## Using custom code with JiBX

For the highest degree of control over JiBX binding behavior you can write custom code that ties directly into the JiBX framework. This gives you ways of handling special cases that are beyond the normal binding flexibility. It's not a technique to be used lightly, though.

Working at this level takes you into the internals of the JiBX runtime implementation classes. These implementation classes are considered to be reasonably stable, but are likely to change more frequently then the interface classes (in the `org.jibx.runtime` package) defined for normal user access. They're also not as well documented as the normal user interface - all you'll have to work with is this page, the associated example code, and the JavaDocs for the implementation classes (in the `org.jibx.runtime.impl` package).

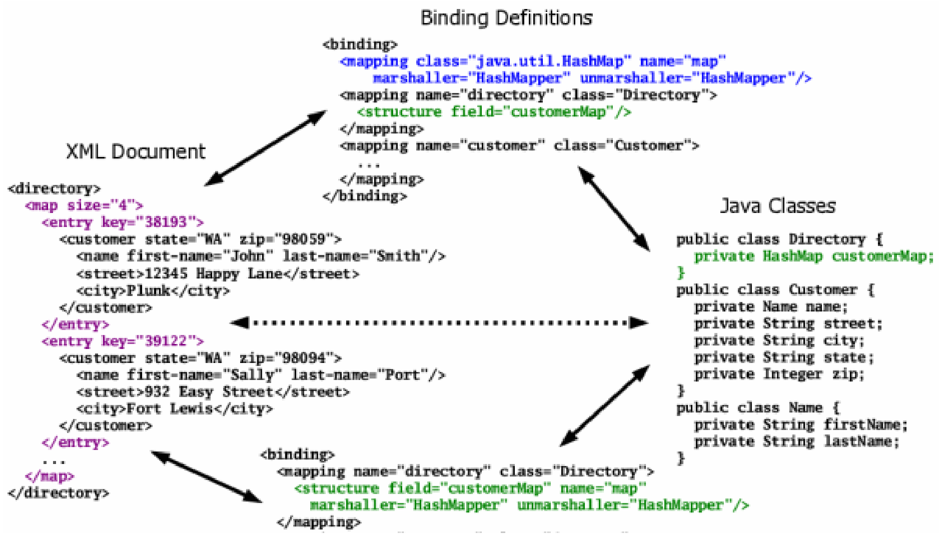
Even with these limitations, the flexibility may be worth the pain involved if you have special binding requirements. I've supplied a pair of examples here which illustrate this flexibility. The JiBX custom code portions of these examples are included in the *jibx-extras.jar* in the distribution, so they're available for use without the need for you to personally dig into the JiBX implementation code (unless you want to change the way they work).

### Custom marshallers and unmarshallers

The principle behind custom marshallers and unmarshallers is simple: Instead of letting JiBX build a marshaller and unmarshaller for a class based on a **mapping** element in your binding definition, you tell JiBX to use your own supplied class. This class must implement the `org.jibx.runtime.IMarshaller` (for a marshaller, used by an output binding) and/or `org.jibx.runtime.IUnmarshaller` (for an unmarshaller, used by an input binding) interfaces.

You can think of custom marshallers and unmarshallers as taking the simpler idea of **custom serializers and deserializers** to a whole new level. Serializers and deserializers give you control over the text representation of primitive or simple object values, where there's no XML structure involved. Custom marshallers and unmarshallers extend this control to complex objects with any number of attributes and/or child elements. The downside of this extended control is that marshallers and unmarshallers are considerably more complex to code than serializers and deserializers.

