

JiBX 1.2, Part 2: XML schema to Java code

Generate cleaner, customized Java code from XML schema

Skill Level: Intermediate

[Dennis Sosnoski \(dms@sosnoski.com\)](mailto:dms@sosnoski.com)

Lead consultant

Sosnoski Software Associates Ltd

03 Mar 2009

Code generation from XML schema definitions is widely used for all types of XML data exchange, including Web services. Most data-binding tools rigidly structure generated code based on the schema — even aspects of the schema that may be irrelevant to your application. In this tutorial, second in a two-part [series](#), learn how JiBX 1.2 generates cleaner code by doing a better job of interpreting the schema and eliminating unnecessary class clutter. You'll also see how you can customize the generated code to suit your needs better, including customizations that easily eliminate unnecessary components of the schema.

Section 1. Before you start

About this tutorial

JiBX is a tool for binding XML data to Java™ objects. JiBX data binding has long been known as the fastest and most flexible approach for binding Java code to XML. But the complexity of its binding definitions and its limited support for increasingly widely used XML schema definitions have frustrated users at times. Fortunately, the 1.2 version of JiBX goes a long way toward eliminating these issues. In this tutorial, you'll learn about using the new features of JiBX 1.2 to generate Java code easily from XML schema definitions and to read and write XML documents matching the generated schema definitions — all without needing to get into the details of JiBX

binding definitions. [Part 1](#) covers the flip side of starting from Java code and generating XML schema definitions.

Objectives

This tutorial guides you through the process of using JiBX to generate Java code from XML schema definitions. You'll first learn how to work with a simple schema and generate a default Java data model matching that schema, then use that data model for reading and writing XML documents. You'll next see how customizations can be used to modify the code generation so it better fits your needs. Finally, you'll move on to a more complex industry-standard schema example and explore the power of customizations to simplify the data model generated for that schema and improve usability. After reading this tutorial and working through the supplied examples, you'll be able to use JiBX to generate customized Java data models for your own schemas.

Prerequisites

To understand this tutorial, you should have at least a basic knowledge of both Java code and XML. You don't need a detailed understanding of XML schema definitions, but some familiarity with schema will help you understand the examples better.

System requirements

To run the examples you need to install:

- Either:
 - [Sun's JDK 1.5.0_09](#) (or later).
 - [IBM Developer Kit for Java technology 1.5.0 SR3](#).
- A recent version of the [Apache Ant](#) build tool.

JiBX download and installation instructions are included in the tutorial.

Section 2. Introducing JiBX

JiBX is one of many tools used for converting between Java data structures and

XML documents (see [Resources](#)). What sets JiBX apart from the others are performance and flexibility features. JiBX performance consistently measures at the top end of the range, surpassing that of other common tools (such as JAXB 2.0) by a factor or two or more. JiBX is also more flexible than almost all other Java-XML tools, using binding definitions to decouple the Java structure from the XML representation so that each can be changed independently of the other.

With the 1.2 release, JiBX adds major features supporting XML schema definitions. You can use tools included in the JiBX release to generate a schema definition matching your Java code or to generate Java code matching your schema definition. Either way, you also get a binding definition that lets you use JiBX to convert between the Java code and XML documents matching the schema definition. In this tutorial, you'll see how to apply the second type of generation: going from a schema definition to Java code.

Installing JiBX

You'll need to install JiBX before proceeding with this tutorial. [Download](#) the latest 1.2.x distribution ZIP and expand it to a convenient location on your system. You'll end up with a directory named `jibx`, which contains all the JiBX JARs, documentation, examples, and even the source code.

Installing the tutorial code

Now download the tutorial [sample code](#), also provided as a ZIP file. The easiest way to install it on your system is to expand the ZIP into the root directory of your JiBX distribution (or on Windows®, copy the `dwcode2` directory from inside the ZIP file to the root directory of your JiBX distribution). This should create a `dwcode2` subdirectory in the `jibx` directory, with the example files (including `build.xml`, `custom.xml`, and others) inside that `dwcode2` subdirectory.

The sample code includes an Ant build file to automate running the JiBX tools and handle the other steps involved in the examples. If you install the sample code directly into the JiBX installation directory, the build can access the JiBX JARs without any additional configuration. If you install the sample code elsewhere, you can still use the Ant build. In this case, you just need to edit the `build.properties` file inside the sample code directory and change the value of the `jibx-home` property to the path of your JiBX installation.

Section 3. Generating default binding and code from

schema

It's easy to generate a JiBX binding definition, and the corresponding Java code, from an XML schema definition. You'll learn how in this section.

Introducing the simple example schema

As a simple example, I'll start with one of the schemas generated in [Part 1](#). Listing 1 shows an abbreviated version of this schema, intended to represent an order from an online store. The full schema is supplied as starter.xsd in the sample code's dwcode2 directory.

Listing 1. First example schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://jibx.org/starter"
  elementFormDefault="qualified"
  targetNamespace="http://jibx.org/starter">
  <xs:simpleType name="shipping">
    <xs:annotation>
      <xs:documentation>Supported shipment methods. The
"INTERNATIONAL" shipment
  methods can only be used for orders with shipping
  addresses outside the U.S., and
  one of these methods is required in this
  case.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="STANDARD_MAIL"/>
      <xs:enumeration value="PRIORITY_MAIL"/>
      <xs:enumeration value="INTERNATIONAL_MAIL"/>
      ...
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="item">
    <xs:annotation>
      <xs:documentation>Order line item
  information.</xs:documentation>
    </xs:annotation>
    <xs:sequence/>
    <xs:attribute type="xs:string" use="required" name="id">
      <xs:annotation>
        <xs:documentation>Stock identifier. This is expected to
  be 12 characters in
  length, with two leading alpha characters followed by
  ten decimal digits.
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute type="xs:int" use="required" name="quantity">
      <xs:annotation>
        <xs:documentation>Number of units
  ordered.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

```

    <xs:attribute type="xs:float" use="required" name="price">
      <xs:annotation>
        <xs:documentation>Price per unit.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="address">
    <xs:annotation>
      <xs:documentation>Address information.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element type="xs:string" name="street1">
        <xs:annotation>
          <xs:documentation>First line of street information
            (required).</xs:documentation>
        </xs:annotation>
      </xs:element>
      ...
    </xs:sequence>
    <xs:attribute type="xs:string" name="state">
      <xs:annotation>
        <xs:documentation>State abbreviation (required for the
          U.S. and Canada,
          optional otherwise).</xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute type="xs:string" name="postCode">
      <xs:annotation>
        <xs:documentation>Postal code (required for the U.S.
          and Canada, optional
          otherwise).</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="customer">
    <xs:annotation>
      <xs:documentation>Customer
        information.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element type="xs:long" name="customerNumber"/>
      ...
    </xs:sequence>
  </xs:complexType>
  <xs:element type="tns:order" name="order"/>
  <xs:complexType name="order">
    <xs:annotation>
      <xs:documentation>Order information.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element type="xs:long" name="orderNumber"/>
      <xs:element type="tns:customer" name="customer"/>
      <xs:element type="tns:address" name="billTo">
        <xs:annotation>
          <xs:documentation>Billing address
            information.</xs:documentation>
        </xs:annotation>
      </xs:element>
      ...
      <xs:element type="tns:item" name="item" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute type="xs:date" use="required"
      name="orderDate">
      <xs:annotation>
        <xs:documentation>Date order was placed with
          server.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>

