JAXB 2 Advanced

<https://dev64.wordpress.com/2012/03/19/xml-schema-jaxb-example/>

**Практический пример использования XML schema и JAXB**

Однажды в очередной раз возникла необходимость реализации очередного XML вебсервиса в очередном проекте. Я решил использовать эту задачу для тестирования описаний [XML schema, сделанных ранее](https://dev64.wordpress.com/2012/02/02/xml-schema/).

После написания данного поста, позже сделал ещё ряд постов о JAXB. Получилась небольшая серия.

**Оглавление**

* [Валидация атрибутов, подключение схемы при анмаршаллинге (обязательно прочитать после этого поста)](https://dev64.wordpress.com/2012/11/21/jaxb-use-schema/)
* [здесь вы можете прочитать небольшое дополнение про обработку исключений JAXBException](https://dev64.wordpress.com/2012/04/12/jaxb-exceptions-processing/).
* А также: [Чтение XML с подменой namespace с помощью JAXB](https://dev64.wordpress.com/2012/04/26/namespace-filter-example/).
* [Использование аннотаций с JAXB, генерация XML schema из java-кода](https://dev64.wordpress.com/2012/05/15/using-annotations-with-jaxb/). В данном посте также примеры маршаллинга с форматированием XML.
* [Формируем XML без xsd схемы, с использованием аннотаций (наглядный пример)](https://dev64.wordpress.com/2013/03/29/jaxb-to-xml-without-xsd/)
* [Маршаллинг JAXB объектов с CDATA](https://dev64.wordpress.com/2013/03/29/jaxb-marshalling-with-cdata/)
* [Расширенный пример использования JAXB аннотаций для форматирования даты в XML](https://dev64.wordpress.com/2012/05/15/extended-jaxb-annotations/). Этот пост расскажет как с помощью аннотаций добавить специальные обработчики, для форматирования данных.
* [Решение проблемы с XmlRootElement при генерации кода из xsd-схемы с помощью xjc](https://dev64.wordpress.com/2012/10/10/issue-with-xmlrootelement-when-generating-sources-from-xsdschema/)
* [Извлекаем Body из SOAP сообщения и применяем JAXB для работы с SOAP](https://dev64.wordpress.com/2012/10/26/get-soap-body-java/)
* [Форматирование (marshalling) XML с указанием кодировки с использованием JAXB](https://dev64.wordpress.com/2012/12/09/marshal-xml-specify-encoding-using-jaxb/)
* [Вывод XML без заголовка с помощью JAXB](https://dev64.wordpress.com/2013/01/14/jaxb-marshal-without-xml-header/)
* [Другие посты по Java…](https://dev64.wordpress.com/java)

**Введение**

Допустим, мне нужно реализовать Web Service для работы с книгами. В зависимости от типа запроса, сервис может возвращать различные варианты ответов.

В документе *result* обязательный элемент *code*, содержащий код успеха операции OK или NOK.  
Элемент *description* необязательный, он содержит словесное описание ошибки, если таковая случилась.  
Остальные элементы тоже необязательные. Это либо элемент *book* — книга, с описанием книги. Либо элемент *books* — список книг

**Примеры XML**

**Результат содержит bookId**

|  |  |
| --- | --- |
| 1  2  3  4 | <result xmlns="<http://foobar.com>">  <code>OK</code>  <bookId>IBN0921801</bookId>  </result> |

**Ошибочный результат**

|  |  |
| --- | --- |
| 1  2  3  4 | <result xmlns="<http://foobar.com>">  <code>NOK</code>  <description>No book found</description>  </result> |

**Результат содержит данные о книге**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <result xmlns="<http://foobar.com>">  <code>OK</code>  <book>    <bookId>ISBN0993209</bookId>    <author>Joshua Bloch</author>    <title>Effective Java</title>    <cover>effective-java.png</cover>  </book>  </result> |

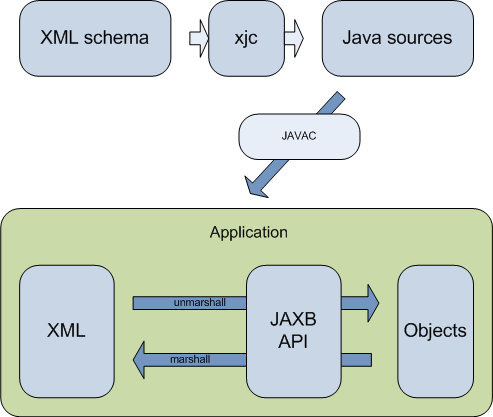
**Результат содержит данные список книг**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <result xmlns="<http://foobar.com>">  <code>OK</code>  <books>  <book>    <bookId>ISBN01213209</bookId>    <author>Scott Mayers</author>    <title>Effective C++</title>    <cover>effective-cpp.png</cover>  </book>  <book>    <bookId>ISBN0993209</bookId>    <author>Joshua Bloch</author>    <title>Effective Java</title>    <cover>effective-java.png</cover>  </book>  </books>  </result> |

Мне нужно реализовать работу с вышеприведенными XML с использованием JAXB. Задача состоит в том, чтобы JAXB автоматически преобразовывал XML в Java объекты и обратно и сообщал при этом об ошибках, если таковые присутствуют в XML документе. Т.е. документ должен валидироваться на соответствие XML схеме.

Работа с JAXB в целом делается так.

* Для XML пишется [XML схема в виде xsd файла](https://dev64.wordpress.com/2012/02/02/xml-schema/)
* Используя утилиту xjc из стандартной поставки Java JDK генерируете из схемы набор классов.
* Классы включаются в программу. JAXB marshaller используется для генерации XML из объектов, unmarshaller — для получения объектов из XML

[](https://dev64.files.wordpress.com/2012/03/jaxb.png)

Я для описанного сервиса написал нижеприведенную схему. Опускаю промежуточные неудачные варианты. Просто упомяну тонкие моменты. См. под примером схемы.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | <?xml version="1.0"?>  <xs:schema xmlns:xs="<http://www.w3.org/2001/XMLSchema>"   targetNamespace="<http://foobar.com>"   xmlns:tns= "<http://foobar.com>"   elementFormDefault="qualified">    <xs:simpleType name="codeType">    <xs:restriction base="xs:string">      <xs:enumeration value="OK"/>      <xs:enumeration value="NOK"/>    </xs:restriction>  </xs:simpleType>    <xs:complexType name="bookType">    <xs:sequence>      <xs:element name="bookId" type="xs:string"/>      <xs:element name="author" type="xs:string"/>      <xs:element name="title" type="xs:string"/>      <xs:element name="cover" type="xs:string"/>    </xs:sequence>  </xs:complexType>    <xs:complexType name="bookListType">    <xs:sequence>      <xs:element name="book" type="tns:bookType" minOccurs="1" maxOccurs="unbounded"/>    </xs:sequence>  </xs:complexType>      <xs:element name="result">    <xs:complexType>      <xs:sequence>        <xs:element name="code" type="tns:codeType" minOccurs="1" maxOccurs="1" />        <xs:choice>          <xs:element name="bookList"            type="tns:bookListType" minOccurs="0" maxOccurs="1"/>          <xs:element name="book" type="tns:bookType" minOccurs="0" maxOccurs="1"/>          <xs:element name="bookId" type="xs:string" minOccurs="0" maxOccurs="1"/>        </xs:choice>      </xs:sequence>    </xs:complexType>  </xs:element>    </xs:schema> |

