Data binding with JAXB

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

In this tutorial you will learn to use data binding to easily map data stored in XML documents to Java objects and back again. You won't have to worry about parsing XML or navigating a tree asking for child nodes and parents. You'll start with an XML Schema and process it into Java source files. Once you have a correspondence between the XML structure and Java classes, you can take an XML document that conforms to the schema and automatically create Java objects that are instances of the classes. Conversely, you can also start with the Java classes and create the XML documents. You will start with an uncomplicated example and build on it as you explore data binding.

I am assuming that you are comfortable programming in the Java language and can write an XML document that conforms to an XML Schema. This tutorial will lead you through each of these steps and you'll end up with an appreciation and an understanding of data binding.

The JAXB tools

This tutorial uses JAXB, the Java APIs for XML Binding from Sun Microsystems. JAXB is a specification available at the *JAXB downloads page*. The reference implementation is available as part of the Java Web Services Developer Pack. Other implementations can be made available by other vendors. In addition, JAXB is not the only technology available for data binding. In this tutorial you will learn the fundamental ideas of data binding by using JAXB and the reference implementation. You can easily apply these techniques and ideas to other tools and technologies.

You will perform two basic activities when working with a data binding solution. You will compile the schema into Java source files that contain interfaces and classes. You will then use these generated classes along with the APIs provided as part of the distribution for handling generated files. The schema compiler is contained in the jar file, **jaxb-xjc.jar**. The remaining functionality is available in the three jar files: **jaxb-api.jar**, **jaxb-libs.jar**, and **jaxb-ri.jar**.

If you explored a previous version of JAXB, you'll notice significant changes in the final release. Most evident and welcome is the change from depending on Document Type Definitions (DTDs) to processing W3C XML Schema. A second change is the introduction of factory methods and interfaces in the generated code. The result of these changes is that you need to update applications written with the first version before they can work with the 1.0 final release of JAXB.

Download and installation

In this tutorial you will use the reference implementation of JAXB for the examples. Although you only need four jar files to work with JAXB, you aren't able to only download those files. At this time, you are required to download and install the entire Java Web Services Developer Pack (Java WSDP) currently at version 1.1. You can download the Java WSDP, the current Java WSDP tutorial, and sample applications from the *Sun Web Services downloads page*. Sun's umbrella site for Web Services is *http://java.sun.com/webservices/*.

Choose a platform for your download and installation. The UNIX distribution is suitable for Linux, Mac OS X, Solaris, and other flavors of UNIX. The Windows version has been tested on the Windows 2000 and Windows XP Professional Editions. The examples in this tutorial have been tested on both distributions using Windows 2000 and Mac OS X.

Note: If you are used to pressing the "Next" button without carefully reading the instructions during installation, you may miss one important point. If you are using JDK 1.4.x, the JAXP classes are built into the JRE. You need to override these by moving the files unpacked by the Java WSDP installation into **[your installation directory]** \jaxp-1.2.2\lib\endorsed to **[JAVA_HOME]**\jre\lib\endorsed. The actual location of the target may be slightly different depending on your flavor of UNIX. Of course, on a UNIX box the direction of the file separators is reversed.

Conventions used in this tutorial

You have now downloaded the entire Web Services Developer Pack. As mentioned before, the only files that you will be explicitly using are the four jar files located inside of the **[your installation directory]** \jaxb-1.0\lib directory. In this tutorial, that directory is referred to as **[JAXB_LIB]**.

UNIX users should reverse the direction of the file separators in the path description of **[JAXB_LIB]** and other paths described in this tutorial. Although this tutorial was developed on a UNIX box, the paths are given using Windows platform conventions. UNIX users also need to change the `";"` separators to `":"`.

You may be tempted to customize your classpath to point at this **[JAXB_LIB]** directory. Although you may find it convenient to do so for the purposes of this tutorial, it is generally not a good idea to pollute your classpath. Good reasons for not doing so are:

- You often forget that you have made additions to your classpath and so your

application doesn't work when you deploy it because you have forgotten the dependencies.

- Order matters in your classpath. By making additions there, you may break other working applications on your machine.
- On Windows machines, the size of your classpath has an upper limit. A bug that is hard to find the first time can occur when you add something to your classpath and push other characters off the end.

Better solutions include customizing your project settings in your IDE, writing an ANT build.xml file, writing a .bat file or shell script, or typing in the classpath from the command line.

About the author

Daniel Steinberg is the co-author of *Extreme Software Engineering: A Hands-on Approach* (Prentice-Hall), the *Java 2 Bible Enterprise Edition* and the *Java 2 Bible* (HungryMinds). Daniel is a Java trainer and consultant for Dim Sum Thinking. He spends as much time as possible with Kimmy the Wonder Wife and their two daughters. Late at night when his family falls asleep, he sneaks back to his computer to write books and articles about Java programming and industry news. His hobbies include cooking with his daughters, paper engineering, and Java development for Mac OS X. Contact Daniel at *DSteinberg@core.com*.

Section 2. Introduction to data binding

Overview

This section starts with a look at the 50,000-foot view of the idea behind data binding. The underlying concept is surprisingly straightforward: Define a correspondence between an XML schema and a collection of Java classes, and then use this correspondence to map valid XML documents to and from instances of these classes.

After a quick look at two alternatives for handling XML from Java code, you'll plummet to earth and look at a concrete-but-oversimplified example. You'll take a short XML schema and process it, using the schema compiler, into the corresponding Java source files. To better understand what you have to work with, you'll examine some of these files before proceeding to Unmarshalling: From XML to Java objects on page 13 where you'll use them.

The two activities of *producing* the Java files from the XML schema and *using* them are separate and often performed by different developers in more complex settings. To ensure consistency across your enterprise, a single group may be in charge of generating the Java files using the schema compiler and distributing these files as a jar to the teams writing the applications. The generated files are used to read, modify, and produce XML files that can be validated against the schema. This means that changes to the schema require that you reprocess it into the corresponding Java files and redeploy these to your team. Avoid such changes as they break applications and invalidate existing documents.

The fundamental idea of data binding

One of the initial sticking points in learning object-oriented programming is the difference between classes and objects. Now that you have been working with object-oriented programming in Java programming for a while, you may not remember ever being confused. At some point you understood that classes were the molds from which these instances called objects were produced.

A similar correspondence happens on the XML side. A schema of some sort is used to describe the allowable structure for an XML document that will be validated against it. Many XML editors and IDEs can help you produce valid XML documents by providing code assistance prompting you for elements and attributes that the schema defines as allowable. You are being helped to create a document that is, in a sense, an instance of the schema.

The big idea behind data binding is to create a correspondence between these XML

schemas and Java classes and then exploit this mapping when converting XML documents to and from Java objects. Your goal is to process XML using Java code. Working with Java and working with XML are fundamentally different. The data binding provides the correspondence between the templates on either side of this divide. The schema compiler creates Java classes and interfaces based on the structure of an XML schema. Data binding allows you to use the data that is being stored in an XML file without worrying about the structure of this data.

