

JiBX 1.2, Part 1: Java code to XML schema

Improve schema quality with custom conversion of Java data models to and from XML documents

Skill Level: Intermediate

[Dennis Sosnoski](#)

Lead consultant

Sosnoski Software Associates Ltd

03 Mar 2009

XML schema definitions are the basis for many types of data exchanges, including most forms of Web services. But XML Schema is a complex standard, and most tools for creating and modifying schema definitions are not as powerful or easy to use as those for working with Java™ code. The new features of JiBX 1.2 that you'll learn about in this tutorial — Part 1 of a two-part [series](#) — let you start from Java code and easily generate quality schema definitions to match your data structures. You can then use the schemas directly, whether you use JiBX data binding or not.

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 XML schema definitions easily from existing Java code 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 2](#) covers the flip side of starting from XML schema definitions and generating Java code.

Objectives

This tutorial guides you through the process of using JiBX to generate XML schema definitions from Java code. You'll first see how to start with a simple Java data model and generate a default schema matching that model. From that base, you'll learn how you can easily apply a range of customizations to control the actual values used from your Java classes and how they're accessed, whether they are required or optional, names and namespaces used in the XML, and even the generated schema definition's structure. Along the way, you'll see how JiBX adds value to your generated schemas by leveraging your investment in Javadocs to document the schema definition automatically. After reading this tutorial and working through the supplied examples, you will be able to use JiBX to generate quality XML schema definitions from your own Java data-structure classes.

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. Getting started

In this section, you'll get an overview of JiBX, and you'll install JiBX and the tutorial sample code.

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 first type of generation: going from Java code to a schema definition.

Installing JiBX

You 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 `dwcode1` directory from inside the ZIP file to the root directory of your JiBX distribution). This should create a `dwcode1` subdirectory in the `jibx` directory, with the example files (including `build.xml`, `custom1.xml`, and others) inside that `dwcode1` subdirectory.

The sample code includes an Apache 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 set a `JIBX_HOME`

environmental variable to the path to your JiBX installation. Alternatively, you can edit the build.xml file inside the sample-code directory and uncomment the line near the top of the file that sets the `jibx-home` property directly.

Section 3. Generating default binding and schema from code

Non-Java 5 usage

The tutorial example code uses Java 5 typed collection and enum features, but JiBX itself is fully compatible with older Java versions. The standard JiBX runtime works with 1.3 and later JVMs and can also be built for J2ME compatibility. Most of JiBX's other components, including BindGen, can be run on 1.4.1 and later JVMs. The BindGen documentation in the JiBX download includes an example showing how customizations can supply BindGen with the equivalent of typed collections when you use pre-Java 5 code.

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

Introducing the Java example code

As an example, I'll start with the Java code for a set of bean-style (private fields, public get and set access methods) classes used to represent an order from an online store. [Listing 1](#) shows an abbreviated version of the code, with most of the get/set methods left out. The full sample code is in the sample code's src directory.

Listing 1. Base Java code

```
package org.jibx.starter;

/**
 * Order information.
 */
public class Order
{
    private long orderNumber;
    private Customer customer;

    /** Billing address information. */
    private Address billTo;
    private Shipping shipping;

    /** Shipping address information. If missing, the billing address is also used as the
     * shipping address. */
    private Address shipTo;
```

```

private List<Item> items;

/** Date order was placed with server. */
private Date orderDate;

/** Date order was shipped. This will be <code>>null</code> if the order has not
yet shipped. */
private Date shipDate;
private Float total;

public long getOrderNumber() {
    return orderNumber;
}
...
}
/**
 * Customer information.
 */
public class Customer
{
    private long customerNumber;

    /** Personal name. */
    private String firstName;

    /** Family name. */
    private String lastName;

    /** Middle name(s), if any. */
    private List<String> middleNames;
    ...
}
/**
 * Address information.
 */
public class Address
{
    /** First line of street information (required). */
    private String street1;

    /** Second line of street information (optional). */
    private String street2;
    private String city;

    /** State abbreviation (required for the U.S. and Canada, optional otherwise). */
    private String state;

    /** Postal code (required for the U.S. and Canada, optional otherwise). */
    private String postCode;

    /** Country name (optional, U.S. assumed if not supplied). */
    private String country;
    ...
}
/**
 * Order line item information.
 */
public class Item
{
    /** Stock identifier. This is expected to be 12 characters in length, with two
leading alpha characters followed by ten decimal digits. */
    private String id;

    /** Text description of item. */
    private String description;

    /** Number of units ordered. */
    private int quantity;
}

```

```
    /** Price per unit. */
    private float price;
    ...
}
/**
 * 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 enum Shipping
{
    STANDARD_MAIL, PRIORITY_MAIL, INTERNATIONAL_MAIL, DOMESTIC_EXPRESS,
    INTERNATIONAL_EXPRESS
}
```