В схеме обязательно должно присутствовать ключевое слово elementFormDefault=»qualified». Ещё один тонкий момент — обязательно при описании типов использовать в них явно указанный namespace и ссылаться на него: xmlns:tns=»<http://foobar.com>. Без этих 2-х тонкостей валидация не работает.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <xs:schema xmlns:xs="<http://www.w3.org/2001/XMLSchema>"   targetNamespace="<http://foobar.com>"   xmlns:tns= "<http://foorar.com>"   elementFormDefault="qualified">  ...  <xs:element name="result">  ...        <xs:element name="code" type="tns:codeType" minOccurs="1" maxOccurs="1" />  ...          <xs:element name="book" type="tns:profileType" minOccurs="0" maxOccurs="1"/>  ...          <xs:element name="bookId" type="xs:string" minOccurs="0" maxOccurs="1"/>  </xs:element> |

**Тестируюсь**

1. Помещаю XML shema, приведенную выше в файл jaxb-test.xsd  
2. Генерирую Java-исходники из jaxb-test.xsd:

> xjc -p com.wordpress.dev64.jaxbtest.model jaxb-test.xsd

3. Копирую получившееся дерево директорий себе в классы проекта

4. Пишу юнит-тест и запускаю его

Ниже простейший unit-test, демонстрирующий парсинг из XML. Всего навсего 3 инструкции. Создаем JAXBContext, создаем unmarshaller и вызываем  
unmarshal.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.wordpress.dev64.test;    import com.wordpress.dev64.jaxbtest.model.CodeType;  import com.wordpress.dev64.jaxbtest.model.Result;  import org.junit.Assert;  import org.junit.Test;    import javax.xml.bind.JAXBContext;  import javax.xml.bind.JAXBException;  import javax.xml.bind.Unmarshaller;  import java.io.StringReader;    public class TestJaxb {      private static String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foobar.com>\">" +              "<code>NOK</code>" +              "<description>abc</description></result>";        @Test      public void test() throws JAXBException {          JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");          Unmarshaller unmarshaller = jc.createUnmarshaller();            Result result = (Result)                  unmarshaller.unmarshal(new StringReader(TEST\_XML));            Assert.assertEquals(CodeType.NOK, result.getCode());          Assert.assertEquals("abc", result.getDescription());      }  } |

«Дело в шляпе». Юнит-тест работает. Однако есть подводный камень. А что если скормить ему некорректный XML. Будет ли exception?

Пробую во входном XML изменить имя namespace. Вместо <http://www.foobar.com> указываю <http://www.foorar.com>. Делаю юнит-тест:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Test (expected=UnmarshalException.class)  public void testInvalidNameSpace() throws JAXBException {      String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foorar.com>\">" +              "<code>NOK</code>" +              "<description>abc</description></result>";      JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");      Unmarshaller unmarshaller = jc.createUnmarshaller();        Result result = (Result)              unmarshaller.unmarshal(new StringReader(TEST\_XML));        Assert.assertEquals(CodeType.NOK, result.getCode());      Assert.assertEquals("abc", result.getDescription());  } |

В этом случае честно кидается UnmarshalException:

javax.xml.bind.UnmarshalException: unexpected element (uri:"http://foorar.com", local:"result"). Expected elements are

at com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallingContext.handleEvent(UnmarshallingContext.java:631)

at com.sun.xml.internal.bind.v2.runtime.unmarshaller.Loader.reportError(Loader.java:236)

Следующий случай. Порчу закрывающийся тег у result:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Test //(expected=UnmarshalException.class)  public void testInvalidXML() throws JAXBException {      String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foobar.com>\">" +              "<code>NOK</code>" +              "<description>abc</description></reult>";      JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");      Unmarshaller unmarshaller = jc.createUnmarshaller();        Result result = (Result)              unmarshaller.unmarshal(new StringReader(TEST\_XML));        Assert.assertEquals(CodeType.NOK, result.getCode());      Assert.assertEquals("abc", result.getDescription());  } |

Exception:

javax.xml.bind.UnmarshalException

- with linked exception:

[org.xml.sax.SAXParseException: The element type "result" must be terminated by the matching end-tag "".]

Всё пока идёт по плану. При внесении ошибок в XML кидаются exception-ы. Но не всё так просто. Я беру правильный XML и добавляю в него  
неожиданный тег *code1*. Ожидаю, что возникнет UnmarshalException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @Test (expected=UnmarshalException.class)  public void testUnknownTag() throws JAXBException {      String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foobar.com>\">" +              "<code>NOK</code>" +              "<code1>XXX</code1>" +              "<description>abc</description></result>";      JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");      Unmarshaller unmarshaller = jc.createUnmarshaller();        Result result = (Result)              unmarshaller.unmarshal(new StringReader(TEST\_XML));        Assert.assertEquals(CodeType.NOK, result.getCode());      Assert.assertEquals("abc", result.getDescription());  } |

Но возникает иная ситуация. Оказывается при валидации на соответствие схеме никаких exception-ов не возникает!  
Оставляю такой же неожиданный тег. Добавляю ValidationEventHandler:

unmarshaller.setEventHandler(new ValidationEventHandler() {

@Override

public boolean handleEvent(ValidationEvent event) {

System.out.println(event.getMessage());

return true; //To change body of implemented methods use File | Settings | File Templates.

}

});

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | @Test  public void testUnknownTag2() throws JAXBException {      String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foobar.com>\">" +              "<code>NOK</code>" +              "<code1>XXX</code1>" +              "<description>abc</description></result>";      JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");      Unmarshaller unmarshaller = jc.createUnmarshaller();        unmarshaller.setEventHandler(new ValidationEventHandler() {          @Override          public boolean handleEvent(ValidationEvent event) {              System.out.println(event.getMessage());              return true;  //To change body of implemented methods use File | Settings | File Templates.          }      });      Result result = (Result)              unmarshaller.unmarshal(new StringReader(TEST\_XML));        Assert.assertEquals(CodeType.NOK, result.getCode());      Assert.assertEquals("abc", result.getDescription());  } |

Тестируюсь. Печатается:

unexpected element (uri:"http://foobar.com", local:"code1"). Expected elements are ,,,,

А хотелось бы, чтобы бросался exception. Сделать это оказывается просто. В ValidationEventHandler просто возвращаю false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | @Test  public void testUnknownTag2() throws JAXBException {      String TEST\_XML="<?xml version = \"1.0\" ?>" +              "<result xmlns=\"<http://foobar.com>\">" +              "<code>NOK</code>" +              "<code1>XXX</code1>" +              "<description>abc</description></result>";      JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");      Unmarshaller unmarshaller = jc.createUnmarshaller();        unmarshaller.setEventHandler(new ValidationEventHandler() {          @Override          public boolean handleEvent(ValidationEvent event) {              return false;          }      });        Result result = (Result)              unmarshaller.unmarshal(new StringReader(TEST\_XML));        Assert.assertEquals(CodeType.NOK, result.getCode());      Assert.assertEquals("abc", result.getDescription());  } |

Получаю exception:

javax.xml.bind.UnmarshalException: unexpected element (uri:"http://foobar.com", local:"code1"). Expected elements are ,,,,

at com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallingContext.handleEvent(UnmarshallingContext.java:631)

Таким образом валидация заработала.

Одно замечание. В вышеприведенных примерах вызов unmarshaller происходит так:

|  |  |
| --- | --- |
| 1  2 | Result result = (Result)          unmarshaller.unmarshal(new StringReader(TEST\_XML)); |

Пока не знаю почему но в одном из проектов вышеприведенный фрагмент отказался компилироваться , unmarshaller пришлось вызывать так:

|  |  |
| --- | --- |
| 1  2  3 | JAXBElement<Result> element = (JAXBElement<Result>)          unmarshaller.unmarshal(new StringReader(TEST\_XML));  Result result = element.getValue(); |

Там, конечно был уже не Result, а другой объект. Я оставил тип в контексте рассмотренного примера, для понимания о чём речь.

