriptor.java
LineItem.java
LineItemDescriptor.java
LineItemType.java
LineItemTypeDescriptor.java
OrderReceipt.java
OrderReceiptDescriptor.java
OrderReceiptLineItem.java
OrderReceiptLineItemDescriptor.java
OrderReceiptSku.java
OrderReceiptSkuDescriptor.java
PurchaseOrder.java
PurchaseOrderDescriptor.java
PurchaseOrderLineItem.java
PurchaseOrderLineItemDescriptor.java
PurchaseOrderSku.java
PurchaseOrderSkuDescriptor.java
ReceiptLineItemType.java
ReceiptLineItemTypeDescriptor.java
ReceiptSkuType.java
ReceiptSkuTypeDescriptor.java
Sku.java
SkuDescriptor.java
SkuType.java
SkuTypeDescriptor.java
```

The developers can now use these generated classes with Castor to (un)marshal the supply chain messages sent by their business partner.

Chapter 3. XML code generation - Extensions

3.1. XML code generation extensions - Motivation

With Castor 1.2 and earlier releases it has already been possible to generate Java classes from an XML schema and use these classes for XML data binding **without** having to write a mapping file.

This is possible because the Castor XML code generator generated - in addition to the domain classes - a set of XML descriptor classes as well, with one descriptor class generated per generated domain class. It's this XML descriptor class that holds all the information required to map Java classes and/or field members to XML artifacts, as set out in the original XML schema definitions. This includes ....

- artefact names
- XML namespace URIs
- XML namespace prefix
- validation code

Starting with Castor 1.3, a mechanism has been added to the XML code generator that allows extension of these core offerings so that either additional content is added to the generated domain classes additonal descriptor classes are gernated.

3.1.1. JDO extensions for the Castor XML code generator

3.1.1.1. JDO extensions - Motivation

With Castor 1.2 and previous releases it was already possible to generate Java classes from an XML schema and use these classes for XML data binding **without** having to write a mapping file.

This is possible because the Castor XML code generator generated - in addition to the domain classes - a set of XML descriptor classes as well, with one descriptor class generated per generated domain class. It's this XML descriptor class that holds all the information required to map Java classes and/or field members to XML artifacts, as set out in the original XML schema definitions. This includes ....

- artefact names
- XML namespace URIs
- XML namespace prefix
- validation code

In addition, it was already possible to use the generated set of domain classes in Castor JDO for object-/relational mapping purpose by supplying a (manually written) JDO-specific mapping file. Whilst technically not very difficult, this was still an error-prone task, especially in a context where tens or hundreds of classes were generated from a set of XML schemas.

The *JDO extensions for the Castor XML code generator* extend the code generator in such a way that a second set of descriptor classes is generated: the JDO descriptor classes. These new descriptor classes define the mapping between Java (domain) objects and database tables/columns, and as such remove the requirement of having to write a JDO-specific mapping file.

> **Note**
>
> Please note that Castor JDO - upon startup - internally converts the information provided in the JDO mapping file to (JDO) descriptor classes. As such, the approach outlined above simply re-uses an existing code base and just automates the production of those descriptor classes.

The following sections introduce the general principles, define the XML schema artifacts available to annotate an existing XML schema and highlight the usage of these artifacts by providing examples. At the same time, a limited set of current product limitations are spelled out.

3.1.1.2. Limitations

With release 1.3 of Castor, the following limitations exist for the JDO extensions of the XML code generator:

1. The extensions currently can only be used in **type** mode of the XML code generator.
2. There's currently no support for **key generators**. There's work in progress to add this functionality, though.
3. There's currently no support for bidirectional relations, modelled through the use of <xs:id> and <xs:idref> constructs.

3.1.1.3. Prerequisites

To facilitate the detailed explanations in the following sections, we now define a few <complexType> definitions that we want to map against an existing database schema, and the corresponding SQL statements to create the required tables.

3.1.1.3.1. Sample XML schemas