The first step is to generate source files for Java classes from one or more XML schemas using the schema compiler that comes with JAXB. You can then use the JAXB APIs to take the data stored in valid XML files and convert them to Java objects that you can use without worrying about the hierarchy of the XML file. You can modify the data or even create new objects and then persist them as XML documents. Just as with your learning of objects and classes, you'll find that the questions you have at the beginning will disappear in no time at all.

Alternative approaches to data binding

Consider this snippet from an XML document that represents messages on an answering machine.

```
<messages>
  <message>
    <time> 0915 </time>
    <person> mom </person>
    <content> call me when you can </content>
  </message>
  <message>
    <time> 1023 </time>
    <person> boss </person>
    <content>
      where are the release notes
    </content>
  </message>
</messages
```

With data binding, you can handle this in a Java-centric way. You are able to see who left this particular message with a call such as `message.getPerson()`. You can also use other approaches to working with XML from Java code. Two of the most popular approaches are the Simple API for XML Parsing (SAX) and the Document Object Model (DOM). You have access to SAX and DOM through the Java API for XML Parsing (JAXP) that is a part of the Java 1.4 release.

Alternative approaches to data

```
<messages>
 <message>
  <time>
   0915
  </time>
  <person>
   mom
  </person>
  <content>
   call me when you can
  </content>
 </message>
 <message>
  <time>
   1023
  </time>
  <person>
   boss
  </person>
  <content>
   where are the release
   notes
  </content>
 </message>
</messages
```

binding: SAX

SAX is an event-based approach. As the parser works its way through the XML document, you can have it notify you of certain events. For example, you can easily write a SAX-based application that works its way through all of your messages and provides you with a list of who called. In this approach you can picture the parser processing the document and saying, "Oh, here's another message. There's the time. There's the person -- let's notify the application that the person's name is Mom. There's the content. There's the end of the message. Oh, here's another message. There's the time. The person is 'boss' -- no need to notify anyone ... "

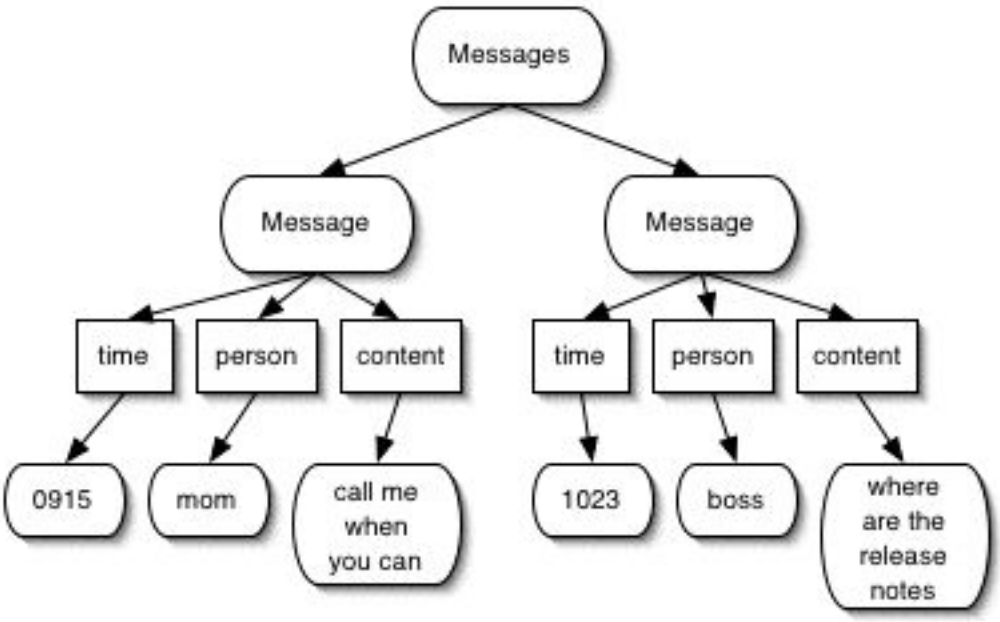
If you get notified that there is a message from Mom and wonder when she called or what she had to say, then you have to reprocess the document. Unless you explicitly persist data as you process it, the information is gone. One way to handle this with SAX is to store the information temporarily until you decide what to do with it. For example, you could store all of the information for a message as it goes by and if you get a call from Mom, then report the time and the contents as well. If not, then free the memory and process the next message.

SAX is a great choice when you just need to respond to events as the file is parsed. The memory requirements are minimal because you are not storing the parsed document in memory. In addition to being lightweight, SAX also tends to be a fast solution. On the down side, you won't get the rich type checking of a data binding solution. With SAX you might look for the `person` element by asking `if (qName.equals("person"))`. The compiler can't flag errors that may arise at runtime as the name of the tag has been passed in as a `String`.

Alternative approaches to data binding: DOM

DOM is a tree-based approach. The result of parsing an XML file with a DOM-based parser is a document object that contains a structured representation of the file. Navigating the DOM usually involves code that looks like `messageNodeList.item(i).getFirstChild().getFirstChild().getData()`.

Here's a DOM tree view of the document:



DOM is a great choice when you need to take advantage of the known structure of a document. You can search through the messages and find any messages from mom. Then you can look for the contents of the element `time` that is a sibling of the `person` element that contains the string *mom* as data. Using DOM has two primary drawbacks. The first is that it is very memory intensive -- the XML document is loaded into memory as a tree. The second is that the code for reading from, manipulating, or writing to an XML document using DOM is tedious and difficult to debug. It would be easier to use code like this:

```
if(message.getPerson().equals("mom")) {
    return "Your mom called at " + message.getTime() +
           "and left the message " + message.getContents();
}
```

JDOM addresses some of the issues involved in using DOM. It requires less memory, is faster than DOM, and supports a more Java-like syntax (see Resources on page 37).

Describing the XML structure of an item in a to-do list

The running example for this tutorial is a to-do list. This list consists of items which in turn contain elements with information about that particular item. For now, I'll begin with a schema that defines an `item` as the root element. An `item` element contains a name in the form of a string, a priority in the form of an integer (`int`), and a task included in that item that is also represented by a string. Here is the initial version of your schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="item">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="priority" type="xs:int"/>
        <xs:element name="task" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

With this uncomplicated schema, you can now begin to understand data binding. Next, you will process this file with the schema compiler. You'll spend the remainder of this section examining the files generated by the schema compiler. In Unmarshalling: From XML to Java objects on page 13 , you will use these generated files to transform an XML document that conforms to this schema to Java objects that you can easily manipulate. In Marshalling: From Java objects to XML on page 22 , you'll go in the other direction, producing an XML document from Java objects. Once you understand these processes of unmarshalling and marshalling, you can move on to a more complicated schema.