```

```
</xs:attribute>
...
</xs:complexType>
</xs:schema>
```

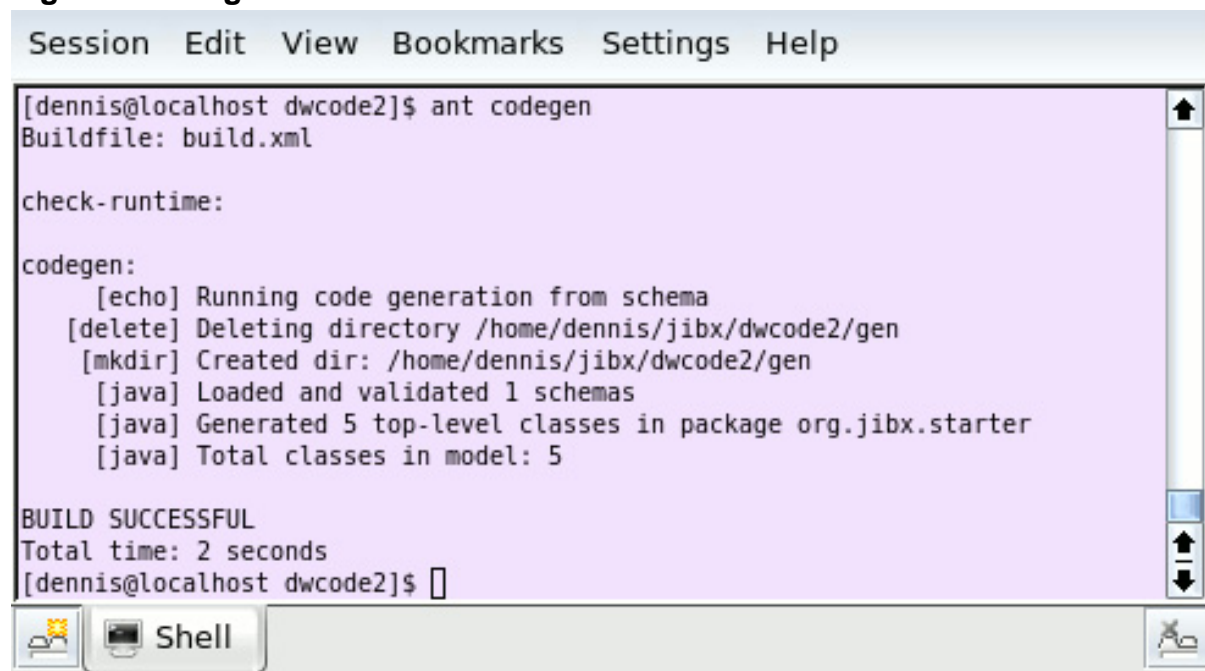
Generating the default binding and code

To generate a JiBX binding and Java classes from an XML schema, you just need to run the `org.jibx.schema.codegen.CodeGen` tool included in the `jibx-tools.jar` from the JiBX distribution. You can run the tool directly from the command line or indirectly via a build tool such as Apache Ant.

The tutorial download includes an Ant `build.xml` script with the `codegen` target to run the generation.

To try this out, open a console in the `dwcode2` directory of the installed download and type `ant codegen`. If you have Ant installed on your system and have installed the download code according to the instructions, you should see output similar to that shown in Figure 1:

Figure 1. Using the Ant build



```
Session Edit View Bookmarks Settings Help
[dennis@localhost dwcode2]$ ant codegen
Buildfile: build.xml

check-runtime:

codegen:
  [echo] Running code generation from schema
  [delete] Deleting directory /home/dennis/jibx/dwcode2/gen
  [mkdir] Created dir: /home/dennis/jibx/dwcode2/gen
  [java] Loaded and validated 1 schemas
  [java] Generated 5 top-level classes in package org.jibx.starter
  [java] Total classes in model: 5

BUILD SUCCESSFUL
Total time: 2 seconds
[dennis@localhost dwcode2]$
```

You can also run CodeGen directly from the console. To do this, you need to:

1. Include the `jibx-tools.jar` in your Java classpath.
2. Specify `org.jibx.schema.codegen.CodeGen` as the class to be run.

3. List the schema definition(s) to be generated.

The supplied Ant `codegen` target uses a couple of additional parameters to tell CodeGen to use the `gen/src` directory as the root of the generated data-model package structure and to wipe all existing files out of that directory before running the generation. Here's the Java command line for duplicating the Ant `codegen` target from a console in the `dwcode2` directory (assuming you've followed the recommended installation instructions):

```
java -cp ../lib/jibx-tools.jar org.jibx.schema.codegen.CodeGen -t gen/src -w starter.xsd
```

On Windows, the command is:

```
java -cp ..\lib\jibx-tools.jar org.jibx.schema.codegen.CodeGen -t gen\src -w starter.xsd
```

You can pass many other options to CodeGen from the command line. You'll look into those later in the tutorial. Now let's take a look at the generated Java code.

Generated artifacts

The generated code consists of five classes, corresponding to the five global type definitions in the [Listing 1](#) schema. Listing 2 shows some samples of the generated code, with excerpts from the `org.jibx.starter.Order` class and the entire `org.jibx.starter.Shipping` class:

Listing 2. Generated code

```
/**
 * Order information.
 *
 * Schema fragment(s) for this class:
 * <pre>
 * <xs:complexType xmlns:ns="http://jibx.org/starter"
 *   xmlns:xs="http://www.w3.org/2001/XMLSchema" name="order">
 *   <xs:sequence>
 *     <xs:element type="xs:long" name="orderNumber"/>
 *     <xs:element type="ns:customer" name="customer"/>
 *     <xs:element type="ns:address" name="billTo"/>
 *     <xs:element type="ns:shipping" name="shipping"/>
 *     <xs:element type="ns:address" name="shipTo" minOccurs="0"/>
 *     <xs:element type="ns:item" name="item" minOccurs="0" maxOccurs="unbounded"/>
 *   </xs:sequence>
 *   <xs:attribute type="xs:date" use="required" name="orderDate"/>
 *   <xs:attribute type="xs:date" name="shipDate"/>
 *   <xs:attribute type="xs:float" name="total"/>
 * </xs:complexType>
 * </pre>
 */
public class Order
{
```

```

private long orderNumber;
private Customer customer;
private Address billTo;
private Shipping shipping;
private Address shipTo;
private List<Item> itemList = new ArrayList<Item>();
private Date orderDate;
private Date shipDate;
private Float total;
...
/**
 * Get the 'shipTo' element value. Shipping address information. If missing, the
 * billing address is also used as the shipping address.
 */
public Address getShipTo() {
    return shipTo;
}
/**
 * Set the 'shipTo' element value. Shipping address information. If missing, the
 * billing address is also used as the shipping address.
 */
public void setShipTo(Address shipTo) {
    this.shipTo = shipTo;
}
/**
 * Get the list of 'item' element items.
 */
public List<Item> getItems() {
    return itemList;
}
/**
 * Set the list of 'item' element items.
 */
public void setItems(List<Item> list) {
    itemList = list;
}
...
}
/**
 * Supported shipment methods. The "INTERNATIONAL" shipment methods can only be used
 * for orders with shipping addresses outside the U.S., and one of these methods is
 * required in this case.
 *
 * Schema fragment(s) for this class:
 * <pre>
 * <xs:simpleType xmlns:xs="http://www.w3.org/2001/XMLSchema" name="shipping">
 *   <xs:restriction base="xs:string">
 *     <xs:enumeration value="STANDARD_MAIL"/>
 *     <xs:enumeration value="PRIORITY_MAIL"/>
 *     <xs:enumeration value="INTERNATIONAL_MAIL"/>
 *     <xs:enumeration value="DOMESTIC_EXPRESS"/>
 *     <xs:enumeration value="INTERNATIONAL_EXPRESS"/>
 *   </xs:restriction>
 * </xs:simpleType>
 * </pre>
 */
public enum Shipping {
    STANDARD_MAIL, PRIORITY_MAIL, INTERNATIONAL_MAIL, DOMESTIC_EXPRESS,
    INTERNATIONAL_EXPRESS
}

```

As you can see from [Listing 2](#), CodeGen automatically converts schema documentation to Javadocs in the generated code (shown here as the leading comments in each class Javadoc, and as part of the comments for the `getShipTo()` and `setShipTo()` methods). CodeGen by default also incorporates

the actual schema definitions in the class Javadocs and for get/set property-access methods it describes the schema component corresponding to the property.

For repeated values, such as the repeating item element within the Listing 1 `complexType` definition, CodeGen generates a Java 5 typed list by default. For `simpleType` restriction enumerations, such as the shipping type in Listing 1, CodeGen generates a Java 5 enum type by default. The generated code for both of these instances is shown in Listing 2.

Generated JiBX binding

Besides the generated code, CodeGen also produces a JiBX binding definition (as the `binding.xml` file, in this case), which tells the JiBX binding compiler how to convert between the Java classes and XML. Binding definitions contain full details of the conversions to be done by JiBX, so they're necessarily complex. Fortunately, you don't need to understand the binding definition in order to work with JiBX using CodeGen binding and code generation, so this tutorial doesn't cover the details.

Section 4. Working with XML documents

In this section, you'll learn about running the JiBX binding compiler and using JiBX at run time to work with XML documents.

