15

XML Data Binding

Introduction

Application data, whether stored as a plain text file, in a RDBMS, or in a custom binary format typically needs to be converted to native data formats before being manipulated by the application. Storing or representing data in XML is no exception.

With the growing use of XML in many applications today, application developers often have the need to access and manipulate the content of XML documents. There are standard ways for a programmer to access this content, such as the W3C DOM API and the de facto standard SAX API from David Megginson (http://www.megginson.com) , but these APIs are used for lower level XML manipulation. They deal with the structure of the XML document. For most applications which need to manipulate XML data, these APIs will be cumbersome, forcing us to deal with the structural components of the XML document to access the data, and they offer no way to depict the meaning of the data. We can write custom APIs or utilities to access the data, but this would be a laborious task. What is needed is a way to access the data without knowing how it is represented, and in a form more natural for our programming language, in a format that depicts its intended meaning.

One solution is data binding. In the next few sections we will explain what data binding is and how it can be used to simplify programming applications that need to interact with XML data.

> *Note: The Professional XML 2nd Edition code download available from http://www.wrox.com comes with not only the full code for all the examples, but also Castor, Jakarta Regex library, Xerces, and xslp, to save you some time with your setup for this chapter.*

What is Data Binding?

Data binding is the process of mapping the components of a given data format, such as SQL tables or an XML schema, into a specific representation for a given programming language that depicts the intended meaning of the data format (such as objects, for example). Data binding allows programmers to work naturally in the native code of the programming language, while at the same time preserving the meaning of the original data. It allows us to model the logical structure of the data without imposing the actual specific structure imposed by the format in which the data is stored.

Let's look at an XML example. In most cases the content of an XML document, though stored in XML as simply character data, represents a number of different data types such as strings, integers, real numbers, dates, and encoded binary data to name a few. These different data types are usually grouped together in some logical hierarchy to represent some special meaning for the domain in which the XML data is intended. Ideally, interacting with the XML content as a data model represented as objects, data structures, and primitive types native to the programming language we are using, and in a manner which closely reflects the meaning of that data, would make programming with XML more natural, much less tedious, and improve code readability and maintainability.

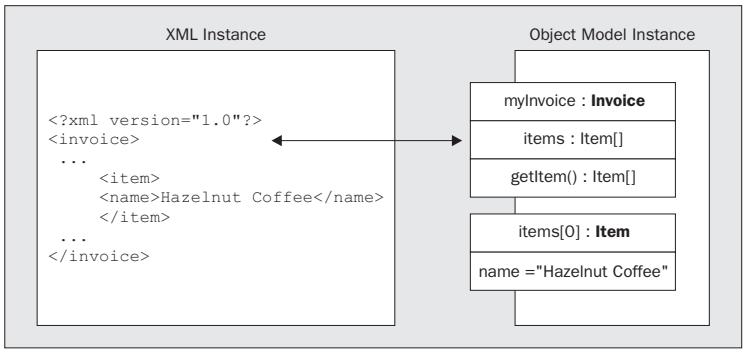
What does all this mean? It simply means that instead of dealing with such things as parse trees, event models, or record sets, we interact with objects, integers, floats, arrays, and other programming data types and structures. To summarize, data binding gives us a way to:

- Represent data and structure in the natural format of the programming language we decide to program in.
- Represent data in a way that depicts the intended meaning.
- Allows us to be agnostic about how the data is actually stored.

XML Data Binding

Now that we have an understanding of what data binding is, let's continue with our look at how it works with XML. XML data binding simply refers to the mapping of structural XML components, such as elements and attributes, into a programmatic data model that preserves the logical hierarchy of the components, exposes the actual meaning of the data, and represents the components in the native format of the programming language. This chapter will focus on XML data binding specifically for the Java programming language. In Java, our data model would be represented as an **Object Model**.

An object model in Java is simply a set of classes and primitive types that are typically grouped into a logical hierarchy to model or represent real-world or conceptual objects. An object model could be as simple as consisting of only one class, or very complex consisting of hundreds of classes.



Most XML based applications written today do some form of data binding, perhaps without the programmer even being aware of it. Unless our application was designed specifically to handle generic XML instances it's hard to imagine interacting with XML data without first needing to convert it to a more manageable format. Each time that we convert the value of an attribute to an integer, or create an object to represent an element structure we are performing data binding.

Simple Data Binding Concepts

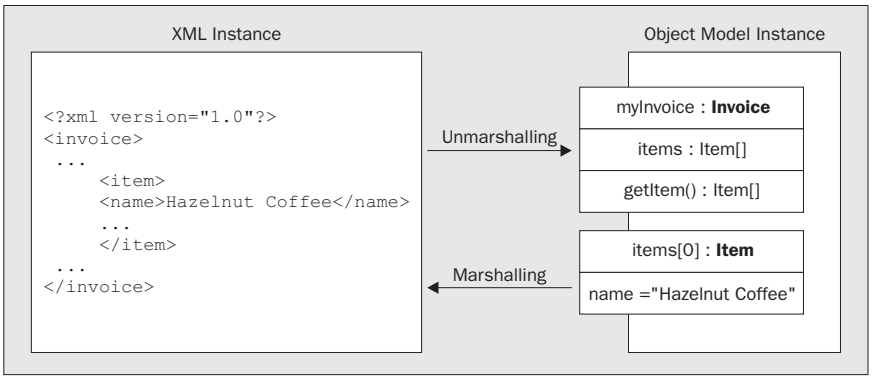
At this point we already have a good understanding of what data binding is, and hopefully you're convinced that data binding is more practical and more natural to use than generic structure based APIs for typical application development. Structure based APIs are very useful for what they were designed for, interacting with data in the generic format in which it is stored, but if our intent is to interact with the data in a form that closely models the meaning of the data, then data binding is clearly the better choice. This is true for XML data binding, RDBMS data binding, and almost any kind of data binding you can think of. It's always more natural to manipulate data in the native formats of the programming language.

> *While data binding may be the clear choice for interacting with data for many XML applications it may not always be the best choice. If there is no need to interact with the data in a form that models the meaning of the data, or if we only want to grab small bits and pieces of the data, then data binding will probably be more trouble than it's worth.*

How do we bind our XML instances to a Java object model? We have two options; we can write our own data binding utilities, or we can use a data-binding framework (the best solution in most cases). A data-binding framework is an application (or set of applications) that allows data binding solutions to be developed more easily – these frameworks typically come with a number of features, such as source code generation and automatic data binding. We'll see examples of using such features later in the chapter.

XML data binding consists of three primary concepts – **the specification of XML to object model bindings**, **marshaling**, and **unmarshaling**. Different binding frameworks will specify bindings differently and typically a data binding framework will have more than one way to specify such bindings.

When we convert an object model instance into an XML instance we call it marshaling, and when we go in the other direction, from an XML instance to an object model we call it unmarshaling. Many people often get these two terms mixed up. The best way to remember it is that we always look at it from the point of view of writing the program. Our desired format to interact with the data is when it's in the programming language's own natural format. So we start with our Java object model. When we want to store the data we need to marshal it into the proper format. When we want to access the data again we need to unmarshal it back into the object model.



Data Objects

In order to interact with our XML instances and manipulate the data, we'd like to convert the XML instances into a more manageable model that is native to the programming language that we are using. This allows for a more natural way of working with the data and makes our code more readable and maintainable. In our case we'd like to convert the XML model into a Java object model for manipulation and then back into the XML representation when we are finished. In most cases, however, the objects in our model simply do nothing more than hold the data that was once stored as XML, and preserve the logical structure of the XML document. Since these objects have no complex programming or business logic, and just contain typed data, we call them **Data Objects**.

Typically our data objects will also be **Java Beans** (or COM objects if we were working in an MS environment). A Java Bean is a Java class that adheres to the Java Bean design pattern. This means that we follow some basic guidelines when writing our classes so that information can be obtained about our data objects. In most cases, following the method naming conventions of the design pattern for Java Beans is sufficient for tools to obtain information about the fields (also called properties) of our classes by examining the method signatures of a given class. This examination process is called **Introspection** (as defined by the Java Beans Specification). The main guideline for the design pattern is that all publicly accessible fields have proper **getter** and **setter** access methods. For a given field, a getter method is quite simply a method that returns the value of that field, while a setter method is one that allows us to set the value of the field.

The Java Beans design pattern indicates that getter methods should begin with "`get`", followed by the field name, with the first letter of the field name capitalized. The setter methods follow the same pattern, but begin with "`set`". For example, if we had a field called `Name`, the getter method should be called `getName` and the setter method `setName`.

Let's look at a simple Java Bean. If we had an `Invoice` class that contained a shipping address, a billing address and a collection of items, our Java Bean would look something like the following:

```
public class Invoice {

   public Invoice() { ... }


   public BillingAddress getBillingAddress();

   public void setBillingAddress(BillingAddress address);

   public ShippingAddress getShippingAddress();

   public void setShippingAddress(ShippingAddress address);

   public Vector getItem();

   public void setItem(Vector items);

}
```

For indexed properties such as our `Vector` of items it is often useful to provide indexed setter and getter methods, such as:

```
    public Item getItem(int index);

    public void setItem(int index, Item item);
```

The indexed setter and getter methods would apply to arrays, vectors, and ordered lists, but doesn't make sense for other types of collections such as sets, or hash tables, which do not preserve the order of the collection.

It's not required that our data objects be Java Bean compliant, but it is good practice. The main reason is that a data-binding framework will most likely need to determine certain information about a given object such as the field names and Java types by examining the class type of the object, using introspection. This will be discussed in more detail later in the chapter.

> *For more information on Java Beans see the Java Beans specification version 1.01 (http://java.sun.com/products/javabeans/docs/spec.html)*

What's Wrong with APIs Such As DOM and SAX?

Most readers should be familiar by now with the DOM and SAX APIs discussed in earlier chapters (11 and 12, respectively). The W3C DOM is tree-based while SAX is event-based, but both APIs are structure-centric in that they deal purely with the structural representation of an XML document. The basic format of an XML document is a hierarchical structure comprised mainly of elements, attributes and character data. There is nothing actually wrong with these APIs if they are used for their intended purpose, manipulating XML documents in a way that represents the generic structure of the XML document.

Most applications that need to access the contents of an XML document, however, will not need to know whether data was represented as an attribute or an element, but simply that the data is structured and has a certain meaning. APIs such as the DOM and SAX, while providing meaningful representations of the data, can only provide extremely generic representations that make accessing our data long winded, which for high-level XML manipulation adds complexity to our applications that is unnecessary. Programmers must essentially walk the structure of the XML in order to access the data. This data also often needs to be converted into its proper format before it can be manipulated, which can be a very tedious process. This is necessary because the data returned using these APIs is simply a set of characters, or a string. If the contents of an element should represent some sort of date or time instance, then the programmer must take the character data and convert it to a date representation in the programming language. In Java this would most likely be an instance of `java.util.Date` (the Java date function). If the element itself represents a set of other elements, we would need to create the appropriate object representation and continue converting sub-elements.

> *There are specifications, such as the W3C XPath Recommendation, which improve our ability to easily access the data within an XML document without the need for walking the complete structure. XPath is great for extracting small pieces of information from an XML document. Unfortunately the data still needs to be converted to the proper format if it is to be manipulated. It also tends to add complexity and extra overhead to our applications if our goal is to access data throughout the entire document.*

Let's look at an example to demonstrate how one would use SAX to handle data binding (`Invoice.java`, below). Recall our `Invoice` class from the previous section. For simplicity let's assume for that the invoice only contains a set of items, where each item in our set has a unique product identifier called an SKU number, a product name, a quantity, unit price, and a description. We will also assume the price is in US dollars.