Using the schema compiler

Create a directory named **ex1** in a convenient location and save the schema listed in Describing the XML structure of an item in a to-do list on page 9 as **item.xsd** inside of this directory. Open a terminal window and navigate inside of the **ex1** directory.

The next step is to generate the Java class files using the schema compiler. Use the **xjc** application that is part of the JAXB distribution. The schema compiler is shipped in the executable jar file **xjc.jar**. You need to pass in the name of the XML schema being processed as a command-line argument. You can also view the optional flags that you can set by passing in the flag `-help` in place of the name of a schema. In this particular example, process the file **item.xsd** with the following command:

```
java -jar [JAXB_LIB]\xjc.jar item.xsd
```

You can view the optional flags that you can set by passing in the flag `-help` in place of the name of a schema, like this:

```
java -jar [JAXB_LIB]\xjc.jar -help
```

When you process `item.xsd`, you get feedback that the schema file is being parsed. One advantage of using an XML schema instead of a DTD is that an XML schema is itself an XML file that can be validated and checked to make sure it is well-formed. Once this step is completed and **item.xsd** is successfully parsed, you get feedback that the schema is being compiled. Finally, you see a list of the files generated by the schema compiler. The output should look something like the following:

```
parsing a schema...
compiling a schema...
generated/impl/ItemImpl.java
generated/impl/ItemTypeImpl.java
generated/Item.java
generated/ItemType.java
generated/ObjectFactory.java
generated/jaxb.properties
generated/bgm.ser
```

The generated Item interface

You need to create objects of type `Item` that correspond to documents that conform to the schema specified in **item.xsd**. There is surprisingly little to the **Item.java** file generated by the schema compiler. Once you remove the comments, here are the entire contents of `Item.java`:

```
package generated;

public interface Item
    extends javax.xml.bind.Element, generated.ItemType{}
```

Four things are worth noting:

- `Item` is a member of the `generated` package. You can choose to change or extend this name using the `-p` option with the schema compiler. Leaving the name as it is helps you identify the code in your project that has been generated by a tool. In addition, you can use the `-d` option to specify the target directory for the generated files.
- `Item` extends the `javax.xml.bind.Element` interface. This interface is a marker interface and so does not declare any method signatures.
- `Item` extends the `generated.ItemType` interface. This interface specifies the method signatures for the getters and setters for the members that correspond to the elements that make up the `item` element.

- `Item` is itself an interface. JAXB uses factory methods to construct an object of type `Item`. You do not need to understand the details of the class the implements the `Item` interface to effectively use it.

The generated ItemType interface

The **item.xsd** schema could have been written slightly differently so that you could define a complex type with a particular name, say `typeName`, and then reference it like this:

```
<xsd:element name="item" type="typeName">
```

Instead, a snippet that includes the definition of `item` looks like this:

```
 <xs:element name="item">
    <xs:complexType>
```

Since you did not name the complex type, JAXB automatically names the corresponding Java interface `ItemType`. The code generated for `ItemType` looks like this:

```
package generated;

public interface ItemType {
    java.lang.String getTask();
    void setTask(java.lang.String value);
    int getPriority();
    void setPriority(int value);
    java.lang.String getName();
    void setName(java.lang.String value);
}
```

The interface `ItemType` is also in the `generated` package and contains the getters and setters for the three components of an item.

The other generated files

Details about the remaining files are sketched here for completeness. You can still use JAXB without understanding the other files generated by the schema compiler. You will be creating and using objects of type `Item` and `ItemType`. As these are both interfaces, you must have actual concrete classes that implement these interfaces. In addition, you must have a way of specifying which concrete classes should be instantiated when you want a class to be specified by the interface type. These tasks

are fulfilled by the remaining files generated by the schema compiler.

Find the **impl** subdirectory of the directory named **generated**. This contains the files that implement the `Item` and `ItemType` interfaces. You will not directly interact with these files and don't need to understand much more than their existence. The `ItemImpl` class extends the `ItemTypeImpl` class just as the `Item` interface extended the `ItemImpl` interface. Take a quick look at the generated code for these implementation classes. In Unmarshalling: From XML to Java objects on page 13 , you'll customize one of them.

You do not directly instantiate the implementation classes. You use a factory method that instantiates an object of type `Item` or `ItemType`. You find the details of this in the **generated** directory in the following snippet from the `ObjectFactory` class that is also generated by the schema compiler.

```
package generated;

public class ObjectFactory
          extends com.sun.xml.bind.DefaultJAXBContextImpl
{
    private static java.util.HashMap defaultImplementations
                                       = new java.util.HashMap();
    static {
        defaultImplementations.put((generated.ItemType.class),
                                   (generated.impl.ItemTypeImpl.class));
        defaultImplementations.put((generated.Item.class),
                                   (generated.impl.ItemImpl.class));
    }
//... the remainder of the class not included in this listing
}
```

A `HashMap` has been created to provide the mappings in this class. The keys are the interfaces `Item` and `ItemType`, and the corresponding values are the implementing classes. When a call is made to instantiate and return an object of type `Item`, an instance of `impl.ItemImpl` is returned. The `ObjectFactory` itself gets called through a similar trick. A data binding solution that implements the JAXB spec is required to behave correctly in this respect.

Section 3. Unmarshalling: From XML to Java objects

Overview

In Introduction to data binding on page 5 , you created the Java files that are associated with the XML schema. In the **unmarshalling** process you begin with a valid XML file and convert it to Java objects that are instances of the classes that were created by the schema compiler. In this section, you will take a valid XML file, unmarshal it, and then access the information contained in it. To unmarshal a document, you need a handle to the custom unmarshaller that was created by the schema compiler. You then pass it some source for XML and process that source. In this example, the source is a file. You cannot rely on the validity of your XML source files, so you will add the ability to validate them before you try to unmarshal them.

A sample XML document

Create a sample XML document that can be validated against **item.xsd**. Call it **item.xml** and save it in the **ex1** directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<item xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="item.xsd">
  <name>Clean room.</name>
  <priority>3</priority>
  <task>Pick up clothes.</task>
</item>
```

Validate it if you are set up to easily do so. You will validate it using JAXB before the end of this section.

The application shell

Start by copying the following code listing into a file named **ProcessItem.java** and save it in the **ex1** directory. Compile the code and run it. You can either use your favorite IDE or just open a terminal window, navigate to the **ex1** directory, and execute `javac ProcessItem.java` followed by `java ProcessItem`. The code should compile without any complaints and the application should run without doing anything, and then return the command prompt. If nothing happens, that means everything is working correctly.

Where you are heading is contained in this code. The `main()` method creates an instance of `ProcessItem`. In the constructor, you see that the code performs two activities. The instance of `Item` is created in response to the method call `createItem` and then the contents of `Item` are read during the `readItem()` method call.

Creating an `Item` object consists of four steps:

1. Create a context that is used to create objects in the `greeting` package.
2. Create an unmarshaller from the context that will be used to process the XML file into the corresponding Java objects.
3. Create a `file` object from the XML file so that the unmarshaller can have access to the contents of the file.
4. Create an `Item` object by unmarshalling the file.

At each step, you check for exceptions that can be thrown. At the end, you can combine steps and streamline the code, but it is constructed this way so that you can add the functionality one step at a time. The package name and XML file name are included as private instance variables to make customization easy if you choose other names.