```
<complexType name="bookType">
   <sequence>
      <element name="isbn" type="xs:string" />
      <element name="pages" type="xs:integer" />
      <element name="lector" type="lectorType" />
      <element name="authors" type="authorType" maxOccurs="unbounded" />
   </sequence>
</complexType>

<complexType name="lectorType">
   <sequence>
      <element name="siNumber" type="xs:integer" />
      <element name="name" type="xs:string" />
   </sequence>
</complexType>

<complexType name="authorType">
   <sequence>
      <element name="siNumber" type="xs:integer" />
      <element name="name" type="xs:string" />
   </sequence>
</complexType>
```

3.1.1.3.2. Sample DDL statements

```
CREATE TABLE author_table (
   sin INTEGER NOT NULL,
   name VARCHAR(20) NOT NULL
);

CREATE TABLE lector_table (
   sin INTEGER NOT NULL,
   name VARCHAR(20) NOT NULL
);

CREATE TABLE book_table (
   isbn VARCHAR(13) NOT NULL,
   pages INTEGER,
   lector_id INTEGER NOT NULL,
   author_id INTEGER NOT NULL
);
```

3.1.1.4. Configuring the XML code generator

To have the Castor XML code generator generate JDO class descriptors when processing a set of XML schemas, please use one of the following methods:

Table 3.1. Accessing options

Usage	Method	Description
`SourceGenerator`	`setJdoDescriptorCreation(boolean)`	Supply a value of `true` to enable this feature.
`SourceGeneratorMain`	Flag `-gen-jdo-desc`	Set this optional flag to enable this feature.
Ant task for XML code generator	`generateJdoDescriptors` option	Set this to a value of `true`.

3.1.1.5. The JDO annotations for XML schemas

This section enlists the XML artifacts available to annotate an existing XML schema with JDO extension-specific information. These constructs are defined themselves in an XML schema `jdo-extensions.xsd` that has a target namespace of `http://www.castor.org/binding/persistence`.

To enable proper validation of your XML schemas when editing JDO annotations, and to enable XML completion in your preferred XML editor, please add `schemaLocation` information to your XML schema definition as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://your/target/namespace"
    xmlns:jdo="http://www.castor.org/binding/persistence"
    xmlns="http://your/target/namespace"                          ❶
    xsi:schemaLocation="http://www.castor.org/binding/persistence http://www.castor.org/jdo-extensions.xsd">

...

</xs:schema>
```

where ...

❶ The values supplied in the `schemaLocation` attribute define the location of the XML schema for any XML artefacts bound to the `http://www.castor.org/binding/persistence` namespace.

3.1.1.5.1. <table> element

The <table> element allows you to map an <complexType> definition to a database table within a database, and to specify the identity (frequently referred to as `primary key`), as follows:

```
<xs:complexType name="authorType">
   <xs:annotation>
      <xs:appinfo>
         <jdo:table name="author_table">                                          ❶
            <jdo:primary-key>                                                     ❷
               <jdo:key>siNumber</jdo:key>
            </jdo:primary-key>
         </jdo:table>
      </xs:appinfo>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="siNumber" type="xs:integer" />
      <xs:element name="name" type="xs:string" />
   </xs:sequence>
</xs:complexType>
```

where ...

❶ The `<jdo:table ...>` defines the name of the database table to which the complex type definition `authorType` should be mapped.

❷ The `<jdo:primary-key>` indicates which artifacts of the content model of the complex type definition should be used as the corresponding object identity; in database terms, this is often referred to as `primary key`.

Above example maps the complex type `authorType` to the table `author_table`, and specifies that the member `siNumber` be used as object identity.