**Выводы**

Чтобы работала валидация в JAXB нужно в схеме не забыть про elementFormDefault=»qualified» и применение явных namespace-ов.  
Чтобы бросался exception при выявлении несоответствий схеме, нужно выставлять ValidationEventHandler возвращающий false.

JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");

Unmarshaller unmarshaller = jc.createUnmarshaller();

unmarshaller.setEventHandler(new ValidationEventHandler() {

@Override

public boolean handleEvent(ValidationEvent event) {

return false;

}

});

Result result = (Result)

unmarshaller.unmarshal(new StringReader(TEST\_XML));

Спасибо, если дочитали пост до конца.

Рекомендую обязательно посмотреть ещё один пост, написанный позже.  
[Валидация атрибутов, подключение схемы при анмаршаллинге](https://dev64.wordpress.com/2012/11/21/jaxb-use-schema/)

Ещё один линк, чтобы не потерять:  
[Maven JAXB 2.1 Plugin](http://mojo.codehaus.org/jaxb2-maven-plugin/index.html)

P.S. B еще одна ссылка:  
[XML с Java JAXB](http://javism.blogspot.com/2012/01/xml-java-jaxb.html)

P.P.S. Обязательно рекомендую заглянуть на страничку с документацией на класс Unmarshaller для списка разных способов его вызова:  
[Unmarshaller javadoc](http://jaxb.java.net/nonav/2.2.1/docs/api/javax/xml/bind/Unmarshaller.html)

А также весьма полезное руводство по JAXB: <http://jaxb.java.net/tutorial/>

В качестве одной из альтернатив JAXB может быть Apache Digester:

[Парсим XML с несколькими namespace с помощью Apache Digester](https://dev64.wordpress.com/2013/02/15/parse-xml-with-several-namespaces-using-digester/)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://dev64.wordpress.com/2012/04/12/jaxb-exceptions-processing/>

# Обработка JAXB исключений

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Апрель 12, 2012](https://dev64.wordpress.com/2012/04/12/jaxb-exceptions-processing/)

Некоторое время назад я писал [пост про обработку XML c применением JAXB](https://dev64.wordpress.com/2012/03/19/xml-schema-jaxb-example/). В этом посте небольшое дополнение про обработку исключений.

Внутри JAXBException-ов сообщение может быть равно null, тогда надо ориентироваться по сообщению внутри ex.getCause():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | try {  JAXBContext jc = JAXBContext.newInstance("com.wordpress.dev64.jaxbtest.model");  Unmarshaller unmarshaller = jc.createUnmarshaller();    unmarshaller.setEventHandler(new ValidationEventHandler() {      @Override      public boolean handleEvent(ValidationEvent event) {          return false;      }  });    JAXBElement<Result> result = (JAXBElement<Result>)          unmarshaller.unmarshal(new StringReader(TEST\_XML));  } catch (JAXBException ex) {      String message = ex.getMessage();      if (message == null) {          message = ex.getCause().getMessage();          if (message == null) {              message = "Unidentified JAXB error";          }      }      System.out.println(message); |

Мне кажется такое поведение слегка странным. Однако это стандартный API — ничего не поделаешь.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://dev64.wordpress.com/2013/03/29/jaxb-marshalling-with-cdata/>

# Маршалинг JAXB объекта с CDATA

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Март 29, 2013](https://dev64.wordpress.com/2013/03/29/jaxb-marshalling-with-cdata/)

Ещё один пост о JAXB. На этот раз про маршаллинг данных с CDATA. Анмаршаллинг CDATA из XML происходит автоматически, однако в обратную сторону, когда нужно автоматически сгенерировать XML — не все так просто.

Для начала постановка задачи. Есть некоторый заданный заказчиком формат XML, в котором некоторые теги должны быть помещены внутрь CDATA. Например, запрос на отсылку почтового сообщения.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="no"?>  <MailRequest>  <To>first@address.com</To>  <To>other@address.net</To>  <From><![CDATA[somebody@somewhere.ie]]></From>  <Message><![CDATA[some test message]]></Message>  </MailRequest> |

Я уже писал [пост, как с помощью аннотаций быстро создать java-объект, пригодный для маршаллинга](https://dev64.wordpress.com/2013/03/29/jaxb-to-xml-without-xsd/). Поэтому сразу пишу беру объект с аннотациями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | @XmlRootElement(name = "MailRequest")  @XmlAccessorType(XmlAccessType.FIELD)  @XmlType(propOrder = {          "to",          "from",          "message"  })  public class MailServiceRequest {      @XmlElement(name = "To", required = true)      protected List<String> to = new ArrayList<String>();      @XmlElement(name = "From", required = true)      protected String from;      @XmlElement(name = "Message", required = true)      protected String message;        public MailServiceRequest to(String to) {          this.to.add(to);          return this;      }        public MailServiceRequest from(String from) {          this.from = from;          return this;      }        public MailServiceRequest message(String message) {          this.message = message;          return this;      }  } |

В приведенном коде уже есть аннотации, определяющие какие имена тегов использовать и в каком порядке выводить поля объекта в виде xml. Также видно что одно из полей является списком. Это распространенный use case.

Теперь нужно сделать маршаллинг, чтобы на выходе получились как положено поля помещенные внутрь CDATA. В JAXB встроенных средств для этого нет. Однако есть в DOM transformation. При этом JAXB умеет маршалить объект в Document. Это позволяет выполнить задачу в 2 этапа: сначала из JAXB вывести в Document, вторым шагом из Document вывести в XML с форматированием и добавлением тега CDATA:

Добавляю 3 метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public byte[] marshalWithCDATA() throws JAXBException,      ParserConfigurationException, TransformerException {      return marshalWithCDATA(marshalToDocument());  }    private Document marshalToDocument() throws ParserConfigurationException,      JAXBException {      DocumentBuilderFactory docBuilderFactory =              DocumentBuilderFactory.newInstance();      Document document =              docBuilderFactory.newDocumentBuilder().newDocument();        JAXBContext context = JAXBContext.newInstance(MailServiceRequest.class);      Marshaller marshaller = context.createMarshaller();      marshaller.marshal(this, document);      return document;  }    // @see <http://stackoverflow.com/questions/7536973/jaxb-marshalling-and-unmarshalling-cdata>  private byte[] marshalWithCDATA(Document document)          throws TransformerException {        TransformerFactory transformerFactory =              TransformerFactory.newInstance();      Transformer transformer = transformerFactory.newTransformer();      transformer.setOutputProperty(              OutputKeys.CDATA\_SECTION\_ELEMENTS,              "Message From");      ByteArrayOutputStream result = new ByteArrayOutputStream();      transformer.transform(new DOMSource(document), new StreamResult(result));      return result.toByteArray();  } |

Основной метод marshalWithCDATA(), он вызывает из себя 2 других, чтобы не получилось слишком длинного метода. Собственно в методе, осуществляющем трансформацию, поля, которые необходимо помечать в CDATA указываются с помощью вызова:

|  |  |
| --- | --- |
| 1  2  3 | transformer.setOutputProperty(          OutputKeys.CDATA\_SECTION\_ELEMENTS,          "Message From"); |

Пишу тест, проверяющий, что приведенный код работает:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Test  public void testMarshalWithCDATA() throws Exception {      MailServiceRequest request = new MailServiceRequest()              .to("first@address.com")              .to("other@address.net")              .from("somebody@somewhere.ie")              .message("some test message");        System.out.println(new String(request.marshalWithCDATA(), "UTF-8"));  } |

Тестирую, получаю на выходе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="no"?>  <MailRequest>  <To>first@address.com</To>  <To>other@address.net</To>  <From><![CDATA[somebody@somewhere.ie]]></From>  <Message><![CDATA[some test message]]></Message>  </MailRequest> |