```
public class ProcessItem {
  private String packageName = "generated";
  private String xmlFileName = "item.xml";

  ProcessItem() {
    createItem();
    readItem();
  }
  private void createItem() {
      createContext();
      createUnmarshaller();
      createFile();
      unmarshalFile();
  }
  private void createContext() {}
  private void createUnmarshaller() {}
  private void createFile()  {}
  private void unmarshalFile() {}
  private void readItem() {}

  public static void main(String[] args) {
    new ProcessItem();
  }
}
```

The JAXBContext

You get a handle to the right tools for unmarshalling, marshalling (converting from Java

to XML), and validating your XML by using `JAXBContext`. The `ObjectFactory` extends an implementation of `JAXBContext`. The easiest way to think about it is that you have to start with an object of type `JAXBContext` for the particular schema you want to work with in order to access any of the interesting functionality of data binding.

In the `createContext()` method listed below, you have to pass in the name of the package you are creating a context for as a string. Any time you pass an argument as a string, the compiler won't be able to help you find errors. If you misspell the name of a method, then the compiler screams. If you mistype the name of a package passed in as a string, the compiler is not able to catch it. You do, however, get a `JAXBException`. Here is the method for creating the context.

You need to make four changes to your code:

- Add import statements for `JAXBContext` and `JAXBException`.
- Declare an instance variable `jaxbContext` of type `JAXBContext`.
- Create the body of the `createContext()` method and declare that it throws the exception.
- Wrap the call to `addContext()` in a `try` block and catch the `JAXBException`.

Here are the parts of the code that are changed.

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;

public class ProcessItem {
  private String packageName = "generated";
  private String xmlFileName = "item.xml";
  private JAXBContext jaxbContext;
  //...
  private void createItem() {
    try {
      createContext();
      createUnmarshaller();
      createFile();
      unmarshalFile();
    } catch (JAXBException e) {
       System.out.println("There has been a problem either creating the "
          + "context for package '" + packageName +
          "', creating an unmarshaller for it, or unmarshalling the '" +
           xmlFileName + "' file. Formally, the problem is a " + e);
    }
  }
  private void createContext() throws JAXBException {
    jaxbContext = JAXBContext.newInstance(packageName);
  }
  //...
}
```

To compile this, you need to add ***[JAXB_LIB]*\jaxb-api.jar** to the classpath. To prepare for future iterations, set your classpath for compiling and running the application to:

[JAXB_LIB]\jaxb-api.jar;*[JAXB_LIB]*\jaxb-ri.jar;*[JAXB_LIB]*\jaxb-libs.jar;.

Make certain this first time that you compile the source files in the `generated` directory as well. You can check that the exception is being thrown by misspelling the package name "greeting", compiling, and running the application. A `JAXBException` should be thrown. Change the spelling back, compile, and run the application one more time to make sure everything is working again.

The unmarshaller

The `JAXBContext` you just created is now used to return an unmarshaller. An unmarshaller is used to transform the XML file to the corresponding Java objects. Creating an unmarshaller is done in the `createUnmarshaller()` method. This method calls the `createUnmarshaller()` method of the `jaxbContext` object and returns the unmarshaller created by that method. You will again have to account for problems that may occur by throwing a `JAXBException`.

The changes to the existing code are:

- Add an import statement for `Unmarshaller`.
- Declare a private instance variable named `unmarshaller` of type `Unmarshaller`.
- Fill out the body of the `createUnmarshaller()` method and declare that it throws a `JAXBException`.

The changes to the code are highlighted here:

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

public class ProcessItem {
  private String packageName = "generated";
  private String xmlFileName = "item.xml";
  private JAXBContext jaxbContext;
  private Unmarshaller unmarshaller;
//...
  private void createUnmarshaller() throws JAXBException {
    unmarshaller = jaxbContext.createUnmarshaller();
  }
//...
  }
}
```

The `unmarshal()` method in the `Unmarshaller` interface has many signatures. You can unmarshal an XML document from a `File`, as you have done here, or from an `InputStream`, `URL`, `StringBuffer`, `dom.Node`, or `sax.SAXSource`. Code

examples are given in the JavaDocs for the APIs that are accessible from the JAXB distribution's **docs** directory.

The File

Once you have an unmarshaller, you do not need the `JAXBContext` for anything else in this application. You do need a file for the unmarshaller to unmarshal, and that is your next step. Create a new `File` object to represent the XML file, **item.xml**. Unlike `FileInputStream`, the `File` constructor does not throw a `FileNotFoundException`. You need to make sure that a `File` object has been constructed and if not, throw a `FileNotFoundException`.

The changes to the existing code are:

- Add `File` and `FileNotFoundException` to the import list.
- Declare a private `File` variable named `file`.
- Fill out the body of the `createFile()` method, declare that it throws a `FileNotFoundException`, and add the code to do so.
- Catch and handle the `FileNotFoundException` in the `createItem()` method.

```
//...
import java.io.FileNotFoundException;
import java.io.File;

public class ProcessItem {
  private String packageName = "generated";
  private String xmlFileName = "item.xml";
  private JAXBContext jaxbContext;
  private Unmarshaller unmarshaller;
  private File file;
//...
  private void createItem() {
    try {
      createContext();
      createUnmarshaller();
      createFile();
      unmarshalFile();
    } catch (JAXBException e) {
      System.out.println("There has been a problem either creating the "
          + "context for package '" + packageName +
          "', creating an unmarshaller for it, or unmarshalling the '" +
          xmlFileName + "' file. Formally, the problem is a " + e);
    } catch (FileNotFoundException e) {
       System.out.println("There has been a problem locating the '" +
           xmlFileName + "' file. Formally, the problem is a " + e);
    }
  }
 //...
  private void createFile()  throws FileNotFoundException{
```

```
    file = new File(xmlFileName);
    if (!file.exists()) throw new FileNotFoundException();
  }
//...
}
```

The Item object

You now have an `Unmarshaller` object and a `File` object that represents the file to be unmarshalled. What remains is to put them together. Here are the required changes:

- Add `Item` to the import list.
- Declare a private `Item` variable named `item`.
- Fill in the body of the `unmarshalFile()` method and declare that the message throws a `JAXBException`. The `unmarshal()` method of the `Unmarshaller` object returns a generic `Object`. You have to cast it to type `Item`.