For brevity we've removed most of the code and comments and we show only the method signatures. The complete code for can be found in the code download at http://www.wrox.com.

```
public class Invoice {

   public Invoice() { ... }

   public void addItem(Item item);

   public Vector getItem();

   public void setItem(Vector items);
}
```

For convenience I've taken the liberty to include an `addItem()` method that will add the items incrementally to the internal `Vector` of items for us. We could also have added a `removeItem()` method, but we won't use it in our example. Often classes like to abstract how a collection of objects is implemented, and "add" and "remove" methods are a good way to do this. The add/remove methods actually stray slightly from the standard Java Beans design pattern. In Java Beans add/remove methods are usually used for Event Listeners. Nonetheless, the approach of using add/remove methods is a good one if you want to abstract implementation details from your class definitions.

Let's take a look at our `Item` class (`item.java`) :

```
public class Item {


   private int _quantity = 0;

   ...

   public Item() { ... };

   public String getDescription() { ... };

   public void setDescription(String desc) { ... };

   public String getName() { ... };

   public void setName(String name) { ... };

   public double getPrice() { ... };

   public void setPrice(double price) { ... };

   public String getProductSku() { ... };

   public void setProductSku(String sku) { ... };

   /**
    * Returns the desired quantity for this Item.
    *
    * @returns the desired quantity for this Item.
   **/
```

```
    public int getQuantity() { return _quantity };

    /**
     * Sets the desired quantity of this Item.
     *
     * @param quantity the desired quantity for this Item.
    **/
    public void setQuantity(int quantity) {
       _quantity = quantity;
    }

} //-- Item
```

You'll notice our representation of an item also conforms to the Java Beans design pattern we discussed in the previous section, as we have proper getter and setter methods for each field in our class.

Now that we have our object model, let's look at a sample XML instance for our invoice:

```
<?xml version="1.0"?>
<invoice>
   <item>
      <name>ACME Rocket Powered Roller Blades</name>
      <description>ACME's model R2 Roller Blades</description>
      <price>999.98</price>
      <product-sku>9153556177842</product-sku>
      <quantity>1</quantity>
   </item>
   <item>
      <name>ACME Bird Feed</name>
      <description>25 pound bag of Bird feed.</description>
      <price>17.49</price>
      <product-sku>9128973676533</product-sku>
      <quantity>2</quantity>
   </item>
</invoice>
```

We can see that our object model and XML instance look very similar. Both XML and Object Oriented programming are ideal for modeling real-world objects.

How do we unmarshal our XML instance of our invoice into an instance of `Invoice`? To do this in SAX 2.0 we need to create an implementation of the SAX `ContentHandler` interface.

The `ContentHandler` interface specifies many methods that allow us to receive events for elements with their associated attributes, character data, and other XML components. In our implementation of this interface we have to create methods that handle the events for all the different types of XML components that a `ContentHandler` declares methods for. In most cases, we only truly need to implement three methods: `startElement`, `endElement`, and `characters`. In order to be a valid implementation, however, we must create empty methods for all of the declared methods of the interface. Hopefully you're already getting the feeling we're not going to be doing too much!

> *For readability I will not show the empty methods, but the full source can be found in the code download at http://www.wrox.com.*

Let's look at the code for `InvoiceContentHandler.java`:

To start we need to import the SAX package and declare that our class implements the `ContentHandler` interface.

```
/* import sax related class */

import org.xml.sax.*;

...

public class InvoiceContentHandler
   implements org.xml.sax.ContentHandler
{
```

In order to build our `Invoice` object model our `ContentHandler` must keep track of some state while we are handling the XML events. We need the `Invoice` instance that we are creating, the current item that is being created and a buffer to hold character data.

```
    private Invoice _invoice = null;

    private Item _current = null;

    private StringBuffer _buffer = null;
```

The constructor for our `ContentHandler` will initialize the `StringBuffer`, but our `Invoice` instance and `Item` instances will get initialized as our handler receives events.

```
    public InvoiceContentHandler() {
       super();
       _buffer = new StringBuffer();
     } //-- InvoiceContentHandler
```

We also create a method that is used to obtain the `Invoice` instance when we are finished handling the events:

```
    public Invoice getInvoice() {
       return _invoice;
    } //-- getItems
```

Now let's look at the `startElement()` method. When we receive the start element events for `Invoice` and `Item` we simply create new instances of these classes. We also add some error handling code to make sure we receive a correct XML instance:

```
    public void startElement
       (String namespaceURI, String localName, String qName, Attributes atts)
    throws SAXException
{
    if (localName.equals("invoice")) {
```

```
            _invoice = new Invoice();
            _buffer.setLength(0);
        }
        else {
            if (_invoice == null) {
                throw new SAXException("Invalid XML Instance, missing <invoice>
    element.");
            }
            else if (localName.equals("item")) {
                _current = new Item();
                _invoice.addItem( _current );
            }
            else {
                _buffer.setLength(0);
            }
        }
    } //-- startElement
```

You'll notice that for all elements other than `invoice` and `item` we simply clear the `StringBuffer`. The reason is that all the other possible elements in our XML instance simply hold character data that represents either String values or primitive types. Therefore we don't have any classes that need to get created for them.

> *For validation purposes we could also use a Validating XML parser in conjunction with some sort of XML schema. A DTD wouldn't suffice here because DTDs contain no type information, but could be used to simply validate the structure. This might be overkill for our simple example, but for large XML instances it would greatly reduce the amount of hard-coded checking that we need to perform.*

The `characters()` method will simply save the characters to the `StringBuffer` so that they are available when we receive the end element event.

```
    public void characters (char[] chars, int start, int length)
    throws SAXException
{
    //-- save characters for later use
    _buffer.append(chars, start, length);

} //-- characters
```

The `endElement()` method looks for each of the elements in our XML instance and converts the saved `StringBuffer` into the appropriate type.

```
    public void endElement
        (String namespaceURI, String localName, String qName)
    throws SAXException
{
    if (localName.equals("item")) {
        _current = null;
    }
    else if (localName.equals("name")) {
        _current.setName( _buffer.toString() );
```

```
    }
    else if (localName.equals("description")) {
       _current.setDescription( _buffer.toString() );
    }
    else if (localName.equals("product-sku")) {
       _current.setProductSku( _buffer.toString() );
    }
    else if (localName.equals("price")) {
       //-- we need to convert string buffer to a double
       try {
          Double price = Double.valueOf ( _buffer.toString() );
          _current.setPrice( price.doubleValue() );
       }
       catch (NumberFormatException nfe) {
          //-- handle error
          throw new SAXException( nfe );
       }
    }
    else if (localName.equals("quantity")) {
       //-- we need to convert string buffer to an int
       try {
          _current.setQuantity( Integer.parseInt( _buffer.toString() ) );
       }
       catch (NumberFormatException nfe) {
          //-- handle error
          throw new SAXException( nfe );
       }
    }

} //-- endElement
```

Notice that we need to convert the `StringBuffer` to a `double` for the price field, and to an `int` for the quantity field.

We've finished the necessary binding and unmarshaling code for our example. I don't know about you, but for me that was too much work to simply create the object model for our XML instance! In addition, maintaining this piece of code would be rather tedious and the larger the document structure, the greater the possibility of errors. Any time the structure of the document changes you will most assuredly need to modify the `startElement/endElement` methods, which are the most complicated methods in the `ContentHandler` to implement.

We now need to call the SAX `XMLReader` to read our XML instance using our implementation of `ContentHandler`. This code is sufficient (`SAXInvoiceExample.java`):

```
//-- Create an instance of our ContentHandler
InvoiceContentHandler invoiceHandler = new InvoiceContentHandler();

//-- Create and configure SAX XMLReader
XMLReader parser = new SAXParser();
parser.setContentHandler( invoiceHandler );

//-- Parse our example XML instance
parser.parse( new InputSource("invoice.xml") );
```

```
//-- Get Invoice instance
Invoice invoice = invoiceHandler.getInvoice();
```

The full example code in the download will also display the newly created `Invoice` instance on the screen.

Our example was actually quite small, but imagine having to manually write data binding code for hundreds of elements and attributes in order to manipulate it. It's just too tedious, time consuming, and potentially very error-prone. In our example we only covered unmarshaling. Marshaling would be a bit simpler as we might not need to include any validation code, but it's basically just as tedious.

Using the W3C DOM API is quite similar to SAX, but instead of dealing with events we deal with a tree structure. In our SAX `ContentHandler` the events were passed to us, and we simply built our Invoice object model. Using the DOM we would simply start with the `Document` node, obtain the document element, which in our case is the `<invoice>` element, and recursively process all of its associated attributes and child nodes.

Both of these APIs are great if our application needs to deal with XML at this lower level. In most cases however our applications will not need such low-level interaction and these APIs simply add more work than is necessary to solve our problems.

The Need for a Data Binding Framework

I hope that by now I have convinced you that using APIs like DOM and SAX are just too tedious for most large-scale XML-based programming projects. I asked you at the end of the last section to imagine having to write data binding code for hundreds of elements and attributes. Now imagine having to do it for all your XML-enabled applications. This would be very tedious and extremely boring, as well as error-prone, and tricky to test and debug. You'd probably end up needing counseling, your spouse would divorce you because your nightmares about elements, attributes, and `NumberFormatExceptions` would be just too strange for him or her to deal with, and ultimately you would end up quitting your job for something more exciting!

If that wasn't enough, imagine having to maintain all that code. Each time your boss decides he or she wants to change the XML schema you'd have to go back through all that binding code and find the right code to change. Code maintenance would be pretty difficult for large XML schemas.

The good news is that we don't need to use these types of API at all!

> **What we need is a simple framework that allows us to specify how an XML schema should map into an object model and automatically handle the data binding for us.**

As we have discussed above, such a framework is called a data-binding framework. Using such a framework allows us to easily perform XML data binding, as well as to easily maintain our code when we need to change the representation of our XML data – the framework will handle the marshaling and unmarshaling automatically.

The even better news, as we'll see in the next section, is that such a data-binding framework exists, and it's free!

XML Data Binding with Castor

So far we've introduced XML data binding, discovered that APIs such as SAX and DOM are not always the best choice when it comes to writing XML based applications, witnessed the tediousness of hand-coded data binding using SAX, and have discussed the need for a data-binding framework. Now I'd like to introduce you to a project called **Castor**, an open source data-binding framework for Java that supports binding XML, relational databases, and LDAP. For this chapter we will only concern ourselves with the XML data binding capabilities. Castor is freely downloadable at http://www.castor.org.

Before we jump into the code, I want to give you a brief background on the project. Castor was started when Assaf Arkin, the CTO of Intalio, Inc., decided that he wanted to use XML files to store configuration data for another open source project that he was writing. To read the configuration files he wanted to use XML data binding and not one of the XML APIs, for all the reasons we have specified in the previous sections. However, there was a big problem. There wasn't an XML data binding framework freely available for him to use. So he asked me to write a simple framework for him.

Castor XML was initially based on the Java specification request for XML data binding (JSR 31) from Sun Microsystems, Inc. The first draft of JSR 31 was very small and vague, but outlined enough concepts for me to try to follow during my initial implementation. Depending on what happens with the JSR 31 specification in the future Castor may or may not adhere to it.

Today, Castor has benefited from a large user community as its current design has been influenced by feedback and numerous contributions from the users. The core development of Castor is funded by Exolab.org (http://www.exolab.org), which hosts and sponsors a number of open source projects, but Castor wouldn't be the product it is today without the valuable contributions, such as bug reports, suggestions, and code that it receives from its user base.