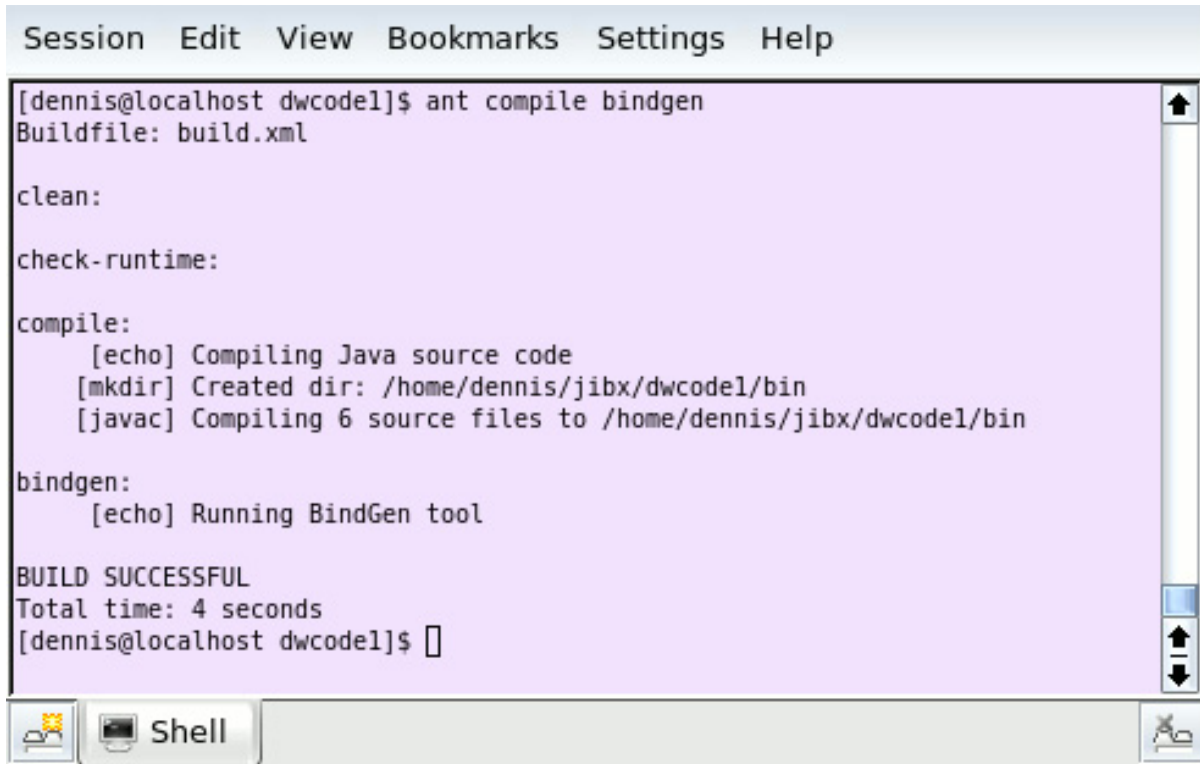
Generating the default binding and schema

To generate a JiBX binding and XML schema from some Java classes, you first need to compile the classes, then run the `org.jibx.binding.generator.BindGen` 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 Ant.

The tutorial download includes an Ant `build.xml` script with the `compile` target to compile the example code and the `bindgen` target to run the BindGen program on the compiled code.

To try this out, open a console in the `dwcode1` directory of the installed download and type `ant compile bindgen`. 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 dwcode1]$ ant compile bindgen
Buildfile: build.xml

clean:

check-runtime:

compile:
  [echo] Compiling Java source code
  [mkdir] Created dir: /home/dennis/jibx/dwcode1/bin
  [javac] Compiling 6 source files to /home/dennis/jibx/dwcode1/bin

bindgen:
  [echo] Running BindGen tool

BUILD SUCCESSFUL
Total time: 4 seconds
[dennis@localhost dwcode1]$
```

You can also run BindGen directly from the console. To do this, you need to include `jibx-tools.jar` in your Java classpath, along with the path for the compiled class files you'll use as input to the generation. If you want to duplicate the effect of the supplied Ant `bindgen` target, you also need to pass the root directory for the source files of your classes in the command line. Finally, you need to list the root class(es) you want used for generation. On UNIX® and Linux® systems, the Java command line (which consists of a single line, even if it appears as wrapped in your display) to duplicate the Ant `bindgen` target from a console in the `dwcode1` directory (assuming you've followed the recommended installation instructions) is:

```
java -cp ../lib/jibx-tools.jar:bin org.jibx.binding.generator.BindGen
-s src org.jibx.starter.Order
```

On Windows, the command (a single line, regardless of its appearance here) is:

```
java -cp ../lib\jibx-tools.jar;bin org.jibx.binding.generator.BindGen
-s src org.jibx.starter.Order
```

Many other options can be passed to BindGen from the command line. You'll look into those later in the tutorial. Right now, let's look at the generated schema.

Generated artifacts

Listing 2 shows the generated schema output from BindGen (as starter.xsd), slightly reformatted to fit the page width and with some of the details removed. The start tag for the schema definition matching each Java class is displayed in bold to emphasize the structure.

Listing 2. Generated 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:complexType name="address">
  <xs:annotation>
    <xs:documentation>Address information.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element type="xs:string" name="street1" minOccurs="0">
      <xs:annotation>
        <xs:documentation>First line of street information (required).</xs:documentation>
      </xs:annotation>
    </xs:element>
    ...
  </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 name="customer" minOccurs="0">
      <xs:complexType>
        ...
      </xs:complexType>
    </xs:element>
    <xs:element type="tns:address" name="billTo" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Billing address information.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="shipping" minOccurs="0">
      <xs:simpleType>
        <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:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element type="tns:address" name="shipTo" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Shipping address information. If missing, the billing address is
          also used as the shipping address.</xs:documentation>
      </xs:annotation>
    </xs:element>
<xs:element name="item" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:string" name="id" minOccurs="0">
        <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:annotation>
  </xs:element>
  <xs:element type="xs:string" name="description" minOccurs="0">
    <xs:annotation>
      <xs:documentation>Text description of item.</xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:sequence>
<xs:attribute type="xs:int" use="required" name="quantity">
  <xs:annotation>
    <xs:documentation>Number of units ordered.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<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:element>
</xs:sequence>
<xs:attribute type="xs:long" use="required" name="orderNumber"/>
<xs:attribute type="xs:date" name="orderDate">
  <xs:annotation>
    <xs:documentation>Date order was placed with server.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute type="xs:date" name="shipDate">
  <xs:annotation>
    <xs:documentation>
      <![CDATA[Date order was shipped. This will be <code>null</code> if the order
        has not yet shipped.]]></xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>
</xs:schema>