Если добавить в код трансформатора несколько строчек:

|  |  |
| --- | --- |
| 1  2 | transformer.setOutputProperty(OutputKeys.INDENT, "yes");  transformer.setOutputProperty("{<http://xml.apache.org/xslt>}indent-amount", "2"); |

то на выход можно получить форматированный XML ([подробности в другом посте](https://dev64.wordpress.com/2012/10/22/pretty-print-dom-document-by-java/)):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="no"?>  <MailRequest>    <To>first@address.com</To>    <To>other@address.net</To>    <From><![CDATA[somebody@somewhere.ie]]></From>    <Message><![CDATA[some test message]]></Message>  </MailRequest> |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://dev64.wordpress.com/2013/03/29/jaxb-to-xml-without-xsd/>

# Формируем XML без xsd схемы с помощью аннотаций

### [Галерея](https://dev64.wordpress.com/type/gallery/)

Posted on [Март 29, 2013](https://dev64.wordpress.com/2013/03/29/jaxb-to-xml-without-xsd/)

Я уже писал [несколько постов про JAXB](https://dev64.wordpress.com/2012/05/15/using-annotations-with-jaxb/), тем не менее темы для постов остаются. Решил добавить еще несколько примеров. Например — маршаллинг и анмаршаллинг вообще без xsd-файлов.

Постановка небольшой задачи. Пусть имеется веб-сервис, отсылающий e-mail-ы, в веб-интерфейсом. Пусть у меня есть документ, описывающий формат запроса, который выглядит как-то так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <MailRequest>      <To>first@address.com</To>      <To>other@address.net</To>      <From>somebody@somewhere.ie</From>      <Message>some test message</Message>  </MailRequest> |

Мне нужно написать код для формирования такого запроса. Можно конечно сделать XSD-схему, сгенерировать из нее классы. Подключить классы к проекту… Однако хочется удобства, да и скорость реализации играет значение. Поэтому пишу простой класс c данными:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class MailServiceRequest {      protected List<String> to;      protected String from;      protected String message;  } |

Добавляю JAXB аннотации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @XmlRootElement(name="MailRequest")  @XmlAccessorType(XmlAccessType.FIELD)  @XmlType(propOrder = {          "to",          "from",          "message"  })  public class MailServiceRequest {      @XmlElement(name = "To", required=true)      protected List<String> to=new ArrayList<String>();      @XmlElement(name = "From", required = true)      protected String from;      @XmlElement(name = "Message", required = true)      protected String message;  } |

@XmlRootElement задает имя тега для элемента, который будет создан из нашей структуры.  
@XmlAccessorType задает, что доступ будет осуществляться непосредственно к полям (не нужно делать getter-ов)  
@XmlType задает в каком порядке добавлять поля класса в выходной документ  
@XmlElement задает имя каждому элементу, и обязателен ли элемент.

Этого уже достаточно чтобы работать с JAXB — маршаллером. Однако нам надо как-то структуру создавать, поэтому для удобства добавлю в нее методы, превращающие ее в билдер:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | @XmlRootElement(name="MailRequest")  @XmlAccessorType(XmlAccessType.FIELD)  @XmlType(propOrder = {          "to",          "from",          "message"  })  public class MailServiceRequest {      @XmlElement(name = "To", required=true)      protected List<String> to=new ArrayList<String>();      @XmlElement(name = "From", required = true)      protected String from;      @XmlElement(name = "Message", required = true)      protected String message;        public MailServiceRequest to(String to) {          this.to.add(to);          return this;      }        public MailServiceRequest from(String from) {          this.from = from;          return this;      }        public MailServiceRequest message(String message) {          this.message = message;          return this;      }  } |

Чтобы делать маршаллинг, добавляю собственно сам метод для маршаллинга:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public byte[] marshal() throws JAXBException {      JAXBContext context = JAXBContext.newInstance(MailServiceRequest.class);      Marshaller marshaller = context.createMarshaller();      marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE);      ByteArrayOutputStream outputStream = new ByteArrayOutputStream();      marshaller.marshal(this, outputStream);      return outputStream.toByteArray();  } |

Теперь пишу тест:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class MailServiceRequestTest {      @Test      public void testMarshal() throws Exception {          MailServiceRequest request = new MailServiceRequest()              .to("first@address.com")              .to("other@address.net")              .from("somebody@somewhere.ie")              .message("some test message");            System.out.println(new String(request.marshal(), "UTF-8"));      }  } |

Получаю на выходе из теста:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <MailRequest>      <To>first@address.com</To>      <To>other@address.net</To>      <From>somebody@somewhere.ie</From>      <Message>some test message</Message>  </MailRequest> |

**Выводы**

* Если схемы изначально нет, можно ее и не делать. Простые объекты можно маршалить в XML без xsd. Схема требуется только для валидации.
* Такой подход позволяет уменьшить количество кода, а также повысить его читаемость, например с помощью применения билдера, как в вышеприведенном примере.
* Описания на отдельные аннотации смотреть здесь: <http://docs.oracle.com/javaee/6/api/javax/xml/bind/annotation/package-summary.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://blog.bdoughan.com/2012/11/creating-generic-list-wrapper-in-jaxb.html>

### Creating a Generic List Wrapper in JAXB

To marshal/unmarshal a list with a [JAXB (JSR-222)](http://jcp.org/en/jsr/detail?id=222) implementation you need to create a wrapper object to hold the list. People find this onerous having to create multiple wrapper objects for this purpose.  Below I'll demonstrate that in reality you only need to create one. This post is based on an [answer I gave on Stack Overflow](http://stackoverflow.com/questions/13272288/is-it-possible-to-programmatically-configure-jaxb/13273022" \l "13273022" \t "_blank).  
  
**Generic Wrapper**  
  
We will create a generic wrapper class with a *List* property annotated with *@XmlAnyElement(lax=true)*. The type of the object used to populate this list will be based on its root element (see: [Using @XmlAnyElement to Build a Generic Message](http://blog.bdoughan.com/2010/08/using-xmlanyelement-to-build-generic.html)).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package blog.list.wrapper;    import java.util.\*;  import javax.xml.bind.annotation.XmlAnyElement;    public class Wrapper<T> {        private List<T> items;        public Wrapper() {          items = new ArrayList<T>();      }        public Wrapper(List<T> items) {          this.items = items;      }        @XmlAnyElement(lax=true)      public List<T> getItems() {          return items;      }    } |

**Java Model**  
  
Below is the Java model that we'll use for this example.  With this approach we need to annotate each class that can appear in the root level collection with *@XmlRootElement*.  
  
**Address**  
  
In the Address class we will use the default element names that JAXB implementations derive from classes and properties.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package blog.list.wrapper;    import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  public class Address {        private String street;        public String getStreet() {          return street;      }        public void setStreet(String street) {          this.street = street;      }    } |

**Person**  
Here we will explicitly specify the element names that will be used.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package blog.list.wrapper;    import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement(name="PERSON")  public class Person {        private String name;        @XmlElement(name="NAME")      public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }    } |

**XML**  
The following XML files will be the input to the demo code.  There are a couple of items to note about the XML:

1. There are multiple root elements (*addresses* and *PERSONS*) that we need to unmarshal to the same class.
2. The elements corresponding to items in the collection (*address* and *PERSON*) correspond to the *@XmlRootElement* annotations on *Address* and *Person*.