Running the JiBX binding compiler

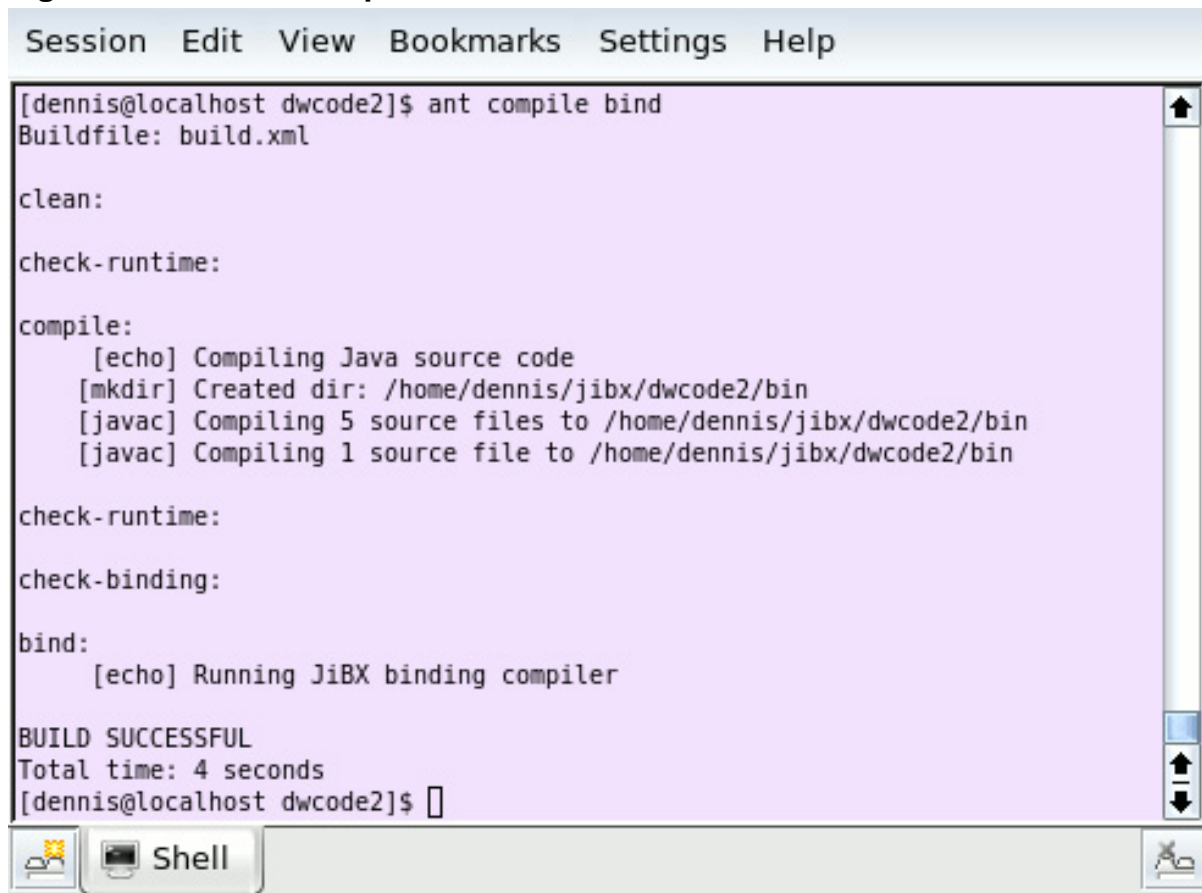
To use the generated binding definition in working with XML documents, you first need to run the JiBX binding compiler. The binding compiler adds bytecode to your compiled class files that actually implements the conversions to and from XML, as specified by the binding definition. You must run the binding compiler each time you recompile your Java classes or modify the binding definition, so it's usually best to add the binding-compiler step as part of your project's standard build process.

The binding compiler is included in the JiBX distribution as part of `jibx-bind.jar`. The JiBX documentation provides full details about different ways to run the binding compiler, including how you can invoke it when running your application rather than as part of the build. JiBX also provides plug-ins for Eclipse and IntelliJ IDEA that automatically run the binding compiler when you're working in these IDEs.

For this tutorial's purposes, you'll keep things simple and just run the binding compiler through Ant. The `build.xml`'s `compile` target prepares for the binding by compiling both the generated code and a supplied test program, while the `bind`

target actually runs the binding compiler. Figure 2 shows the output that you should see when you run these targets, assuming you've already run the `codegen` targets. (You can also run all three targets in sequence by listing them in order on the command line: `ant codegen compile bind`.)

Figure 2. Ant build compile and bind tasks



```
Session Edit View Bookmarks Settings Help
[dennis@localhost dwcode2]$ ant compile bind
Buildfile: build.xml

clean:

check-runtime:

compile:
  [echo] Compiling Java source code
  [mkdir] Created dir: /home/dennis/jibx/dwcode2/bin
  [javac] Compiling 5 source files to /home/dennis/jibx/dwcode2/bin
  [javac] Compiling 1 source file to /home/dennis/jibx/dwcode2/bin

check-runtime:

check-binding:

bind:
  [echo] Running JiBX binding compiler

BUILD SUCCESSFUL
Total time: 4 seconds
[dennis@localhost dwcode2]$
```

Using JiBX at run time

Listing 3 shows a simple test document matching the schema, included in the tutorial's code download as `starter.xml`:

Listing 3. Test document for order schema

```
<order orderDate="2008-10-18" shipDate="2008-10-22" xmlns="http://jibx.org/starter">
  <orderNumber>12345678</orderNumber>
  <customer>
    <customerNumber>5678</customerNumber>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
  </customer>
  <billTo state="WA" postCode="98059">
    <street1>12345 Happy Lane</street1>
```

```

    <city>Plunk</city>
    <country>USA</country>
  </billTo>
  <shipping>PRIORITY_MAIL</shipping>
  <shipTo state="WA" postCode="98034">
    <street1>333 River Avenue</street1>
    <city>Kirkland</city>
  </shipTo>
  <item quantity="1" price="5.99" id="FA9498349851"/>
  <item quantity="2" price="9.50" id="GC1234905049"/>
  <item quantity="1" price="8.95" id="AX9300048820"/>
</order>

```

The download package also includes a simple test program, shown here as Listing 4, that demonstrates using JiBX for both *unmarshalling* and *marshalling* documents. Marshalling is the process of generating an XML representation for an object in memory, potentially including objects linked from the original object. Unmarshalling is the reverse process of marshalling, building an object (and potentially a graph of linked objects) in memory from an XML representation. The Ant `run` target executes this test program, using the Listing 3 document as input and writing the marshalled copy of the document to a file named `out.xml`.

Listing 4. Test program

```

public class Test
{
    /**
     * Unmarshal the sample document from a file, compute and set order total, then
     * marshal it back out to another file.
     *
     * @param args
     */
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: java -cp ... " +
                "org.jibx.starter.Test in-file out-file");
            System.exit(0);
        }
        try {

            // unmarshal customer information from file
            IBindingFactory bfact = BindingDirectory.getFactory(Order.class);
            IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
            FileInputStream in = new FileInputStream(args[0]);
            Order order = (Order)uctx.unmarshalDocument(in, null);

            // compute the total amount of the order
            float total = 0.0f;
            for (Iterator<Item> iter = order.getItems().iterator(); iter.hasNext();) {
                Item item = iter.next();
                total += item.getPrice() * item.getQuantity();
            }
            order.setTotal(new Float(total));

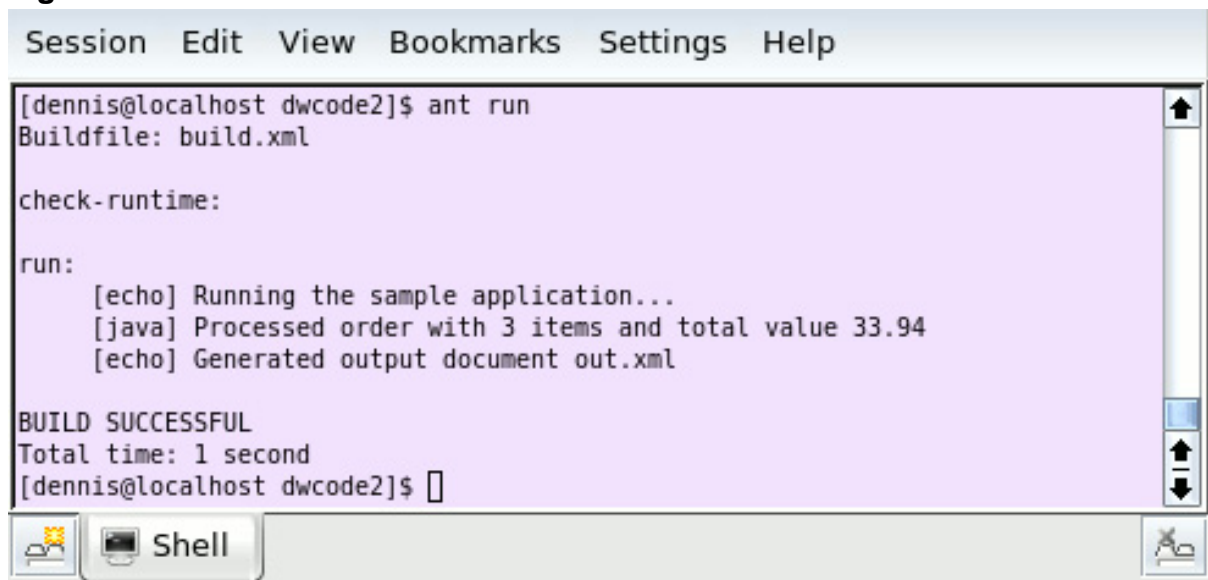
            // marshal object back out to file (with nice indentation, as UTF-8)
            IMarshallingContext mctx = bfact.createMarshallingContext();
            mctx.setIndent(2);
            FileOutputStream out = new FileOutputStream(args[1]);
            mctx.setOutput(out, null);
            mctx.marshalDocument(order);
            System.out.println("Processed order with " + order.getItems().size() +
                " items and total value " + total);
        }
    }
}