Castor is written in Java and performs data binding specific to the Java language; in other words, you cannot use Castor to perform XML data binding in C++. Internally Castor XML uses the SAX API as it's underlying XML API. Castor actually interacts with the SAX API during marshaling/unmarshaling so that you don't have to! We'll discuss this more in the following sections.

Castor is designed to not only to work with existing Java object models, but it can generate complete object models based on a given XML Schema! Castor has three core data binding methodologies: automatic data binding using built-in introspection, user specified data binding, and XML Schema based data binding with complete source code generation. These methodologies can be used independently or together, which makes Castor a very powerful and easy to use data binding framework. We will discuss these approaches in the following sections.

You will notice throughout the next few sections that I often refer to different ways to configure Castor. Most of the configuration takes place in a file called `castor.properties`. This file is located in the Castor `.jar` file (`org/exolab/castor/castor.properties`). If you need to configure Castor, simply place a copy of this file in your current working directory and make any necessary modifications. The actual configuration changes are explained further in the following sections.

Now that we know a little history of the project and some basics, let's get to the code!

Using Castor's Built-in Introspection

The easiest way to get started with Castor is to simply rely on its default capabilities to automatically handle simple data binding. Using the default capabilities of Castor is so easy you might wonder why you were ever using an XML API to begin with! The best way to learn anything is to simply try it, so we'll start with a simple example and then explain what Castor is doing behind the scenes.

We'll continue with our invoice theme. Recall our data object representation of an invoice item from the previous section – I've repeated `Item.java` here so that I can make a few additional points about the class.

```
public class Item {

   public Item() { ... };

   public String getDescription() { ... };

   public void setDescription(String name) { ... };

   public String getName() { ... };

   public void setName(String name) { ... };

   public double getPrice() { ... };

   public void setPrice(double price) { ... };

   public String getProductSku() { ... };

   public void setProductSku(String sku) { ... };

   public int getQuantity() { ... };

   public void setQuantity(int quantity) { ... };

} //-- Item
```

This is the same class definition that we used in the first example. No modifications are necessary. As mentioned previously, we have followed the Java Beans design pattern when creating this class definition, which will be important for Castor to be able to automatically determine information about our data object. Castor will need to introspect this class and examine the method signatures during the marshaling and unmarshaling processes. We also have a default constructor. A default constructor is simply a constructor that takes no arguments. This is important since Castor needs to be able to instantiate our objects during unmarshaling. If our objects only had constructors that required arguments, Castor wouldn't know how to create them.

In the following two sections we will demonstrate both marshaling and unmarshaling instances of our `Item` class. We will start with the unmarshaling process first because it's usually more complicated than marshaling and then we'll move into the marshaling process.

Default Unmarshaling

Let's take a look at an XML instance of an item that we would like to unmarshal into an instance of our `Item` class (`item.xml`):

```
<?xml version="1.0"?>
<item>
   <name>ACME Hazelnut Coffee</name>
   <product-sku>9123456123456</product-sku>
   <description>A pound of ACME Hazelnut Coffee – Whole Bean</description>
   <price>9.99</price>
   <quantity>2</quantity>
</item>
```

Since most programmers, at least the ones that I know, including myself, spend more time drinking coffee than actually writing code, our invoice item appropriately represents an order for 2 pounds of coffee from the company that produces everything from rocket-powered roller skates to quick dry cement.

We now have our object model representation and our XML instance document, so let's take a look at the code necessary to unmarshal our XML instance into our object model (`ItemTest.java`):

```
//-- import Castor XML classes
import org.exolab.castor.xml.*;
```

At the top of the class we need to import Castor related packages.

```
...
...
   public static void main() {

      //-- unmarshal instance of item
      Item item = (Item) Unmarshaller.unmarshal(Item.class,
                    new FileReader("item.xml"));
```

That's all that's needed. Let's add a few lines of code to print some of the values of our item:

```
//-- Display unmarshalled address to the screen
System.out.println("Invoice Item");
System.out.println("------------");
System.out.println("Product Name: " + item.getName());
System.out.println("Sku Number : " + item.getProductSku());
System.out.println("Price    : " + item.getPrice());
System.out.println("Quantity  : " + item.getQuantity());
```

Sample output:

> java ItemTest

```
Invoice Item
------------

Product Name: ACME Hazelnut Coffee
Sku Number : 9123456123456
Price    : 9.99
Quantity  : 2
```

Now that was pretty easy, right? What happens, however, if we change our XML file slightly by renaming our element name of the SKU number, currently `product-sku`, to simply `sku`? Let's give it a try and find out (`modified_item.xml`):

```
<?xml version="1.0" standalone="yes"?>
<item>
   <name>ACME Hazelnut Coffee</name>
   <sku>9123456123456</sku>
   <description>A pound of ACME Hazelnut Coffee - Whole Bean</description>
   <price>9.99</price>
   <quantity>2</quantity>
</item>
```

Now we re-run our test case and we get the following result:

```
> java ItemTest

Invoice Item
------------

Product Name: ACME Hazelnut Coffee
Sku Number : null
Price    : 9.99
Quantity  : 2
```

What happened? Why is the SKU number listed as `null`? The simple answer is that we are relying on the default capabilities of Castor. When we changed the element name, Castor was unable to match this name to the proper field in our `Item` class because it was expecting a method, that did not exist. Element naming is important if we are relying on Castor to automatically figure out our data binding. We will see in the section "Using Castor Mapping Files" how we can properly unmarshal the modified version of our XML instance, but first let's take a look at default marshaling.

Default Marshaling

We have seen how simple it is to unmarshal an XML instance into an object model. How about going in the other direction? Let's expand on our example a bit by modifying the quantity field. We all know that 2 pounds of coffee won't last too long for most IT professionals. We will change the example so that after we unmarshal our XML instance into an `Item` object, we will double the quantity to 4 pounds, and then we will marshal this `Item` into a new XML instance.

Let's look at the code necessary to update and marshal our XML instance (`ItemUpdate.java`):

```
...
/* unmarshaling code removed for brevity */
...
//-- update quantity to 4 pounds instead of 2
item.setQuantity( 4 );

//-- marshal item
FileWriter writer = new FileWriter("item_updated.xml");
Marshaller.marshal(item, writer);
writer.close();
```

That's it! `ItemUpdate.java` will save our new XML instance into a file called `item_updated.xml`:

```
<?xml version="1.0"?>
<item>
   <name>ACME Hazelnut Coffee</name>
   <quantity>4</quantity>
   <description>A pound of ACME Hazelnut Coffee - Whole Bean</description>
   <price>9.99</price>
   <product-sku>9123456123456</product-sku>
</item>
```

You might have noticed that our elements are not in the same order. Using Castor's default capabilities ordering of elements is not preserved. In most cases the ordering is not important (although element ordering is important from a valid XML point of view). We'll see how ordering can be preserved in the sections on *Mapping* and *The Complete Data Binding Solution.*

For the advanced reader, who might be a bit curious, the ordering for the default behavior will actually depend on the order in which the method names for the class are examined. This is actually dependent on the class loader. Castor uses the Java Reflection API to obtain the information about the methods of a given class. The order in which the methods are obtained using the Java Reflection API is the order in which Castor will marshal fields. For more information on the Java Reflection API please see http://java.sun.com/products/jdk/1.1/docs/guide/reflection/

A Closer Look At the Default Behavior of Castor XML

Using the default behavior of Castor XML is pretty simple, but let's take a closer look at what is actually happening during the unmarshaling and marshaling of our XML instances. The key to the default behavior in Castor is introspection – recall that this is the process in which the design of a class is examined in order to deduce certain information about it.

Unmarshaling

During unmarshaling, Castor starts with the **document element** of the XML instance. The document element is automatically mapped to an instance of the class passed to the `unmarshal` method. Recall how we invoked the unmarshaller on our `Item` class:

```
//-- unmarshal instance of item
Item item = (Item) Unmarshaller.unmarshal(Item.class,
         new FileReader("item.xml"));
```

We gave the unmarshaller the starting point for our object model. It isn't always necessary to do this, however, when using the default data binding capabilities, it's necessary to give Castor the starting class of our object model when:

- Our starting class does not have a name that can be inferred from the document element name; for example, if we have an element named `<foo>`, but our class name is `FooImpl`.
- Our starting class belongs to a **package** (a Java facility that allows classes to be logically grouped together as well as way to differentiate from classes provided by others), which will be the majority of the time, because it is always good practice to use packages.

There are some special features in Castor that allow us to circumvent this requirement, but discussion of these features is beyond the scope of this chapter.

After Castor has mapped the document element, introspection is used to examine the method signatures of the class. For each sub-element and attribute that exists in the document element, Castor attempts to map to a specific `setter` method by comparing the name of the element or attribute to the name of the method. Castor uses the design patterns specified in the Java Beans specification to find the proper method. For example, Castor will map our `<quantity>` element to a method called `setQuantity`. If the XML name contains a hyphen, Castor will remove the hyphen and capitalize the following character. For example, `<product-sku>` will be mapped to a method called `setProductSku()`.

Once a method is found, the method signature is examined to determine the proper Java type for the argument. Castor will then attempt to convert the data to the proper format. For example, our `setQuantity()` method in our `Item` class has a Java primitive `int` type as the argument, so Castor converts the character data from our `<quantity>` element to an `int` and invokes the `setQuantity()` method using this value.

For all complex elements, which are elements that contain sub-elements and attributes, the same process that is used on the document element is followed. The only difference is that the class type that the element is mapped to is obtained by the examination of the method signatures as discussed above. If Castor cannot find an appropriate setter method, the data for that element or attribute will simply be ignored. Castor takes a "do the best you can" approach to automatic data binding.

This entire process is continued recursively until the entire document is processed.

Marshaling

The marshaling process is much like the unmarshaling process except that we start with our object model and produce an XML instance. The document element for the XML instance is the object instance that we pass into the `marshal()` method:

```
...
Marshaller.marshal(item, writer);
...
```

In our example, the `item` object will be marshalled as the document element. The name for the document element is derived from the class name of the object. By default, when Castor converts Java names to XML names the first character is converted to lower case. All subsequent upper case characters, which typically identify new words, are converted to lower case and preceded by a hyphen. The following chart shows some examples of Java names and their corresponding default XML name. If two or more characters in a row are in upper case they will be not be converted to lower case.

Java class or field name	Castor's default conversion for XML
`Invoice`	`<invoice>`
`ShippingAddress`	`<shipping-address>`
`Item`	`<item>`
`NAME`	`<NAME>`

Castor can be configured to allow mixed case in XML names modifying the `org.exolab.castor.xml.naming` property in the `castor.properties` file as follows:

```
…
# change naming algorithm to allow mixed-case
org.exolab.castor.xml.naming=mixed
…
```

In this configuration only the first character is converted to lower case, so no hyphen is needed to separate words (known as camelCasing). The above names would all be converted the same except for `ShippingAddress`, which would be converted to `shippingAddress`.

> *Castor also allows custom naming algorithms by simply implementing the `org.exolab.castor.xml.XMLNaming` interface. This is actually beyond the scope of this chapter, but for more information on configuring Castor, please see the Castor documentation available from* www.castor.org.

Based on the above naming conversions the instance of our `Item` class will have an XML instance name of `<item>`. Once the document element is created, Castor uses introspection to examine all public `getter` methods of the class. For each `getter` method, Castor will create an appropriate XML node, either an attribute or an element, based on the return type of the method. Castor can be configured to handle primitive types (such as `int`, `float`, etc.) as either attributes or elements. In our example we have configured Castor to treat all primitive types as elements. If however the return type is an object, Castor will always create an element. The names of the `getter` methods will be used to create XML names, using the above naming conversions, for their respective fields. For example, our `getQuantity` method will be used to create an element called `<quantity>`.

```
...
  <quantity>4</quantity>
...
```

This entire process is repeated recursively for all objects in the object model.