```
//...
import generated.Item;

public class ProcessItem {
  //...
  private File file;
  private Item item;
//...
  private void unmarshalFile() throws JAXBException {
    item = (Item) unmarshaller.unmarshal(file);
  }
  private void readItem() {}
//...
}
```

You can now go back and reduce the code size by inlining most of these method calls. On the other hand, the current state of the code is readable and reusable, and the time needed to the create of a few extra instance variables will not be noticeable compared to the time you might otherwise spend unmarshalling your document.

Reading from the Item object

Now it's time to reap the benefits of data binding. Obtaining a context, unmarshaller, file, and item was no more involved than parsing a document using SAX or DOM. With JAXB, however, you can now use Java getters and setters to interact with the XML data. In the `readItem()` method, you print out the contents of the subelements of

`item`. The only change to your existing code is to fill out the body of the `readItem()` method. Here is the entire code listing:

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import java.io.FileNotFoundException;
import java.io.File;
import generated.Item;

public class ProcessItem {

  private String packageName = "generated";
  private String xmlFileName = "item.xml";
  private JAXBContext jaxbContext;
  private Unmarshaller unmarshaller;
  private File file;
  private Item item;

  ProcessItem() {
    createItem();
    readItem();
  }

  private void createItem() {
    try {
      createContext();
      createUnmarshaller();
      createFile();
      unmarshalFile();
    } catch (JAXBException e) {
      System.out.println("There has been a problem either creating the "
          + "context for package '" + packageName +
          "', creating an unmarshaller for it, or unmarshalling the '" +
          xmlFileName + "' file. Formally, the problem is a " + e);
    } catch (FileNotFoundException e) {
      System.out.println("There has been a problem locating the '" +
          xmlFileName + "' file. Formally, the problem is a " + e);
    }
  }

  private void createContext() throws JAXBException {
    jaxbContext = JAXBContext.newInstance(packageName);
  }

  private void createUnmarshaller() throws JAXBException {
    unmarshaller = jaxbContext.createUnmarshaller();
  }

  private void createFile()  throws FileNotFoundException{
    file = new File(xmlFileName);
    if (!file.exists()) throw new FileNotFoundException();
  }

  private void unmarshalFile() throws JAXBException {
    item = (Item) unmarshaller.unmarshal(file);
  }
```

```
  private void readItem() {
    System.out.println("The name of the item is "
                       + item.getName());
    System.out.println("On a scale of 1 to 10 the priority is "
                       + item.getPriority());
    System.out.println("The task associated with this item is "
                       + item.getTask());
    System.out.println(item);
  }

  public static void main(String[] args) {
    new ProcessItem();
  }
}
```

Compile and run the application and you should see this output:

```
The name of the item is Clean room.
On a scale of 1 to 10 the priority is 3.
The task associated with this item is Pick up clothes.
```

Modifying the ItemTypeImpl

To print out the contents of the `Item` object, you should be able to replace the `item.readItem()` call with `System.out.println(item)`. If you do so, you will see something like this:

```
generated.impl.ItemImpl@81b3d4
```

You can fix this by providing a `toString()` method in the `ItemTypeImpl` class. For example, you might use the following as your `toString()` method:

```
 public String toString(){
    return "Name = "+ _Name + ", Priority = "+ _Priority
               + ", and Task = "+_Task;
  }
```

Now when you call `System.out.println(item)`, you will get the following output:

```
Name = Clean room., Priority = 3, and Task = Pick up clothes.
```

You may also find it useful to override the `equals()` and `hashCode()` methods for comparing `Item` objects.

Validating the input file

You can validate the XML file as you unmarshal it. In your application, add the highlighted line to the `createUnmarshaller()` method:

```
private void createUnmarshaller() throws JAXBException {
    unmarshaller = jaxbContext.createUnmarshaller();
    unmarshaller.setValidating(true);
  }
```

Recompile your code. Make **item.xml** invalid by adding this highlighted line:

```
<?xml version="1.0" encoding="UTF-8"?>
<item xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="item.xsd">
  <name>Clean room.</name>
  <priority>3</priority>
  <task>Pick up clothes.</task>
  <time>3</time>
  </item>
```

Rerun your application to see this message.

```
There has been a problem either creating the
  context for package 'generated', creating an unmarshaller
  for it, or unmarshalling the 'item.xml' file. Formally, the
  problem is a javax.xml.bind.UnmarshalException:
  Unexpected element {}:time
  Exception in thread "main" java.lang.NullPointerException
        at ProcessItem.readItem(ProcessItem.java:51)
        at ProcessItem.<init>(ProcessItem.java:19)
        at ProcessItem.main(ProcessItem.java:58)
```

Restore your XML document so that it is valid and rerun your application. When the document is valid, everything should work as before. You may choose to turn off validation so that you do not go through this extra step each time you process a document.

Section 4. Marshalling: From Java objects to XML

Overview

In Unmarshalling: From XML to Java objects on page 13 , you took a valid XML file and transformed it into a Java object. In this section, you will create a Java object and initialize its fields. You'll then marshal this object. Marshalling is the process of turning one or more Java objects into an XML document. You will also validate the object before converting it, and you'll format the output to be more human readable.

The application shell

Save the following code listing as **MakeNewItem.java** in the **ex1** directory:

```
public class MakeNewItem {
  private String packageName = "generated";
  private String destinationName = "newItem.xml";

  MakeNewItem() {
    createNewItem();
    configureItem();
    persistItem();
  }
  private void createNewItem() {}
  private void configureItem() {}
  private void persistItem() {}

  public static void main(String[] args) {
    new MakeNewItem();
  }
}
```

As before, set your classpath to:

```
[JAXB_LIB]\jaxb-api.jar;[JAXB_LIB]\jaxb-ri.jar;[JAXB_LIB]\jaxb-libs.jar;.
```

Compile and run your application. It should not do anything but compile quietly and return a command prompt after running. In particular, it should compile without error messages and should run without throwing exceptions.

Creating an item from scratch

Create an object of type `Item`. This may seem to be a puzzle: You know that you are not supposed to access the `ItemImpl` class yourself, so you need to use an `ObjectFactory` to create a new `Item`. Because the `ObjectFactory` was created by the schema compiler, it has methods for creating new top-level objects.

Make the following changes to the code:

- Add import statements for `ObjectFactory`, `Item`, and `JAXBException`.
- Declare a private instance variable of type `Item` named `item`.
- Fill out the body of the `createNewItem()` method to create an `ObjectFactory` and use it to create an `Item`. To see what you have at this point, add a call to `println(item)` to see the current contents of the newly constructed `Item`. (This only functions correctly if you made the changes to `ItemTypeImpl` described in Unmarshalling: From XML to Java objects on page 13 .)