```

By default, BindGen generates a schema with nested `complexType` and `simpleType` definitions for types that are used only once and separate definitions for types that are used more than once. In this case, the nested style results in a schema with just three global definitions: the `address` and `order` complex types, and the `order` element. The `Address` class is used in two places within the `Order` class (for billing and shipping addresses), which is why that class is represented by a separate global definition in the schema. (Schema allows you to reuse definitions only if they're global.) The other classes in the Java data structure (`Customer`, `Item`, and `Shipping`) are each referenced at only one point in the `Order` class, so the corresponding type definitions are embedded directly within the `order` schema type definition.

One of the nicer BindGen features is that it can generate schema documentation from Javadocs in the input classes. You can see the schema documentation in [Listing 2](#) for each field with a Javadoc in [Listing 1](#), and for each global type corresponding to a class with a Javadoc. Not all forms of Javadocs can be matched with schema components by BindGen's default handling — and some Javadocs, such as those on "get" access methods, may look odd when converted to schema documentation — but the resulting schema documentation can be extremely useful for clarifying the proper use of the XML representation. You can even define your own formatter class for Javadocs used as schema documentation, if you want to

make some changes to the text in the process of conversion (such as stripping "Get the ..." lead sentences from "get" method Javadocs).

This Javadoc conversion feature works only if you have the source code available for the classes and provide an argument to BindGen telling it the root directory path (or paths). In the command-line samples I provided earlier (see [Generating the default binding and schema](#)), the source path is supplied as `-s src`.

Generated JiBX binding

Besides the schema definition, BindGen also produces a JiBX binding definition (as the binding.xml file, in this case) that tells the JiBX binding compiler how to convert between the Java classes and XML. That binding definition is actually the main output of BindGen, with the schema generated from the binding. 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 BindGen binding and schema 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

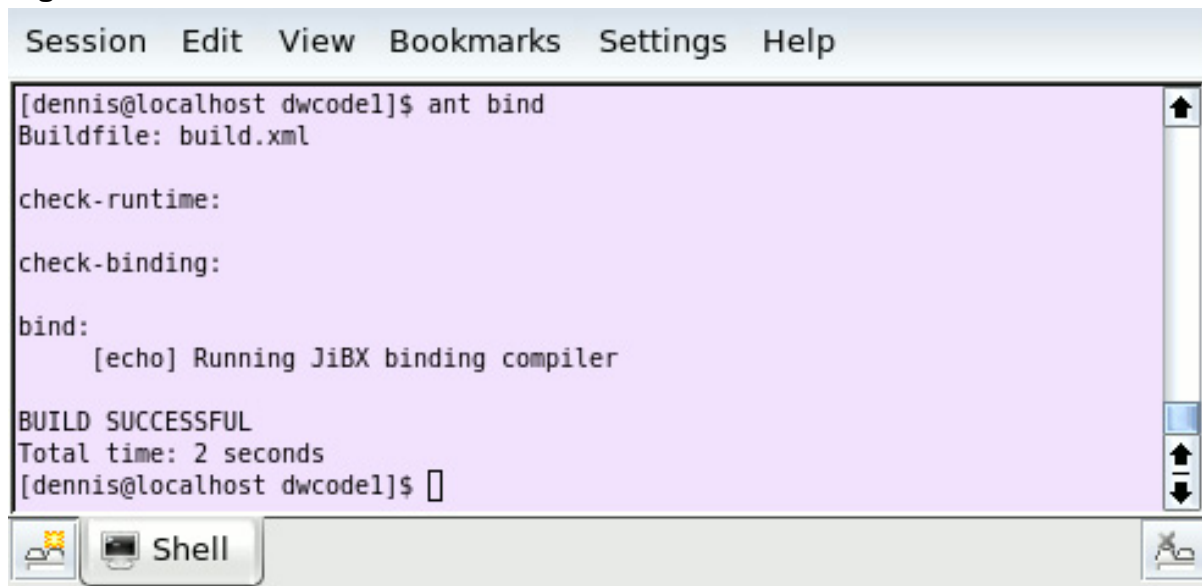
To use the generated binding definition in working with XML documents, you first need to run the JiBX binding compiler tool. The binding compiler adds bytecode to your compiled class files that actually implements the conversions to and from XML, as specified by the binding definition. You must run the binding compiler each time you recompile your Java classes or modify the binding definition, so it's generally 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 run it at run time 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, using the build.xml's `bind` target. [Figure 2](#) shows the output that you should see when you run this target, assuming you've already run the `compile` and `bindgen` targets. (You can also run all three targets in sequence by listing them in order on the command line: `ant compile bindgen bind`.)