The XML schema definition for the `<table>` element is defined as follows:

```
<xs:element name="table">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="primaryKey" type="jdo:pkType"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="accessMode" use="optional" default="shared">
         <xs:simpleType>
            <xs:restriction base="xs:string">
               <xs:enumeration value="read-only"/>
               <xs:enumeration value="shared"/>
               <xs:enumeration value="exclusive"/>
               <xs:enumeration value="db-locked"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="detachable" type="xs:boolean" default="false"/>
   </xs:complexType>
</xs:element>

<xs:complexType name="pkType">
   <xs:sequence>
      <xs:element name="key" type="xs:string" maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>
```

3.1.1.5.2. <column> element

The <column> element allows you to map a member of content model of a <complexType> definition to a column within a database table.

```
<xs:complexType name="authorType">
   <xs:annotation>
      <xs:appinfo>
         <jdo:table name="author_table">
            <jdo:primary-key>
               <jdo:key>siNumber</jdo:key>
            </jdo:primary-key>
         </jdo:table>
      </xs:appinfo>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="siNumber" type="xs:integer" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:column name="sin" type="integer" />                    ❶
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="name" type="xs:string" />
   </xs:sequence>
</xs:complexType>
```

where ....

❶ Defines that the element definition `siNumber` be mapped against the database column `sin`, and that the (database) type of this column is `integer`.

Above example maps the element `isNumber` to the database column `sin`, and specifies the database type to be used for persistence (`integer`, in this case).

The XML schema definition for `<column>` is defined as follows:

```
<xs:element name="column">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="jdo:readonlyDirtyType">
            <xs:attribute name="name" type="xs:string" use="required" />
            <xs:attribute name="type" type="xs:string" use="required" />
            <xs:attribute name="acceptNull" type="xs:boolean" use="optional"
               default="true" />
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

where the content is described as follows:

Table 3.2. <column> - Definitions

Name	Description
name	Name of the column
type	JDO-type of the column
acceptNull	Whether this field accepts NULL values or not

3.1.1.5.3. <one-to-one> element

The <one-to-one> element allows you to map a member of content model of a <complexType> definition to a 1:1 relation to another <complexType>.

```
<xs:complexType name="bookType">
   <xs:annotation>
      <xs:appinfo>
         <jdo:table name="book_type_table">
            <jdo:primary-key>
               <jdo:key>isbn</jdo:key>
            </jdo:primary-key>
         </jdo:table>
      </xs:appinfo>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="isbn" type="xs:string" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:column name="isbn" type="varchar" />
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="pages" type="xs:integer" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:column name="pages" type="integer" />
            </xs:appinfo>
         </xs:annotation>                                                        ❶
      </xs:element>
      <xs:element name="lector" type="lectorType" >
         <xs:annotation>
            <xs:appinfo>
               <jdo:one-to-one name="lector_id" />
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="authors" type="authorType" maxOccurs="unbounded" >
         ...
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

where ....

❶ Defines a 1:1 relation to another <complexType>, additionally providing the necessary foreign key column at the database level.

Above example maps the element `lector` to a 1:1 relation to the complex type `lectorType`, and specifies the (column name of the) foreign key to be used (`lector_id` in this case).

The XML schema definition for `<one-to-one>` is defined as follows:

```
<xs:element name="one-to-one">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="jdo:readonlyDirtyType">
            <xs:attribute name="name" type="xs:string"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

where the content is described as follows:

Table 3.3. <one-to-one> - Definitions

Name	Description
name	Name of the column that represents the foreign key of this relation

3.1.1.5.4. <one-to-many> element

The <one-to-many> element allows you to map a member of the content model of a <complexType> definition as part of a 1:M relation to another <complexType>.

```
<xs:complexType name="bookType">
   <xs:annotation>
      <xs:appinfo>
         <jdo:table name="book_type_table">
            <jdo:primary-key>
               <jdo:key>isbn</jdo:key>
            </jdo:primary-key>
         </jdo:table>
      </xs:appinfo>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="isbn" type="xs:string" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:column name="isbn" type="varchar" />
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="pages" type="xs:integer" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:column name="pages" type="integer" />
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="lector" type="lectorType" >
         <xs:annotation>
            <xs:appinfo>
               <jdo:one-to-one name="lector_id" />
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
      <xs:element name="authors" type="authorType" maxOccurs="unbounded" >
         <xs:annotation>
            <xs:appinfo>
                <jdo:one-to-many name="book_id" />                        ❶
            </xs:appinfo>
         </xs:annotation>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

where ....