**addresses.xml**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?xml version="1.0" encoding="UTF-8"?>  <addresses>      <address/>          <street>1 Some Street</street>      </address>      <address/>          <street>2 Another Road</street>      </address>  </addresses> |

**persons.xml**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?xml version="1.0" encoding="UTF-8"?>  <PERSONS>      <PERSON>          <NAME>Jane Doe</NAME>      </PERSON>      <PERSON>          <NAME>John Smith</NAME>      </PERSON>  </PERSONS> |

**Demo**  
  
The demo code below demonstrates how to use the *Wrapper* class. Since the root element can be different you will need to specify that you want to unmarshal to the wrapper class. Alternatively you could leverage the *@XmlElementDecl* annotation to associate multiple root elements with the wrapper class (see: [JAXB and Root Elements](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html)).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | package blog.list.wrapper;    import java.util.List;  import javax.xml.bind.\*;  import javax.xml.namespace.QName;  import javax.xml.transform.stream.StreamSource;    public class Demo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance(Wrapper.class, Person.class,                  Address.class);            // Unmarshal          Unmarshaller unmarshaller = jc.createUnmarshaller();          List<Address> addresses = unmarshal(unmarshaller, Address.class, "addresses.xml");          List<Person> persons = unmarshal(unmarshaller, Person.class, "persons.xml");            // Marshal          Marshaller marshaller = jc.createMarshaller();          marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);          marshal(marshaller, addresses, "addresses");          marshal(marshaller, persons, "PERSONS");       }        /\*\*       \* Unmarshal XML to Wrapper and return List value.       \*/      private static <T> List<T> unmarshal(Unmarshaller unmarshaller,              Class<T> clazz, String xmlLocation) throws JAXBException {          StreamSource xml = new StreamSource(xmlLocation);          Wrapper<T> wrapper = (Wrapper<T>) unmarshaller.unmarshal(xml,                  Wrapper.class).getValue();          return wrapper.getItems();      }        /\*\*       \* Wrap List in Wrapper, then leverage JAXBElement to supply root element       \* information.       \*/      private static void marshal(Marshaller marshaller, List<?> list, String name)              throws JAXBException {          QName qName = new QName(name);          Wrapper wrapper = new Wrapper(list);          JAXBElement<Wrapper> jaxbElement = new JAXBElement<Wrapper>(qName,                  Wrapper.class, wrapper);          marshaller.marshal(jaxbElement, System.out);      }    } |

**Output**  
  
Below is the output from running the demo code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <addresses>      <address/>          <street>1 Some Street</street>      </address>      <address/>          <street>2 Another Road</street>      </address>  </addresses>  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <PERSONS>      <PERSON>          <NAME>Jane Doe</NAME>      </PERSON>      <PERSON>          <NAME>John Smith</NAME>      </PERSON>  </PERSONS> |

**Further Reading**  
  
If you enjoyed this post then you may also be interested in the following:

* [Using @XmlAnyElement to Build a Generic Message](http://blog.bdoughan.com/2010/08/using-xmlanyelement-to-build-generic.html)
* [JAXB and Root Elements](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html)

Posted by Blaise Doughan at [6:38 AM](http://blog.bdoughan.com/2012/11/creating-generic-list-wrapper-in-jaxb.html" \o "permanent link) [http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=1710464868724994296&postID=946946021678336364)

[Email This](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=946946021678336364&target=email" \t "_blank" \o "Email This)[BlogThis!](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=946946021678336364&target=blog" \t "_blank" \o "BlogThis!)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=946946021678336364&target=twitter" \t "_blank" \o "Share to Twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=946946021678336364&target=facebook" \t "_blank" \o "Share to Facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=946946021678336364&target=pinterest" \t "_blank" \o "Share to Pinterest)

Labels: [Collection Property](http://blog.bdoughan.com/search/label/Collection%20Property), [JAXB](http://blog.bdoughan.com/search/label/JAXB), [XmlAnyElement](http://blog.bdoughan.com/search/label/XmlAnyElement), [XmlRootElement](http://blog.bdoughan.com/search/label/XmlRootElement)

#### 6 comments:

1. http://3.bp.blogspot.com/_wg73kpMXUVw/SXpybgs8s4I/AAAAAAAAAIU/IFWovzlocao/S45-s35/10680.jpg

[*Gerard Davison*](https://www.blogger.com/profile/16521924431312439170)[November 16, 2012 at 7:58 AM](http://blog.bdoughan.com/2012/11/creating-generic-list-wrapper-in-jaxb.html?showComment=1353070705240#c5219769977685329803)

Blaise,  
  
This looks interesting - is there any way to signal to the XML Schema generator what the plural name should be? For example if I have a JAX-RS method that returns:  
  
@GET  
public Wrapper get() {};  
  
This will fail to generate the matching XML Schema definition in the WADL because Wrapper isn't a @XmlRootElement. Is there any obvious mechanism which would statically allow us to specify the plural name / namespace without access to the Marshall/Unmarshall mechanism.   
  
I guess a set of trivial sub classes would solve this problem; but I was interested to hear if you had another solution that wasn't obvious.  
  
Thanks,  
  
Gerard

[Reply](javascript:;)

[Replies](javascript:;)

* 1. http://lh5.googleusercontent.com/-vqbzMZDtO4s/AAAAAAAAAAI/AAAAAAAAAGw/DmS7X-NuOh4/s35-c/photo.jpg

[*Blaise Doughan*](https://www.blogger.com/profile/02016807982653343777)[November 22, 2012 at 4:57 PM](http://blog.bdoughan.com/2012/11/creating-generic-list-wrapper-in-jaxb.html?showComment=1353621453835#c3495285241756194568)

Hi Gerard,  
  
You could have multiple *@XmlElementDecl* annotations that correspond to the *Wrapper* class on an *ObjectFactory* (a class annotated with *@XmlRegistry*).  
  
See:  
- [JAXB and Root Elements](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html)  
  
-Blaise

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html>

### JAXB and Root Elements

*@XmlRootElement* is an annotation that people are used to using with JAXB (JSR-222).  It's purpose is to uniquely associate a root element with a class.  Since JAXB classes map to complex types, it is possible for a class to correspond to multiple root elements. In this case *@XmlRootElement* can not be used and people start getting a bit confused.  In this post I'll demonstrate how *@XmlElementDecl* can be used to map this use case.  
  
  
**XML Schema**   
  
The XML schema below contains three root elements:  *customer*, *billing-address*, and *shipping-address*.  The *customer* element has an anonymous complex type, while *billing-address* and *shipping-address* are of the same named type (*address-type*).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <?xml version="1.0" encoding="UTF-8"?>  <xs:schema      xmlns:xs="<http://www.w3.org/2001/XMLSchema>"      targetNamespace="<http://www.example.org/customer>"      xmlns="<http://www.example.org/customer>"      elementFormDefault="qualified">        <xs:element name="customer">          <xs:complexType>              <xs:sequence>                  <xs:element ref="billing-address"/>                  <xs:element ref="shipping-address"/>              </xs:sequence>          </xs:complexType>      </xs:element>        <xs:complexType name="address-type">          <xs:sequence>              <xs:element name="street" type="xs:string"/>          </xs:sequence>      </xs:complexType>        <xs:element name="billing-address" type="address-type"/>        <xs:element name="shipping-address" type="address-type"/>    </xs:schema> |

**Generated Model**  
  
Below is a JAXB model that was generated from the XML schema.  The same concepts apply when adding JAXB annotations to an existing Java model.  
  