Figure 2. Ant build bind task



```
Session Edit View Bookmarks Settings Help
[dennis@localhost dwcode1]$ ant bind
Buildfile: build.xml

check-runtime:

check-binding:

bind:
  [echo] Running JiBX binding compiler

BUILD SUCCESSFUL
Total time: 2 seconds
[dennis@localhost dwcode1]$
```

Using JiBX at run time

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

Listing 3. Default binding test document

```
<order orderNumber="12345678" orderDate="2008-10-18" shipDate="2008-10-22"
  xmlns="http://jibx.org/starter">
  <customer customerNumber="5678">
    <firstName>John</firstName>
    <lastName>Smith</lastName>
  </customer>
  <billTo>
    <street1>12345 Happy Lane</street1>
    <city>Plunk</city>
    <state>WA</state>
    <postCode>98059</postCode>
    <country>USA</country>
  </billTo>
  <shipping>PRIORITY_MAIL</shipping>
  <shipTo>
    <street1>333 River Avenue</street1>
    <city>Kirkland</city>
    <state>WA</state>
    <postCode>98034</postCode>
    <country>USA</country>
  </shipTo>
  <item quantity="1" price="5.99">
    <id>AC4983498512</id>
```

```

    <description>Left-handed widget</description>
  </item>
  <item quantity="2" price="9.50">
    <id>IW2349050499</id>
    <description>Right-handed widget</description>
  </item>
  <item quantity="1" price="8.95">
    <id>RC3000488209</id>
    <description>High-speed MP3 rewinder</description>
  </item>
</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);
        }
    }
}

```

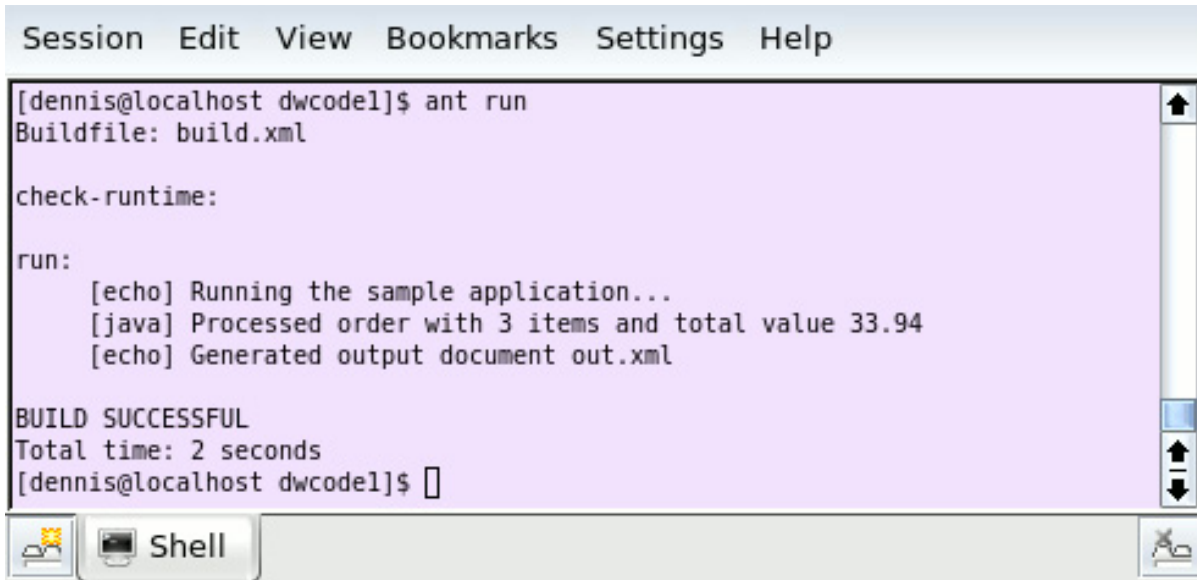
```

    } 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 dwcode1]$ 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: 2 seconds
[dennis@localhost dwcode1]$

```

You can inspect the generated `out.xml` file to see how it matches the original input shown in [Listing 3](#). Aside from namespace declaration and attribute order and the added `total` attribute in the output (computed and set by the test program), the two documents should be identical. This won't always be the case! JiBX, like most forms of data binding, works with only the "significant" data in the document, meaning those values being used by your application. Nonsignificant parts of the document (such as whitespace within a start or end tag, text between elements, and comments) are lost when you unmarshal a document. Part of the reason the input and output documents are so similar in this case is that the [Listing 4](#) code sets the output format to use indentation of two spaces per element nesting level, matching the input document.

Sharp observers will notice one difference between the input and output that *seems* significant, in the item-list portion of the output document, shown in [Listing 5](#):

Listing 5. Item list from output document

```

<item quantity="1" price="5.99">
  <id>AC4983498512</id>
  <description>Left-handed widget</description>

```

```
</item>
<item quantity="2" price="9.5">
  <id>IW2349050499</id>
  <description>Right-handed widget</description>
</item>
<item quantity="1" price="8.95">
  <id>RC3000488209</id>
  <description>High-speed MP3 rewinder</description>
</item>
```

If you compare the line shown in bold in [Listing 5](#) with the corresponding line in the [Listing 3](#) original document, you can see that the price has changed from 9.50 to 9.5, with the trailing zero removed. This is not an error, though. The representation used for the price value is a `float`, and in terms of both Java and XML schema, leading zeros before the decimal point and trailing zeros after the decimal point are not significant for a `float`.

Section 5. BindGen customizations

In this section, you'll learn how to customize BindGen operation to control the XML representation of data, change the style of names and namespaces, and control some aspects of schema structure.

