Runtime

After running the binding compiler the modified classes are ready to be used. You'll need to include the */lib/jibx-run.jar* and */lib/xpp3.jar* jar files from the distribution in your application classpath (but not the */lib/bcel.jar* and */lib/jibx-bind.jar* files, which are only used by the binding compiler). You'll also need to add a little code at whatever point you want to marshal or unmarshal a document. This uses the `org.jibx.runtime.BindingDirectory` class that's included in the JiBX runtime jar, along with a binding factory class that JiBX generates in the same package as your code (or in a package you specify in your binding). You don't need to worry about the details of getting at this generated class, though. Instead, you can access it by passing any of the classes defined by a global mapping (one that's a child of the root **binding** element) in your binding to the `BindingDirectory`. The code is simple:

```
IBindingFactory bfact =
    BindingDirectory.getFactory(Customer.class);
```

Here *Customer* is the name of a class with a global mapping in the binding. If you've compiled more than one binding into the code, you'll also need to pass the name of the binding you want to use. Or if you prefer, you can use a different lookup method in the `BindingDirectory` class which finds the binding factory using a binding name and Java package name.

The `org.jibx.runtime.IBindingFactory` interface that gets returned provides methods to construct marshalling and unmarshalling contexts, which in turn allow you to do the actual marshal and unmarshal operations. Here's an unmarshal example:

```
IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
Object obj = uctx.unmarshalDocument
    (new FileInputStream("filename.xml"), null);
```

This is just one of several variations of an unmarshal call, in this case to unmarshal an XML document in the file *filename.xml*. You can pass a reader instead of a stream as the source of the document data if you want, and can also specify an encoding for the document - see the JavaDocs for details. The returned object will be an instance of one of your classes defined with a global mapping in the binding - you can either check the type with **instanceof** or cast directly to your object type, if you know what it is.

Marshalling is just as easy. Here's an example:

```
IMarshallingContext mctx = bfact.createMarshallingContext();
mctx.marshalDocument(obj, "UTF-8", null,
    new FileOutputStream("filename.xml"));
```

As with the unmarshal example, this is just one of several variations that can be used for the marshal call. This marshals the object to an XML document written to the file *filename.xml*, with **UTF-8** character encoding (the most common choice for XML). The code as shown writes the output document with no extra whitespace; you can use the `setIndent()` method of the context to add whitespace for readability, if you wish. You can pass a writer instead of a stream, as well as some other variations - see the following section for details on character encoding usages, and the JavaDocs for the different types of marshal calls. The object to be marshalled must always be an instance of a class defined with a global mapping

in the binding.

Normally XML documents are marshalled as complete units, and to facilitate this usage the `marshalDocument()` variations all close the output stream or writer once the end tag for the XML document has been written. In special cases you may need to write an XML document without closing the output. You can also do this with JiBX marshalling, though it's a little more involved than the common case. Here's the code:

```
IMarshallingContext mctx = bfact.createMarshallingContext();
mctx.setOutput(new FileOutputStream("filename.xml"), null);
((IMarshallable)obj).marshal(mctx);
mctx.getXmlWriter().flush();
```

In the above code, the cast to `IMarshallable` uses an interface added to each mapped class by the JiBX binding compiler. The call to the `marshal()` method of the interface does the actual marshalling of XML, and the last line makes sure that all the output has been written from internal buffers.

Viewing binding information

The 1.2 release of JiBX adds a PrintInfo tool for accessing a binding factory directly and viewing compiled binding information. This tool is the default execution target for the *jibx-run.jar*, allowing you to invoke it very easily from the command line:

```
java -jar lib/jibx-run.jar
```

When executed in this manner it just prints out the JiBX runtime version information. If you add a `-?` argument to the command line it'll show usage information for the tool, including

other command line parameters you can use to direct it to a binding factory which can be loaded from the Java classpath. Here's an example of running PrintInfo for the *starter* example, assuming you're doing this from a console open to the *examples/starter* directory and you've compiled the classes and the binding for this example (in a single line, shown split here only for formatting):

```
java -cp bin:../../lib/jibx-run.jar org.jibx.runtime.PrintInfo
 -c org.jibx.starter.Customer
```

The output tells you the version of the binding compiler used to compile the binding, the namespaces used by the binding, and the mapping definitions included in the binding.

