JAXB TUTORIAL FOR JAVA XML BINDING

THE ULTIMATE GUIDE

DANI BUIZA

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

JAXB Tutorial

Contents

Preface

Java offers several options for handling XML structures and files. One of the most common and used ones is JAXB. JAXB stands for Java Architecture for XML Binding. It offers the possibility to convert Java objects into XML structures and the other way around. JAXB comes with the JRE standard bundle since the first versions of the JRE 1.6.

The first specification of JAXB was done in March 2003 and the work process is tracked in the Java Specification Request 31: https://jcp.org/en/jsr/detail?id=31. In this specification request you can find a lot of information regarding the long life of JAXB and all the improvements that have been made.

As already mentioned, JAXB is included in the JRE bundle since the update 1.6. Before that it was necessary to include their libraries in the specific Java project in order to be able to use it.

Before JAXB was available (long time ago), the way Java had to handle XML documents was the DOM: http://www.w3.org/-DOM/. This was not a very good approach because there was almost not abstraction from XML nodes into Java objects and all value types were inferred as Strings. JAXB provides several benefits like Object oriented approach related to XML nodes and attributes, typed values, annotations and may others that we are going to explain in this article.

All examples in this tutorial have been implementing using the following software versions: JRE 1.8.0 for 32b. The IDE used is Eclipse SDK Version: Luna (4.4). However any other Java versions containing the JAXB API and IDEs should work perfectly fine since all the code is standard Java 8 one.

About the Author

Daniel Gutierrez Diez holds a Master in Computer Science Engineering from the University of Oviedo (Spain) and a Post Grade as Specialist in Foreign Trade from the UNED (Spain). Daniel has been working for different clients and companies in several Java projects as programmer, designer, trainer, consultant and technical lead.

Chapter 1

Mapping

Java objects can be bonded to XML structures by using certain annotations and following specific rules. This is what we call mapping. In this tutorial we are going to explain the following points, providing examples, resources and extra information:

- We are going to show some examples about how to convert Java objects into XML structures, this is called marshaling. We will show how to handle primitive types, collections and more complex types using adapters.
- We will also explain how to do the complementary operation, called un-marshaling, i.e. converting XML files into Java objects.
- All this is done using Java annotations, we will list and explain the most important annotations used within JAXB.
- We will also provide an introduction to XSDs (XML Schemas) which are used for validation and are a powerful tool supported by JAXB. We will see how XSDs can be used for marshalling as well.
- Finally, we will list several tools that can be used in combination with JAXB that help programmers in different ways.

Chapter 2

Marshal

In this chapter, we are going to see how to convert Java objects into XML files and what should be taken into consideration while doing this operation. This is commonly called marshaling.

First of all we indicate JAXB what java elements from our business model correspond to what XML nodes.

```
@XmlType( propOrder = { "name", "capital", "foundation", "continent" , "population"} )
@XmlRootElement( name = "Country" )
public class Country
{

    @XmlElement (name = "Country_Population")
    public void setPopulation( int population )
    {
        this.population = population;
    }


    @XmlElement( name = "Country_Name" )
    public void setName( String name )
    {
        this.name = name;
    }

    @XmlElement( name = "Country_Capital" )
    public void setCapital( String capital )
    {
        this.capital = capital;
    }
    @XmlAttribute( name = "importance", required = true )
    public void setImportance( int importance )
    {
        this.importance = importance;
    }
...
```

The class above contains some JAXB annotations that allow us to indicate what XML nodes we are going to generate. For this purpose we are using the annotations

- `@XmlRootElement` as root element.
- `@XmlElement` in combination with setter methods.
- `@XmlAttribute` to pass attributes to the XML nodes. These attributes can have properties like to be required or not.
- `@XmlType` to indicate special options like to order of appearance in the XML.

We will explain these annotations and others in more in detail in next chapters. For the moment, I just want to mention them here.