Customizing BindGen operation

BindGen supports extensive customizations for all aspects of binding and schema generation. The set of customizations to be applied are passed to BindGen as an XML document, with nested elements that mirror the structure of your Java code. Listing 6 gives a simple example:

Listing 6. Simple customizations example

```
<custom>
  <package name="org.jibx.starter" property-access="true">
    <class name="Address" includes="street1 street2 city state
postCode country"/>
    <class name="Item" excludes="description"/>
  </package>
</custom>
```

This example works with a single Java code package, so [Listing 6](#) uses just one `<package>` element child of the root `<custom>` element. `<package>` and `<class>` customization elements use name attributes that are relative to any enclosing `<package>` element, so in the [Listing 6](#) example only a simple class name is required for each `<class>` element. `<package>` elements can be nested

inside one another, so if you're dealing with classes across a hierarchy of packages, it's easy to handle any options using nested `<package>` elements. The nested structure is especially convenient because many customization attributes are inherited through the element nesting, as I'll discuss later in this section. Using nesting is optional, though — you can skip the `<package>` elements completely and use `<class>` elements with fully qualified class names directly, if you prefer.

A customization file is passed to BindGen as a command-line parameter, using the form `-c file-path`. Customizations are always optional, and you never need to use a customization file unless you want to change the default BindGen behavior.

Controlling how BindGen works with your code

BindGen does a reasonable job with its default handling of Java classes, but there are limits on what can be done without user guidance. For example, the default handling is to include every field in the XML representation except for static, transient, or final fields. This approach works fine for classes that represent simple data objects; however, if your classes include state information or computed values, you might end up with an XML representation that includes values you'd rather not expose outside the class.

Customizations allow you to control in two ways what BindGen uses in the XML representation. First, you can easily switch to using bean-style `getXXX()`, `setXXX()`, and `isXXX()` access methods rather than directly accessing fields. Second, you can choose either to list the values you want to include in the XML representation for a class or to list the values you want to exclude. The [Listing 6](#) customizations demonstrate both these techniques.

One-way conversions

This tutorial uses JiBX for two-way conversions between Java data structures and XML documents. BindGen also supports generating one-way conversions, which can be more convenient in some cases. For instance, value object classes are generally immutable and define only `final` fields and read ("get") access methods. This makes value object classes difficult to use for two-way conversions with BindGen. If you instead tell BindGen to generate an output-only conversion, it will happily work with either the fields or the properties, whichever you prefer. To generate a one-way conversion, use `direction="output"` or `direction="input"` on the root `<custom>` element of the customizations.

The `<package>` element in [Listing 6](#) uses a `property-access="true"` attribute to tell BindGen to look for bean-style properties defined by public, nonstatic access methods, rather than fields, when determining which values to include in the XML representation. This attribute is an example of an inherited customization setting, which applies to everything nested inside the element with the attribute. In the

[Listing 6](#) example, the setting applies to the two nested `<class>` elements. It also applies to all other classes in the package, even though no `<class>` customization elements are present for those other classes. Besides determining how values are found from the class representation, the `property-access` setting also controls how the values are accessed by the generated JiBX code — directly from the fields or by calling the access methods.

The first `<class>` element in [Listing 6](#) uses an `includes="street1 street2 city state postCode country"` attribute to list the specific values from the class that BindGen needs to include in the XML representation. The second `<class>` element in the listing uses an `excludes="description"` attribute, which lists values to be excluded from the XML representation. Because you're using property access rather than field access for values, these names are matched with properties defined by `get/set/is` access methods. If you were using fields, the value names would be matched with field names.

The `custgen1` Ant target runs BindGen using the [Listing 6](#) customizations. This target is an alternative to the `bindgen` target shown earlier, so to run the complete build you'd use the Ant command line: `ant compile custgen1 bind`. [Listing 7](#) shows the item type definition from the schema generated when this target is run:

Listing 7. Customized schema output fragment

```
<xs:element name="item" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:string" name="id" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:int" use="required" name="quantity"/>
    <xs:attribute type="xs:float" use="required" name="price"/>
  </xs:complexType>
</xs:element>
```

You can see from [Listing 7](#) that the `description` value is now missing from the schema representation, as specified by the customizations.

You need to use a different XML document as input when using this customization because the `<description>` element present in the original XML is no longer used. The Ant `run1` target runs the test program using a `data1.xml` input and generating out1.xml as output. You can also run the entire sequence, from compiling the source code to running the test program, with the `custom1` Ant target.

Controlling instance creation

Instance creation during unmarshalling can also be controlled using customizations. By default, JiBX expects to have a no-argument (default) constructor defined for each class (which the Java compiler generates automatically, if you don't define any other constructors). When a new instance of the class is needed during unmarshalling, JiBX uses that no-argument constructor to create the instance. If

some of your classes only define constructors with arguments, you can use BindGen customizations to make them usable by JiBX. One way of doing this is by defining a factory method to be used for creating instances of the class, using a `factory="xxx"` attribute on the `<class>` customizations element to supply the fully qualified (with leading package and class information) name of a static method returning an instance of the class. You can also just add `add-constructors="true"` on the root `<custom>` element, which will generate a binding that adds no-argument constructors to classes as needed. This second approach works fine for normal data classes, but you'll still need to supply a factory for any interface or abstract classes (which can never be constructed directly). Of course, if you're generating an output-only binding (see the [One-way conversions](#) sidebar), instance creation is not an issue and you don't need to be concerned about constructors.

Other customizations for working with input classes

BindGen supports many other customizations used to control how it works with the Java input classes. For example, if you use a naming convention for your Java field names, you can configure BindGen to ignore particular prefix or suffix strings by using `strip-prefixes` or `strip-suffixes` attributes. (So to ignore leading `m_` and `s_` prefixes, for instance, you'd use `strip-prefixes="m_ s_"`). These modifications to field names are applied before the fields are matched to value names used in other customizations and naturally also apply when XML names are generated from the field names.