**Figure 21. Using a custom marshaller/unmarshaller**

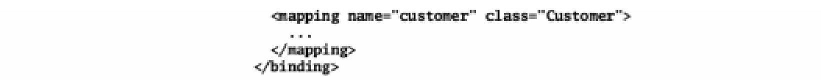


```
<binding>
  <mapping class="java.util.HashMap" name="map"
      marshaller="HashMapper" unmarshaller="HashMapper"/>
  <mapping name="directory" class="Directory">
    <structure field="customerMap"/>
  </mapping>
  <mapping name="customer" class="Customer">
    ...
  </mapping>
</binding>
```

```
<directory>
  <map size="4">
    <entry key="38193">
      <customer state="WA" zip="98059">
        <name first-name="John" last-name="Smith"/>
        <street>12345 Happy Lane</street>
        <city>Plunk</city>
      </customer>
    </entry>
    <entry key="39122">
      <customer state="WA" zip="98094">
        <name first-name="Sally" last-name="Port"/>
        <street>932 Easy Street</street>
        <city>Fort Lewis</city>
      </customer>
    </entry>
    ...
  </map>
</directory>
```

```
public class Directory {
  private HashMap customerMap;
}
public class Customer {
  private Name name;
  private String street;
  private String city;
  private String state;
  private Integer zip;
}
public class Name {
  private String firstName;
  private String lastName;
}
```

```
<binding>
  <mapping name="directory" class="Directory">
    <structure field="customerMap" name="map"
      marshaller="HashMapper" unmarshaller="HashMapper"/>
  </mapping>
```



```
    <mapping name="customer" class="Customer">
      ...
    </mapping>
  </binding>
```

Figure 21 shows a pair of examples of using a a custom marshaller/unmarshaller. In this case the marshaller/unmarshaller involved handles `java.util.HashMap` instances. The Java class structure is based off a `Directory` that contains a `HashMap` (highlighted in green) using customer identifier strings as keys and the associated customer information (in the form of `Customer` class instances) as values. The custom marshaller code converts the hashmap into a **map** element wrapper with an attribute giving the number of key-value pairs, and the actual key-value pairs as contained **entry** elements (all highlighted in magenta). Each **entry** element gives the key as an attribute and the mapped value as the content. The custom unmarshaller converts this XML structure back into a hashmap instance.

The binding definitions in Figure 21 show two different ways of using a custom marshaller/unmarshaller class. In the top version, the **mapping** definition highlighted in blue tells JiBX to use the custom marshaller/unmarshaller by default for handling all `java.util.HashMap` instances referenced by the binding. Here the actual binding definition for the **customerMap** field (highlighted in green) automatically uses the custom code. In the bottom version, the custom marshaller and unmarshaller are instead referenced directly by the binding definition for the field (again highlighted in green).

Hashmap handling makes an especially good example of the situations that require custom marshaller/unmarshallers. This particular implementation handles one particular form of hashmaps, with string keys and values of an object type with a **mapping** definition within the JiBX binding file. Other forms of hashmaps could be handled by modifying the basic marshaller/unmarshaller code, but it's very difficult to handle all possible types of hashmaps with a single implementation. Consider how the XML structure would need to be different if the key values in the hashmap were other mapped objects rather than simple string values, for instance (they couldn't be expressed in XML as attribute values, for starters).