**Customer**   
  
JAXB domain classes correspond to complex types.  Since the *customer* element had an anonymous complex type the *Customer* class has an *@XmlRootElement* annotation.  This is because only one XML element can be associated with an anonymous type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package org.example.customer;    import javax.xml.bind.annotation.\*;    @XmlAccessorType(XmlAccessType.FIELD)  @XmlType(name = "", propOrder = {"billingAddress","shippingAddress"})  @XmlRootElement(name = "customer")  public class Customer {        @XmlElement(name = "billing-address", required = true)      protected AddressType billingAddress;        @XmlElement(name = "shipping-address", required = true)      protected AddressType shippingAddress;        public AddressType getBillingAddress() {          return billingAddress;      }        public void setBillingAddress(AddressType value) {          this.billingAddress = value;      }        public AddressType getShippingAddress() {          return shippingAddress;      }        public void setShippingAddress(AddressType value) {          this.shippingAddress = value;      }    } |

**AddressType**   
  
Again because JAXB model classes correspond to complex types, a class is generated for the *address-type* complex type.  Since multiple root level elements could exist for this named complex type, it is not annotated with *@XmlRootElement*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package org.example.customer;    import javax.xml.bind.annotation.\*;    @XmlAccessorType(XmlAccessType.FIELD)  @XmlType(name = "address-type", propOrder = {"street"})  public class AddressType {        @XmlElement(required = true)      protected String street;        public String getStreet() {          return street;      }        public void setStreet(String value) {          this.street = value;      }    } |

**ObjectFactory**  
  
The *@XmlElementDecl* annotation is used to represent root elements that correspond to named complex types.  It is placed on a factory method in a class annotated with *@XmlRegistry* (when generated from an XML schema this class is always called *ObjectFactory*).  The factory method returns the domain object wrapped in an instance of *JAXBElement.*  The *JAXBElement* has a *QName* that represents the elements name and namespace URI.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | package org.example.customer;    import javax.xml.bind.JAXBElement;  import javax.xml.bind.annotation.\*;  import javax.xml.namespace.QName;    @XmlRegistry  public class ObjectFactory {        private final static QName \_BillingAddress\_QNAME = new QName("<http://www.example.org/customer>", "billing-address");      private final static QName \_ShippingAddress\_QNAME = new QName("<http://www.example.org/customer>", "shipping-address");        public ObjectFactory() {      }        public Customer createCustomer() {          return new Customer();      }        public AddressType createAddressType() {          return new AddressType();      }        @XmlElementDecl(namespace = "<http://www.example.org/customer>", name = "billing-address")      public JAXBElement<AddressType> createBillingAddress(AddressType value) {          return new JAXBElement<AddressType>(\_BillingAddress\_QNAME, AddressType.class, null, value);      }        @XmlElementDecl(namespace = "<http://www.example.org/customer>", name = "shipping-address")      public JAXBElement<AddressType> createShippingAddress(AddressType value) {          return new JAXBElement<AddressType>(\_ShippingAddress\_QNAME, AddressType.class, null, value);      }    } |

**package-info**  
  
The *package-info* class is used to specify the namespace mapping (see [JAXB & Namespaces](http://blog.bdoughan.com/2010/08/jaxb-namespaces.html)).

|  |  |
| --- | --- |
| 1  2  3  4 | @XmlSchema(namespace = "<http://www.example.org/customer>", elementFormDefault = XmlNsForm.QUALIFIED)  package org.example.customer;    import javax.xml.bind.annotation.\*; |

**Unmarshal Operation**   
  
Now we look at the impact of the type of root element when unmarshalling XML.  
  
**customer.xml**   
  
Below is a sample XML document with *customer* as the root element.  Remember the *customer* element had an anonymous complex type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?xml version="1.0" encoding="UTF-8"?>  <customer xmlns="<http://www.example.org/customer>">      <billing-address>          <street>1 Any Street</street>      </billing-address>      <shipping-address>          <street>2 Another Road</street>      </shipping-address>  </customer> |

**shipping.xml**   
  
Here is a sample XML document with *shipping-address* as the root element.  The *shipping-address* element had a named complex type.

|  |  |
| --- | --- |
| 1  2  3  4 | <?xml version="1.0" encoding="UTF-8"?>  <shipping-address xmlns="<http://www.example.org/customer>">      <street>2 Another Road</street>  </shipping-address> |

**Unmarshal Demo**

When unmarshalling XML that corresponds to a class annotated with *@XmlRootElement* you get an instance of the domain object.  But when unmarshalling XML that corresponds to a class annotated with *@XmlElementDecl* you get the domain object wrapped in an instance of J*AXBElement*.   In this example you may need to use the *QName* from the *JAXBElement* to determine if you unmarshalled a billing or shipping address.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package org.example.customer;    import java.io.File;  import javax.xml.bind.\*;    public class UnmarshalDemo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance("org.example.customer");          Unmarshaller unmarshaller = jc.createUnmarshaller();            // Unmarshal Customer          File customerXML = new File("src/org/example/customer/customer.xml");          Customer customer = (Customer) unmarshaller.unmarshal(customerXML);            // Unmarshal Shipping Address          File shippingXML = new File("src/org/example/customer/shipping.xml");          JAXBElement<AddressType> je = (JAXBElement<AddressType>) unmarshaller.unmarshal(shippingXML);          AddressType shipping = je.getValue();      }    } |

**Unmarshal Demo - JAXBIntrospector**

If you don't want to deal with remembering whether the result of the *unmarshal* operation will be a domain object or *JAXBElement*, then you can use the *JAXBIntrospector.getValue(Object)* method to always get the domain object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package org.example.customer;    import java.io.File;  import javax.xml.bind.\*;    public class JAXBIntrospectorDemo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance("org.example.customer");          Unmarshaller unmarshaller = jc.createUnmarshaller();            // Unmarshal Customer          File customerXML = new File("src/org/example/customer/customer.xml");          Customer customer = (Customer) JAXBIntrospector.getValue(unmarshaller                  .unmarshal(customerXML));            // Unmarshal Shipping Address          File shippingXML = new File("src/org/example/customer/shipping.xml");          AddressType shipping = (AddressType) JAXBIntrospector                  .getValue(unmarshaller.unmarshal(shippingXML));      }    } |

**Marshal Operation**

You can directly marshal an object annotated with *@XmlRootElement* to XML.  Classes corresponding to *@XmlElementDecl* annotations must first be wrapped in an instance of *JAXBElement*.  The factory method you you annotated with *@XmlElementDecl* is the easiest way to do this.  The factory method is in the *ObjectFactory* class if you generated your model from an XML schema.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package org.example.customer;    import javax.xml.bind.\*;    public class MarshalDemo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance("org.example.customer");          Marshaller marshaller = jc.createMarshaller();          marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);            // Create Domain Objects          AddressType billingAddress = new AddressType();          billingAddress.setStreet("1 Any Street");          Customer customer = new Customer();          customer.setBillingAddress(billingAddress);            // Marshal Customer          marshaller.marshal(customer, System.out);            // Marshal Billing Address          ObjectFactory objectFactory = new ObjectFactory();          JAXBElement<AddressType> je =  objectFactory.createBillingAddress(billingAddress);          marshaller.marshal(je, System.out);      }    } |

**Output**  
  
Below is the output from running the demo code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <customer xmlns="<http://www.example.org/customer>">      <billing-address>          <street>1 Any Street</street>      </billing-address>  </customer>  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <billing-address xmlns="<http://www.example.org/customer>">      <street>1 Any Street</street>  </billing-address> |

Posted by Blaise Doughan at [6:35 AM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html" \o "permanent link) [http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=1710464868724994296&postID=7980815110428885997)

[Email This](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=7980815110428885997&target=email" \t "_blank" \o "Email This)[BlogThis!](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=7980815110428885997&target=blog" \t "_blank" \o "BlogThis!)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=7980815110428885997&target=twitter" \t "_blank" \o "Share to Twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=7980815110428885997&target=facebook" \t "_blank" \o "Share to Facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=1710464868724994296&postID=7980815110428885997&target=pinterest" \t "_blank" \o "Share to Pinterest)

Labels: [JAXB](http://blog.bdoughan.com/search/label/JAXB), [JAXBIntrospector](http://blog.bdoughan.com/search/label/JAXBIntrospector), [XmlElementDecl](http://blog.bdoughan.com/search/label/XmlElementDecl), [XmlRootElement](http://blog.bdoughan.com/search/label/XmlRootElement)

#### 20 comments:

1. https://img1.blogblog.com/img/anon36.png

Anonymous[August 9, 2012 at 1:18 PM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1344532729316" \l "c4492292346512898937)

Is it possible to marshal an AddressType if you don't know if it is of type billing-address or shipping-address?