❶ Defines a 1:M relation to another <complexType>, additionally providing the necessary foreign key column for the many member at the database level.

Above example maps the element `authors` as part of a 1:M relation to the complex type `authorType`, and specifies the (column name of the) foreign key of the many member to be used (`book_id` in this case).

The XML schema definition for `<one-to-many>` is given as follows:

```
<xs:element name="one-to-many">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="jdo:readonlyDirtyType">
            <xs:attribute name="name" type="xs:string" />
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

with the following details applying:

Table 3.4. <one-to-many> - Definitions

Name	Description
name	Name of the column that represents the (many) foreign key of this relation

3.1.1.6. Using the generated (domain) classes with Castor JDO

Once you have generated domain classes and descriptor classes (both XML and JDO) from your set of XML schemas, you'll be able to use them as are. There's a few minor changes, which we are going to highlight below, but the main benefit is that you **not** have to write a JDO mapping file.

3.1.1.6.1. Empty mapping file

As you have already generated JDO descriptor classes for each of your domain objects, you won't have to supply mappings for those classes anymore. As such, your mapping file will stay empty, as shown:

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                           "http://castor.org/mapping.dtd">
<mapping>

   <!-- no mappings required -->
</mapping>
```

Note

Please note that you can of course supply mappings for those classes that stand outside of the generation process from your XML schemas. It is possible, too, to match both modes. In other words, a domain class mapped manually will be able to refer to a domain class as generated.

3.1.1.6.2. Use of a `JDOClassDescriptorResolver`

In order for Castor to be able to access the generated (JDO) class descriptors and to load those classes from the file system, you will have to configure an instance of `JDOClassDescriptorResolver` and pass it to your `JDOManager` instance when loading the JDO configuration.

The following example shows how to configure Castor JDO so that the classes generated from the sample XML schema above can be used with CASTOR JDO seamlessly.

```
JDOClassDescriptorResolver resolver = new JDOClassDescriptorResolverImpl();
resolver.addClass(org.castor.jdo.extension.sample.BookType.class);
resolver.addClass(org.castor.jdo.extension.sample.LectorType.class);
resolver.addClass(org.castor.jdo.extension.sample.AuthorType.class);

InputSource jdoConfiguration = ....;
JDOManager.loadConfiguration(jdoConfiguration, null, null, resolver);

JDOManager jdoManager = JDOManager.createInstance("jdo-extensions");
...
```

Alternatively, if the classes generated from the sample XML schema shown above reside in the same package, you can configure the `JDOClassDescriptorResolver` as follows:

```
JDOClassDescriptorResolver resolver = new JDOClassDescriptorResolverImpl();
resolver.addPackage("org.castor.jdo.extension.sample");
...
```

Tip

For the latter approach to work, you will have to make sure that the `.castor.jdo.cdr` files generated alongside your domain (and descriptor classes) are included in your application deployment units. If not, Castor JDO will not be able to load the descriptor classes from the file system, and throw an exception.

3.1.2. SOLRJ extensions for the Castor XML code generator

3.1.2.1. SOLRJ extensions - Motivation

With Castor 1.2 and previous releases it was already possible to generate Java classes from an XML schema and use these classes for XML data binding **without** having to write a mapping file.

This is possible because the Castor XML code generator generated - in addition to the domain classes - a set of XML descriptor classes as well, with one descriptor class generated per generated domain class. It's this XML descriptor class that holds all the information required to map Java classes and/or field members to XML artifacts, as set out in the original XML schema definitions. This includes ....