Conclusion

We've been introduced to the default binding capabilities of Castor, and seen that it's very easy to automatically have Castor handling our binding for us. Even though it's very easy to use the default capabilities it might be difficult to understand what's actually happening if something doesn't marshal or unmarshal properly; for example, when we wanted to use `<sku>` instead of `<product-sku>` we received a `null` value. We also have to make sure that we follow the Java Beans design pattern by using proper getter and setter methods and making sure we have a default constructor, otherwise the default introspection won't work properly. There is also some overhead to having Castor examine all fields of a class, and use the Java Reflection API to interact with class instances, especially if there are many classes in the object model. In the next section we will see how to overcome some of these limitations and have more control over the binding process by using a **mapping file**.

Using Castor Mapping Files

Now that we've seen how easy it is using Castor's default introspection capabilities to automatically handle data binding for us, let's look at how we can have greater control over our XML bindings. It is often the case that the default introspection capabilities are not sufficient to perform the desired data binding. Recall in the previous section we changed our `<product-sku>` element to simply `<sku>`, like so:

```
<?xml version="1.0"?>
<item>
   <name>ACME Hazelnut Coffee</name>
   <sku>9123456123456</sku>
   <description>A pound of ACME Hazelnut Coffee - Whole Bean</description>
   <price>9.99</price>
   <quantity>2</quantity>
</item>
```

The default capabilities were no longer able to perform the automatic binding for us. To solve problems like this, Castor has a binding mechanism called a mapping file. The mapping file is actually an XML document that is written from the point of view of the object model. It describes our object model and allows us specify some XML binding information when needed.

The mapping file allows us to specify how a given Java class and its fields "map" to and from XML; we call this the **XML binding** for the class. The following XML bindings can be specified:

- The node type that a given field should be "bound" to (element, attribute, or text).
- The XML name for a class or a field.
- Namespace declarations for a class

A node is the core component of the XML structure. Elements, attributes, character data, processing instructions, and comments are all examples of nodes.

The mapping file also lets us specify the name of the getter and setter methods for each field in the class. This allows us to perform XML data binding on class definitions that do follow the Java Beans design pattern. We'll examine this in more detail in the next section.

The Mapping File

The mapping file contains two types of information, a description of the object model, and information on how the object model relates to the XML document (or database).

The easiest way to explain the mapping file is to simply show an example. We will show how to write the mapping for our `Item` class so that we can properly marshal and unmarshal instances of the modified version of our XML schema for our invoice item.

We will look at the features necessary for expressing XML bindings only, but the mapping file is also used for relational database bindings. As mentioned in the introduction, Castor supports RDB data binding for Java. Both the XML and RDB bindings can be specified in the same file for classes that will be used in both forms of data binding.

Let's take a look at how to specify the mapping file.

The mapping file conforms to an XML schema, which can be found in the Castor `.jar` file: org/exolab/castor/mapping/mapping.xsd.

The <mapping> Element

A mapping file starts with the `<mapping>` element. This is the document element and simply contains zero or more class mappings.

```
<?xml version="1.0"?>
<mapping>

   <description>Mapping example for our Item class</description>

   <!-- class mappings -->
   ...
</mapping>
```

The `<mapping>` element also allows an optional `<description>` element as we can see in our example above. It also supports the inclusion of other mapping files, but we won't be discussing includes in this chapter.

The <class> Element

The class mapping consists of the `<class>` element. This element specifies the name of the class, and contains zero or more field mappings. It may also contain an optional `map-to` element, which is used to specify an XML name mapping and namespace declaration for the described class. We don't need the `<map-to>` element in our example, but it's often needed for classes that may appear as a document element. This is explained in more detail in the Castor documentation. Our example will deal only with field mappings:

```
<?xml version="1.0"?>
<mapping>

   <description>Mapping example for the Item class</description>

   <class name="Item">
      <!—field mappings -->
   </class>
</mapping>
```

The `name` attribute of the `<class>` element simply specifies the class name. This must be the full package name of the class. Since our `Item` class does not use a package we don't specify one. If we were using the package `org.acme.invoice`, our class mapping would look like:

```
 <class name="org.acme.invoice.Item">
   <!-- field mappings -->
 </class>
```

The `<class>` element also supports an optional `extends` attribute, which can be used to specify a super class (also called a base class) for the described class. This is really only needed if the described class extends another class which also has a class mapping. This allows us to reuse field mappings (see the next section) from the super class without having to re-specify the mappings. All that needs to be specified is the new fields from the subclass.

We don't need to use the `extends` attribute in our class mapping for the `Item` class, but an example of what this would look like is as follows:

```
<class name="org.acme.invoice.InvoiceObject">
   <!-- local field mappings -->
 </class>
…
 <class name="org.acme.invoice.Item"
        extends="org.acme.invoice.InvoiceObject">
   <!-- inherits field mappings from InvoiceObject -->
   <!-- local field mappings -->
 </class>
```

The <field> Element

The `<field>` element allows us to specify field mappings. A field mapping is where we actually express some data binding information. The field mapping describes the field name, its class type, and optionally the setter and getter methods. The field mapping may also optionally contain the `<bind-xml>` element which allows use to specify specific XML bindings for the given field.

Let's look at the field mapping for our product SKU number:

```
...
   <field name="productSku"
      type="string"/>
...
```

Here we have specified the field name as `productSku` and its type as "`string`".

The `name` attribute is simply the name of the field. The value of the `name` attribute is not always important if we provide full specification of the field mapping, but is important if we do not specify `getter` and `setter` methods (we'll see how to specify these later in this section), as introspection will be used to determine these methods. Therefore in the above field mapping, introspection will be used and Castor will deduce that `getProductSku` and `setProductSku` are the appropriate access methods.

Since the mapping file is written from the point of view of the object model, the `type` attribute specifies the Java class type of the field. You'll notice that our field mapping simply used "`string`" as the type, which doesn't look like the Java string type, `java.lang.String`. Since writing "`java.lang.String`" is rather tedious Castor has aliases, or short cuts, for many of the native Java types.

The following is a list of some of the common type aliases. A complete list may be found in the mapping documentation for Castor.

Alias	Java Class Type	Java Primtive Type
`string`	`java.lang.String`	N/A
`integer`	`java.lang.Integer.TYPE`	int
`double`	`java.lang.Double.TYPE`	double
`boolean`	`java.lang.Boolean.TYPE`	boolean
`long`	`java.lang.Long.TYPE`	long
`float`	`java.lang.Float.TYPE`	float
`date`	`java.util.Date`	N/A

You'll notice in the table that for a primitive type the actual Java class instances of those types are actually used. For example, the `int` type is actually represented by the class `java.lang.Integer.TYPE`.

If a type is a collection, such as a `java.util.Vector`, the optional `collection` attribute is used to specify the type of the collection, and the `type` attribute is used to specify the type of the elements of in the collection. If, for example, we expanded our mapping example to include the `Invoice` class, which contains a collection of `Item` objects, such as:

```
public class Invoice {
   ...
   public void addItem(Item item);
   ...
   public Vector getItems();
   ...
}
```

We could use the following field mapping for the items:

```
...
   <field name="items"
      type="Item" collection="vector"/>
...
```

We'll see an example of using collections later in this section.

The following is a partial list of the collection type aliases:

Alias	Java Class Type
array	type[ ].class
vector	java.util.Vector
arraylist	java.util.ArrayList
set	java.util.Set
hashtable (not yet fully supported)	java.util.Hashtable

Though we didn't need to specify getter and setter methods for our example it's quite easy to do so. We simply add `get-method()` and `set-method()` attributes respectively. An example of this is as follows:

```
...
   <field name="productSku" type="string"
          get-method="getProductSku" set-method="setProductSku"/>
...
```

Again, the specification of setter and getter methods is redundant for our specific case, but if our method names do not follow the Java Beans design pattern then we need to specify this information. For example, if our `setProductSku()` method is changed to `updateProductSku()` we would need to have the following mapping:

```
...
   <field name="productSku" type="string"
          get-method="getProductSku" set-method="updateProductSku"/>
...
```

Note that if we specify either a `set-method()` or `get-method()`, but not both, then the non-specified accessor method will be ignored. This means that if the `get-method()` is specified, but the `set-method()` is not, then during marshaling the value will appear in the XML instance, but during unmarshaling the value will be ignored. The opposite is true if the `set-method()` is specified but no `get-method()`. We will see an example of this later in the chapter.

The <bind-xml> Element

Recall that in the previous section on using Castor's default capabilities we wanted to change the XML element name of `<product-sku>` to simply `<sku>`, but Castor was unable to handle this properly. For this to work we need to tell Castor what the new element name is. We do that with the `<bind-xml>` element.

The <bind-xml> element allows us to specify the XML name and node type (element, attribute, or text) of a field:

```
<field name="productSku" type="string">
   <bind-xml name="sku"/>
</field>
```

The name attribute specifies the XML name binding. You can see that we have now specified that the XML name for our productSku field should be "sku".

Notice that we didn't specify a node type. Recall in the previous section, about Castor's default capabilities, that we discussed how primitive types and objects are mapped to a specific node type. Since the type of the productSku field is a string, or object, it maps to an element. We could have used the following:

```
<field name="productSku" type="string">
   <bind-xml name="sku" node="element"/>
</field>
```

The node attribute is used to specify the node type. In our example specifying the node type is not necessary.

The Mapping File for the Item Class

We now know enough to write the complete mapping for our Item class, so here goes (mapping.xml):

```
<?xml version="1.0"?>
<mapping>

<description>Mapping example for the Item class</description>

<class name="Item">
   <field name="productSku" type="string">
      <bind-xml name="sku"/>
   </field>
   <field name="description" type="string"/>
   <field name="name"        type="string"/>
   <field name="price"       type="double"/>
   <field name="quantity"    type="integer"/>
</class>
</mapping>
```

Notice that for the description, name, price, and quantity fields we didn't specify a <bind-xml> element. This is because Castor can automatically handle the XML bindings for us.

Unmarshaling with a Mapping File

Now that we have our mapping file, let's look at the code necessary to unmarshal our XML instance. We will simply change our ItemTest class to load our mapping file:

We first must import the mapping package as follows:

```
//-- we need to import the mapping package
import org.exolab.castor.mapping.*;
```

```
...
import org.exolab.castor.xml.*;
...
...
```

We then load the mapping just before we call the unmarshaller:

```
    //-- Load Mapping
    Mapping mapping = new Mapping();
    mapping.loadMapping("mapping.xml");
```

After we've loaded the mapping we simply give this mapping to the unmarshaller:

```
    Unmarshaller unmarshaller = new Unmarshaller(Item.class);
    unmarshaller.setMapping( mapping );

    Item item = (Item) unmarshaller.unmarshal(new FileReader("item.xml"));

    /* print item - code removed for brevity */
```

Now we can re-run our test case on the modified version of our XML instance (shown in the beginning of this section) and we get the following:

```
> java ItemTest

Invoice Item
------------

Product Name: ACME Hazelnut Coffee
Sku Number : 9123456123456
Price    : 9.99
Quantity  : 2
```

As you can see, Castor can now properly handle our XML instance.

