Inner classes

JiBX fully supports working with inner classes, but there are a couple of potential surprises in how these are handled. First off is the name representation. The Java source code naming rules treat classes the same as packages, which leads to some serious ambiguity. For instance, there's no way to determine whether the class name `org.jibx.Abc.Def` refers to a class `Def` in the package `org.jibx.Abc` or to an inner class `Def` of the class `org.jibx.Abc`. To avoid this ambiguity, JiBX uses the naming rules Java applies in generated bytecode, where the character '$' is used as a separator between a class name and the name of an inner class. So when you're working with JiBX binding definitions, the class name `org.jibx.Abc.Def` always refers to a class `Def` in the package `org.jibx.Abc`, while the class name `org.jibx.Abc$Def` always refers to an inner class `Def` of the class `org.jibx.Abc`.

The second issue users sometimes run into when using inner classes with JiBX is that non-static inner classes can't be used directly for unmarshalling. This is because each non-static inner class instance contains a hidden reference to the "owning" outer class instance. You can still work with such non-static inner classes using JiBX, but you need to use a factory method to create an instance of the class (see User extension method hooks for details of working with factory methods).

Enumerations

You can use all types of enumerations with JiBX, including both "official" Java 5 enum types and various forms of typesafe enumerations which are compatible with earlier Java versions. Enumerations normally represent simple string values, so they're represented in the binding definition using a **value** element (though if you have an enumeration with complex structure, you can instead use a **structure** element just as you would for any other complex class).

Simple Java 5 enum types, where the value to be used in the XML representation is the same as the enum name, are handled automatically by JiBX. If the value for the XML representation is different from the enum name, you can use the **enum-value-method** attribute of the binding **value** element (technically part of the string attribute group) to tell JiBX which method should be called to obtain the XML representation. The specified method must take no parameters and return a `String` value. This method will be used both when marshalling, to get the text to output, and when unmarshalling, to see if an enum instance matches the input text (with each enum value checked in turn until a match is found).

Other types of enumerations require a little more work. Generally the idea of an enumeration (especially type-safe enumerations) is that only one instance of the enumeration type is created for each possible value. For JiBX to work with such an enumeration type for unmarshalling, you need to define a static deserialization method which takes a text value and returns the corresponding enumeration instance (see the section **Custom serializers and deserializers** on the next page of the tutorial for details). If the enumeration type defines a `toString()` method which returns the text value used in the XML representation, you don't need to specify anything special for marshalling (since JiBX uses the `toString()` method by default for any object being marshalled as a **value**). If `toString()` does *not* return the value to be used in XML, you also need to define a serialization method (again, see **Custom serializers and deserializers** for details).

If an enumeration type is used in more than one place in your binding, you may want to use a **format** definition for the type. Doing this allows you to specify the added information in just one place, and then have it used automatically everywhere that type is referenced. See the **User extension method hooks** section of the next page for details.

Directions and tracking

The **binding** element that's the root of every binding definition supports several attributes that control the overall operation of JiBX when working with that binding. Two of these attributes are worth special mention as part of this tutorial.

The **direction** attribute lets you define one-way bindings, supporting either unmarshalling XML to your object classes (**direction="input"**) or marshalling your object classes to XML (**direction="output"**). If you only need to go in one direction this can simplify your binding definition and sometimes even your code, since you don't need to define binding attributes that are only used for the other direction. By default, you always need to provide all necessary information for going in both directions.