```

```
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
        System.exit(1);  
    } catch (JiBXException e) {  
        e.printStackTrace();  
        System.exit(1);  
    }  
}
```

Figure 3 shows the output you should see when you run the `run` target:

Figure 3. Ant build run task



```
Session Edit View Bookmarks Settings Help  
[dennis@localhost dwcode2]$ ant run  
Buildfile: build.xml  
  
check-runtime:  
  
run:  
  [echo] Running the sample application...  
  [java] Processed order with 3 items and total value 33.94  
  [echo] Generated output document out.xml  
  
BUILD SUCCESSFUL  
Total time: 1 second  
[dennis@localhost dwcode2]$
```

This is the same test program as used in [Part 1](#), and it's subject to the same limitations discussed in that tutorial. Just as in Part 1, the `out.xml` file has the output generated by marshalling back out the order data that was obtained by unmarshalling the original document.

Section 5. Introducing CodeGen customizations

In this section, you'll learn the basics of customizing CodeGen to control the structure of the code generated from a simple schema.

A simple customization example

CodeGen supports extensive customizations for many aspects of code and binding

generation. The set of customizations to be applied are passed to CodeGen as an XML document, with nested elements that relate to schemas or schema components. Listing 5 gives a simple example:

Listing 5. Simple customizations example

```
<schema prefer-inline="true" show-schema="false"
enumeration-type="simple"
generate-all="false" includes="order item"/>
```

The [Listing 5](#) customization consists of a single, unnamespaced schema element with several different attributes giving the specific customizations to be applied. (So far, you're only working with a single schema definition, so this very simple form of customization can be used. Later in this tutorial, you'll see examples of customizations for working with multiple schemas.) The first customization attribute — `prefer-inline="true"` — tells CodeGen to inline schema definitions that are referenced only once, rather than use the default behavior of keeping them as separate classes. The second attribute — `show-schema="false"` — blocks embedding the schema definitions in class Javadocs. `enumeration-type="simple"` generates simple typesafe enumerations rather than Java 5 enums.

The final pair of attributes work together. CodeGen by default generates a separate top-level class for every global type definition in the schemas specified as input, along with a corresponding abstract mapping in the generated JiBX binding for each `complexType`. The `generate-all="false"` attribute changes this default behavior, explicitly generating only those `complexType`s that are specifically included or referenced from an included type. The `includes="order item"` attribute then gives the names to be generated, which were chosen to keep compatibility with the test program — the `<order>` element is needed because that's the root element of the instance documents, and the `complexType` item is needed because the test program expects to find a separate top-level class for this type when it sums up the total amount of the order.

You can try out these customizations by using the Ant `custgen` task instead of the `codegen` task (or just use the `full` task, which runs the complete sequence of targets `clean custgen compile bind run`). Listing 6 shows excerpts of the generated code, which you can compare to the default generated code shown in [Listing 2](#). The big differences (besides the simplified class Javadocs) are that the `Customer` class is now inlined, and the `Shipping` class is now an inner class using a custom typesafe enumeration class.

Listing 6. Code generated with customization

```
/**
 * Order information.
 */
public class Order
```

```

{
    private long orderNumber;
    private long customerCustomerNumber;
    private String customerFirstName;
    private String customerLastName;
    private Address billTo;
    private Shipping shipping;
    private Address shipTo;
    private List<Item> itemList = new ArrayList<Item>();
    private Date orderDate;
    private Date shipDate;
    private Float total;
    ...
    /**
     * Supported shipment methods. The "INTERNATIONAL" shipment methods can only be used
     * for orders with shipping addresses outside the U.S., and one of these methods is
     * required in this case.
     */
    public static class Shipping
    {
        private final String value;
        public static final Shipping STANDARD_MAIL = new Shipping(
            "STANDARD_MAIL");
        public static final Shipping PRIORITY_MAIL = new Shipping(
            "PRIORITY_MAIL");
        public static final Shipping INTERNATIONAL_MAIL = new Shipping(
            "INTERNATIONAL_MAIL");
        public static final Shipping DOMESTIC_EXPRESS = new Shipping(
            "DOMESTIC_EXPRESS");
        public static final Shipping INTERNATIONAL_EXPRESS = new Shipping(
            "INTERNATIONAL_EXPRESS");
        private static final String[] values = new String[]{"DOMESTIC_EXPRESS",
            "INTERNATIONAL_EXPRESS", "INTERNATIONAL_MAIL", "PRIORITY_MAIL",
            "STANDARD_MAIL"};
        private static final Shipping[] instances = new Shipping[]{
            DOMESTIC_EXPRESS, INTERNATIONAL_EXPRESS, INTERNATIONAL_MAIL,
            PRIORITY_MAIL, STANDARD_MAIL};

        private Shipping(String value) {
            this.value = value;
        }

        public String toString() {
            return value;
        }

        public static Shipping convert(String value) {
            int index = java.util.Arrays.binarySearch(values, value);
            if (index >= 0) {
                return instances[index];
            } else {
                return null;
            }
        }

        public static Shipping fromValue(String text) {
            Shipping value = convert(text);
            if (value == null) {
                throw new IllegalArgumentException("Value \'' + text
                    + "' is not allowed");
            } else {
                return value;
            }
        }
    }
}

```

Many additional customizations are available for use with CodeGen. You'll see some examples of these later in this tutorial, but to give a better idea of the power of these customizations, it's necessary to move on to a more complex schema.

Section 6. Trying a real-world schema

Working with a stand-alone schema definition is great for a simple demonstration, but it doesn't give much of a feeling for how a tool functions when applied to the complex schema definitions widely used in enterprise applications. Now it's time to move on to a more realistic example, in the form of one of the industry-standard HR-XML schema definitions.

HR-XML TimeCard schema

The HR-XML Consortium is an organization established to develop open standards for XML representations for human resources. It represents more than 110 corporate members, and almost 50 technology firms are certified to meet its standards.

The HR-XML schemas used for this tutorial consist of 157 schemas, including a mixture of top-level document definitions and common components. CodeGen can easily handle this number of schemas, but the number of generated classes and the complexity of the interrelationships would obscure the more interesting aspects of the schema handling. To focus in on these details, the subset of HR-XML used here consists of a single top-level document definition, for the `TimeCard` element, along with the common components referenced as part of the `TimeCard` definition — a total of seven schema definitions.