Something a Little More Advanced

Now that we have seen how to use a mapping file, let's make a few changes to demonstrate more complicated mappings. First let's change our XML instance so that `sku` is an attribute instead of an element as follows (`modified_item.xml`):

```
<?xml version="1.0" standalone="yes"?>
<item sku="9123456123456">
   <name>ACME Hazelnut Coffee</name>
   <description>A pound of ACME Hazelnut Coffee - Whole Bean</description>
   <price>9.99</price>
   <quantity>2</quantity>
</item>
```

If you re-run the test case using the new XML instance, you'll notice that the SKU number is `null` again:

```
...
Sku Number : null
...
```

To fix this, we make a simple change to our mapping file, which gives us the following (`modified_mapping.xml`):

```
...
   <field name="productSku" type="string">
      <bind-xml name="sku" node="attribute"/>
   </field>
...
```

Notice we've specified the new node type as "`attribute`".

Now we can re-run our test case and we notice that our example is working again.

```
> java ItemTest2

Invoice Item
------------

Product Name: ACME Hazelnut Coffee
Sku Number : 9123456123456
Price    : 9.99
Quantity  : 2
```

So you can see it's quite easy to specify the node type and XML name for the fields of our classes.

Let's expand our example by introducing the `Invoice` class. In this example our `Invoice` class will simply contain a collection of `Items`, and a total price field. The total price field will have some logic to calculate the total price of the invoice. This field will not have a `set` method, but will have a `get` method. Below you can see `Invoice.java`:

```
public class Invoice {


   /**
      * The collection of items
   **/
   Vector _items = null;

   /**
      * Creates a new Invoice.
   **/
   public Invoice() {
      super();
      _items = new Vector();
   } //-- Item()

   /**
      * Adds an Item to this Invoice
      *
```

```
        * @param item the Item to add
     **/
     public void addItem(Item item) {
        _items.addElement(item);
     } //-- addItem

     /**
        * Returns the Vector of Items for this Invoice
        *
        * @return the Vector of Items for this Invoice
     **/
     public Vector getItems() {
        return _items;
     } //-- getItems

     /**
        * Returns the total for all the Items for this Invoice
        *
        * @returns the total for all the Items for this Invoice
     **/
     public double getTotal() {

        double total = 0.0;

        for (int i = 0; i < _items.size(); i++) {
           Item item = (Item) _items.elementAt(i);
           total += item.getPrice() * item.getQuantity();
        }
        return total;
     } //-- getTotal


  } //-- Invoice
```

We can see that the `getTotal()` method does not return a stored value, but calculates the total price of all the items in our invoice. The total price is something that we'd like to have marshalled in the XML instance so we have the `getter` method, but we have no `setter` method because the value is recalculated when it's needed.

A sample XML instance of our invoice will look as follows (`invoice.xml`):

```
<?xml version="1.0"?>
<invoice>
   <item sku="9123456123456">
      <name>ACME Hazelnut Coffee</name>
      <description>A pound of ACME Hazelnut Coffee –
          Whole Bean
      </description>
      <price>9.99</price>
      <quantity>2</quantity>
   </item>
   <item sku="9123456654321">
      <name>ACME Ultra II Coffee Maker</name>
      <description>ACME's best coffee maker,
```

```
            includes intravenous attachment.
         </description>
         <price>149.99</price>
         <quantity>1</quantity>
      </item>
      <total-price>169.97</total-price>
   </invoice>
```

The mapping file is quite similar to the previous example, we just need to add the class mapping for our `Invoice` class (see below, `invoice_mapping.xml`):

```
...
<class name="Invoice">
   <field name="items" type="Item" collection="vector"/>
   <field name="total" type="double" get-method="getTotal">
      <bind-xml name="total-price"/>
   </field>
</class>
...
```

Notice that we needed a `bind-xml` element for the `total` field since the XML name that we used in our `invoice.xml` file was `total-price`.

The code to unmarshal is similar to that in our previous example. We've added some additional code to loop over all the items in the invoice, and print out the total price, giving us `InvoiceDisplay.java`:

```
//-- Load Mapping
Mapping mapping = new Mapping();
mapping.loadMapping("invoice_mapping.xml");

//-- Unmarshal Invoice
Unmarshaller unmarshaller = new Unmarshaller(Invoice.class);
unmarshaller.setMapping( mapping );

FileReader reader = new FileReader("invoice.xml");
Invoice invoice = (Invoice) unmarshaller.unmarshal(reader);
reader.close();

//-- Display Invoice
System.out.println("Invoice");
System.out.println("-------");

Vector items = invoice.getItems();

for (int i = 0; i < items.size(); i++) {

   Item item = (Item) items.elementAt(i);

   System.out.println();
   System.out.print(i+1);
   System.out.println(". Product Name: " + item.getName());
   System.out.println("  Sku Number : " + item.getProductSku());
   System.out.println("  Price    : " + item.getPrice());
   System.out.println("  Quantity  : " + item.getQuantity());
}
System.out.println();
System.out.println("Total: " + invoice.getTotal());
```

When we run our `InvoiceDisplay` class we get the following result:

```
> java InvoiceDisplay

Invoice
-------

1. Product Name: ACME Hazelnut Coffee
   Sku Number : 9123456123456
   Price    : 9.99
   Quantity  : 2

2. Product Name: ACME Ultra II Coffee Maker
   Sku Number : 9123456654321
   Price    : 149.99
   Quantity  : 1

Total: 169.97
```

Now that we've seen how to unmarshal with a mapping file we'll take a quick look at marshaling.

Marshaling with a Mapping File

Marshaling using a mapping file is just as easy as unmarshaling. The marshaller also has a `setMapping` method, which is used to specify the mapping file. Since we can use the same mapping file for both unmarshaling and marshaling, we can just take a look at the source code necessary to call the marshaller. We'll modify our `ItemUpdate` class, from the section on marshaling using the default capabilities, to use the mapping file. This gives us `ItemUpdate.java`:

```
...

//-- Load Mapping
Mapping mapping = new Mapping();
mapping.loadMapping("modified_mapping.xml");

/* unmarshaling code removed for brevity */
...
//-- update quantity to 4 pounds instead of 2
item.setQuantity( 4 );

//-- marshal item
FileWriter writer = new FileWriter("item_updated.xml");
Marshaller marshaller = new Marshaller(writer);
marshaller.setMapping(mapping);
marshaller.marshal(item);
writer.close();
```

That's all that's needed! The contents of `item_updated.xml` are:

```
<?xml version="1.0"?>
<item sku="9123456123456">
   <description>A pound of ACME Hazlenut Coffee - Whole Bean</description>
   <name>ACME Hazlenut Coffee</name>
```

```
    <price>9.99</price>
    <quantity>4</quantity>
</item>
```

Notice that the order in which the elements appear is the same order in which they were specified in our mapping file.

Conclusion

We have seen how easy it is to use a mapping file to specify our XML bindings when the default capabilities are not sufficient to handle the job. We've shown how to specify node types for specific Java fields, how to specify the XML names for a given field, as well as how to specify getter and setter methods. In the next section will discuss using XML Schema to specify our bindings and we will have Castor automatically generate our object model.

> *For more information on the how to use XML mapping files, please see* http://castor.exolab.org/xml-mapping.html.

Using Castor's Source Code Generator

We have seen how to use automatic XML data binding with Castor, and how to use mapping files, but what if we don't have an existing object model to represent our XML instances? It can often be a tedious and time consuming task writing an object model. Luckily, Castor has a solution – a source code generator, which can automatically generate a Java object model to represent a given XML schema. The source code generator will not only create the Java object model, but also all necessary XML binding information needed by the marshaling framework to properly marshal, unmarshal, and validate instances of the given schema. A source code generator is a valuable time-saving tool when it comes to writing XML enabled applications.

This section is meant to give a general introduction to using and understanding Castor's source code generator.

Using XML Schema To Express Bindings

The source code generator takes as input a schema written in the W3C XML Schema language, and produces the Java object model for that schema. Since XML Schema is not yet a W3C Recommendation, we are using the latest version as of this writing, which is the March 30, 2001 Proposed Recommendation. Castor uses XML Schema because of its rich typing and similarities to an object oriented paradigm.

> *This section is intended to introduce how Castor uses XML Schema – you may need to read Chapter 6 on XML Schema, or the W3C XML Schema page (*http://www.w3.org/XML/Schema*), before continuing.*

Why not use DTDs? Using a DTD (Document Type Definition) we can specify XML structure such as elements and attributes, but we can't specify that an attribute's value should represent an integer or a date. Some of you might be thinking about using the DTD Notation mechanism. Notations are a weak form of typing since they are nothing more than some meta-data, which, in order to work properly, must be respected by people and applications that read and write instances of the given DTD. There is actually a W3C Note (basically a submission), called DT4DTD, or simply Datatypes for DTD, on providing some form of data types for DTD (see http://www.w3.org/TR/dt4dtd). The submission actually seems to try and provide some standard notations that can be used to represent the data types. There also may be some other external types specifications for DTDs, but there is nothing within the language itself.

Unlike DTDs XML Schemas are very expressive and contain a large set of built-in data types. These data types allow us to precisely specify what the value of attributes and elements should be. XML Schema uses the core concepts of **Simple Types** and **Complex Types**. Simple types are basically the set of built-in schema data types and restrictions there of. A complex type is similar to the content model used in DTDs in that it allows us to express an ordering of elements essentially defining the structural representation. Complex types can also define attributes, extend other complex types or simple types, or even restrict other complex types.

> *Castor actually has the ability to convert a DTD into an XML Schema. All the types will basically be strings, but these can be changed by hand to take advantage of the XML Schema types. We currently don't provide a GUI for this, but it's still quite useful. There are also a number of other tools available for doing DTD to XML Schema conversion, such as XML Spy (http://www.xmlspy.com) and TIBCO's Turbo XML (http://www.extensibility.com/tibco/solutions/turbo_xml/index.htm).*

Mapping Schema Components To Java

Mapping XML Schema components to Java is typically a straightforward task. In an ideal world all the data types for XML Schema would have a 1 to 1 mapping with a data type of the programming language we are binding to, in our case the Java language. This is not always the case however. XML Schema has quite a rich set of built-in data types and these types don't always match perfectly with Java. For instance, the XML Schema `date` type doesn't actually map cleanly into a `java.util.Date`, but the XML Schema `timeInstant` type does. So data binding isn't always perfect, but allows us to access and represent the data in way that resembles, as close as possible, the data model of the XML schema. Castor has a defined set of classes that are used to handle the cases when a schema type doesn't map properly into a Java type.

Simple Types

As mentioned above simple types are set of built-in XML Schema types, as well as restrictions there of. Castor will map most simple schema types to Java primitives, such as `int`, `double`, `float`, etc. Other simple types that are more complex such as dates are mapped into either native Java types, such as `java.util.Date`, or, as mentioned above, if no native Java type exists, a Castor defined type. Simple types often support a number of *facets* that allow them to be customizable. For example the `integer` type allows the minimum and maximum values to be specified. Below is a brief list of the commonly used schema simple types and their corresponding Java type. For a complete list, as well as a list of supported facets for each type, see the Castor documentation at http://www.castor.org.

Schema type	Java type
`string`	`java.lang.String`
`boolean`	`Boolean`
`Int`	`Int`
`Long`	`Long`
`Float`	`Float`
`double`	`Double`
`Date`	`org.exolab.castor.types.Date`
`Time`	`org.exolab.castor.types.Time`

Schema type	Java type
timeInstant	java.util.Date
binary	byte array (byte[])
NCName	java.lang.String
positiveInteger	Int

For simple types that are restrictions of other simple types, such as the built-in types `positiveInteger` and `NCName`, as well as simple types that have been restricted by using facets, Castor will map to the closest Java type and then generate necessary validation code to perform validation when marshaling and unmarshaling. For example, when using the `positiveInteger` type,the source code generator will generate validation code to make sure the values of the integer are greater than zero.