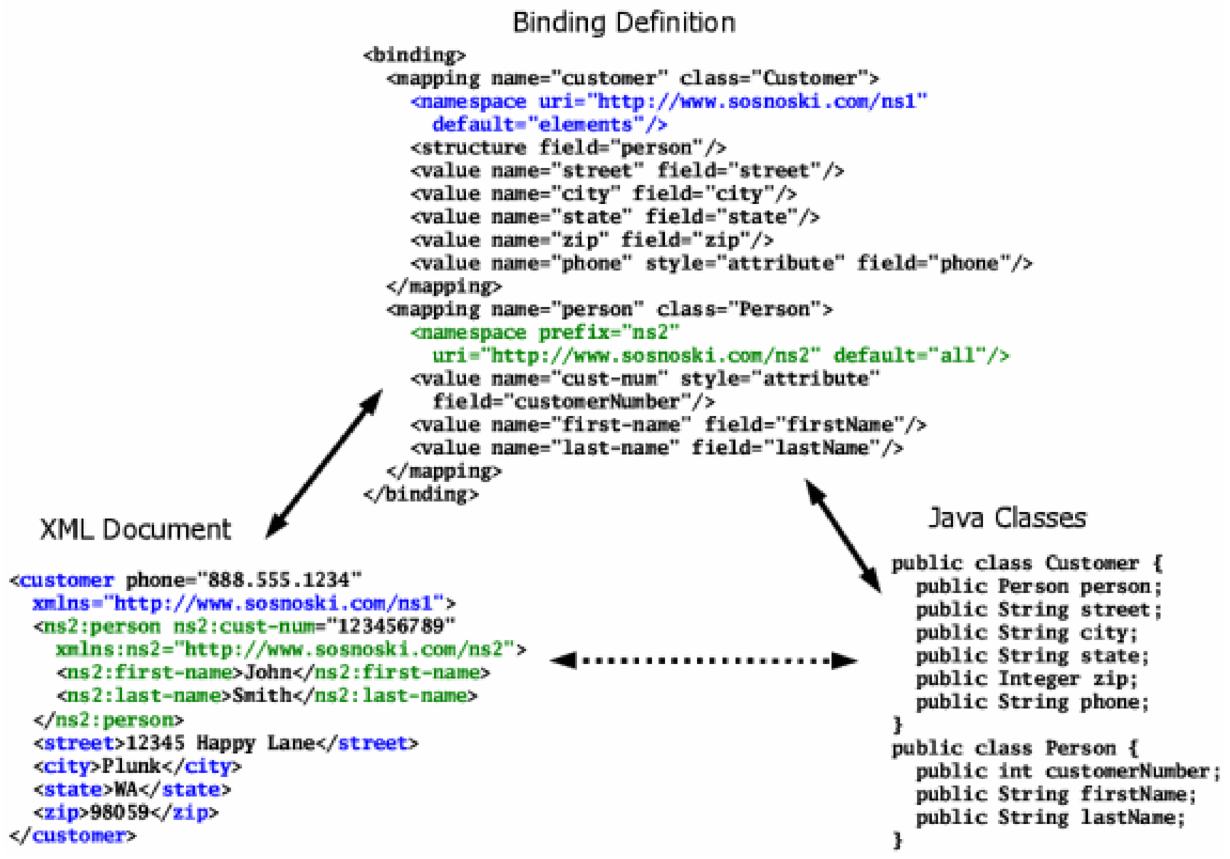
The **track-source** attribute lets you add source position tracking when unmarshalling. This can be useful when you want to be able to relate unmarshalled objects back to particular locations in an input document (as when reporting errors to a user, for instance). If you use **track-source="true"**, the binding compiler will add code to each of your bound object classes to implement the `org.jibx.runtime.ITrackSource` interface, and will set the source position information for each unmarshalled object. You can then access the source position information (in terms of document name, line number, and column number) using the methods provided by the interface (though you'll need to cast your objects to the interface type in order to do this, since the interface will not be visible at compile time).

See the <binding> element details page for information on all the available options.

Working with namespaces

Namespaces are an increasingly important part of working with XML. JiBX provides full support for namespaces, though the **namespace** element and the **ns** attribute for element and attribute names. Figure 18 gives a simple example of a document with two namespaces defined.

Figure 18. Working with namespaces



```
<binding>
  <mapping name="customer" class="Customer">
    <namespace uri="http://www.sosnoski.com/ns1"
      default="elements"/>
    <structure field="person"/>
    <value name="street" field="street"/>
    <value name="city" field="city"/>
    <value name="state" field="state"/>
    <value name="zip" field="zip"/>
    <value name="phone" style="attribute" field="phone"/>
  </mapping>
  <mapping name="person" class="Person">
    <namespace prefix="ns2"
      uri="http://www.sosnoski.com/ns2" default="all"/>
    <value name="cust-num" style="attribute"
      field="customerNumber"/>
    <value name="first-name" field="firstName"/>
    <value name="last-name" field="lastName"/>
  </mapping>
</binding>
```

```
<customer phone="888.555.1234"
  xmlns="http://www.sosnoski.com/ns1">
  <ns2:person ns2:cust-num="123456789"
    xmlns:ns2="http://www.sosnoski.com/ns2">
    <ns2:first-name>John</ns2:first-name>
    <ns2:last-name>Smith</ns2:last-name>
  </ns2:person>
  <street>12345 Happy Lane</street>
  <city>Plunk</city>
  <state>WA</state>
  <zip>98059</zip>
</customer>
```

```
public class Customer {
  public Person person;
  public String street;
  public String city;
  public String state;
  public Integer zip;
  public String phone;
}
public class Person {
  public int customerNumber;
  public String firstName;
  public String lastName;
}
```

Here the first **namespace** element in the binding definition sets the *http://www.sosnoski.com/ns1* namespace as the default for elements within the context of the customer mapping. This is equivalent to a standard default namespace definition in XML, which is what JiBX creates when marshalling using this mapping. The portions of the diagram relating to this namespace are highlighted in blue.

The second **namespace** element sets the *http://www.sosnoski.com/ns2* namespace as the default for both elements and attributes within the context of the person mapping. This makes the namespace automatically apply to every name definition unless you override it with a specific namespace using the **ns** attribute. The use of this namespace is highlighted in green in the diagram.

If you have namespace definitions that apply to the entire XML document you can just define these directly as children of the **binding** element. The effect of placing a definition there is the same as if it were included separately within each top-level mapping definition.

See the <namespace> element details page for more information on working with namespaces in bindings.

Structuring bindings

The **include** element provides a way to structure your binding definitions to be more modular and reusable by letting you break them up into pieces. See the <include> element details page for more information on using this type of modular definition structure, including how namespaces work in combination with included bindings.

You should also keep in mind that you can use multiple bindings with a single set of classes. The only requirement to do this is that all the bindings must be compiled at the same time - if you compile bindings separately, the last one compiled will wipe out the information from the earlier bindings. This ability to work with multiple bindings is an important aspect of JiBX's flexiblity, allowing such uses as working with multiple versions of XML documents in a single set of Java classes (unlike other frameworks, which would generally require a separate set of classes for each version of the documents).

Next: Method hooks for extending JiBX