Character encodings

The Java core classes provides `java.io.Writer` implementations that support a wide variety of character encodings. These can wrap simple output streams, and handle the conversions from Java characters to bytes as appropriate for the particular encoding used by the writer. This direct conversion of characters to bytes is not sufficient for use with XML, though. The problem is that character encodings may not allow for all the legal XML character codes. Any XML characters that are not supported by the output encoding need to be converted to *character references* for output (see the XML recommendation for details).

Because of this need to use character references, JiBX supports the use of *character escapers* for output conversion handling. For the widely-used UTF-8 and ISO-8859-1 (Western European character set) encodings implementations are included that handle both

(Western European character set) encodings implementations are included that handle both stream and writer output formats automatically (though using a stream will provide the best performance). The US-ASCII 7-bit format is also handled automatically, though in this case a `java.io.Writer` is always used internally.

Other character encodings can be used for output if you supply an appropriate `org.jibx.runtime.ICharacterEscaper` instance to be used with the output stream or writer. Depending on the encoding, you may even be able to use one of the existing character escaper implementation classes from the `org.jibx.runtime.impl` package directly, or at least base your own escaper code on one of those implementation classes. For most users the standard encodings are all that will ever be needed, but this approach allows other alternatives to be used when necessary for special requirements.

Output formats

JiBX also provides the ability to generate output formats other than text, by using different implementations of the `org.jibx.runtime.IXMLWriter` interface. The implementations of this interface currently provided support text output to streams or writers, StAX output (see below), and output to XBIS format (using the XBIS *org.xbis.JibxWriter* class). Users with special requirements in this area can implement their own versions of the `org.jibx.runtime.IXMLWriter` interface and use it directly.

StAX support

Support for StAX parser input and StAX writer output has been supported since JiBX version 1.1. To select which parser you want to use for input, you can set the system property *org.jibx.runtime.impl.parser* to the value *org.jibx.runtime.impl.XMLPullReaderFactory* to select the XPP3 XMLPull parser, or the value *org.jibx.runtime.impl.StAXReaderFactory* to select the StAX parser. By default, JiBX uses whichever parser implementation it finds at runtime, with preference given to the XPP3 XMLPull parser if both XPP3 and a StAX parser are present. If you never want to use the XPP3 parser, simply remove the */lib/xpp3.jar* file from your JiBX installation and runtime classpath.

Some StAX parsers support schema validation of input. If you wish to make use of this feature you'll need to substitute an appropriate StAX parser implementation for the */lib/wstx-asl.jar* StAX parser included in the JiBX distribution, and take whatever action is needed to enable schema validation.

To use StAX output (to a `javax.xml.stream.XMLStreamWriter` instance) you'll need to set the XML writer directly on the JiBX marshalling context, using the special JiBX `org.jibx.runtime.impl.StAXWriter` class which wraps the target `javax.xml.stream.XMLStreamWriter` instance. Here's a sample of code for this purpose:

```
// marshal root object back out to document in memory
IMarshallingContext mctx = bfact.createMarshallingContext();
ByteArrayOutputStream bos = new ByteArrayOutputStream();
try {
    XMLOutputFactory ofact = XMLOutputFactory.newInstance();
    XMLStreamWriter wrtr = ofact.createXMLStreamWriter(bos, enc);
    mctx.setXmlWriter(new StAXWriter(bfact.getNamespaces(), wrtr));
    mctx.marshalDocument(obj);
} catch (XMLStreamException e) {
    throw new JiBXException("Error creating writer", e);
}
```