[Reply](javascript:;)

[Replies](javascript:;)

* 1. http://lh5.googleusercontent.com/-vqbzMZDtO4s/AAAAAAAAAAI/AAAAAAAAAGw/DmS7X-NuOh4/s35-c/photo.jpg

[*Blaise Doughan*](https://www.blogger.com/profile/02016807982653343777)[August 12, 2012 at 9:25 PM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1344821113804#c3292701771538120361)

Since AddressType is not annotated with @XmlRootElement you will need to wrap it in an instance of JAXBElement. The QName does not need to be billing-address or shipping-address, you can have it be whatever you like.  
  
-Blaise

* 1. https://img1.blogblog.com/img/anon36.png

[*Roger Feese*](http://directory.unl.edu/%3Fuid%3Drfeese2)[August 13, 2012 at 11:39 AM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1344872360817#c7270426520779479239)

Hello Blaise, I'm the Anonymous poster. I really appreciate this article and your response to my question.  
  
I'm dealing with some legacy systems with perhaps awkward XML schemas. Using your example, what if your Customer had a List of AddressTypes rather than specific Billing and Shipping address fields. In other words, I have a list of addresses and don't know until run time if they are shipping or billing addresses. Would this be possible and how would you get it to Marshal correctly?:  
  
  
protected List <AddressType> address;  
  
  
So, here is how the XML should look:  
  
<customer...>  
...  
  <billing-address>...</billing-address>  
  <shipping-address>...</shipping-address>  
  <shipping-address>...</shipping-address>  
  <billing-address>...</billing-address>  
...  
</customer>

* 1. http://lh5.googleusercontent.com/-vqbzMZDtO4s/AAAAAAAAAAI/AAAAAAAAAGw/DmS7X-NuOh4/s35-c/photo.jpg

[*Blaise Doughan*](https://www.blogger.com/profile/02016807982653343777)[August 13, 2012 at 1:55 PM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1344880515593#c6787320039035980777)

Hi Roger,  
  
What bit of info indicates whether it is a billing-address or shpping-address? Is it a value on the instance?  
  
-Blaise

* 1. https://img1.blogblog.com/img/anon36.png

[*Roger Feese*](http://directory.unl.edu/%3Fuid%3Drfeese2)[August 13, 2012 at 2:39 PM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1344883198234#c1950819813355848433)

The root element of the AdressType itself defines the address type in XML, as in your example.   
  
For a while, I was trying to figure out if I could embed the type in a transient field and somehow tell the marshaller to use that information, but I didn't see a way to do that.  
  
I'm considering the following solution: Allocate a variable for each type of address (as you did, but make them lists). This should work (unfortunately I have a few more "address types" in the system I'm working on).  
  
@XmlElement(name = "shipping-address")  
List<ShippingAddress> shippingAddress;  
  
@XmlElement(name = "billing-address")  
List<BillingAddress> billingAddress;

* 1. http://lh5.googleusercontent.com/-vqbzMZDtO4s/AAAAAAAAAAI/AAAAAAAAAGw/DmS7X-NuOh4/s35-c/photo.jpg

[*Blaise Doughan*](https://www.blogger.com/profile/02016807982653343777)[August 16, 2012 at 6:34 AM](http://blog.bdoughan.com/2012/07/jaxb-and-root-elements.html?showComment=1345113297927#c8310486596862828397)

Hi Roger,  
  
I wrote up the following based on our email conversation:  
- [Removing JAXBElement From Your Domain Model](http://blog.bdoughan.com/2012/08/removing-jaxbelement-from-your-domain.html)  
  
-Blaise

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://blog.bdoughan.com/2012/08/jaxbs-xmltransient-and-property-order.html>

### JAXB's @XmlTransient and Property Order

In previous articles I wrote about how the *@XmlTransient* annotation can be used at the type level to have a [class excluded from the inheritance hierarchy](http://blog.bdoughan.com/2011/06/ignoring-inheritance-with-xmltransient.html), or at the field/property level to [unmap a field/property](http://blog.bdoughan.com/2012/04/jaxb-and-unmapped-properties.html).  In this article I'll demonstrate how doing this impacts the *propOrder* setting on the *@XmlType* annotation.   
  
**Java Model**

**Base**

This class will serve as the root of the inheritance hierarchy.  This will be a mapped class, and as such there is nothing special we need to do, to make this happen.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package blog.proporder.xmltransient;    public abstract class Base {        private int id;        public int getId() {          return id;      }        public void setId(int id) {          this.id = id;      }    } |

**Person**  
  
To exclude a class from being mapped as part of the inheritance hierarchy you simply need to annotate it with *@XmlTransient*.  Any super classes of this class that are not annotated with *@XmlTransient* (i.e. *Base*) will still be mapped.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package blog.proporder.xmltransient;    import javax.xml.bind.annotation.XmlTransient;    @XmlTransient  public class Person extends Base {        private String name;        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }    } |

**Customer**  
  
Since the parent class (*Person*) has been marked *@XmlTransient* the name property will be treated as part of the *Customer* class and can be included in the *propOrder*.  The *Customer* class also extends *Base* which was not marked *@XmlTransient* so the *id* property can not be specified in the *propOrder*.  The *propOrder* setting must not include a field/property that has been annotated with *@XmlTransient*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package blog.proporder.xmltransient;    import java.util.List;  import javax.xml.bind.annotation.\*;    @XmlRootElement  @XmlType(propOrder = { "phoneNumbers", "name"})  public class Customer extends Person {        private String password;      private List<String> phoneNumbers;        @XmlTransient      public String getPassword() {          return password;      }        public void setPassword(String password) {          this.password = password;      }        @XmlElement(name = "phone-number")      public List<String> getPhoneNumbers() {          return phoneNumbers;      }        public void setPhoneNumbers(List<String> phoneNumbers) {          this.phoneNumbers = phoneNumbers;      }      } |

**Demo Code**

Below is the demo code that can be used to run this example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package blog.proporder.xmltransient;    import java.io.File;  import javax.xml.bind.\*;    public class Demo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance(Customer.class);            Unmarshaller unmarshaller = jc.createUnmarshaller();          File xml = new File("input.xml");          Customer customer = (Customer) unmarshaller.unmarshal(xml);            Marshaller marshaller = jc.createMarshaller();          marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);          marshaller.marshal(customer, System.out);      }    } |

**XML (input.xml/Output)**

Below is the input to, and output from running the demo code.  Note how the properties of classes marked with @XmlTransient are included, but properties marked *@XmlTransient* are not.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8"?>  <customer>     <id>123</id>     <phone-number>555-1111</phone-number>     <phone-number>555-2222</phone-number>     <name>Jane Doe</name>  </customer> |

**Further Reading**

If you enjoyed this post you may also be interested in:

* [JAXB and Unmapped Properties](http://blog.bdoughan.com/2012/04/jaxb-and-unmapped-properties.html)
* Mapping Inheritance Using Standard JAXB Mechanisms
  + [JAXB and Inheritance - Using the xsi:type Attribute](http://bdoughan.blogspot.com/2010/11/jaxb-and-inheritance-using-xsitype.html)
  + [JAXB and Inheritance - Using Substitution Groups](http://bdoughan.blogspot.com/2010/11/jaxb-and-inheritance-using-substitution.html)
  + [JAXB and Inheritance - Using XmlAdapter](http://blog.bdoughan.com/2012/01/jaxb-and-inhertiance-using-xmladapter.html)
* Mapping Inheritance Using MOXy Extensions
  + [JAXB and Inheritance - MOXy Extension @XmlDescriminatorNode/@XmlDescrimintatorValue](http://bdoughan.blogspot.com/2010/11/jaxb-and-inheritance-moxy-extension.html)
  + [JAXB and Inheritance - Using MOXy's @XmlClassExtractor](http://blog.bdoughan.com/2012/02/jaxb-and-inheritance-eclipselink-moxy.html)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://blog.bdoughan.com/2011/06/using-jaxbs-xmlaccessortype-to.html>

# [Java XML and JSON Binding](http://blog.bdoughan.com/)

Object-to-XML and object-to-JSON mapping using JAXB and EclipseLink MOXy.