Below is a partial listing of the custom marshaller/unmarshaller class (with only the marshaller implementation shown - see the */tutorial/example21* directory of the distribution for the full source code and related files):

```
public class HashMapper implements IMarshaller, IUnmarshaller, IAliasable
{
    private static final String SIZE_ATTRIBUTE_NAME = "size";
    private static final String ENTRY_ELEMENT_NAME = "entry";
    private static final String KEY_ATTRIBUTE_NAME = "key";
    private static final int DEFAULT_SIZE = 10;

    private String m_uri;
    private int m_index;
    private String m_name;

    public HashMapper() {
        m_uri = null;
        m_index = 0;
        m_name = "hashmap";
    }

    public HashMapper(String uri, int index, String name) {
        m_uri = uri;
        m_index = index;
        m_name = name;
    }

    public boolean isExtension(int index) {
        return false;
    }
    public void marshal(Object obj, IMarshallingContext ictx)
        throws JiBXException {
        
        // make sure the parameters are as expected
        if (!(obj instanceof HashMap)) {
            throw new JiBXException("Invalid object type for marshaller");
        } else if (!(ictx instanceof MarshallingContext)) {
            throw new JiBXException("Invalid object type for marshaller");
        } else {
            
            // start by generating start tag for container
            MarshallingContext ctx = (MarshallingContext)ictx;
            HashMap map = (HashMap)obj;
            ctx.startTagAttributes(m_index, m_name).
                attribute(m_index, SIZE_ATTRIBUTE_NAME, map.size()).
                closeStartContent();
            
            // loop through all entries in hashmap
            Iterator iter = map.entrySet().iterator();
            while (iter.hasNext()) {
                Map.Entry entry = (Map.Entry)iter.next();
                ctx.startTagAttributes(m_index, ENTRY_ELEMENT_NAME);
                if (entry.getKey() != null) {
                    ctx.attribute(m_index, KEY_ATTRIBUTE_NAME,
                        entry.getKey().toString());
                }
                ctx.closeStartContent();
                if (entry.getValue() instanceof IMarshallable) {
                    ((IMarshallable)entry.getValue()).marshal(ctx);
                    ctx.endTag(m_index, ENTRY_ELEMENT_NAME);
                } else {
                    throw new JiBXException("Mapped value is not marshallable");
                }
            }
            
            // finish with end tag for container element
            ctx.endTag(m_index, m_name);
        }
    }
    ...
}
```

At runtime, JiBX creates an instance of the class when needed using either a default (no-argument) constructor or an optional *aliased* constructor that uses element name information passed in from the JiBX binding. The difference between the two is that the aliased constructor allows the element name to be used by the marshaller/unmarshaller to be set by JiBX based on the information in the binding definition file. For the Figure 21 example JiBX uses the aliased constructor behind the scenes, since the binding definitions supply a name for the mapped element. If a custom marshaller or unmarshaller class (which need not be the same class) supports setting the root element name in this way it needs to implement the `org.jibx.runtime.IAliasable` interface (which is only an indicator interface, with no actual methods defined).

This naming flexibility only applies at the top level, though. As you can see from the code, the local names used for the attributes and nested element name are fixed at compile time, while the namespaces are all set to match that of the aliased top-level element (which may not be what you want - often documents that use namespaces for elements do not use them for attributes, for instance).

Besides the two constructor variations shown in this example, you can also define constructors that take an additional `String` parameter. If the binding compiler finds a constructor of this type it will pass the name of the object class that the marshaller/unmarshaller is used with in the binding when calling the constructor. This feature can be used to implement polymorphic marshaller/unmarshallers.

If you want to use this custom marshaller/unmarshaller for hashmaps in your own application you can find it included in *jibx-extras.jar* with the name `org.jibx.extras.HashMapperStringToComplex`. You can also find several other custom marshaller/unmarshaller classes in the `org.jibx.extras` package, including the `org.jibx.extras.TypedArrayMapper` that gives an example of marshaller/unmarshaller polymorphism. I'll also look into ways of making custom marshaller/unmarshallers even more configurable in the future.

## Controlling JiBX with front-end code

Another interesting issue that's come up for several users in the past is the need to work with multiple versions of XML documents. JiBX has supported this from the beginning if the versions used different root element names, but this is not a convenient approach for XML versioning - it's much easier to keep the element names the same and instead just use an attribute of the root element to specify the document version.