Unlike a validating XML parser, the validation code generated validates the object model and not the XML instance. This code will automatically be called by Castor during the marshaling and unmarshaling process, but may be called by the application at any point to force validation. Validation can be disabled by modifying the `castor.properties` file as follows:

```
…
# disable validation during marshaling and unmarshaling
org.exolab.castor.marshaling.validation=false
…
```

By default Castor also disables the underlying XML parser from performing validation of the XML instance. This is to prevent unnecessary overhead of validating both the object model and the XML instance. If object model validation is disabled, the user may want to enable XML parser validation by modifying the `castor.properties` file as follows:

```
…
# enable XML instance validation during unmarshaling
org.exolab.castor.parser.validation=true
…
```

Note that XML instance validation occurs only during the unmarshaling process, but object model validation occurs during both unmarshaling and marshaling.

> *Some of you might be wondering why the validation code is not simply generated in the class definition itself. Early versions of Castor actually did this. The main reason that it's generated externally is to take advantage of the validation framework that exists in Castor for allowing validation of class definitions that were not generated by Castor. It was a design decision to simply keep all validation consistent throughout the framework. Another reason is for a future plan to expose validation information for third party tools, such as GUI editors that might desire access to such information.*
>
> *There are plans to allow control over whether the validation code is generated externally or internally to the class definition, but this feature was not available at the time of this writing.*

Given the following simple type definition:

```
<simpleType name="priceType">
   <restriction base="double">
      <minInclusive value="0"/>
   </restriction>
</simpleType>
```

the source generator will create a field using the Java primitive type `double` for any class definition that is generated that uses the `priceType`. Additional validation code will be generated, external to the generated class, to guarantee values for this field are greater than or equal to zero. The marshaling framework will use the validation code in order to validate instances of the field during the marshaling and unmarshaling process.

For example, for the following element definition:

```
<complexType ... >
   ...
   <element name="price" type="priceType"/>
   ...
</complexType>
```

The source code generator will create the following field definition and accessor methods (getters and setters):

```
public class ... {
   ...
   private double _price;
   ...
   public double getPrice() {
      return _price;
   }
   ...
   public void setPrice(double price) {
      _price = price;
   }
   ...
}
```

Complex Types

Now let's look at complex types and element definitions. Castor has two ways to map complex types and elements into Java. The first, and default, approach is called the **element-method**. When using this approach, the source code generator maps all element definitions that have types specified using a complex type definition into Java class definitions that represent the structure of the complex type.

All top-level complex types are mapped into abstract Java class definitions. Top-level complex types are complex types that are globally defined for the given schema. They may be extended by other complex type definitions or simply used directly by element definitions.

The following is an example of a top-level complex type definition:

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema" ... >
   ...
   <!-- top-level complex type definition -->
   <complexType name="address">
```

```
        <sequence>
          ...
          <element name="name" type="string"/>
          <element name="city" type="string"/>
          ...
        </sequence>
    </complexType>
    ...
</schema>
```

The above complex type definition simply defines an address (we'll assume a US postal address) which contains a sequence of elements (name, city, street, etc.) For simplicity we only defined the name and city elements, which are both defined as strings. The source generator will create an abstract class definition called `Address` as follows:

```
public abstract class Address {

   private String _name = null;
   private String _city = null;
   ...

   public String getCity() { ... };

   public void setCity(String city) { ... };

   public String getName() { ... };

   public void setName(String name) { ... };
   ...

}
```

Any class definitions created for elements that use a top-level complex type definition will extend the class definition for the given complex type.

For example, for the following element definition,

```
<element name="shipping-address" type="address"/>
```

the source generator will create a class definition that extends the `Address` class as follows:

```
public class ShippingAddress extends Address {
  ...
}
```

The element-method approach models closely the XML instance document where each element in the XML instances maps to the class definition generated for the given element.

For all element definitions that have anonymous complex type definitions, which are complex type definitions that appear inside the element definition, the source code generator will simply create a class definition that represents the structure of the complex type. For example, given the following element definition:

```
<element name="item">
   <complexType>
      <sequence>
         ...
         <element name="price" type="priceType"/>
         ...
      </sequence>
   </complexType>
</element>
```

the source code generator will create a class definition as follows:

```
public class Item {

   private double _price = 0.0;

   ...

   public double getPrice() {
      return _price;
   }

   public void setPrice(double price) {
      _price = price;
   }

   ...

   public void validate()
      throws ValidationException;
   ...
}
```

You'll notice the `Item` class looks just like a typical Java Bean. The `validate()` method can be called to make sure that instances of `Item` are valid. This method will actually call Castor's validator, which is part of the marshaling framework, to perform the validation using the validation code that was generated separately from the class definition.

The second approach, called the type-method, is similar to the element-method, but differs on a couple of aspects. Like the element-method, when using top-level element definitions, or element definitions that contain an anonymous complex type definition, the source code generator will create a class definition, which models the structure of the complex type.

With the `type`-method, however, top-level complex types are not treated as abstract classes as they are with the element-method. For example, our address type definition above will get mapped into the following class definition:

```
public class Address {

   private String _name = null;

   ...

   public String getName() { ... };
```

```
    public void setName(String name) { ... };

    ...

}
```

Also unlike the element-method, with the type-method any element definition that is not top-level, but uses a top-level complex type as its defined type will simply be treated as a field in the class definition for the surrounding complex type definition (the complex type definition in which the element is defined). For example, if we have the following element definition:

```
<element name="invoice">
   <complexType>
      <sequence>
        ...
        <element name="shippingAddress" type="address"/>
        ...
      </sequence>
   </complexType>
</element>
```

in the case of the type-method, the source generator will create a class definition as follows:

```
public class Invoice {

   ...
   private Address _shippingAddress = null;
   ...
}
```

whereas using the element-method the source code generator will create a class definition as follows:

```
public class Invoice {

   ...
   private ShippingAddress _shippingAddress = null;
   ...
}
```

We will be using the element-method for our examples. I encourage the reader to try both methods to determine which one is more appropriate for their needs. Configuring the method to use is quite simple. The source code generator has a file called `castorbuilder.properties`, which is located in `org/exolab/castor/builder` path within the Castor `.jar` file. Simply copy this file to your local working directory and modify the `org.exolab.castor.builder.javaclassmapping` property to read as follows:

```
...
# change class generation method to the type-method
org.exolab.castor.builder.javaclassmapping=type
...
```

The basic rule of thumb when choosing which approach to use is to think about how you want your data modeled. If you want each element definition to map to their own separate class definitions, then choose the `element-method`. With this approach the object model looks very much like the XML instance. For example, if we have an element called `<shipping-address>`, it will map to a class called `ShippingAddress`.

If, on the other hand, you want each element definition to map to the class definition of its corresponding complex type definition, then choose the `type-method`. With this approach the object model looks very much like the types defined within your schema. For example if we have an element called `<shipping-address>` it will map to whatever class definition was generated for its corresponding complex type definition. This may be `ShippingAddress` or `Address` or whatever was defined in the XML schema. We can't guess by simply looking at the XML name of the element, we need to look at the XML schema itself.

The developers of Castor couldn't decide which approach was best, so we simply allow you to choose the one that makes most sense to you.

Now that we have a basic understanding of how Castor's Source Code Generator maps XML Schema types into Java we'll look at a simple example.

> *For more information on how Castor maps XML Schema into Java please read the Source Code Generator documentation at http://www.castor.org/sourcegen.html.*

Generating the Class Definitions for an XML Schema

Recall our invoice example from the previous sections. Let's define the XML schema that represents our invoice. For simplicity our invoice will only contain a shipping address and a collection of items. A sample XML instance would read as follows (`invoice.xml`):

```
<?xml version="1.0"?>
<invoice>

   <shipping-address country="US">
      <name>Joe Smith</name>
      <street>10 Huntington Ave</street>
      <city>Boston</city>
      <state>MA</state>
      <zip>02116</zip>
   </shipping-address>
   ...
   <item sku="9123456123456">
      <name>ACME Hazelnut Coffee</name>
      <description>
         A pound of ACME Hazelnut Coffee - Whole Bean
      </description>
      <price>9.99</price>
      <quantity>2</quantity>
   </item>
   ...
</invoice>
```

Our schema to represent the above XML looks like this (`invoice.xsd`):

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
   targetNamespace="acme.org/Invoice">

   <!-- invoice definition -->
   <element name="invoice">
      <complexType>
         <sequence>
            <element ref="shipping-address"/>
            <element ref="item" maxOccurs="unbounded" minOccurs="1"/>
         </sequence>
      </complexType>
   </element>

   <!-- item definition-->
   <element name="item">
      <complexType>
         <sequence>
            <element name="name"    type="string"/>
            <element name="quantity"  type="integer"/>
            <element name="price"    type="priceType"/>
            <element name="description" type="string"/>
        </sequence>
        <attribute name="sku" type="ID" use="required"/>
      </complexType>
   </element>

   <!-- Shipping address definition, A simple US based address structure -->
   <element name="shipping-address">
      <complexType>
         <sequence>
            <element name="name"  type="string"/>
            <element name="street" type="string"/>
            <element name="city"  type="string"/>
            <element name="state" type="stateCode"/>
            <element name="zip"  type="zipCode"/>
         </sequence>
      </complexType>
   </element>

   <!-- A US Zip Code -->
   <simpleType name="zipCode">
      <restriction base="string">
         <pattern value="[0-9]{5}(-[0-9]{4})?"/>
      </restriction>
   </simpleType>

   <!-- State Code, simply restrict string to two letters.
        Obviously a non valid state code can be used....but this is just
        an example.
   -->
   <simpleType name="stateCode">
      <restriction base="string">
         <pattern value="[A-Z]{2}"/>
      </restriction>
   </simpleType>
```

```
    <!-- priceType definition -->
    <simpleType name="priceType">
       <restriction base="double">
          <minInclusive value="0"/>
       </restriction>
    </simpleType>

</schema>
```

Notice that we use the pattern facet for `zipCode` and `stateCode` simple type definitions. The pattern facet is a regular expression that allows us to define the production of the string value.

Now that we have our XML Schema all we need to do is run the source code generator so that it can create our object model for us. We do this using the command line interface:

>java org.exolab.castor.builder.SourceGenerator –i invoice.xsd –package org.acme.invoice

This invokes the source code generator on our invoice schema and creates our source code using the package "`org.acme.invoice`". The source code generator will create the source files, which will then need to be compiled. If the directory for the package (in our example this would be `org/acme/invoice`) does not exist it will automatically create and all generated source will be placed in this directory.

> *The source code generator allows a number of different command line options to be specified. For more information on these options please refer to the Castor documentation.*

Let's take a look at one of the source files generated (`invoice.java`). The method bodies, comments, and some helper methods have been removed so that the file doesn't take up too much page space:

```
package org.acme.invoice;

/* imports and javadoc removed for brevity */

public class Invoice implements java.io.Serializable {

   private ShippingAddress _shippingAddress;

   private java.util.Vector _itemList;

   public Invoice() { ... };

   public void addItem(Item vItem)
      throws java.lang.IndexOutOfBoundsException { ... };

   public Item getItem(int index)
      throws java.lang.IndexOutOfBoundsException { ... };

   public Item[] getItem() { ... };

   public ShippingAddress getShippingAddress() { ... };

   public void marshal(java.io.Writer out)
```

```
      throws org.exolab.castor.xml.MarshalException,
org.exolab.castor.xml.ValidationException
   {

      Marshaller.marshal(this, out);
   } //-- void marshal(java.io.Writer)

  public void setItem(Item vItem, int index)
    throws java.lang.IndexOutOfBoundsException { ... };

  public void setItem(Item[] itemArray) { ... };

  public void setShippingAddress(ShippingAddress shippingAddress) { ... };

   public static org.acme.invoice.Invoice unmarshal(java.io.Reader reader)
      throws org.exolab.castor.xml.MarshalException,
         org.exolab.castor.xml.ValidationException
   {
      return (org.acme.invoice.Invoice)
Unmarshaller.unmarshal(org.acme.invoice.Invoice.class, reader);
   } //-- org.acme.invoice.Invoice unmarshal(java.io.Reader)

   public void validate()
      throws org.exolab.castor.xml.ValidationException { ... };

}
```

As you can see, the generated class definition is a typical Java Bean, which represents the invoice that we have described in our schema. Also notice that `marshal()` and `unmarshal()` methods have been created for convenience (the generation of these methods can be disabled using the special option on the command line when invoking the source code generator, currently called `–nomarshall`). These methods simply call Castor's marshaller and unmarshaller as we have done in earlier sections.