Next step is to generate XML files from Java objects. For this purpose we create a simple test program using the `JAXBContext` and its marshaling functionalities:

```
Country spain = new Country();
spain.setName( "Spain" );
spain.setCapital( "Madrid" );
spain.setContinent( "Europe" );
spain.setImportance( 1 );
spain.setFoundation( LocalDate.of( 1469, 10, 19 ) );
spain.setPopulation( 45000000 );

/* init jaxb marshaler */
JAXBContext jaxbContext = JAXBContext.newInstance( Country.class );
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

/* set this flag to true to format the output */
jaxbMarshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );

/* marshaling of java objects in xml (output to file and standard output) */
jaxbMarshaller.marshal( spain, new File( "country.xml" ) );
jaxbMarshaller.marshal( spain, System.out );
```

Basically, the most important part here is the use of the class `javax.xml.bind.JAXBContext`. This class provides a framework for validating, marshaling and un-marshaling XML into (and from) Java objects and it is the entry point to the JAXB API. More information about this class can be found here: https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/-JAXBContext.html.

In our small example, we are just using this class in order to create a JAXB context that allows us to marshal objects of the type passed as parameter. This is done exactly here:

```
JAXBContext jaxbContext = JAXBContext.newInstance( Country.class );
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

The result of the main program would be:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Country importance="1">
 <Country_Name>Spain</Country_Name>
 <Country_Capital>Madrid</Country_Capital>
 <Country_Foundation_Date></Country_Foundation_Date>
 <Country_Continent>Europe</Country_Continent>
 <Country_Population>45000000</Country_Population>
</Country>
```

In the JAXB application shown above we are just converting simple types (Strings and Integers) present in a container class into XML nodes. We can see for example that the date based attributes like the foundation date are missing, we will explain later how to solve this problem for complex types.

This looks easy. JAXB supports all kinds of Java objects like other primitive types, collections, date ranges, etc.

If we would like to map a list of elements into an XML, we can write:

```
@XmlRootElement( name = "Countries" )
public class Countries
{
    List countries;

    /**
     * element that is going to be marshaled in the xml
     */
```

```
    @XmlElement( name = "Country" )
    public void setCountries( List countries )
    {
        this.countries = countries;
    }
```

We can see in the snippet above that a new container class is needed in order to indicate JAXB that a class containing a list is present. The result a similar program as the one shown above would be:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Countries>
        <Country importance="1">
                <Country_Name>Spain</Country_Name>
                <Country_Capital>Madrid</Country_Capital>
                <Country_Foundation_Date></Country_Foundation_Date>
                <Country_Continent>Europe</Country_Continent>
                <Country_Population>0</Country_Population>
        </Country>
        <Country importance="0">
                <Country_Name>USA</Country_Name>
                <Country_Capital>Washington</Country_Capital>
                <Country_Foundation_Date></Country_Foundation_Date>
                <Country_Continent>America</Country_Continent>
                <Country_Population>0</Country_Population>
        </Country>
</Countries>
```

There are several options when handling collections:

- We can use wrapper annotations: The annotation `javax.xml.bind.annotation.XMLElementWrapper` offers the possibility to create a wrapper around an XML representation. This wrapper can contain a collection of elements.
- We can use collection based annotations like `javax.xml.bind.annotation.XMLElements` or `javax.xml.bind.annotation.XMLRefs` that offer collections functionalities but less flexibility.
- We can use a container for the collection. A container class (`Countries` in our case) which has a member of the type `java.util.Collection` (`Country` in our case). This is my favorite approach since it offers more flexibility. This is the approach shown before.

Chapter 3

Un-marshal

In this chapter, we are going to see how to do the complementary operation: un-marshal XML files into java objects and what should be taken into consideration while doing this operation.

First of all we create the XML structure that we want to un-marshal:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Countries>
        <Country>
                <Country_Name>Spain</Country_Name>
                <Country_Capital>Madrid</Country_Capital>
                <Country_Continent>Europe</Country_Continent>
        </Country>
        <Country>
                <Country_Name>USA</Country_Name>
                <Country_Capital>Washington</Country_Capital>
                <Country_Continent>America</Country_Continent>
        </Country>
        <Country>
                <Country_Name>Japan</Country_Name>
                <Country_Capital>Tokyo</Country_Capital>
                <Country_Continent>Asia</Country_Continent>
        </Country>
</Countries>
```

Good to mention that we deleted the foundation dates. If these would be present we would get an error since JAXB do not know how to un-marshal them. We will see afterwards how to solve this problem.