This makes a good example of controlling the high-level operation of JiBX from your own code. Below is a partial listing (with constructor and get/set methods left out) of code that first selects a binding for unmarshalling based on an attribute of the document root element, then creates an unmarshalling context for the specific version found and uses that context to unmarshal the document. On the marshalling side, this uses a supplied version string to select the binding:

```
public class BindingSelector
{
    /** URI of version selection attribute. */
    private final String m_attributeUri;

    /** Name of version selection attribute. */
    private final String m_attributeName;

    /** Array of version names. */
    private final String[] m_versionTexts;

    /** Array of bindings corresponding to versions. */

    private final String[] m_versionBindings;

    /** Basic unmarshalling context used to determine document version. */
    private final UnmarshallingContext m_context;

    /** Stream for marshalling output. */
    private OutputStream m_outputStream;

    /** Encoding for output stream. */
    private String m_outputEncoding;

    /** Output writer for marshalling. */
    private Writer m_outputWriter;

    /** Indentation for marshalling. */
    private int m_outputIndent;

    ...

    /**
     * Marshal according to supplied version.
     *
     * @param obj root object to be marshalled
     * @param version identifier for version to be used in marshalling
     * @throws JiBXException if error in marshalling
     */

    public void marshalVersioned(Object obj, String version)
        throws JiBXException {

        // look up version in defined list
        String match = (version == null) ? m_versionTexts[0] : version;
        for (int i = 0; i < m_versionTexts.length; i++) {
            if (match.equals(m_versionTexts[i])) {

                // version found, create marshaller for the associated binding
                IBindingFactory fact = BindingDirectory.
                    getFactory(m_versionBindings[i], obj.getClass());
                MarshallingContext context =
                    (MarshallingContext)fact.createMarshallingContext();

                // configure marshaller for writing document
                context.setIndent(m_outputIndent);
                if (m_outputWriter == null) {
                    if (m_outputStream == null) {

                        throw new JiBXException("Output not configured");
                    } else {
                        context.setOutput(m_outputStream, m_outputEncoding);
                    }
                } else {
                    context.setOutput(m_outputWriter);
                }

                // output object as document
                context.startDocument(m_outputEncoding, null);
                ((IMarshallable)obj).marshal(context);
                context.endDocument();
                return;

            }
        }

        // error if unknown version in document
        throw new JiBXException("Unrecognized document version " + version);
    }

    /**
     * Unmarshal according to document version.
     *
     * @param clas expected class mapped to root element of document (used only
     * to look up the binding)
     * @return root object unmarshalled from document
     * @throws JiBXException if error in unmarshalling
     */

    public Object unmarshalVersioned(Class clas) throws JiBXException {

        // get the version attribute value (using first value as default)
        m_context.toStart();
        String version = m_context.attributeText(m_attributeUri,
            m_attributeName, m_versionTexts[0]);

        // look up version in defined list
        for (int i = 0; i < m_versionTexts.length; i++) {
            if (version.equals(m_versionTexts[i])) {

                // version found, create unmarshaller for the associated binding
                IBindingFactory fact = BindingDirectory.
                    getFactory(m_versionBindings[i], clas);
                UnmarshallingContext context =

                    (UnmarshallingContext)fact.createUnmarshallingContext();

                // return object unmarshalled using binding for document version
                context.setFromContext(m_context);
                return context.unmarshalElement();

            }
        }

        // error if unknown version in document
        throw new JiBXException("Unrecognized document version " + version);
    }
}
```

To use this binding selection code you need to define a pair of arrays of giving the text values for each version number and the corresponding binding name, then create the `BindingSelector` instance and set the document to be unmarshalled before finally calling `unmarshalVersioned` method to get back the unmarshalled root object:

```
// attribute text strings used for different document versions
    private static String[] VERSION_TEXTS = {
        "1.0", "1.1", "1.2"
    };

    // binding names corresponding to text strings
    private static String[] VERSION_BINDINGS = {
        "binding0", "binding1", "binding2"
    };

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

            // process input file according to declared version
            BindingSelector select = new BindingSelector(null, "version",
                VERSION_TEXTS, VERSION_BINDINGS);
            IUnmarshallingContext context = select.getContext();
            context.setDocument(new FileInputStream(args[0]), null);
            Customer customer = (Customer)select.
                unmarshalVersioned(Customer.class);
            ...
```

You can find the full source code and related files in the */tutorial/example22* directory of the distribution. A version of this class is also included in the *jibx-extras.jar* as `org.jibx.extras.BindingSelector`, so if you just want to make use of it as-is you can easily do so.