If you look at the generated files you will notice that for each class definition created, such as `Invoice.java`, there is an associated class definition, called a **Class Descriptor**, which is also generated, such as `InvoiceDescriptor.java`. The class descriptors are classes that hold the binding and validation information for their associated class and are used by the marshaling framework. Essentially they are equivalent to the mapping files we discussed earlier.

> *Discussion of the Class Descriptor API goes beyond the scope of this chapter, but more information can be found in the Castor documentation.*

Marshaling and Unmarshaling

Now that we've generated our object model from our schema, let's look at the code necessary to unmarshal and marshal instances of our schema.

As can be seen from the following code (`InvoiceDisplay.java`), when we want to unmarshal an instance of an invoice we simply call the `unmarshal()` method:

```
//-- import our generated classes
import org.acme.invoice.*;
```

```
    ...
    try {
        //-- Unmarshal Invoice
        FileReader reader = new FileReader ("invoice.xml" );
        Invoice invoice = Invoice.unmarshal(reader);

        //-- Display Invoice
        System.out.println("Invoice");
        System.out.println("-------");

        System.out.println("Shipping:");
        System.out.println();

        ShippingAddress address = invoice.getShippingAddress();

        System.out.println("   " + address.getName());
        System.out.println("   " + address.getStreet());
        System.out.print("   " + address.getCity() + ", ");
        System.out.println(address.getState() + "  " + address.getZip());

        System.out.println();
        System.out.println("Items: ");

        Item[] items = invoice.getItem();

        for (int i = 0; i < items.length; i++) {
             Item item = items[i];
             System.out.println();
             System.out.print(i+1);
             System.out.println(". Product Name: " + item.getName());
             System.out.println("   Sku Number  : " + item.getSku());
             System.out.println("   Price       : " + item.getPrice());
             System.out.println("   Quantity    : " + item.getQuantity());
        }
     }
     catch(MarshalException mx) {...}
     catch(ValidationException vx) {...}
     catch(IOException iox) {...}
```

The full source code for `InvoiceDisplay.java` is available in the code download for the book from the Wrox web site at http://www.wrox.com.

After running `InvoiceDisplay` we get the following result:

```
>java InvoiceDisplay

Invoice
-------
Shipping:

   Joe Smith
   10 Huntington Ave
   Boston, MA  02116
```

```
Items:

1. Product Name: ACME Hazelnut Coffee
   Sku Number  : 9123456123456
   Price       : 9.99
   Quantity    : 2

2. Product Name: ACME Ultra II Coffee Maker
   Sku Number  : 9123456654321
   Price       : 149.99
   Quantity    : 1
```

When we want to marshal an instance of our `Invoice` class we simply call the `marshal()` method, again in our example we will update the quantity of coffee from 2 pounds to 4 (`InvoiceUpdate.java`):

```
      //-- Unmarshal Invoice
      Invoice invoice = Invoice.unmarshal( new FileReader ("invoice.xml" ));

      //-- find item for our Hazelnut coffee
      String sku = "9123456123456";

       Item[] items = invoice.getItem();

       for (int i = 0; i < items.length; i++) {
           //-- compare sku numbers
           if (items[i].getSku().equals(sku)) {
               //-- update quantity
               items[i].setQuantity(4);
               break;
           }
       }

       //-- marshal our updated invoice
       FileWriter writer = new FileWriter("invoice_updated.xml");
       invoice.marshal( writer );
```

That's all there is to it!

Simply run `InvoiceUpdate` as follows:

```
>java InvoiceUpdate
```

If we run our `InvoiceDisplay` program using the `invoice_updated.xml` document created by our `InvoiceUpdate` program we will get the following:

```
>java InvoiceDisplay invoice_updated.xml

Invoice
-------
Shipping:

   Joe Smith
```

```
    10 Huntington Ave
    Boston, MA  02116

Items:

1. Product Name: ACME Hazelnut Coffee
    Sku Number  : 9123456123456
    Price       : 9.99
    Quantity    : 4

2. Product Name: ACME Ultra II Coffee Maker
    Sku Number  : 9123456654321
    Price       : 149.99
    Quantity    : 1
```

Conclusion

We've discussed briefly how to specify structure and types for our XML documents using a W3C XML Schema, and we've seen how to use the source code generator to automatically create an object model for our schema. The main benefit of using the source code generator is that we have very little work to do as developers in order to access our XML instances in Java. The entire object model to represent the schema is created automatically, including all data binding and validation code, which can save us valuable time when writing our XML applications. If our XML schema changes, it's very easy to regenerate the object model and maintain our applications.

Using XSLT with XML Data Binding

XSLT, the W3C's language for performing XML transformations, seems to have relevance in almost any topic on XML so it should be no surprise that XSLT can be used in conjunction with XML data binding.

One interesting use of XSLT with XML data binding is to create presentable views of our object model for use in web applications. Two other important issues in data binding where XSLT plays an important role is with the handling of complex schema and sharing data with other systems. Both of these issues typically deal with schema conversion – when faced with the need to convert between one XML structure and another we turn to XSLT.

This section is meant to give you some ideas about how you can use XSLT with XML data binding to solve real-world problems. We'll begin with a brief discussion about using XSLT to create presentable views of our object model, and then we will discuss schema conversion.

It might be necessary to read Chapter 9, "Transforming XML", which covers XSLT, before continuing.

Creating Presentable Views of Object Models

When writing applications we often have the need to present application data to the user. Consider our Invoice object model. An invoice is something that should clearly be displayed to the user of our application after an order has been placed.

Imagine that we needed to provide an HTML view of our invoice. We could write a lot of Java code to output our `Invoice` object as HTML. This would be pretty ugly, because our code would have lots of `print` statements with embedded HTML markup. Not only would writing the initial display code be tedious and time consuming, but also maintaining this code would pose some serious headaches.

A better solution would be to separate the presentation information from our application. Since XML data binding allows us to marshal our object models in an XML format, adding presentation information to our data can be quite easy.

Castor supports marshaling object models to both SAX and DOM APIs. This makes connecting the XML output of Castor to an XSLT processor that supports either of these standard APIs straightforward.

Schema Conversion

Converting Complex Schemas To Simpler Ones

It may be the case that a schema is too complex to be handled properly by a data-binding framework (or it might be that our object model and the schema are too far removed from each other to be directly connected by mapping). This often happens with automatically generated schemas, such as schemas generated from DTDs. XML Schema is very expressive and there is more than one way to write a schema that describes the same structure.

In a complex schema, information contained in attributes, child elements or potentially elsewhere in the XML document may be needed to determine the proper binding type for a given element. For example we could have the following element declaration:

```
...
<xsd:element name="address" type="address"/>
...
<xsd:complexType name="address">
  <xsd:all>
      <xsd:element name="name"  type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"  type="xsd:string"/>
      <xsd:element name="state" type="stateAbbrev"/>
      <xsd:element name="zip"  type="zip-code"/>
  </xsd:all>
  <xsd:attribute name="type">
      <xsd:simpleType>
         <xsd:restriction>
            <xsd:enumeration value="shipping"/>
            <xsd:enumeration value="billing"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:attribute>
</xsd:complexType>
...
```

Though we didn't show it, assume we have declared `stateAbbrev` type to be an enumeration of the available US state codes, and that we have declared the `zip-code` type to be a pattern consisting of 5 digits, followed by an optional hyphen and 4 digits.

A sample instance of the address element would look like:

```
<address type="shipping">
   <name>Pita Thomas</name>
   <street>2000 Endless Loop</street>
```

```
      <city>Redmond</city>
      <state>WA</state>
      <zip>98052</zip>
   </address>
```

Now assume in our object model that we have two separate objects for the address; we have a `BillingAddress`, and a `ShippingAddress` object, both of which extend an abstract base class called `Address`.

During unmarshaling we need know that an address element with a `type` attribute value of "`shipping`" should be mapped to the `ShippingAddress` class. This poses a problem since this involves more than simply mapping the address element to a particular class. It also involves the value of the `type` attribute. This situation is a one-to-many mapping between element and class definition. Our example isn't very complex, but sometimes one-to-many mappings can be very difficult to handle with a data binding framework.

To make things simple we can use an XSLT stylesheet to convert our above XML instance into an XML instance that adheres to the following schema representation:

```
...
<xsd:element name="shipping-address" type="address"/>
<xsd:element name="billing-address" type="address"/>
...
<xsd:complexType name="address">
   <xsd:all>
      <xsd:element name="name"  type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"  type="xsd:string"/>
      <xsd:element name="state" type="stateAbbrev"/>
      <xsd:element name="zip"  type="zip-code"/>
   <xsd:all>
</xsd:complexType>
...
```

So instead of having an attribute that specifies the type of address, we simply have two separate elements. We've basically reduced our one-to-many mapping situation into a one-to-one mapping situation, which is very easy to handle.

The new XML instance would look like the following:

```
<shipping-address>
   <name>Pita Thomas</name>
   <street>2000 Endless Loop</street>
   <city>Redmond</city>
   <state>WA</state>
   <zip>98052</zip>
</shipping-address>
```

Now that there is a one-to-one mapping between the shipping-address element and the `ShippingAddress` class, we can easily perform unmarshaling and marshaling of the XML instances.