You can find the subset of HR-XML schema definitions used in this tutorial under the `hrxml/schemas` directory. Listing 7 shows an edited version of the main schema for the `TimeCard` element definition. This gives a sample of the HR-XML schema style, which uses a mixture of nested and global type definitions and contains a wider range of schema structures than the first example, including:

- `<xs:choice>` compositors (as shown in some of the embedded `complexType`s within the `TimeCardType` definition)
- `<xs:any>` particles (see the `AdditionalDataType` definition near the start of the listing)
- `<xs:simpleType>` `<union>`s (see the `TimeCardDuration` definition at the end of the listing)

- Nonenumeration `<xs:simpleType>` restrictions

Listing 7. HR-XML TimeCard schema

```

<xs:schema targetNamespace="http://ns.hr-xml.org/2007-04-15" ...
  elementFormDefault="qualified" version="2007-04-15">
  <xs:import namespace="http://www.w3.org/XML/1998/namespace" ...>
  <xs:include schemaLocation="../CPO/EntityIdType.xsd"/>
  ...
  <xs:complexType name="AdditionalDataType" mixed="true">
    ...
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:any namespace="##any" processContents="strict" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>
  ...
  <xs:element name="TimeCard" type="TimeCardType"/>
  <xs:complexType name="TimeCardType">
    <xs:sequence>
      <xs:element name="Id" type="EntityIdType" minOccurs="0"/>
      <xs:element name="ReportedResource">
        <xs:complexType>
          <xs:choice>
            <xs:element name="Person" type="TimeCardPersonType"/>
            <xs:element name="Resource">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Id" type="EntityIdType"
                    minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="ResourceName" type="xs:string" minOccurs="0"/>
                  <xs:element name="AdditionalData" type="AdditionalDataType" minOccurs="0"
                    maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="type" type="xs:string"/>
              </xs:complexType>
            </xs:element>
          </xs:choice>
        </xs:complexType>
      </xs:element>
      <xs:element name="ReportedTime" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="PeriodStartDate" type="AnyDateTimeType"/>
            <xs:element name="PeriodEndDate" type="AnyDateTimeType"/>
            <xs:element name="ReportedPersonAssignment" minOccurs="0">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Id" type="EntityIdType" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:choice maxOccurs="unbounded">
              <xs:element name="TimeInterval">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Id" type="EntityIdType" minOccurs="0"/>
                    <xs:element name="StartDateTime" type="AnyDateTimeType"/>
                    <xs:choice>
                      <xs:sequence>
                        <xs:element name="EndTimeTime" type="AnyDateTimeType"/>
                        <xs:element name="Duration" type="TimeCardDuration" minOccurs="0"/>
                      </xs:sequence>
                      <xs:element name="Duration" type="TimeCardDuration"/>
                    </xs:choice>
                  </xs:complexType>
                </xs:element>
              </xs:choice>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      <xs:element name="PieceWork" minOccurs="0" maxOccurs="unbounded">

```

```

    ...
  </xs:element>
  <xs:element name="RateOrAmount" minOccurs="0" maxOccurs="unbounded">
    ...
  </xs:element>
  <xs:element name="Allowance" minOccurs="0" maxOccurs="unbounded">
    ...
  </xs:element>
  ...
</xs:sequence>
<xs:attribute name="type" type="xs:string" use="required"/>
...
</xs:complexType>
</xs:element>
<xs:element name="TimeEvent">
  ...
</xs:element>
<xs:element name="Expense">
  ...
</xs:element>
<xs:element name="Allowance">
  ...
</xs:element>
</xs:choice>
...
</xs:sequence>
...
</xs:complexType>
</xs:element>
...
</xs:sequence>
<xs:attribute ref="xml:lang"/>
</xs:complexType>
...
<xs:simpleType name="TimeCardDuration">
  <xs:union memberTypes="xs:duration xs:decimal"/>
</xs:simpleType>
</xs:schema>

```

Generated code for TimeCard

The Ant build.xml file in the hrxml directory defines Ant targets for trying out the basic code generation for the TimeCard schema, including both the default generation and a couple of customization examples (discussed later). The sample directory also contains a test program, org.jibx.hrxml.Test. It unmarshals sample documents using the generated data-model classes and then marshals the documents back out and compares the result with the original document. And there's a set of test documents from the HR-XML distribution in the samples directory. The codegen target runs CodeGen using defaults, compile compiles the generated code and test code, bind compiles the JiBX binding, and roundtrip runs the test program on the sample documents. You can also use the full task to run all of these steps in sequence.

Most forms of code generation from schema generate a separate class for each complexType definition and for enumeration simpleTypes. CodeGen often is able to reduce the number of generated classes by examining references and inlining definitions where possible and by ignoring unused definitions in included and

imported schema definitions. In the case of the `TimeCard` schema, there are a total of 10 global (named) `complexType`s and an additional 23 local (anonymous) `complexType`s, along with 8 enumeration `simpleTypes`. The generated default data model consists of 15 top-level classes and 23 inner classes, just a few fewer than the number you'd expect to see based on the schema component counts. You'll see later some ways of using customizations to further simplify the data model in cases in which not all the schema components are needed.

<xs:choice> handling

Listing 8 shows how CodeGen handles a choice between two elements in the `TimeCardType` `complexType` definition. CodeGen by default uses a selection variable to track which choice is currently active. The set methods for values included in the choice allow you to write a new value for the current selection but prevent changing the selection directly (throwing an `IllegalStateException` if you try). To change the current selection once it has been set, you first need to call a `clear` method (here `clearReportedResourceSelect()`) which resets the selection state.

Listing 8. HR-XML TimeCard-generated code sample

```
/**
 * Schema fragment(s) for this class:
 * <pre>
 * <xs:complexType xmlns:ns="http://ns.hr-xml.org/2007-04-15"
 *   xmlns:ns1="http://www.w3.org/XML/1998/namespace"
 *   xmlns:xs="http://www.w3.org/2001/XMLSchema" name="TimeCardType">
 *   <xs:sequence>
 *     <xs:element type="ns:EntityIdType" name="Id" minOccurs="0"/>
 *     <xs:element name="ReportedResource">
 *       <xs:complexType>
 *         <xs:choice>
 *           <xs:element type="ns:TimeCardPersonType" name="Person"/>
 *           <xs:element name="Resource">
 *             <!-- Reference to inner class Resource -->
 *           </xs:element>
 *         </xs:choice>
 *       </xs:complexType>
 *     </xs:element>
 *     ...
 *   </pre>
 */
public class TimeCardType
{
    private EntityIdType id;
    private int reportedResourceSelect = -1;
    private final int REPORTED_RESOURCE_PERSON_CHOICE = 0;
    private final int RESOURCE_CHOICE = 1;
    private TimeCardPersonType reportedResourcePerson;
    private Resource resource;
    ...
    private void setReportedResourceSelect(int choice) {
        if (reportedResourceSelect == -1) {
            reportedResourceSelect = choice;
        } else if (reportedResourceSelect != choice) {
            throw new IllegalStateException(
                "Need to call clearReportedResourceSelect() before changing existing choice");
        }
    }
}
```

```

}

/**
 * Clear the choice selection.
 */
public void clearReportedResourceSelect() {
    reportedResourceSelect = -1;
}

/**
 * Check if ReportedResourcePerson is current selection for choice.
 *
 * @return <code>>true</code> if selection, <code>>false</code> if not
 */
public boolean ifReportedResourcePerson() {
    return reportedResourceSelect == REPORTED_RESOURCE_PERSON_CHOICE;
}

/**
 * Get the 'Person' element value.
 *
 * @return value
 */
public TimeCardPersonType getReportedResourcePerson() {
    return reportedResourcePerson;
}

/**
 * Set the 'Person' element value.
 *
 * @param reportedResourcePerson
 */
public void setReportedResourcePerson(
    TimeCardPersonType reportedResourcePerson) {
    setReportedResourceSelect(REPORTED_RESOURCE_PERSON_CHOICE);
    this.reportedResourcePerson = reportedResourcePerson;
}

/**
 * Check if Resource is current selection for choice.
 *
 * @return <code>>true</code> if selection, <code>>false</code> if not
 */
public boolean ifResource() {
    return reportedResourceSelect == RESOURCE_CHOICE;
}

/**
 * Get the 'Resource' element value.
 *
 * @return value
 */
public Resource getResource() {
    return resource;
}

/**
 * Set the 'Resource' element value.
 *
 * @param resource
 */
public void setResource(Resource resource) {
    setReportedResourceSelect(RESOURCE_CHOICE);
    this.resource = resource;
}

```

For most applications, this type of choice handling works well, preventing the user

from trying to set more than one alternative in a choice. Customizations can be used to modify the default choice handling, though, so if you don't like this form of choice handling, you can easily change it. The `choice-check` attribute controls how the selection state for an `<xsd:choice>` is checked in the generated code. The `choice-check="disable"` value disables all checking and does not track a selection state, leaving it up to the user to set one and only one value for each choice. `choice-check="checkset"` matches the default handling shown in [Listing 8](#), where only the set methods check for a current setting and throw an exception. `choice-check="checkboth"` also checks the selection state when a get method is called, throwing an exception if the get method does not match the current selection state. Finally, `choice-check="override"` changes the default handling always to change the current state when any value in the choice is set, rather than throwing an exception when a different state was previously set.