You can also customize the handling of individual fields or bean properties within a class, using nested `<value>` elements. You'll see how to work with these value customization elements in a later example.

Controlling the XML representation

Besides controlling how BindGen interprets your Java code, you can use customizations to control the XML representation of data. These XML customizations include the actual representation (as an element or an attribute) of values, the order and names of elements and attributes, whether a value is optional or required, and more.

The earlier [Listing 6](#) customization example demonstrates one XML customization in the form of the `includes="street1 street2 city state postCode country"` attribute used on the first `<class>` element. I discussed how this selects the values from the class that are included in the XML representation. It also controls the XML representation in that the order in which the values are listed becomes the order in which they're expressed in the XML representation. That's not a significant issue for attributes (which are always considered unordered in XML), but it is important for elements.

If you *don't* specify the order of values by using an `includes` attribute on the `<class>` customization, BindGen generates the values in the order they're delivered by using Java reflection on the classes. For most Java compilers and JVMs, this reflection order will match the order of the definitions in the Java source code. However, Java compilers and JVMs are not *required* to preserve this order from the source code, so some compilers or JVMs might cause BindGen to change the order of child elements. If you want to be certain the XML representation will always be the same no matter what Java compiler and JVM are used, the `includes` attribute gives you an easy way to fix the order.

You can also control the XML representation of a value using the `includes` attribute. BindGen allows leading flag characters to be used on each name in the list to indicate the representation: `@` for an attribute, and `/` for an attribute. So if you change the [Listing 6](#) customization to `includes="street1 street2 city state @postCode country"`, the representation of the post code value changes from a child element to an attribute.

Controlling required status

Controlling whether a value is considered optional or required is another easy customization using the `<class>` element's `requireds` and `optionals` attributes. As with the `includes` attribute, you can precede names in the `requireds` and `optionals` lists by a flag character to indicate whether they should be expressed as a child element or an attribute.

By default, BindGen treats all primitive values and simple object values (classes with direct XML equivalent types, other than `String`) as attributes and treats all complex object values as child elements. All primitive values are treated as required, and all object values as optional. In addition to overriding these defaults at the `<class>` customization level by using the `includes`, `requireds`, and `optionals` elements, you can change the default representation to use elements for all values by setting a `value-style="element"` attribute at any level of customizations (`<custom>`, `<package>`, or `<class>` element). You can also use the `require` attribute to control which types should be treated as required values in the XML:

- `require="none"` makes everything optional.
- `require="primitives"` is the default, making only primitive values required.
- `require="objects"` inverts the default, making primitives optional and object types required.
- `require="all"` treats all values as required by default.

Listing 8 shows the `custom2.xml` customization file from the tutorial download's `dwcode1` directory, illustrating several of the features I've discussed in this section:

Listing 8. Customizing order, required status, and representation

```
<custom property-access="true">
  <package name="org.jibx.starter">
    <class name="Address" includes="street1 street2 city @state @postCode country"
      requireds="street1 city"/>
    <class name="Customer" includes="customerNumber firstName lastName"
      requireds="lastName firstName /customerNumber"/>
    <class name="Item" excludes="description" requireds="@id quantity price"/>
    <class name="Order" requireds="/orderNumber customer billTo shipping orderDate"/>
  </package>
</custom>
```

You can try this set of customizations by using the Ant `custgen2` target (ant compile custgen2 bind, to run the complete build). Listing 9 shows selected portions of the generated schema using these customizations, showing the resulting order, required status (with `minOccurs="0"` for optional elements, which are required by default in schema, and `use="required"` for required attributes, which are optional by default in schema), and element or attribute representation:

Listing 9. Schema generated using customizations

```
<xs:complexType name="order">
  <xs:annotation>
    <xs:documentation>Order information.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element type="xs:long" name="orderNumber">
      <xs:annotation>
        <xs:documentation>Get the order number.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="customer">
      <xs:complexType>
        <xs:sequence>
          <xs:element type="xs:long" name="customerNumber"/>
          <xs:element type="xs:string" name="firstName"/>
          <xs:element type="xs:string" name="lastName"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    . . .
  </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>
<xs:element type="tns:order" name="order"/>
<xs:complexType name="address">
  <xs:annotation>
    <xs:documentation>Address information.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element type="xs:string" name="street1"/>
    <xs:element type="xs:string" name="street2" minOccurs="0"/>
    <xs:element type="xs:string" name="city"/>
    <xs:element type="xs:string" name="country" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute type="xs:string" name="state"/>
  <xs:attribute type="xs:string" name="postCode"/>
</xs:complexType>
```

After compiling the binding using the `bind` Ant task, you can test this using the `run2` task which takes the `data2.xml` test document as input and generates an output `out2.xml`. You can also run the complete sequence from `compile` to test with the `custom2` target. Listing 10 shows the test document:

Listing 10. Test document matching customizations

```
<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="8394983498512"/>
  <item quantity="2" price="9.50" id="9912349050499"/>
  <item quantity="1" price="8.95" id="1293000488209"/>
</order>
```

Compare [Listing 10](#) with the original test document, shown in [Listing 3](#), to see how your customizations have changed the XML representation of the data (including changing the form of the line item representations to empty elements, a much more compact representation than the original).