- artefact names
- XML namespace URIs
- XML namespace prefix
- validation code

The *SOLRJ extensions for the Castor XML code generator* extend the code generator in such a way that the set of domain classes is augmented with SOLRJ-specific `@Field` annotations.

The following sections introduce the general principles, define the XML schema artifacts available to annotate an existing XML schema and highlight the usage of these artifacts by providing examples.

3.1.2.2. Prerequisites

To facilitate the detailed explanations in the following sections, we now define a few <complexType> definitions that we want to be able to store in a SOLR index in addition to vanilla XML data binding funtionality.

3.1.2.2.1. Sample XML schemas

```
<complexType name="bookType">
   <sequence>
      <element name="isbn" type="xs:string" />
      <element name="pages" type="xs:integer" />
   </sequence>
</complexType>
```

3.1.2.3. The SOLRJ annotations for XML schemas

This section enlists the XML artifacts available to annotate an existing XML schema with SOLRJ extension-specific information. These constructs are defined themselves in an XML schema `solrj-extensions.xsd` that has a target namespace of `http://www.castor.org/binding/solrj`.

To enable proper validation of your XML schemas when editing SOLRJ annotations, and to enable XML completion in your preferred XML editor, please add `schemaLocation` information to your XML schema definition as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://your/target/namespace"
    xmlns:solr="http://www.castor.org/binding/solrj"
    xmlns="http://your/target/namespace"
    xsi:schemaLocation="http://www.castor.org/binding/solrj http://www.castor.org/solrj-extens❶ions.xsd">

...

</xs:schema>
```

where ...

❶ The values supplied in the `schemaLocation` attribute define the location of the XML schema for any XML artefacts bound to the `http://www.castor.org/binding/solrj` namespace.

3.1.2.3.1. <field> element

The <field> element allows you to map a member of the content model of a `<complexType>` definition to SOLRJ field.

```
<complexType name="bookType">
   <sequence>
      <element name="isbn" type="xs:string">
         <xs:annotation>
            <xs:appinfo>
                <solrj:field name="id" />                                      ❶
            </xs:appinfo>
         </xs:annotation>
      </element>
      <element name="pages" type="xs:integer">
         <xs:annotation>
            <xs:appinfo>
                <solrj:field />                                                ❷
            </xs:appinfo>
         </xs:annotation>
```

```
      </element>
   </sequence>
</complexType>
```

where ....

❶ Defines that the element definition `isbn` be mapped against the SOLRJ field `id`.
❷ Defines that the element definition `name` be mapped to the SOLRJ field `name`.

Above example maps the element `isbn` to the SOLR index field `id`, and the element `name` to the identically-named SOLR index field. Please note that a SOLR index field name does not have to be specified if the field name and the Java property name are identical.

Above complex type definition will be transformed to the corresponding Java property definitions (within a class):

```
public class BookType {

    @Field("id")
    private String isbn;

    @Field
    private long pages;

}
```

The XML schema definition for `<field>` is defined as follows:

```
<xs:element name="field">
  <xs:annotation>
     <xs:documentation>
            Element 'field' is used to specify the use of the SOLRJ
            @Field annotation.
        </xs:documentation>
  </xs:annotation>
  <xs:complexType>
     <xs:attribute name="name" type="xs:string" use="optional">
        <xs:annotation>
           <xs:documentation>
                    Attribute 'name' is used to specify the name of
                    the index field to be mapped against; if not used,
                    the name of the Java property will be used as filed
                    name.
                </xs:documentation>
        </xs:annotation>
     </xs:attribute>
  </xs:complexType>
</xs:element>
```

where the content is described as follows:

Table 3.5. <field> - Definitions

Name	Description	Optional?
name	Name of the SOLR index field.	Yes

3.1.2.4. Using the generated domain classes with SOLR

Once you have generated domain classes (and descriptor classes for the XML binding) from your set of XML schemas, you'll be able to use them as are.