The `choice-exposed` customization attribute works in combination with the `choice-check` settings, which track a current selection state. A value of `choice-exposed="false"` keeps the selection state constants, state variable value, and state change method all private, matching the default code generation shown in [Listing 8](#). `choice-exposed="true"` makes these all publicly accessible, adding a get method for the state variable. This allows you to use a Java `switch` statement easily to execute different code depending on the current state, in place of multiple `if` statements.

Both these attributes can be used at any level of customization, allowing you to set the behavior for all the generated code on the outermost customization easily while still retaining the ability to do something different on a case-by-case basis.

`<xs:any>` and `mixed="true"` handling

Like many enterprise schemas, the HR-XML schemas use `<xs:any>` schema components to create extension points for data that can be defined by users independently of the original schema. CodeGen by default handles `<xs:any>` schema components using an `org.w3c.dom.Element` object (or list of `Element`, if the `maxOccurs` value on the `<xs:any>` is greater than 1). The `Element` object can be used to represent any arbitrary XML element (including all attributes, namespace declarations, and content), so it provides all the flexibility needed to work with any document matching the schema definition.

Listing 9 shows the generated code matching an `<xs:any>` component in the [Listing 7](#) schema sample. Because the `<xs:any>` uses `maxOccurs="unbounded"`, the generated code uses a list of `Elements`.

Listing 9. `<xs:any>`-generated code sample

```
/**
```

```

* ...
* Schema fragment(s) for this class:
* <pre>
* <xs:complexType xmlns:xs="http://www.w3.org/2001/XMLSchema" mixed="true"
*   name="AdditionalDataType">
*   <xs:sequence>
*     <xs:any minOccurs="0" maxOccurs="unbounded" processContents="strict"
*       namespace="##any"/>
*   </xs:sequence>
*   <xs:attribute type="xs:string" name="type"/>
* </xs:complexType>
* </pre>
*/
public class AdditionalDataType
{
    private List<Element> anyList = new ArrayList<Element>();
    private String type;

    /**
     * Get the list of sequence items.
     *
     * @return list
     */
    public List<Element> getAny() {
        return anyList;
    }

    /**
     * Set the list of sequence items.
     *
     * @param list
     */
    public void setAny(List<Element> list) {
        anyList = list;
    }
    ...
}

```

Some aspects of the schema definition in Listing 9 are ignored or only partially handled by CodeGen. First, the enclosing `<xs:complexType>` definition specifies `mixed="true"`, which means that character data is allowed to be intermixed with the elements represented by the `<xs:any>` particle. The data model generated by CodeGen has no place to hold such character-data content, so it'll just be discarded when a document is unmarshalled. Second, the `<xs:any>` uses `processContents="strict"`, meaning any elements present in instance documents need to have their own schema definitions. CodeGen ignores this attribute, though it is possible to get a similar effect using a different form of `<xs:any>` handling (discussed below). CodeGen also ignores `<xs:any>` namespace restrictions. Listing 9 uses `namespace="##any"`, meaning elements matching the `<xs:any>` are not namespace-restricted, but if the value had instead been `namespace="##other"`, for example, the result would be the same.

You can use the `any-handling` customization attribute at any level of customizations to select other ways of handling `<xs:any>`. The value `any-handling="discard"` simply ignores the `<xs:any>` in the generated data model and discards any elements corresponding to the `<xs:any>` when unmarshalling occurs. `any-handling="dom"` matches the default handling, using `org.w3c.dom.Element` to represent an element matching the `<xs:any>`. Finally,

`any-handling="mapped"` generates code that requires a global schema definition for each element matching the `<xs:any>` (roughly corresponding to the `processContents="strict"` schema condition). In this last case, the data model uses `java.lang.Object` to represent an element, with the actual runtime type of the object matching the global schema definition.

<xs:simpleType> handling

Like most forms of code generation from schema, CodeGen ignores or only partially handles many aspects of `<xs:simpleType>` definitions. `<xs:simpleType>` restrictions are one example of this limited support. Of the many varieties of `simpleType` restrictions defined by schema (which include length restrictions, value ranges, and even regular expression patterns), only `<xs:enumeration>` restrictions are currently enforced in the generated data model.

`<xs:simpleType>` `<union>`s are also currently ignored by CodeGen. Listing 10 shows the generated code matching an `<xs:union>` reference, along with the original schema fragments matching the code (at the bottom of the listing). You can see in Listing 10 that each of the references to a union type (including both the `TimeCardDuration` type shown in the listing and the `AnyDateTimeType`) is represented by a simple `String` value in the generated code.

Listing 10. <xs:union>-generated code sample and original schema

```
/**
 * Schema fragment(s) for this class:
 * <pre>
 * <xs:element xmlns:ns="http://ns.hr-xml.org/2007-04-15"
 *   xmlns:xs="http://www.w3.org/2001/XMLSchema" name="TimeInterval">
 *   <xs:complexType>
 *     <xs:sequence>
 *       <xs:element type="ns:EntityIdType" name="Id" minOccurs="0"/>
 *       <xs:element type="xs:string" name="StartDateTime"/>
 *       <xs:choice>
 *         <xs:sequence>
 *           <xs:element type="xs:string" name="EndDateTime"/>
 *           <xs:element type="xs:string" name="Duration" minOccurs="0"/>
 *         </xs:sequence>
 *         <xs:element type="xs:string" name="Duration"/>
 *       </xs:choice>
 *     </xs:sequence>
 *   </pre>
 */
public static class TimeInterval
{
    private EntityIdType id;
    private String startDateTime;
    private int choiceSelect = -1;
    private final int END_DATE_TIME_CHOICE = 0;
    private final int DURATION_CHOICE = 1;
    private String endDateTime;
    private String duration;
    private String duration1;
    ...
    ...
}
```

```
<xsd:element name="TimeInterval">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Id" type="EntityIdType" minOccurs="0"/>
      <xsd:element name="StartDateTime" type="AnyDateTimeType"/>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="EndDateTime" type="AnyDateTimeType"/>
          <xsd:element name="Duration" type="TimeCardDuration" minOccurs="0"/>
        </xsd:sequence>
        <xsd:element name="Duration" type="TimeCardDuration"/>
      </xsd:choice>
    </xsd:sequence>
    ...
  </xsd:complexType>
  <xsd:simpleType name="TimeCardDuration">
    <xsd:union memberTypes="xsd:duration xsd:decimal"/>
  </xsd:simpleType>
```

Schema modifications

If you compare the schema fragments embedded in the Javadoc at the top of [Listing 10](#) with the actual schema fragments at the bottom of the listing, you'll see that the `union simpleType` references in the original schema have been replaced by `xs:string` references in the Javadoc version. This is deliberate, and it's representative of several types of transformations to schema structures performed by CodeGen. Some transformations, such as the elimination of `<union>` `simpleTypes` and of `simpleType` restriction facets other than `<xs:enumeration>`, are hardcoded into the CodeGen operation. Other transformations are controlled by customizations. Either way, schema fragments included in Javadocs always show the transformed schema because this is what is actually used to generate the code.

You'll see more types of transformations controlled by customizations in the following sections of the tutorial.

Section 7. Customizing the TimeCard data model

An [example](#) earlier in this tutorial showed some simple CodeGen customizations. Now that you've seen how CodeGen handles the HR-XML `TimeCard` schema with default settings, it's time to look into some more powerful customizations.

Customizing the data model

The data-model code generated by CodeGen using default settings has some weaknesses. For one thing, the schema type names all end with `Type`, and this

carries over to the corresponding generated class names, making the names longer than necessary. The package name generated from the schema namespace, `org.hrxml.ns`, is reasonable, but it would be better to have a package name that indicates the data model is specifically for `TimeCard` documents.

Listing 11 shows another awkward aspect of the generated data-model classes, in which a `java.math.BigInteger` is used to represent an `xs:integer` type. This is the most accurate representation for `xs:integer` using standard Java classes, but `BigInteger` is awkward to use compared to simple `int` primitive or `java.lang.Integer` object types. Unfortunately, schemas are often written using the `xs:integer` type even when `xs:int` would be more appropriate, so developers can get stuck with `BigInteger` values in the generated code. That's definitely the case here, where the actual values allowed for `GenderCode` are all single digits (as shown by the original schema fragment at the bottom of the listing).