Controlling names and namespaces

Java names customarily use a "camelcase" style: names are mostly lowercase, but the initial letter of each word is uppercased. For field or property names, the initial uppercase applies only to words after the first (resulting in names like `postCode` and `customerNumber`). XML names are not as standardized, and several different styles are commonly used. These include the camelcase style with initial lowercase (the Java field and property name style), camelcase with an initial uppercase character (the Java class name style), hyphen-separator (words separated by hyphens) style, dot-separator (words separated by periods) style, and underscore-separator (words separated by underscores) style.

BindGen assumes the camelcase style for XML names by default, but you can easily change this by setting a `name-style` attribute at any level of customization (`<custom>`, `<package>`, or `<class>` element). The allowed values for this attribute match the different XML styles listed above:

- camel-case (the default)
- upper-camel-case
- hyphenated
- dotted
- underscored

You can also set the XML name for a value by using a customization specifically for that value. Using an individual value customization gives you full control over both how that value will be accessed and how it will be represented in XML. Listing 11 gives a couple of examples of using customization elements for individual values, based on the same example code you've seen in the earlier examples:

Listing 11. Customizing names and namespace

```
<custom property-access="true" name-style="hyphenated" namespace="http://jibx.org/custom"
  namespace-style="fixed">
  <package name="org.jibx.starter">
    <class name="Address" includes="street1 street2 city @state @postCode country"
      requireds="street1 city"/>
    <class name="Customer" includes="customerNumber firstName lastName"
      requireds="lastName firstName /customerNumber"/>
    <class name="Item" excludes="description" requireds="@id quantity price"/>
    <class name="Order" requireds="orderNumber customer billTo shipping orderDate">
      <value property-name="orderNumber" element="order-num"/>
      <value property-name="items" item-name="line-item" element="order-items"/>
    </class>
  </package>
</custom>
```

The first value customization in Listing 11 is for the `orderNumber` property, inside the `<class name="Order" . . . >` element. By using an `element="order-num"` attribute, the `orderNumber` customization tells BindGen to express the value as an element, rather than the default attribute form used for a primitive value. The second customization is for the `items` collection property. This customization uses both `item-name` and `element` attributes. The `item-name` attribute controls the name used for the individual values represented by the collection, while the `element` attribute forces the use of the supplied name as a wrapper element around the values in the collection.

XML without namespaces

All the tutorial examples use XML namespaces because the use of namespaces is generally considered a best practice for data exchange. If you want to work with XML without namespaces, you can use a `namespace-style="none"` attribute at any level of the customizations to turn off namespaces completely for all nested components.

The [Listing 11](#) customizations also define the namespace to be used in XML

documents. The previous examples rely on the default BindGen handling of namespaces, which is to derive the namespace URI used in the XML representation of Java code from the Java package. This default handling converted the `org.jibx.starter` package to the namespace URI `http://jibx.org/starter`. In [Listing 11](#), the namespace is customized by adding a pair of attributes — `namespace="http://jibx.org/custom"` and `namespace-style="fixed"` — on the root `<custom>` element. The first of these attributes defines the base namespace, while the second prevents the normal behavior of modifying the namespace based on the Java package. These attributes are both inherited through nesting of customization elements, so they could just as easily have been placed on the `<package>` element instead of the `<custom>` element.

You can try out the [Listing 11](#) customizations by using the Ant `custgen3` target for the binding and schema generation, and the `run3` target to run a test (after using the standard `bind` target to run the JiBX binding compiler — or just use the `full3` target to do the whole sequence). [Listing 12](#) shows the input document used with the test code:

Listing 12. XML sample with customized names and namespace

```
<order order-date="2008-10-18" ship-date="2008-10-22" xmlns="http://jibx.org/custom">
  <order-num>12345678</order-num>
  <customer>
    <customer-number>5678</customer-number>
    <first-name>John</first-name>
    <last-name>Smith</last-name>
  </customer>
  <bill-to state="WA" post-code="98059">
    <street1>12345 Happy Lane</street1>
    <city>Plunk</city>
    <country>USA</country>
  </bill-to>
  <shipping>PRIORITY_MAIL</shipping>
  <ship-to state="WA" postCode="98034">
    <street1>333 River Avenue</street1>
    <city>Kirkland</city>
  </ship-to>
  <order-items>
    <line-item quantity="1" price="5.99" id="AC4983498512"/>
    <line-item quantity="2" price="9.50" id="IW2349050499"/>
    <line-item quantity="1" price="8.95" id="RC3000488209"/>
  </order-items>
</order>
```

If you compare [Listing 12](#) with the [Listing 10](#) sample, you'll see how the representation has been changed by the latest customizations.

Customizing schema representations

You've now seen how BindGen customizations can change the XML representation of your Java data. Customizations can also be used to control some aspects of the

actual schema structure.

Recall that BindGen defaults to using nested definitions in preference to global types and elements. If you review the [Listing 9](#) generated schema, you'll see this nesting structure. The schema uses only three global definitions: the `address` and `order` complex types, and the `order` element. The other classes in the Java data structure (`Customer`, `Item`, and `Shipping`) are each referenced at only one point in the `Order` class, so the corresponding type definitions are embedded directly within the `order` schema type definition.

You can change the schema style by using a `force-mapping="true"` attribute on any of the nesting customization elements. [Listing 13](#) shows the `custom4.xml` customizations file, which adds this change to the `custom2.xml` customizations matching the [Listing 9](#) generated schema:

Listing 13. Customization for schema structure

```
<custom property-access="true" force-mapping="true">
  <package name="org.jibx.starter">
    <class name="Address" includes="street1 street2 city @state @postCode country"
      requireds="street1 city"/>
    <class name="Customer" includes="customerNumber firstName lastName"
      requireds="lastName firstName /customerNumber"/>
    <class name="Item" excludes="description" requireds="@id quantity price"/>
    <class name="Order" requireds="/orderNumber customer billTo shipping orderDate"/>
  </package>
</custom>
```

[Listing 14](#) shows the resulting schema structure (generated as `starter.xsd` by running the `custgen4` Ant target). This version of the schema represents the same XML document structure as the [Listing 9](#) schema but includes separate type definitions matching each Java class.