The changes are highlighted below:

```
import generated.ObjectFactory;
import generated.Item;
import javax.xml.bind.JAXBException;

public class MakeNewItem {
  private String packageName = "generated";
  private String destinationName = "newItem.xml";
  private Item item;

  MakeNewItem() {
    createNewItem();
    configureItem();
    persistItem();
  }

  private void createNewItem() {
    try {
      ObjectFactory itemMaker = new ObjectFactory();
      item = itemMaker.createItem();
      System.out.println(item);
    } catch (JAXBException e) {
      System.out.println("There was this problem creating the item: "
                         + e);
    }
  }
//...
}
```

After compiling and running the code, you'll get these results:

```
Name = null, Priority = 0, and Task = null
```

When the `Item` object is created, the two strings are initialized to `null` because they are reference variables, and the `int` is initialized to 0.

Validating the new Item

When you start with an XML document, you can validate it as part of the unmarshalling process. When you create an object using an `ObjectFactory`, or if you want to validate a document after unmarshalling it, you need to use a `Validator`. Use your `JAXBContext` to return a `Validator` object that is configured to validate against a particular XML schema. You know that at this point your object is not valid because the two strings are still `null`.

To validate, make these changes to your code:

- Add import statements for `JAXBContext`, `Validator`, and `ValidationException`.
- Declare a private `JAXBContext` named `jaxbContext`.
- Add a `createContext()` method that initializes `jaxbContext` and handles a `JAXBException`.
- Insert a call to the `createContext()` method as your first line of the constructor.
- Create an `itemIsValid()` method that creates a `Validator` object and validates `item`. It will also handle `ValidationExceptions` and throw a `JAXBException`.
- Add a call to the `itemIsValid()` method to `createNewItem()` and the code to display the results.

```
import generated.ObjectFactory;
import generated.Item;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Validator;
import javax.xml.bind.ValidationException;

public class MakeNewItem {
  private String packageName = "generated";
  private String destinationName = "newItem.xml";
  private JAXBContext jaxbContext;
  private Item item;

  MakeNewItem() {
    createContext();
    createNewItem();
    configureItem();
    persistItem();
  }
  private void createNewItem() {
    try {
      ObjectFactory itemMaker = new ObjectFactory();
      item = itemMaker.createItem();
      System.out.println(item);
      System.out.println("It is "+ itemIsValid() +
                  " that the  new Item object is valid");
```

```
    } catch (JAXBException e) {
      System.out.println("There was this problem creating the item: " + e);
    }
  }
  private void createContext() {
    try {
      jaxbContext = JAXBContext.newInstance(packageName);
    } catch (JAXBException e) {
      System.out.println("There was this problem creating a context "+e);
    }
  }
  private boolean itemIsValid() throws JAXBException {
    try {
    Validator validator = jaxbContext.createValidator();
    return validator.validate(item);
    }  catch (ValidationException e){
      System.out.println("There was a problem validating the item.");
      return false;
    }
  }
//...
}
```

Here is the result of compiling and running the above code:

```
Name = null, Priority = 0, and Task = null
DefaultValidationEventHandler: [ERROR]: a required object is missing.
DefaultValidationEventHandler: [ERROR]: a required object is missing.
There was a problem validating the item.
It is false that the Item object is valid.
```

Eliminate the `println()` statement that appears in bold in the `createNewMenuItem()` method in the code listing on this page.

Configuring the Item

You created an `Item` object that is empty and hence not valid. Next, configure it by adding values for each of the elements. This is where the value of data binding is evident. You use standard Java setters for setting the values of the elements. If you are working with an IDE, code assist will help you locate the available methods. As highlighted below, these are your only changes to the code.

```
//...

public class MakeNewItem {
//...
  private void configureItem() {
    item.setName("Pick up kids.");
    item.setPriority(10);
    item.setTask("Get kids from school at 1525.");
```

```
    System.out.println(item);
  }
//...
}
```

Compile and run the code and you'll get this:

```
Name = null, Priority = 0, and Task = null
Name = Pick up kids., Priority = 10, and Task = Get kids from school at 1525.
```

At this point, you could validate the `Item` object; you will handle this in the next step before you save the state of the Java objects to a valid XML file.

Marshalling your data

To save your data to a file named **newItem.xml**, check that your `Item` object is valid. If it is, then create a `Marshaller` and use it to marshal your object to the XML file. By this point, you can handle several steps at once. The entire code listing is presented below with the following changes highlighted:

- Add import statements for `Marshaller`, `FileOutputStream`, and `FileNotFoundException`.
- Create a utility method `getFileStream()` that returns a `FileInputStream` with **newItem.xml** as its target.
- Create a utility method `marshallItem()` that creates a marshaller from `jaxbContext`, formats the code nicely by setting the property `JAXB_FORMATTED_OUTPUT`, and forwards to the `marshal()` method in the marshaller.
- Fill in the body of the `persistItem()` method to marshal the data if `item` is valid, and handle a `JAXBException` if anything goes wrong.

```
import generated.ObjectFactory;
import generated.Item;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Validator;
import javax.xml.bind.ValidationException;
import javax.xml.bind.Marshaller;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class MakeNewItem {
  private String packageName = "generated";
  private String destinationName = "newItem.xml";
  private JAXBContext jaxbContext;
  private Item item;
```

```
MakeNewItem() {
  createContext();
  createNewItem();
  configureItem();
  persistItem();
}

private void createNewItem() {
  try {
    ObjectFactory itemMaker = new ObjectFactory();
    item = itemMaker.createItem();
    System.out.println(item);
  } catch (JAXBException e) {
    System.out.println("There was this problem creating the item: " + e);
  }
}
private void createContext() {
  try {
    jaxbContext = JAXBContext.newInstance(packageName);
  } catch (JAXBException e) {
    System.out.println("There was this problem creating a context "+e);
  }
}
private boolean itemIsValid() throws JAXBException {
  try {
  Validator validator = jaxbContext.createValidator();
  return validator.validate(item);
  }  catch (ValidationException e){
    System.out.println("There was a problem validating the item.");
    return false;
  }
}
private void configureItem() {
  item.setName("Pick up kids.");
  item.setPriority(10);
  item.setTask("Get kids from school at 1525.");
  System.out.println(item);
}
 private void persistItem() {
  try {
    if (itemIsValid()) marshallItem();
  } catch (JAXBException e) {
    System.out.println("There was this problem persisting the item: "
                       + e);
  }
}
 private void marshallItem() throws JAXBException {
  Marshaller marshaller = jaxbContext.createMarshaller();
  marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                         new Boolean(true));
  marshaller.marshal(item, getFileStream());
}
private FileOutputStream getFileStream() {
  try {
    return new FileOutputStream(destinationName);
  } catch (FileNotFoundException e) {
    System.out.println("The problem creating a destination file was "
                       + e);
```

```
      return null;
    }
  }
  public static void main(String[] args) {
    new MakeNewItem();
  }
}
```

The generated XML file

This example illustrates the basic steps for creating and persisting an object using data binding. The objects were created using an `ObjectFactory` and persisted using the steps in the `persistItem()` method. These processes are like two slices of bread in a sandwich and don't change very much. The filling is where data binding really shines. It is easy to deal with XML data using the Java accessors. What you do in the middle can be as complex as your needs -- but you no longer need to struggle to navigate the tree or to figure out where you are in an XML document. Accessing and changing the value of the elements remains straightforward.