* [Home](http://blog.bdoughan.com/)
* [Contact Me](http://blog.bdoughan.com/p/contact_01.html)

## June 24, 2011

### Using JAXB's @XmlAccessorType to Configure Field or Property Access

JAXB offers a lot of flexibility when interacting with your object model.  One area is configuring the use of fields or properties to access the data in your domain objects.  This is specified as an *XmlAccessType* (PUBLIC\_MEMBER, PROPERTY, FIELD, or NONE) via the *@XmlAccessorType* annotation.  In this post we'll examine what these options really mean.

**Java Model**

The following will be used for our domain model.  It has been contrived for demonstrating the concepts described in this post, and is not mean to represent best practices for object-to-XML mapping.  We will vary the *XmlAccessType* set on the *@XmlAccessorType* annotation between runs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAttribute;  import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  public class Root {        private String fieldA;        @XmlAttribute      private String fieldB;        public String fieldC;        @XmlAttribute      public String getPropertyA() {          return fieldA;      }        public void setPropertyA(String a) {          this.fieldA = a;      }        public String getPropertyB() {          return fieldB;      }        public void setPropertyB(String b) {          this.fieldB = b;      }    } |

**Demo Code**

The following demo code will be used for this post.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package blog.xmlaccessortype;    import javax.xml.bind.JAXBContext;  import javax.xml.bind.Marshaller;    public class Demo {        public static void main(String[] args) throws Exception {          JAXBContext jc = JAXBContext.newInstance(Root.class);            Root root = new Root();          root.setA("a");          root.setB("b");          root.fieldC = "c";            Marshaller marshaller = jc.createMarshaller();          marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);          marshaller.marshal(root, System.out);      }    } |

**XmlAccessType.PUBLIC\_MEMBER**

PUBLIC\_MEMBER is the default access type in JAXB.  It means that a JAXB implementation will generate bindings for:

* public fields
* annotated fields
* properties

**When to Use**  
  
This access type will support most of your use cases.  Check out the descriptions of the other access types for use cases where you may consider switching.  
  
**Java Model**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlAttribute;  import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  @XmlAccessorType(XmlAccessType.PUBLIC\_MEMBER)  public class Root {        private String fieldA;        @XmlAttribute      private String fieldB;        public String fieldC;        @XmlAttribute      public String getPropertyA() {          return fieldA;      }        public void setPropertyA(String a) {          this.fieldA = a;      }        public String getPropertyB() {          return fieldB;      }        public void setPropertyB(String b) {          this.fieldB = b;      }    } |

**XML Output**

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?xml version="1.0" encoding="UTF-8"?>  <root fieldB="b" propertyA="a">      <fieldC>c</fieldC>      <propertyB>b</propertyB>  </root> |

**XmlAccessType.PROPERTY**

When the PROPERTY access type is used, JAXB implementations will generate bindings for:

* annotated fields
* properties

**When to Use**  
This access type is very similar to PUBLIC\_MEMBER.  The primary reason to switch to PROPERTY is to guard against any public fields that may be in your domain model.    
  
There are advantages to using PROPERTY access over FIELD access when using your domain objects with JPA implementations that may alter the byte codes of your domain classes:

* Some JPA implementations inject byte code into the properties to trigger "lazy loading".  If you use the FIELD access type your JAXB marshal operation will not bring in this data.
* Some JPA implementations inject fields to support such things as change tracking.  If you use FIELD access you may find your JAXB implementation complaining about fields that you did not explicitly add to your domain model.

**Java Model**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlAttribute;  import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  @XmlAccessorType(XmlAccessType.PROPERTY)  public class Root {        private String fieldA;        @XmlAttribute      private String fieldB;        public String fieldC;        @XmlAttribute      public String getPropertyA() {          return fieldA;      }        public void setPropertyA(String a) {          this.fieldA = a;      }        public String getPropertyB() {          return fieldB;      }        public void setPropertyB(String b) {          this.fieldB = b;      }    } |

**XML Output**

|  |  |
| --- | --- |
| 1  2  3  4 | <?xml version="1.0" encoding="UTF-8"?>  <root fieldB="b" propertyA="a">      <propertyB>b</propertyB>  </root> |

**XmlAccessType.FIELD**

The use of access type FIELD will cause JAXB implementations to create bindings for:

* fields
* annotated properties

**When to Use**   
  
I tend to use this access type in my examples as it allows me to omit the properties to save space.  In reality this access type is most useful in scenarios where not all of the fields are exposed through properties.  
  
  
**Java Model**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlAttribute;  import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  @XmlAccessorType(XmlAccessType.FIELD)  public class Root {        private String fieldA;        @XmlAttribute      private String fieldB;        public String fieldC;        @XmlAttribute      public String getPropertyA() {          return fieldA;      }        public void setPropertyA(String a) {          this.fieldA = a;      }        public String getPropertyB() {          return fieldB;      }        public void setPropertyB(String b) {          this.fieldB = b;      }    } |

**XML Output**

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?xml version="1.0" encoding="UTF-8"?>  <root fieldB="b" propertyA="a">      <fieldA>a</fieldA>      <fieldC>c</fieldC>  </root> |

**XmlAccessType.NONE**

When access type NONE is used JAXB will create bindings for:

* annotated fields
* annotated properties

**When to Use**

This access type is useful when your domain object has many fields/properties and you only want to map a few to XML.   Handling this use case with the other access types would involve using *@XmlTransient* to exclude fields/properties from the XML mapping.   
  
**Java Model**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlAttribute;  import javax.xml.bind.annotation.XmlRootElement;    @XmlRootElement  @XmlAccessorType(XmlAccessType.NONE)  public class Root {        private String fieldA;        @XmlAttribute      private String fieldB;        public String fieldC;        @XmlAttribute      public String getPropertyA() {          return fieldA;      }        public void setPropertyA(String a) {          this.fieldA = a;      }        public String getPropertyB() {          return fieldB;      }        public void setPropertyB(String b) {          this.fieldB = b;      }    } |

**XML Output**

|  |  |
| --- | --- |
| 1  2 | <?xml version="1.0" encoding="UTF-8"?>  <root fieldB="b" propertyA="a"/> |

**Access Type and Inheritance**

The access type specified on a class will be inherited by all its subclasses.  @XmlAccessorType can be used to override the access type that was configured on the parent class.

**Overriding the Default Access Type**

The *@XmlAccessorType* annotation can be used at the type level to configure a single class, or at the package level to change the default access type for all classes within that package (individual classes can override the access type).  Package level annotations can be supplied through a *package-info* class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @XmlAccessorType(XmlAccessType.FIELD)  package blog.xmlaccessortype;    import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType; |