After that we can create a program that "reads" this XML file and parses it into the proper Java objects:

```
File file = new File( "countries.xml" );
JAXBContext jaxbContext = JAXBContext.newInstance( Countries.class );

/**
* the only difference with the marshaling operation is here
*/
Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
Countries countres = (Countries)jaxbUnmarshaller.unmarshal( file );
System.out.println( countres );
```

We can see that the code above does not differ that much from the one shown in the last chapter related to the marshal operation. We also use the class `javax.xml.bind.JAXBContext` but in this case the method used is the createUnmarshaller() one, which takes care of providing an object of the type `javax.xml.bind.Unmarshaller`. This object is the responsible of un-marshaling the XML afterwards.

This program uses the annotated class `Country`. It looks like:

```
@XmlType( propOrder = { "name", "capital", "foundation", "continent" , "population"} )
@XmlRootElement( name = "Country" )
public class Country
{

    @XmlElement (name = "Country_Population")
    public void setPopulation( int population )
    {
        this.population = population;
    }


    @XmlElement( name = "Country_Name" )
    public void setName( String name )
    {
        this.name = name;
    }

    @XmlElement( name = "Country_Capital" )
    public void setCapital( String capital )
    {
        this.capital = capital;
    }
    @XmlAttribute( name = "importance", required = true )
    public void setImportance( int importance )
    {
        this.importance = importance;
    }
...
```

We cannot appreciate too many differences to the classes used in the last chapter, same annotations are used.

This is the output produced by the program in the standard console:

```
Name: Spain
Capital: Madrid
Europe
Name: USA
Capital: Washington
America
Name: Japan
Capital: Tokyo
Asia
```

The same explained for the marshalling operation applies here. It is possible to un-marshal objects or other primitive types like numeric ones; it is also possible to un-marshal collection based elements like lists or sets and the possibilities are the same.

As we can see in the results provided above and in the last chapter, the attribute of the type `java.time.LocalDate` has not been converted. This happens because JAXB does not know how to do it. It is also possible to handle complex types like this one using adapters; we are going to see this in the next chapter.

Chapter 4

Adapters

When handling complex types that may not be directly available in JAXB we need to write an adapter to indicate JAXB how to manage the specific type.

We are going to explain this by using an example of marshaling (and un-marshaling) for an element of the type `java.time.LocalDate`.

```
public class DateAdapter extends XmlAdapter
{

    public LocalDate unmarshal( String date ) throws Exception
    {
        return LocalDate.parse( date );
    }

    public String marshal( LocalDate date ) throws Exception
    {
        return date.toString();
    }

}
```

The piece of code above shows the implementation of the marshal and un-marshal methods of the interface `javax.xml.bind.annotation.adapters.XmlAdapter` with the proper types and results and afterwards, we indicate JAXB where to use it using the annotation `@XmlJavaTypeAdapter`:

```
 ...
@XmlElement( name = "Country_Foundation_Date" )
@XmlJavaTypeAdapter( DateAdapter.class )
public void setFoundation( LocalDate foundation )
{
    this.foundation = foundation;
}
...
```

The resulting XML for the first program shown in this tutorial would be something like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Country importance="1">
        <Country_Name>Spain</Country_Name>
        <Country_Capital>Madrid</Country_Capital>
        <Country_Foundation_Date>1469-10-19</Country_Foundation_Date>
        <Country_Continent>Europe</Country_Continent>
        <Country_Population>45000000</Country_Population>
</Country>
```

This can be applied to any other complex type not directly supported by JAXB that we would like to have in our XML structures. We just need to implement the interface `javax.xml.bind.annotation.adapters.XmlAdapter` and implement their methods `javax.xml.bind.annotation.adapters.XmlAdapter.unmarshal(ValueType)` and `javax.xml.bind.annotation.adapters.XmlAdapter.marshal(BoundType)`.

Chapter 5

XSDs

XSD is an XML schema. It contains information about XML files and structures with rules and constraints that should be followed. These rules can apply to the structure of the XML and also to the content. Rules can be concatenated and very complex rules can be created using all kind of structures, in this article we are going to show the main concepts about how to use XSDs for validation and marshaling purposes.

Here is an example of an XML Schema that can be used for the class `Country` used in this tutorial:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Country">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="Country_Name" type="xs:string" />
                        <xs:element name="Country_Capital" type="xs:string" />
                        <xs:element name="Country_Foundation_Date" type="xs:string" />
                        <xs:element name="Country_Continent" type="xs:string" />
                        <xs:element name="Country_Population" type="xs:integer" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
</xs:schema>
```