Listing 11. xs:integer generation example

```
/**
 * Schema fragment(s) for this class:
 * <pre>
 * <xs:element xmlns:xs="http://www.w3.org/2001/XMLSchema"
 * type="xs:integer"
 * name="GenderCode"/>
 * </pre>
 */
public class GenderCode
{
    private BigInteger genderCode;

    /**
     * Get the 'GenderCode' element value.
     *
     * @return value
     */
    public BigInteger getGenderCode() {
        return genderCode;
    }

    /**
     * Set the 'GenderCode' element value.
     *
     * @param genderCode
     */
    public void setGenderCode(BigInteger genderCode) {
        this.genderCode = genderCode;
    }
}

<xsd:simpleType name="GenderCodeType">
  <xsd:annotation>
    <xsd:documentation>Must conform to ISO 5218 -
Representation of Human Sexes
(0 - Not Known; 1 - Male; 2 - Female; 9 - Not
specified)</xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:integer">
    <xsd:pattern value="[0129]"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 12 shows a customization to improve these aspects of the generated data model. The `package="org.hrxml.timecard"` attribute gives the Java package to be used for the generated classes. The `type-substitutions="xs:integer xs:int"` attribute defines schema type substitutions to be applied by CodeGen, in this case using the `xs:int` type wherever `xs:integer` is referenced in the schema. You can define multiple pairs of substitutions by just adding more type names to the list, using spaces as separators between the pairs as well as between the type names in each pair.

The nested `name-converter` element configures the handling of XML names being converted to Java names. In this case, the `strip-suffixes="Type"` attribute tells CodeGen to remove `Type` wherever it occurs at the end of a name. You can provide multiple alternatives to be stripped with this attribute, as a space-separated list. You can also use a `strip-prefixes` attribute to remove unnecessary leading text from names, along with several other forms of customizations. It's even possible to replace the default name conversion class with your own implementation, if you want to do something really special in the name conversions. For the full details on these `name-converter` options, see the JiBX CodeGen documentation.

Finally, the nested `class-decorator` element adds a decorator to the code generation sequence. In this case, the decorator is a predefined one provided as part of the CodeGen distribution, which adds useful support methods for collection values. Any configured code generation decorators are called in sequence by CodeGen as it constructs the source code for data model classes, and have the opportunity to modify or add to the field, method, and class constructs generated by CodeGen. Each of these constructs is passed to the decorator as an Abstract Syntax Tree (AST) component, using the Eclipse AST implementation. The supplied decorators (including the

`org.jibx.schema.codegen.extend.CollectionMethodsDecorator` decorator used here to add methods, and a `org.jibx.schema.codegen.extend.SerializableDecorator` used to add the `java.io.Serializable` interface and optionally a version id to data model classes) provide examples of working with the Eclipse AST to extend CodeGen, so the source code of these classes is a great starting point for writing your own decorators.

Listing 12. TimeCard customizations example

```
<schema-set xmlns:xs="http://www.w3.org/2001/XMLSchema" package="org.hrxml.timecard"
  type-substitutions="xs:integer xs:int">
  <name-converter strip-suffixes="Type"/>
  <class-decorator class="org.jibx.schema.codegen.extend.CollectionMethodsDecorator"/>
</schema-set>
```

You can try out the Listing 12 customization using the `custgen1` Ant target, or use the `custom1` target to run the complete sequence of generate, compile, bind, and

test. Listing 13 shows the result of applying the customizations. The `TimeCardType` class name has changed to just `TimeCard`, and in addition to the `List` get and set methods there are now added `size`, `add`, `indexed get`, and `clear` methods. In the `GenderCode` class, the `BigInteger` reference has been replaced with a simple `int` primitive type.

Listing 13. Customized data model

```
/**
 * Schema fragment(s) for this class:
 * <pre>
 * ...
 * </pre>
 */
public class TimeCard
{
    ...
    private List<ReportedTime> reportedTimeList = new ArrayList<ReportedTime>();
    ...
    /**
     * Get the list of 'ReportedTime' element items.
     *
     * @return list
     */
    public List<ReportedTime> getReportedTimes() {
        return reportedTimeList;
    }

    /**
     * Set the list of 'ReportedTime' element items.
     *
     * @param list
     */
    public void setReportedTimes(List<ReportedTime> list) {
        reportedTimeList = list;
    }

    /**
     * Get the number of 'ReportedTime' element items.
     * @return count
     */
    public int sizeReportedTime() {
        return reportedTimeList.size();
    }

    /**
     * Add a 'ReportedTime' element item.
     * @param item
     */
    public void addReportedTime(ReportedTime item) {
        reportedTimeList.add(item);
    }

    /**
     * Get 'ReportedTime' element item by position.
     * @return item
     * @param index
     */
    public ReportedTime getReportedTime(int index) {
        return reportedTimeList.get(index);
    }

    /**
     * Remove all 'ReportedTime' element items.
     */
}
```

```

    public void clearReportedTime() {
        reportedTimeList.clear();
    }
    ...
}
/**
 * Schema fragment(s) for this class:
 * <pre>
 * &lt;xs:element xmlns:xs="http://www.w3.org/2001/XMLSchema" type="xs:int"
 *   name="GenderCode"/>
 * </pre>
 */
public class GenderCode
{
    private int genderCode;

    /**
     * Get the 'GenderCode' element value.
     *
     * @return value
     */
    public int getGenderCode() {
        return genderCode;
    }

    /**
     * Set the 'GenderCode' element value.
     *
     * @param genderCode
     */
    public void setGenderCode(int genderCode) {
        this.genderCode = genderCode;
    }
}

```

Eliminating unused definitions

In the first customization example, using the original simple schema, you saw how to control the type definitions included in the generated data model by using `generate-all="false"` to disable generating every global definition and an `includes` list to force generating specific definitions. Listing 14 shows a modified customization for the `TimeCard` schema that adds these attributes, with only the `TimeCard` element to be included in the generated data model (along with everything used by the `TimeCard` representation, of course).

Listing 14. Customization with only TimeCard components

```

<schema-set xmlns:xs="http://www.w3.org/2001/XMLSchema" package="org.hrxml.timecard"
  type-substitutions="xs:integer xs:int" generate-all="false">
  <name-converter strip-suffixes="Type"/>
  <class-decorator class="org.jibx.schema.codegen.extend.CollectionMethodsDecorator"/>
  <schema name="TimeCard.xsd" includes="TimeCard"/>
</schema-set>

```

You can use the `custgen2` Ant target to try this customization with CodeGen, or use the `custom2` target to run the complete sequence of generate, compile, bind, and test. This change reduces the number of top-level classes in the data model

from 15 to 10 — not a bad start on simplifying the data model.

Section 8. Customizing individual components

So far, you've seen only examples of customizations that apply across the entire set of schemas, or to individual schemas. You can also customize CodeGen's handling of specific components *within* a schema definition, including both global definitions and items embedded within the global definitions. The customizations available include eliminating the component from the data model, changing the class or value name used for the component, and changing the schema type of the component.

Customizations to eliminate components from the data model aren't very useful if you control the schema — in that case, it's always going to be simpler to just change the schema directly. But enterprise data-exchange schemas often include specialized components that may not be appropriate for particular applications using those schemas, and these schemas are typically not under your control. Using customizations in this case allows you to simplify your data model without touching the supplied schemas.

Component customizations

Customizations for schema components work by associating a customization element with the schema-definition element representing the component. You can use several different approaches for establishing the association between the customization and the schema element because one approach may be more convenient than another in a particular situation. One part of the association is fixed, though: The name of the customization element must always match the schema component element name. So to customize an `<xs:element>` definition in a schema, for instance, you need to use an `<element customization element (with no namespace)>`.

Listing 15 shows a definition from one of the other schemas referenced by `TimeCard`, which makes a good sample to demonstrate customization of individual components. The `PersonNameType` consists of several simple `xs:string` elements, along with some other elements with complex structure. As it happens, the test documents used in the tutorial code don't include any instances of the `Affix` or `AlternateScript` elements from this type, so they're good candidates to eliminate in order to simplify the generated data model.

Listing 15. PersonName schema

```

<xsd:complexType name="PersonNameType">
  <xsd:sequence>
    <xsd:element name="FormattedName" type="xsd:string" minOccurs="0"/>
    <xsd:element name="LegalName" type="xsd:string" minOccurs="0"/>
    <xsd:element name="GivenName" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="PreferredGivenName" type="xsd:string" minOccurs="0"/>
    <xsd:element name="MiddleName" type="xsd:string" minOccurs="0"/>
    <xsd:element name="FamilyName" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        ...
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="Affix" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        ...
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="AlternateScript" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        ...
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="script" type="xsd:string"/>
</xsd:complexType>

```

Listing 16 shows one way of defining customizations to eliminate the `Affix` and `AlternateScript` elements from the data model. This approach uses a path specification, which is an XPath-like set of directions for navigating the schema definition structure. Path steps are separated by slash (/) characters, and steps matching named components of the schema definition (global type, group, or `attributeGroup` definitions, or element or attribute definitions whether global or not) can use an `[@name=...]` predicate to single out a particular instance of the component type.

Listing 16. Direct customization for unneeded components