Listing 14. Customized schema structure

```
<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: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:attribute type="xs:int" use="required" name="quantity"/>
    <xs:attribute type="xs:float" use="required" name="price"/>
  </xs:complexType>
</xs:schema>
```

```

</xs:complexType>
<xs:element type="tns:order" name="order"/>
<xs:complexType name="address">
  <xs:annotation>
    <xs:documentation>Address information.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element type="xs:string" name="street1"/>
    <xs:element type="xs:string" name="street2" minOccurs="0"/>
    <xs:element type="xs:string" name="city"/>
    <xs:element type="xs:string" name="country" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute type="xs:string" name="state"/>
  <xs:attribute type="xs:string" name="postCode"/>
</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:element type="xs:string" name="firstName"/>
    <xs:element type="xs:string" name="lastName"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="order">
  <xs:annotation>
    <xs:documentation>Order information.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element type="xs:long" name="orderNumber">
      <xs:annotation>
        <xs:documentation>Get the order number.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element type="tns:customer" name="customer"/>
    <xs:element type="tns:address" name="billTo"/>
    <xs:element type="tns:shipping" name="shipping"/>
    <xs:element type="tns:address" name="shipTo" minOccurs="0"/>
    <xs:element type="tns: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>
</xs:schema>

```

Schemas of the type shown in [Listing 14](#), called "Venetian Blind" style schemas, are popular for use with complex XML structure definitions. By separating out each type definition, this schema style lets you easily reuse component structures when modifying or extending a schema. The flexibility of the Venetian Blind style is probably not important if you just plan to use your Java code as the base for any further changes (rerunning BindGen each time your code changes), but it can be nice if you intend to use the schema as a basis for further development.

Section 6. BindGen command-line parameters

BindGen supports several command-line parameters in addition to those used in the tutorial code. Table 1 lists the most important options:

Table 1. BuildGen command-line options

Command	Purpose
-b name	Generated root binding definition file name (default name is binding.xml)
-c path	Path to input customizations file
-n uri=name,...	Give schema namespace URI and file-name pairs (default generates file names from schema namespace URIs)
-p path,...	Paths for loading Java class files (default is the classpath used to run BindGen)
-s path,...	Paths for loading Java source files (source is not used by default)
-t path	Target directory path for generated output (default is current directory)
-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 BindGen as command-line parameters, without the need to create a customizations file, by using `--` as a special prefix to the customization attribute value. So to set the same global options as used in the [Listing 13](#) customizations, you'd add `--property-access=true` `--force-mapping=true` to the BindGen command line. No quotes are needed for the attribute value when you use this technique. If you want to set a customization that takes a list of multiple values, just use commas rather than spaces as separators between the individual values (so to ignore the prefixes `m_` and `s_` on field names, for instance, you'd use the command line parameter `--strip-prefixes=m_,s_`).

Section 7. Going further

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

You can go even further with JiBX by working with the actual binding definitions, which give you control over every step of the conversion process. You can easily add your own code to be executed as part of the conversions, using user extension hooks built into the binding definitions. You can even create your own custom marshalling and unmarshalling code that can selectively take over control from the code generated by JiBX to handle unique XML or Java data structures. The JiBX documentation includes a tutorial that illustrates many aspects of working with binding definitions, including these extension features, along with reference documentation for all the details.

JiBX is especially convenient when you want to develop a schema definition quickly for data exchange without needing to learn a lot about schema. The XML Schema standard is complex, and tools for working with schema definitions provide little support for restructuring and refactoring schemas. By using Java code and BindGen as the basis for your schema development as shown in this tutorial, you can apply all the flexibility of Java IDEs to create schema definitions quickly and easily without in any way committing yourself to using JiBX.

JiBX also includes a tool for generating complete WSDL and schema definitions for Web services based on Java code. This tool, named Jibx2WsdI, builds on top of BindGen. You can use all the BindGen customizations discussed in this article for the data classes used as inputs and outputs for your service methods, so that the generated schema will reflect your preferences. The JiBX documentation provides details on how to use Jibx2WsdI.

In [Part 2](#), you'll learn how to use JiBX to generate Java code from XML schema definitions.

Downloads

Description	Name	Size	Download method
Sample code for this tutorial	j-jibx1.zip	13KB	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.
- "[Classworking toolkit: Inside JiBX code generation](#)" (Dennis Sosnoski, developerWorks, September 2005): Find out how JiBX implements class file enhancement for XML data binding.
- "[XML and Java technologies: Data binding Part 3: JiBX architecture](#)" and "[XML and Java technologies: Data binding Part 4: JiBX Usage](#)" (Dennis Sosnoski, developerWorks, April 2003): These articles, based on an early development version of JiBX, discuss the JiBX internal structure and how to use JiBX for flexible binding of Java objects to XML.
- "[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.
- "[Code First' Web Services Reconsidered](#)" (Dennis Sosnoski, InfoQ, August 2007): Find out how building on existing code can be a practical way to get started on Web service development.
- [Data binding with Castor](#) (Brett D. McLaughlin, developerWorks, 2007-2008): Learn about another open source data-binding tool in this article series.
- 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 JiBX.
- [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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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