5.1 Validation using XSDs

XSDs can be used for XML validation. JAXB uses XSDs for validating XML and to assure that the objects and XML structures follow the set of expected rules. In order to validate an object against an XSD we need to create the XSD first, as an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:simpleType name="continentType">
                <xs:restriction base="xs:string">
                        <xs:pattern value="Asia|Europe|America|Africa|Oceania"/>
                </xs:restriction>
        </xs:simpleType>

<xs:element name="Country">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="Country_Name" type="xs:string" minOccurs='1' />
                        <xs:element name="Country_Capital" type="xs:string" minOccurs='1' / ↩
                            >
```

```
                        <xs:element name="Country_Foundation_Date" type="xs:string"  ↩
                            minOccurs='1' />
                        <xs:element name="Country_Continent" type="continentType" minOccurs ↩
                            ='1' />
                        <xs:element name="Country_Population" type="xs:integer" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
</xs:schema>
```

We can use this in our program my indicating JAXB what XSD we want to use in our code. We can do this by creating an instance of the class `javax.xml.validation.Schema` and initializing its validation in the following way:

```
/**
* schema is created
*/
SchemaFactory sf = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI );
Schema schema = sf.newSchema( new File( "countries_validation.xsd" ) );
```

The validation expects an error handler that can take care of errors. This is done by implementing the interface `org.xml.sax.ErrorHandler` and its error methods:

```
public class MyErrorHandler implements ErrorHandler
{

    @Override
    public void warning( SAXParseException exception ) throws SAXException
    {
        throw exception;

    }
   ...
```

After that, the validation can be used to validate instances of the class `javax.xml.bind.util.JAXBSource`:

```
/**
 * context is created and used to create sources for each country
 */
JAXBContext jaxbContext = JAXBContext.newInstance( Country.class );
JAXBSource sourceSpain = new JAXBSource( jaxbContext, spain );
JAXBSource sourceAustralia = new JAXBSource( jaxbContext, australia );
```

Here is the complete program:

```
/**
 * error will be thrown because continent is mandatory
 */
Country spain = new Country();
spain.setName( "Spain" );
spain.setCapital( "Madrid" );
spain.setFoundation( LocalDate.of( 1469, 10, 19 ) );
spain.setImportance( 1 );


/**
 * ok
 */
Country australia = new Country();
australia.setName( "Australia" );
australia.setCapital( "Camberra" );
australia.setFoundation( LocalDate.of( 1788, 01, 26 ) );
australia.setContinent( "Oceania" );
```

```
australia.setImportance( 1 );


/**
 * schema is created
 */
SchemaFactory sf = SchemaFactory.newInstance( XMLConstants.W3C_XML_SCHEMA_NS_URI );
Schema schema = sf.newSchema( new File( "countries_validation.xsd" ) );

/**
 * context is created and used to create sources for each country
 */
JAXBContext jaxbContext = JAXBContext.newInstance( Country.class );
JAXBSource sourceSpain = new JAXBSource( jaxbContext, spain );
JAXBSource sourceAustralia = new JAXBSource( jaxbContext, australia );

/**
 * validator is initialized
 */
Validator validator = schema.newValidator();
validator.setErrorHandler( new MyErrorHandler() );

//validator is used
try
{
        validator.validate( sourceSpain );
        System.out.println( "spain has no problems" );
}
catch( SAXException ex )
{
        ex.printStackTrace();
        System.out.println( "spain has problems" );
}
try
{
        validator.validate( sourceAustralia );
        System.out.println( "australia has no problems" );
}
catch( SAXException ex )
{
        ex.printStackTrace();
        System.out.println( "australia has problems" );
}
```

and its output:

```
org.xml.sax.SAXParseException
        at javax.xml.bind.util.JAXBSource$1.parse(JAXBSource.java:248)
        at javax.xml.bind.util.JAXBSource$1.parse(JAXBSource.java:232)
        at com.sun.org.apache.xerces.internal.jaxp.validation.ValidatorHandlerImpl.validate ↩
            (
...
spain has problems
australia has no problems
```

We can see that Australia has no problems but Spain does. . .

5.2 Marshaling using XSDs

XSDs are used also for binding and generating java classes from XML files and vice versa. We are going to see here how to use it with a marshaling example.

Using the same XML schema shown before, we are going to wriite a program that initializes a `javax.xml.validation.Schema` using the given XSD and a `javax.xml.bind.JAXBContext` for the classes that we want to marshal (`Country`). This program will use a `javax.xml.bind.Marshaller` in order to perform the needed operations:

```
/**
* validation will fail because continent is mandatory
*/
Country spain = new Country();
spain.setName( "Spain" );
spain.setCapital( "Madrid" );
spain.setFoundation( LocalDate.of( 1469, 10, 19 ) );

SchemaFactory sf = SchemaFactory.newInstance( XMLConstants.W3C_XML_SCHEMA_NS_URI );
Schema schema = sf.newSchema( new File( "countries_validation.xsd" ) );

JAXBContext jaxbContext = JAXBContext.newInstance( Country.class );

Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, true );
marshaller.setSchema( schema );
//the schema uses a validation handler for validate the objects
marshaller.setEventHandler( new MyValidationEventHandler() );
try
{
        marshaller.marshal( spain, System.out );
}
catch( JAXBException ex )
{
        ex.printStackTrace();
}

/**
* continent is wrong and validation will fail
*/
Country australia = new Country();
australia.setName( "Australia" );
australia.setCapital( "Camberra" );
australia.setFoundation( LocalDate.of( 1788, 01, 26 ) );
australia.setContinent( "Australia" );

try
{
        marshaller.marshal( australia, System.out );
}
catch( JAXBException ex )
{
        ex.printStackTrace();
}

/**
* finally everything ok
*/
australia = new Country();
australia.setName( "Australia" );
australia.setCapital( "Camberra" );
australia.setFoundation( LocalDate.of( 1788, 01, 26 ) );
australia.setContinent( "Oceania" );
```

```
try
{
        marshaller.marshal( australia, System.out );
}
catch( JAXBException ex )
{
        ex.printStackTrace();
}
```

We were talking before about validation related to XSDs. It is also possible to validate while marshaling objects to XML. If our object does not comply with some of the expected rules, we are going to get some validation errors here as well although we are not validating explicitly. In this case we are not using an error handler implementing `org.xml.sax.ErrorHandler` but a `javax.xml.bind.ValidationEventHandler`:

```
public class MyValidationEventHandler implements ValidationEventHandler
{

    @Override
    public boolean handleEvent( ValidationEvent event )
    {
        System.out.println( "Error catched!!" );
        System.out.println( "event.getSeverity():  " + event.getSeverity() );
        System.out.println( "event:  " + event.getMessage() );
        System.out.println( "event.getLinkedException():  " + event.getLinkedException() );
        //the locator contains much more information like line, column, etc.
        System.out.println( "event.getLocator().getObject():  " + event.getLocator(). ↩
            getObject() );
        return false;
    }

}
```

We can see that the method `javax.xml.bind.ValidationEventHandler.handleEvent(ValidationEvent)` has been implemented. And the output will be:

```
javax.xml.bind.MarshalException
 - with linked exception:
org.xml.sax.SAXParseException; lineNumber: 0; columnNumber: 0; cvc-pattern-valid: Value ' ↩
    Australia' is not facet-valid with respect to pattern 'Asia|Europe|America|Africa| ↩
    Oceania' for type 'continentType'.
        at com.sun.xml.internal.bind.v2.runtime.MarshallerImpl.write(MarshallerImpl.java ↩
            :311)
        at ...
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Country>
<Country_Name>Australia</Country_Name>
<Country_Capital>Camberra</Country_Capital>
<Country_Foundation_Date>1788-01-26</Country_Foundation_Date>
<Country_Continent>Oceania</Country_Continent>
<Country_Population>0</Country_Population>
</Country>
```

Obviously there is much more to explain about XSDs because of all the possible rules constellations and its applications are huge. But this is out of the scope of this tutorial. If you need more information about how to use XML Schemas within JAXB you can visit the following links, I found them really interesting:

- http://www.w3schools.com/schema/schema_example.asp.
- http://blog.bdoughan.com/2010/12/jaxb-and-marshalunmarshal-schema.html .

Chapter 6

Annotations

In this tutorial we have been using several annotations used within JAXB for and XML marshaling and un-marshaling operations. We are going to list here the most important ones:

- `XmlAccessorOrder`: This annotation controls the ordering of fields and properties in a class in which they will appear in the XML. For more information: https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlAccessorOrder.html
- `XmlAccessorType`: Indicates if an element should be serialized or not. It is used in combination with `javax.xml.bind.annotation .XMLAccessType`. For more information: https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlAccessorType.html
- `XmlAnyAttribute`: Maps an element to a map of wildcard attributes. For more information http://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlAnyAttribute.html .
- `XmlAnyElement`: Serves as a fallback for un-marshalling operations when no mapping is predefined. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlAnyElement.html
- `XmlAttribute`: This annotation is one of the basic and most used ones. It maps a Java element (property, attribute, field) to an XML node attribute. In this tutorial has been used in several examples. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlAttribute.html
- `XmlElement`: Maps a Java element to an XML node using the name. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlElement.html
- `XmlElementRef`: Maps a java element to an XML node using its type (different to the last one, where the name is used for mapping). For more information about this annotation https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlElementRef.html
- `XmlElementRefs`: Marks a property that refers to classes with `XmlElement` or `JAXBElement`. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlElementRefs.html
- `XmlElements`: This is a container for multiple `XMLElement` annotations. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlElements.html
- `XmlElementWrapper`: It generates a wrapper around an XML representation, intended to be used with collections, in this tutorial we saw there are different ways to handle collections. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlElementWrapper.html
- `XmlEnum`: Provides mapping for an enum to an XML representation. It works in combination with `XmlEnumValue`. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlEnum.html
- `XmlEnumValue`: Maps an enum constant to an XML element. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlEnumValue.html
- `XmlID`: Maps a property to an XML id. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlID.html

- `XmlList`: Another way of handling lists within JAXB. For more information https://docs.oracle.com/javase/7/docs/api/javax/-xml/bind/annotation/XmlList.html
- `XmlMimeType`: Controls the representation of the annotated property. For more information https://docs.oracle.com/javase/-7/docs/api/javax/xml/bind/annotation/XmlMimeType.html
- `XmlMixed`: The annotated element can contain mixed content. The content can be text, children content or unknown. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlMixed.html
- `XmlRootElement`: This is probably the most used annotation within JAXB. It is used to map a class to an XML element. It is a kind of entry point for every JAXB representation. For more information https://docs.oracle.com/javase/7/docs/api/javax/-xml/bind/annotation/XmlRootElement.html
- `XmlSchema`: Maps a package to an XML namespace. For more information https://docs.oracle.com/javase/7/docs/api/javax/-xml/bind/annotation/XmlSchema.html
- `XmlSchemaType`: Maps a Java type to a simple schema built-in type. For more information https://docs.oracle.com/javase/-7/docs/api/javax/xml/bind/annotation/XmlSchemaType.html
- `XmlSeeAlso`: Indicates JAXB to bind other classes when binding the annotated one. This is needed because Java makes very difficult to list all subclasses for a given class, using this mechanism you can indicate JAXB what subclasses (or other classes) should be bounded when handling a specific one. For more information https://docs.oracle.com/javase/7/docs/api/javax/xml/-bind/annotation/XmlSeeAlso.html
- `XmlType`: Used for mapping a class or an enum to a type in an XML Schema. For more information https://docs.oracle.com/-javase/7/docs/api/javax/xml/bind/annotation/XmlType.html