Compile and run your application and the file **newItem.xml** is created in the same directory as **item.xml**. Open **newItem.xml** and you should see something like this:

```
<xml version="1.0" encoding="UTF-8" standalone="yes"?>
<item>
<name>Pick up kids.</name>
<priority>10</priority>
<task>Get kids from school at 1525.</task>
</item>
```

Section 5. Further exploration

Overview

So far, you have unmarshalled an XML file and accessed the data, and you have created Java objects and marshalled them into an XML file. Often you will be interested in combining these processes into one application: You might want to read data that is stored in an XML file or delivered to you from another source, use it somehow and possibly change it, and then save it back to disk or serve it up to another client application. In this section, you'll modify the schema to be slightly more complex and examine the changes to the Java objects generated by the schema compiler. Then you'll build a utility class designed to help you marshal, unmarshal, and validate, as well as create new objects of the type specified by the XML schema. Finally, you'll create a simple client application to use this utility class. The code is presented in this way to allow you to easily customize it for your own use.

A more complex Schema

Extend your earlier XML schema as follows:

- The root element is now a `todolist`, which contains zero or more `item` elements that are described by the complex type `entry`.
- Each element of type `entry` consists of a string representing the name of the element, an integer between 1 and 10 describing the `priority` as is specified in the `rank` type, and one or more `task` elements that are of type `subentry`. An entry also has an attribute that represents `totalItemTime` that has a default value of zero.
- Each element of type `subentry` consists of a string `description` and an integer `timeEstimate`.

Here is the schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="todolist">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="item"
          type="entry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
  <xs:complexType name="entry">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="priority" type="rank"/>
      <xs:element minOccurs="1" maxOccurs="unbounded" name="task"
        type="subentry"/>
    </xs:sequence>
    <xs:attribute default="0" name="totalItemTime" type="xs:int"/>
  </xs:complexType>

  <xs:complexType name="subentry">
    <xs:sequence>
      <xs:element name="description" type="xs:string"/>
      <xs:element default="0" name="timeEstimate" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="rank">
    <xs:restriction base="xs:int">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="10"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Save this file as **todolist.xsd** in a directory named **ex2**. Process this file with the schema compiler as before and save the generated files in **ex2** as well. Finally, compile the generated files so that you can use them in your sample application.

A valid XML file

In this section, you'll start with a valid XML file and modify it in some way. As an example, you can use the following file. Save it as **todolist.xml** in the **ex2** directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<todolist xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="todolist.xsd">
  <item>
    <name>Clean room.</name>
    <priority>4</priority>
    <task>
      <description>Pick up clothes.</description>
      <timeEstimate>10</timeEstimate>
    </task>
    <task>
      <description>Straighten magazines</description>
      <timeEstimate>20</timeEstimate>
    </task>
  </item>
  <item>
```

```
    <name>Get kids.</name>
    <priority>10</priority>
    <task>
      <description>Pick up kids at school at 1525.</description>
      <timeEstimate>30</timeEstimate>
    </task>
  </item>
</todolist>
```

The generated files: Todolist and TodolistType

As before, the schema compiler generates interfaces, implementation files, a property file, and the `ObjectFactory`. In this schema, the complex types have been named in two cases. The `item` element is of type `entry` and the `task` element is of type `subentry`. The result is methods named `getItem()` and `getTask()` that return lists populated by elements of Java object type `Entry` and `Subentry`. At the top level, the part of the schema shown below maps to two interfaces: `Todolist` and `TodolistType`.

```
<xs:element name="todolist">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="item"
          type="entry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

The `Todolist` interface is just a wrapper that also extends the `Element` marker interface. Once you strip away the comments, you are left with this:

```
package generated;

public interface Todolist
    extends javax.xml.bind.Element, generated.TodolistType {}
```

The `TodolistType` interface contains a list of the objects that correspond to `item` elements of type `entry`. The only method declared in this interface is `getItem()` and it returns a `List`.

```
package generated;

public interface TodolistType {
    java.util.List getItem();
}
```

The generated files: Entry and SubEntry

You access the items on the to-do list using the `getItem()` method in `TodolistType` or create a new one using the `ObjectFactory`. Programmatically, your access to the contents of each item is specified in the `Entry` interface. You can get and set both the `priority` and `name` elements, as well as the total item time attribute. Because an item can contain one or more tasks, you get a `List` of objects of type `Subentry` by calling the `getTask()` method:

```
package generated;

public interface Entry {

    java.util.List getTask();
    int getTotalItemTime();
    void setTotalItemTime(int value);
    int getPriority();
    void setPriority(int value);
    java.lang.String getName();
    void setName(java.lang.String value);
}
```

The Java object representing tasks is `Subentry`. It includes the getters and setters for the description and time estimate of tasks.

```
package generated;

public interface Subentry {

    java.lang.String getDescription();
    void setDescription(java.lang.String value);
    int getTimeEstimate();
    void setTimeEstimate(int value);
}
```

The utility class

Now you can put together what you learned in the first two applications. Create a utility class that can unmarshal your **todolist.xml** file and later marshal any changes you make back to that file. Save this file as **TodolistUtil.java** in the **ex2** directory.

```
import generated.ObjectFactory;
import generated.Todolist;
import generated.Entry;
import generated.Subentry;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.Marshaller;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

```
public class TodolistUtil {
  private JAXBContext jaxbContext;
  private ObjectFactory objectFactory;
  private Todolist todolist;
  private static String packageName = "generated";
  private String xmlFileName = "todolist.xml";

  TodolistUtil() {
    createContextAndObjectFactory();
    createTodolist();
  }

  private void createContextAndObjectFactory() {
    try {
      jaxbContext = JAXBContext.newInstance(packageName);
      objectFactory = new ObjectFactory();
    } catch (JAXBException e) {
      System.out.println("There was this problem creating a context " + e);
    }
  }

  private void createTodolist() {
    try {
      Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
      unmarshaller.setValidating(true);
      todolist = (Todolist) unmarshaller.unmarshal(new File(xmlFileName));
    } catch (JAXBException e) {
      System.out.println("There is this problem with unmarshalling: " + e);
    }
  }

  private void persistTodolist() {
    try {
      if (jaxbContext.createValidator().validate(todolist)) {
        Marshaller marshaller = jaxbContext.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                               new Boolean(true));
        marshaller.marshal(todolist, new FileOutputStream(xmlFileName));
      }
    } catch (JAXBException e) {
      System.out.println("There was this problem persisting the item: "
                         + e);
    } catch (FileNotFoundException e) {
      System.out.println("There was this problem creating a destination file "
                         + e);
    }
  }
}
```

Adding services to the utility file

Note that so far all of the methods and variables on `TodolistUtil` are private. It does not expose any functionality, yet it is not much of a utility: It does not have a

`main()` method, so it has to be called by another object. You can add the ability to create an item, to create a task, and to save a newly created item by adding the following three highlighted methods:

```
//...
public class TodolistUtil {
//...
  public Entry makeNewItem(String name, int priority) {
    Entry newItem = null;
    try {
      newItem = objectFactory.createEntry();

      newItem.setName(name);
      newItem.setPriority(priority);
    } catch (JAXBException e) {
      System.out.println("There was this problem creating a new item: "
                         + e);
    }
    return newItem;
  }
  public Subentry makeNewTask(String description, int time) {
    Subentry newTask = null;
    try {
      newTask = objectFactory.createSubentry();
      newTask.setDescription(description);
      newTask.setTimeEstimate(time);
    } catch (JAXBException e) {
      System.out.println("There was this problem creating a new task: "
                         + e);
    }
    return newTask;
  }
  public void addItem(Entry item) {
    todolist.getItem().add(item);
    persistTodolist();
  }
}
```

In your sample application, you are only creating one instance of the utility class. If you want to ensure this behavior, you should create a private static instance of `TodolistUtil` and serve it up using the Singleton pattern.

A client application

Once again, you are positioned to enjoy the benefits of data binding. All of the JAXB-specific code has been wrapped in the `TodolistUtil` that only exposes three methods: `makeNewItem()`, `makeNewTask()`, and `addItem()`. Here is an example application that creates a new item, creates tasks that it adds to the new item, and then adds the item to the to-do list and saves it back to the XML file. This is fairly typical of how you will use JAXB: You have some data persisted in an XML file; you want to use or modify that data but you do not want to think about how the file is structured. This

allows you to think in terms of your model.

In this code, arguments are passed in on the command line. In a real application, they would more likely be entered through a GUI or from another file. For this application to work, at least two pairs of arguments need to be passed in. The first pair represents the name and priority of the item. Any subsequent pair represents the description and time required for a task belonging to that item. All command-line arguments are read as strings, so the integers need to be converted using the static call `Integer.parseInt(args[i])`:

```
import generated.Entry;
import generated.Subentry;

public class AddNewItem {

  AddNewItem(String[] args) {
    TodolistUtil util = new TodolistUtil();
    if (args.length > 3) {
      Entry newItem = util.makeNewItem(args[0],
                           Integer.parseInt(args[1]));
      for (int j = 2; j < args.length; j = j + 2) {
        Subentry newTask = util.makeNewTask(args[j],
                           Integer.parseInt(args[j + 1]));
        newItem.getTask().add(newTask);
      }
      util.addItem(newItem);
    }
  }

  public static void main(String[] args) {
    new AddNewItem(args);
  }
}
```

Perhaps the most striking feature of this code is that JAXB APIs are not used. You can easily create your own client code that manipulates the XML in different ways. Maybe you can figure out how much time is required to perform all of the tasks on your to-do list. Perhaps you want to sort the list in priority order. Now that you have abstracted the binding layer, creating these solutions is easy.

Running the application

Compile and run the application. You need to pass in the command-line arguments either in your IDE or from the command line. From inside of the **ex2** directory, you can run your application like this (you should enter the following all on one line):

```
java -classpath [JAXB_LIB]\jaxb-api.jar;[JAXB_LIB]\jaxb-libs.jar;
   [JAXB_LIB]\jaxb-ri.jar;. AddNewItem
   "Get food." 4 "Pick up milk." 10 "Pick up cookies" 15
```

After the application runs, the file **todolist.xml** has been changed to this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<todolist>
<item>
<name>Clean room.</name>
<priority>4</priority>
<task>
<description>Pick up clothes.</description>
<timeEstimate>10</timeEstimate>
</task>
<task>
<description>Straighten magazines</description>
<timeEstimate>20</timeEstimate>
</task>
</item>
<item>
<name>Get kids.</name>
<priority>10</priority>
<task>
<description>Pick up kids at school at 1525.</description>
<timeEstimate>30</timeEstimate>
</task>
</item>
<item>
<name>Get food.</name>
<priority>4</priority>
<task>
<description>Pick up milk.</description>
<timeEstimate>10</timeEstimate>
</task>
<task>
<description>Pick up cookies</description>
<timeEstimate>15</timeEstimate>
</task>
</item>
</todolist>
```

Section 6. Summary and resources

Summary

In this tutorial you have learned the basic steps and fundamental ideas of data binding. You have seen how to:

- Create and modify an XML schema to specify the Java classes that will be created.
- Use the schema compiler to produce the Java source files from the XML schema.
- Unmarshal an XML document to the corresponding Java objects.
- Create Java objects and instantiate the variables corresponding to attributes and elements.
- Validate an XML document during unmarshalling, or validate the tree of Java objects at any time.
- Marshal a collection of Java objects to save the data to the corresponding XML document.
- Build a utility class to separate the data binding activities from the business logic.

Resources

- Find news on JAXB as well as the latest downloads, more sample code, documentation, and white papers at *http://java.sun.com/xml/jaxb/*.

- The JAXB distribution is part of the Java Web Services distribution. Find the latest release of the Java Web Services Developer Pack at *http://java.sun.com/webservices/*.

- Check out more great XML tutorials on the developerWorks XML zone's *education* page.

- Learn more about *JAXP (Java APIs for XML Processing), an API that supports processing of XML documents using DOM, SAX, and XSLT* (http://java.sun.com/xml/jaxp/).

- The W3C is the best place to learn about *DOM (Document Object Model) API* (http://www.w3.org/DOM/).

- Get the latest on *JDOM*, a Java-based solution for accessing, manipulating, and

outputting XML data from Java code (http://www.jdom.org).

- Look into *XOM (XML object model)*, an open source, tree-based API for processing XML with Java technology that strives for correctness and simplicity (http://www.cafeconleche.org/XOM/).

- Download the *Xerces2 Java Parser* at the Apache XML Project site (http://xml.apache.org/xerces2-j/index.html).

- IBM WebSphere Studio Application Developer provides support for data binding. You can find more information on the *WebSphere Studio Application Developer information page* (http://www-3.ibm.com/software/awdtools/studioappdev/about/).

- You will find useful links to more tools at the *IBM Java tools* page (http://www-106.ibm.com/developerworks/views/java/tools.jsp).

- Find more information on the technologies covered in this article at the developerWorks *XML* (http://www-106.ibm.com/developerworks/xml/) and *Java technology* (http://www-106.ibm.com/developerworks/java/) zones.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you! Additionally, you are welcome to contact the author, Daniel Steinberg, directly at dsteinberg@core.com.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11.

We'd love to know what you think about the tool.