The following XSLT stylesheet can be used to convert the first format into the second so that we can perform unmarshaling:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0">
   <xsl:template match="address">
      <xsl:choose>
         <xsl:when test="@type='shipping'">
            <shipping-address>
               <xsl:copy-of select="*"/>
            </shipping-address>
         </xsl:when>
         <xsl:otherwise>
           <billing-address>
              <xsl:copy-of select="*"/>
           </billing-address>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
</xsl:stylesheet>
```

This stylesheet needs just one template for our `address` element. Based on the value of the `type` attribute we simply create either a `<shipping-address>` or `<billing-address>` element, and then copy the remaining data.

This stylesheet only performs the transformation in one direction; when we marshal we would need another stylesheet, or at least another template, that would then convert the marshalled XML instance back into the first format. I won't show it here, but it would be quite similar to the above.

So we see how XSLT can play an important role in allowing us to perform data binding on schemas that might be too complex for our data-binding framework. Of course, using XSLT before unmarshaling and after marshaling will impose additional overhead and degrade performance.

Converting Schemas in Order To Share Data

When writing non-trivial applications we often find that we must share data with other applications and systems within our own organization and, as with B2B applications, potentially with other organizations. It is more common than not for two systems or organizations to have different schemas for representing the same data. With this in mind, it may be necessary to transform the XML data before data binding takes place. XSLT can be used to convert our schemas into other schemas so that we can share our data.

To illustrate this suppose that we work in a particular division of a company that has shipping information that needs to shared with another division of our company, or another company for that matter. Suppose that our division within the company has decided to represent shipping addresses as follows:

```
<shipping-address country="US">
   <name>Gignoux Coffee House</name>
   <street>24 Kernel Street</street>
   <city>Redmond</city>
   <state>WA</state>
   <zip>98052</zip>
</shipping-address>
```

All of our data binding code will unmarshal and marshal XML instances as long as they are in the above format.

Now let's assume that the other division or company uses the following XML format to represent the same data:

```
<shipTo>
   <name>Gignoux Coffee House</name>
   <address country="US">
      <street>24 Kernel Street</street>
      <city>Redmond</city>
      <state>WA</state>
      <zip-code>98052</zip-code>
  </address>
</shipTo>
```

All of their data binding code relies on the shipping addresses to be in the above format in order to be properly unmarshalled.

When we need to share the data we can use an XSLT stylesheet to convert to and from both formats. If our division needs to give the other division or company the data in their respective format, we can simply use an XSLT stylesheet just after we marshal our object model to convert the resulting XML instance into the proper format. If the other division or company is responsible for the data conversion, they can simply use an XSLT stylesheet just before unmarshaling to convert the data into the proper format.

We can use the following XSLT stylesheet to convert from the first format into the second:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0">

   <xsl:template match="shipping-address">
      <shipTo>
         <xsl:copy-of select="name"/>
         <address country="{@country}">
           <xsl:apply-templates select="street | city | state | zip"/>
         </address>
      </shipTo>
   </xsl:template>

   <xsl:template match="*">
      <xsl:copy-of select="."/>
   </xsl:template>

   <xsl:template match="zip">
      <zip-code><xsl:value-of select="text()"/></zip-code>
   </xsl:template>

</xsl:stylesheet>
```

Again this stylesheet only transforms the first format into the second, so we would need another stylesheet that we could use after marshaling.

We can see how XSLT can play an import role in enabling us to share XML data and aid us with integration of data that is not in the proper format.

XML Data Binding Instead of Object Serialization

Serialization refers to Java's built-in persistence mechanism for saving object instance information. The value of each field in the object instance is saved along with type information so that the object instance can be recreated when needed. It is quite analogous to the XML marshaling we have been discussing in previous sections, except that instead of an XML instance, the data is stored in a binary format. Of course, I'm simplifying the analogy a bit as there are other differences than simply how the data is stored and we'll see some of that throughout this section.

If both XML data binding and Java Serialization can save and recreate object instances, how do we determine which method to use? It really depends on the situation at hand, such as what we need to do with the persisted state information or how fast our application needs to be. In most cases XML data binding has clear advantages over serialization. There are some key areas to look at when making a decision on which one to use; these are sharing data, data accessibility, performance, and object model constraints. There are some other differences as well, which will be noted at the end of the section.

Sharing Data Across Systems

Java Serialization is class specific. This means that in order to de-serialize (or restore) an object, we must use the exact same class definition that was used during serialization. If we want to share serialized object instances with other systems, then those systems or applications must also have the same set of classes. This is because the job of serialization is to save an object's instance…not simply the data of the object. There are some workarounds to this issue, such as having the class definition maintain the `serialVersionUID` to trick the JVM into thinking we have the same class definition, or implementing the `Externalizable` interface, which would be a lot of work.

XML data binding does not have this limitation because we're not actually saving object instances, but instead we are simply saving the data of the object. As we have seen, an XML instance can be mapped into different object models.

One goal of data binding is to be agnostic about how the data is stored. Using data binding we can control the stored representation of the object. The format for Java serialization is a binary format that we have no control over.

Java Serialization only works with Java! Serialized objects cannot be de-serialized in a different programming language. XML data binding can be done in almost any language. For example we can marshal a Java object model and unmarshal into a C++ object model. This of course requires that either a data-binding framework exists in the language or we create the data binding code. Being language agnostic is important if we need to share data with other applications, especially if we need to share our data with other companies.

XML data binding can also be used to take advantage of the growing number of XML based messaging formats and protocols by simply marshaling our objects directly to the desired XML format.

Data Accessibility

Java Serialization saves the persisted data in a binary format. This means it's not human readable. If we have the need to read the persisted data then a binary format is not very desirable. It is also means we couldn't easily open this data in a text editor and update values. XML on the other hand is character data, simply formatted using a special syntax. This means that humans can easily read it. Some XML instances may be very long, and not very nice to read, but we can still read it if we need to. We can also update values using a simple text editor.

Performance

Performance is where serialization has a distinct advantage over data binding. Since Castor generated classes implementing the `Serializable` interface, it's actually quite easy to compare the two approaches.

Here's a simple performance test (`PerformanceTest.java`) using our `Invoice` object model that we created earlier with Castor's source code generator. The test case performs a number of iterations

of serializing, deserializing, marshaling and unmarshaling and simply averages the result for each separate test.

```
int    attempts  = 1000;
...
// perform serialization
start = System.currentTimeMillis();
for (int i = 0; i < attempts; i++) {
    FileOutputStream fstream = new FileOutputStream("invoice.ser");
    ObjectOutputStream ostream = new ObjectOutputStream(fstream);
    ostream.writeObject( invoice );
    ostream.close();
}
stop = System.currentTimeMillis();
total = stop - start;
average = (double)total/(double)attempts;
...
// print average
System.out.println("avg time to serialize: " + average + " milliseconds");
...
// perform marshaling
start = System.currentTimeMillis();
for (int i = 0; i < attempts; i++) {
   FileWriter writer = new FileWriter("invoice2.xml");
   invoice.marshal( writer );
   writer.close();
}
stop = System.currentTimeMillis();
total = stop - start;
average = (double)total/(double)attempts;
```

Note: Running this test with virus software running in the background may seriously affect the performance of the marshaling as some virus programs scan reading and writing of files with ".xml" extensions. This is particularly true when running Windows.

I chose to perform 1000 iterations, but feel free to modify this number.

On average, serializing an instance of our Invoice is about 3 times faster than marshaling the same instance using Castor. Deserializing is also about 3 times faster than unmarshaling. It is also interesting to note that the file size for the serialized Invoice instance (839 bytes) is larger than the file size for the XML instance (594 bytes). This will reverse as we add more Items to our invoice and eventually the file size for the serialization will become smaller than the XML instance. I encourage the reader to play around with the test, as results will vary.

For larger object models there are some things we could actually do to speed up the performance of marshaling and unmarshaling such as disabling validation, or caching the descriptors. For example, even with our small `Invoice` instance, if we disable validation and we cache the descriptors by modifying our example as follows (`PerformanceTestOptimized.java`):

```
// import class ClassDescriptorResolverImpl in order to cache
// our class descriptors
import org.exolab.castor.xml.util.ClassDescriptorResolverImpl;
...

    // use our own instance of a ClassDescriptorResolver to
    // cache class descriptors across instances of marshaller
    // and unmarshaller.
    ClassDescriptorResolver cdr = new ClassDescriptorResolverImpl();
    //-- marshaling
    System.out.println();
    System.out.println("Marshaling Invoice object " + attempts + " times");
    start = System.currentTimeMillis();

    for (int i = 0; i < attempts; i++) {
        FileWriter writer = new FileWriter("invoice2.xml");
        Marshaller marshaller = new Marshaller(writer);
        marshaller.setResolver(cdr);
        marshaller.setValidation(false);
        marshaller.marshal(invoice);
        writer.close();
    }
...
```

Notice that we construct our own marshaller, pass in our instance of `ClassDescriptorResolver`, and disable validation. The `ClassDescriptorResolver` is used by the marshaller to load all class descriptors; by passing in our own instance we prevent the marshaller from creating a new one. This allows us to save any class descriptors loaded during the marshaling process. We do the same thing for the unmarshaling process.

So we can see that we've improved our marshaling significantly. Serialization only becomes 2 times faster than marshaling, instead of the previous 3 with the unoptimized version. This improvement should be even better for larger files. Again results will vary, so I encourage the reader to experiment on their own system.

Some Additional Differences

New Class Instances

Since the goal of Serialization is to save an object instance, the JVM will not call any constructors of the object. A new instance is actually created, but no constructors are called and therefore no instructions within the constructors will be executed. To work around this the class definition can supply its own `readObject()` method to perform any initialization that may be necessary.

A data binding framework, like Castor, will actually create new instances of the objects. As we have already seen, Castor typically requires that all class definitions have a default constructor, because Castor will create new instances and therefore must invoke the constructor. Actually, Castor really has no choice in the matter as only the JVM can create an object without calling a constructor.

Public vs. Private

When performing Serialization the JVM has access to all fields: public, protected or private. So it simply deals with the fields directly and not the accessor methods (getters and setters) to save and restore values. A data binding framework can access public fields directly, but has no access to the protected or private fields. Therefore it typically needs to use the public accessor methods to save and restore values of the fields.

There are pros and cons to both approaches. With Serialization we are not required to expose public methods to access private data that needs to be persisted. A drawback to the direct field manipulation of serialization is that if there is any business logic in the accessor methods for a given field it will not be executed. This can be overcome by implementing the `writeObject` and `readObject` methods used by the Serialization API to handle the business logic, however this is more difficult to maintain.

Conclusion

Java Serialization and XML data binding both allow us to save and recreate object instances. Choosing the right approach depends on the situation, but XML data binding has many clear advantages.

Using XML data binding instead of Serialization enables us to:

- Share persisted data across systems
- Obtain object model independence
- Become programming language agnostic
- Read and manually update the persisted information using any text editor

The benefits to Java Object Serialization over XML Data Binding:

- No need to expose access to private fields in order to be persisted
- Faster than XML data binding

Summary

It is typically easier to interact with data in the native format of our programming language, in a manner that depicts the intended meaning of the data, without concern for how the data is stored. Data binding provides the concepts for allowing us to interact with data formats in such a fashion. We learned that, for many XML applications, using XML APIs such as SAX and DOM are often very tedious, lack specific meaning, and force us to work in a format that depicts how the data is stored. Using these APIs is desirable if our applications need to interact with generic XML instances.

Using a data-binding framework allows us to easily write and maintain XML-enabled applications. We have shown examples of using a data binding framework and in doing so we have seen that a data-binding framework allows us to:

- Perform simple XML data binding using an existing object without specifying any binding information
- Express bindings for an existing object model for more complex XML data binding
- Generate a complete object model, including all binding and validation code for a given XML schema

Sometimes an XML schema may be too complex for a specific data-binding framework. We have discussed how we could use XSLT to simplify our XML schema instances to aid data binding. Sharing data with other companies or applications is very common and we have seen how XSLT can help us perform data binding on schemas that are different from what our system expects.

It is often necessary for an application to persist object instances. We have discussed using XML data binding instead of Java Serialization for this purpose, especially when we need to access the data in another system, or if we need to make the persisted data human readable or updateable.

There are many uses for XML data binding and hopefully by reading this chapter you have discovered a way to make programming your XML-enabled applications, easier, more natural, and more enjoyable.

Resources:

Castor

http://castor.exolab.org

Java Beans Specification, version 1.01

http://java.sun.com/products/javabeans/docs/spec.html

Java Serialization

http://www.javasoft.com/products/jdk/1.2/docs/guide/serialization/index.html

JSR 31

http://java.sun.com/aboutJava/communityprocess/jsr/jsr_031_xmld.html

W3C XML Schema

http://www.w3.org/XML/Schema

There are other data binding frameworks apart from Castor. We didn't have time to discuss them in this chapter, but more information about some of them can be found here:

Zeus

http://zeus.enhydra.org

Quick

http://jxquick.sourceforge.net/

Breeze XML Studio

http://www.breezefactor.com/overview.html