JAXB Cheatsheet

Rodrigo García Carmona (v1.0.2)

Project setup

We don't need to import any library to use JAXB, since it's included in the JDK.

To marshall (Java2XML) and unmarshall (XML2Java) using JAXB we need to annotate the data classes and write some code to do the actual (un)marshalling.

JAXB Annotations

We must annotate the Java classes that will represent the data contained in the XML documents. These data classes must:

- Implement *Serializable*.
- Have a parameter-less constructor.
- Have a constructor without parameters.
- Have public getters and setters for all annotated attributes.

We must annotate every class that will be part of a XML document as follows:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee implements Serializable {
...
}
```

With *XMLAccessorType* we're indicating that the JAXB will look inside the class for XML annotations, and with FIELD that those will be found in the class attributes (also called fields), instead of in the methods.

Annotating Attributes

Since we chose the FIELD annotation method, we must annotate every class' attribute with one of the following annotations:

- *XmlAttribute*: If the attribute is going to appear as an attribute in the XML.
- *XmlElement*: If the attribute is going to appear as a tag in the XML.
- *XmlTransient*: If the attribute is not going to appear in the XML at all.

In both *XmlAttribute* and *XmlElement* we can specify a particular name by using the *name* property. If we don't, the Java attribute name will be used by default. We can also use the *required* property (set to true or false) to indicate if the attribute or tag is mandatory. By default it's true.

Here's an example of these annotations:

```
@XmlTransient
private Integer id;
@XmlAttribute
private String name;
@XmlElement(name = "Contents", required = "false")
private String content;
@XmlElement(name = "Employee")
private List<Employee> authors;
```

This will produce an XML like this one:

```
<Report name="External Sales Report">
    <Contents>Eh, better than expected!</Contents>
    <Employee>
            ...
    </Employee>
    <Employee>
```

```
            ...
    </Employee>
</Report>
```

It's important to note that the *Employee* class must also be annotated, since it's also a part of the XML document.

Also note that this applies to *Lists* too, like the one we had with *Employee* objects. In this case, each *Employee* will be inside its own *Employee* tag. If we want to surround all of them with a wrapper tag, we must add the *XMLElementWrapper* annotation:

```
@XmlTransient
private Integer id;
@XmlAttribute
private String name;
@XmlElement(name = "Contents")
private String content;
@XmlElement(name = "Employee")
@XmlElementWrapper(name = "Authors")
private List<Employee> authors;
```

Which will produce an XML like this one:

```
<Report name="External Sales Report">
    <Contents>Eh, better than expected!</Contents>
    <Authors>
        <Employee>
            ...
        </Employee>
        <Employee>
            ...
        </Employee>
    </Authors>
</Report>
```

Element order

Optionally, we can use the *propOrder* property of the *XmlType* annotation to specify the children tag ordering in the XML document. Inside *propOrder* we must put the Java attribute names, not the XML element names.

Here's an example:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(propOrder = { "name", "dob", "address", "salary" })
public class Employee implements Serializable {
}
```

Root element

Every XML document must have a root element: the first tag of the document. We must annotate every class that needs to fulfil this role in at least one of our XML documents with the *XMLRootElement* annotation. Like this:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "Employee")
public class Employee implements Serializable {
}
```

Marshalling

To marshall Java objects into a XML document we must follow these steps:

1. Create a *JAXBContext* object.
2. Create a *Marshaller* using the *JAXBContext* object.
3. Call the *marshall* method of the *Marshaller*.

```
// Create the object
Book book = new book();
book.setName(“Dune”);
Book.setAuthor(“Frank Herbert”)
// Create the JAXBContext
JAXBContext jaxbC = JAXBContext.newInstance(Book.class);
// Create the JAXBMarshaller
Marshaller jaxbM = jaxbC.createMarshaller();
// Pretty formatting
jaxbM.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,Boolean.TRUE);
// Write to a file
jaxbM.marshal(book, XMLfile);
// Printout
jaxbM.marshal(book, System.out);
```

Unmarshalling

To unmarshall a XML document into Java objects we must follow these steps:

1. Create a *JAXBContext* object.
2. Create an *Unmarshaller* using the *JAXBContext* object.
3. Call the *unmarshall* method of the *Unmarshaller*.

```
// Create the JAXBContext
JAXBContext jaxbC = JAXBContext.newInstance(Book.class);
// Create the JAXBMarshaller
Unmarshaller jaxbU = jaxbC.createUnmarshaller();
// Create the object by reading from a file
Book book = (Book) jaxbU.unmarshal(XMLfile);
// Printout
System.out.println(book);
```