- `XmlValue`: Enables mapping a class to a XML Schema complex type with a simpleContent or a XML Schema simple type. For more information <a> https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/XmlValue.html

This is a huge list, but these are not all the available JAXB related annotations. For a complete list with all the JAXB available annotations please go to the package summary https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/-package-frame.html.

Chapter 7

Tools

There are many tools that can help programmers while working with JAXB and XML Schemas. We are going to list in this chapter some of them and provide some useful resources:

- `schemagen`: Stands for Schema Generator. It is used for generating JAXB Schemas from Java annotated classes or sources (it also works with byte codes). Can be used directly from the command line or using Ant. For more information visit the Oracle page https://docs.oracle.com/javase/7/docs/technotes/tools/share/schemagen.html or https://jaxb.java.net/2.2.4/docs/-schemagen.html. It is available with the JRE standard bundle and can be launched from the bin directory.
- `xjc`: This is the JAXB Binding Compiler. It is used for generate Java sources (classes) from XML Schemas (XSDs). It can be used with Ant. Performs a validation of the XSD before its execution. For more information about its usage and parameters you should visit https://jaxb.java.net/2.2.4/docs/xjc.html .
- `Jsonix`: Nothing to do with JAXB directly but it is a tool that convert XML to JSON and the other way around. It is very useful when working on JavaScript. Here is the official link http://confluence.highsource.org/display/JSNX/Jsonix .
- `Hyperjaxb3`: It provides relation persistence for JAXB objects. It uses the JAXB annotations to persist the objects in relational databases. It works with Hibernate, probably the most used Java persistence framework. You can find more documentation and sources in the link http://confluence.highsource.org/display/HJ3/Home.
- Maven JAXB2 Plugin: Plugin for Maven that offers all the functionalities from the xjc tool and many more while converting XML Schemas into Java classes. In the following link you can download the sources and consult the documentation https://github.com/highsource/maven-jaxb2-plugin .
- JAXB Eclipse Plugin: There are several Eclipse plugins available (as well for other IDEs like NetBeans or IntelliJ) that allow to compile XML Schemas into Java classes, validate XML schemas or generate Schemas. All of them wrap and use one of the tools mentioned above. Since there is no plugin that can be described as the standard one I leave you to search in the web and pick the one of your choice.

These are not all the tools that work with JAXB and XSDs for converting XML structures into Java classes. There are many more. I just listed here the most important and basic ones that every developer should take into consideration when using JAXB in their applications.

Chapter 8

Best Practices

Although this is not a complete list of best practices and it is a subjective topic I would like to give a couple of preferred usages related to JAXB.

- Try to avoid problematic primitive types like `float`, `decimal` or `negativeInteger`; these do not have a related type in JAXB. In these cases you can probably use other "non problematic" primitive types.
- Use the annotation `@XMLSchemaType` to be more explicit regarding the type that has to be used and do not leave JAXB to decide this point.
- Avoid anonymous types and mixed content.
- JAXB uses the systems default locale and country for generating information and error messages In order to change that you can pass to your application the following JVM parameters: `-Duser.country=US -Duser.language=en -Duser.variant=Traditional_WIN`

Chapter 9

Summary

So that is all. In this tutorial we explained what JAXB is and what it can be used for. JAXB stands for Java Architecture for XML binding and it is a framework used basically for:

- Marshaling Java objects into XML structures.
- Un-marshaling XML files into Java classes.
- These operations can be done using XSDs (XML schemas).
- XSDs are very useful for validation purposes as well.

In plain words, JAXB can be used for converting XMLs into Java classes and vice versa. We explained the main concepts related to all these and we listed the main annotations and classes involved in the JAXB usage.

There are several tools that can be used in combination with JAXB for different purposes, we explained how to use and what can be done with some of them.

Chapter 10

Resources

This tutorial contains deep information about JAXB and its usage for marshaling and un-marshaling XML into Java objects. It also explains how to use XSDs when working with JAXB and some best practices and tricks. But in case you need to go one step forward in your applications programming with JAXB it may be useful to visit following resources:

- http://examples.javacodegeeks.com/core-java/xml/bind/jaxb-unmarshal-example/
- http://examples.javacodegeeks.com/core-java/xml/bind/jaxb-unmarshal-example/
- http://www.oracle.com/technetwork/articles/javase/index-140168.html
- http://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding
- https://jaxb.java.net/
- http://en.wikipedia.org/wiki/XML_schema

Chapter 11

Download

You can download the full source code of this tutorial here: **jaxb_ultimate**.