```

<schema-set ...>
  <schema name="PersonName.xsd">
    <element path="complexType[@name=PersonNameType]/sequence/element[@name=Affix]"
      ignore="true"/>
    <element path=
      "complexType[@name=PersonNameType]/sequence/element[@name=AlternateScript]"
      ignore="true"/>
  </schema>
</schema-set>

```

In Listing 16, each path is spelled out completely, from the schema level. You can also use wildcards in the paths. The `*` wildcard as a path step matches any single element in the schema definition, while the `**` wildcard matches any number of nested elements in the schema definition. So instead of the path `complexType[@name=PersonNameType]/sequence/element[@name=Affix]`, you could instead use `complexType[@name=PersonNameType]/*/element[@name=Affix]`, or `complexType[@name=PersonNameType]**/element[@name=Affix]`. You couldn't use `**/element[@name=Affix]`, though — CodeGen requires you to

identify explicitly the global definition component involved in any customization as a safeguard to prevent applying the customization incorrectly.

Component customizations can be nested, as long as the nesting matches the schema-definition structure. In this case, each customization just needs to specify its target relative to the containing customization. You can also use a `name="..."` attribute on the customization as an alternative to a `[@name=...]` predicate on the final step of a path, and you can skip the element name on the final step (because it must always be the same as the name of the customization element). You can even avoid using a path completely, instead using nesting combined with a `name` attribute. Listing 17 shows the same customizations as Listing 16, restructured to use this alternative approach:

Listing 17. Nested customization for unneeded components

```
<schema-set ...>
  <schema name="PersonName.xsd">
    <complexType name="PersonNameType">
      <sequence>
        <element name="Affix" ignore="true"/>
        <element name="AlternateScript" ignore="true"/>
      </sequence>
    </complexType>
  </schema>
</schema-set>
```

Simplifying the data model

Besides the `PersonName` components used as examples in the preceding subsection, the `TimeCard` schemas have a number of other complex components that are not used in the sample documents included in this tutorial. By using customizations to eliminate these unused components, you can considerably simplify the generated data model. There are also some cases where the Java value names used by CodeGen don't work well. In particular, cases where the same element name is used repeatedly can result in value names distinguished only by digit suffixes, making it difficult to understand the proper use of the values. See Listing 10 for an example, where a pair of fields named `duration` and `duration1` are included in the generated code. You can use a customization to change one of these names to something more meaningful.

Listing 18 shows the `custom3.xml` file from the `hxml` directory of the code, which includes all these customizations. This intentionally uses a variety of the component-identification approaches discussed in the preceding subsection, with nesting, paths, and paths mixed with names. The `value-name` customization is at the bottom, using a `value-name="simpleDuration"` attribute to change the name used for the second duration to a more descriptive form.

Listing 18. Simplifying and clarifying the TimeCard data model

```

<schema-set xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://ns.hr-xml.org/2007-04-15" package="org.hrxml.timecard"
  type-substitutions="xs:integer xs:int" generate-all="false">
  <name-converter strip-suffixes="Type"/>
  <class-decorator class="org.jibx.schema.codegen.extend.CollectionMethodsDecorator"/>
  <schema name="UserArea.xsd" excludes="UserArea UserAreaType"/>
  <schema name="PersonName.xsd">
    <complexType name="PersonNameType">
      <sequence>
        <element name="Affix" ignore="true"/>
        <element name="AlternateScript" ignore="true"/>
      </sequence>
    </complexType>
  </schema>
  <schema name="TimeCard.xsd" includes="TimeCard">
    <complexType name="TimeCardType">
      <element path="**/element[@name=Allowance]" ignore="true"/>
      <element path="**/element[@name=PieceWork]" ignore="true"/>
      <element path="**/element[@name=TimeEvent]/**/" name="RateOrAmount" ignore="true"/>
      <element path="**/choice/[@name=Duration]" value-name="simpleDuration"/>
    </complexType>
  </schema>
</schema-set>

```

You can use the `custgen3` Ant target to try this customization with CodeGen, or use the `custom3` target to run the complete sequence of generate, compile, bind, and test. The customizations reduce the generated class count to 9 top-level classes and 10 inner classes, for a total of 19. This count is exactly half the number of classes in the original data model generated without using customizations.

Section 9. CodeGen command-line parameters

CodeGen supports several additional command line parameters beyond those used in the tutorial code. Table 1 lists the most important options:

Table 1. CodeGen command-line options

Command	Purpose
-c path	Path to input customizations file
-n package	Default package for no-namespace schema definitions (default is the default package)
-p package	Default package for all schema definitions (default is to use package generated from each schema namespace)
-s path	Schema root directory path (default is current directory)

-t path	Target directory path for generated output (default is current directory)
-v	Verbose-output flag
-w	Wipe all files from target directory before generating output (ignored if the target directory is the same as the current directory)

You can also pass global customizations to CodeGen as command-line parameters, without the need to create a customizations file, by using the special `--` prefix before the customization attribute value. So to set the same global options as used in the [Listing 5](#) customizations, you'd add `--prefer-inline=true`
`--show-schema=false` `--enumeration-type=simple`
`--generate-all=false` to the CodeGen command line. (You can't specify the list of schema components to be included in the generation this way, though, because these are specific to a particular schema.) No quotes are needed for attribute value when you use this technique. If you want to set a customization that takes a list of multiple values, use commas rather than spaces as separators between the individual values. (So to ignore the `Type` and `Group` schema name suffixes, for instance, you'd use the command-line `--strip-suffixes=Type,Group` parameter.)

Section 10. Going further

In this tutorial, you've learned the basics of using JiBX to generate a Java data model from an XML schema definition first and then convert documents matching that schema to and from the data model. You have also seen examples of using customizations to control how the data model is generated. There are many other customizations you can use to control different aspects of the data model, beyond those I've covered in this tutorial. The JiBX documentation provides full details on all these customization options, along with more examples of code generation from schema.

Web service definitions are one of the main uses of XML schemas. JiBX can currently be used within the Apache Axis2, Apache CXF, XFire, and Spring-WS Web services stacks, and it also supports its own lightweight Web services engine in the form of JiBX/WS. You can use the code-generation-from-schema features discussed in this tutorial within any of these Web services stacks, though you'll currently need to extract the schema definition from the Web Services Description Language (WSDL) service definition before it can be generated. You also need to go through

additional steps for each stack to get to a working Web service. Future releases of JiBX will simplify the process of creating Web services implementations, so check the documentation in your JiBX distribution to find out about any new features in this area.

Downloads

Description	Name	Size	Download method
Sample code for this tutorial	j-jibx2.zip	27KB	HTTP

[Information about download methods](#)

Resources

Learn

- [JiBX: Binding XML to Java Code](#): Visit the JiBX Web site for API documentation, project news, and other resources.
- ["Transform Java classes into Web services using Axis2 and JiBX"](#) (Tyler Anderson, developerWorks, March 2007): Read this two-part article to find out how to use JiBX to define a Web service from existing Java classes.
- [JAXB Reference Implementation](#): Learn about the JAXB 2.0 Java standard for XML data binding, with runtime support bundled in the Java 6 and later JRE.
- [Data binding with Castor](#) (Brett D. McLaughlin, developerWorks): Learn about another open source data-binding tool in this article series.
- ["Schema for Web Services — Part I: Basic Datatypes"](#) (Dennis Sosnoski, InfoQ, January 2009): Learn about some of the schema limitations of data-binding tools (and some of the issues with schema itself) in this series.
- [HR-XML Consortium](#): Find out more about the HR-XML Consortium standards for human resources data exchange, including the TimeCard schema used in the tutorial.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- [JiBX](#): Download the JiBX framework.
- [Ant](#): Download Apache Ant.
- [Sun JDK 1.5 or later](#): You'll need at least version 1.5.0_09 to follow the examples in this tutorial.
- [IBM® developer kits](#): IBM Java developer kits are available for AIX and Linux.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Dennis Sosnoski

Dennis Sosnoski is a consultant and training facilitator specializing in Java-based SOA and Web services. His professional software development experience spans over 30 years, with the last decade focused on server-side XML and Java technologies. Dennis is the lead developer of the open source JiBX XML data binding tool, as well as a committer on the Apache Axis2 Web services framework. He was also one of the expert group members for the JAX-WS 2.0 and JAXB 2.0 specifications. See [his Web site](#) for information on his training and consulting services.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.