The Java™ Architecture for XML Binding User's Guide

Early-Access Draft

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

May 2001, Revision 01

Send comments about this document to:

Copyright (c) 2001 Sun Microsystems, Inc. All Rights Reserved.

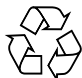
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduct the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Sun Microsystems, Inc. or the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES OR LIABILITIES SUFFERED BY LICENSEE AS A RESULT OF OR RELATING TO USE, MODIFICATION OR DISTRIBUTION OF THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You acknowledge that Software is not designed,licensed or intended for use in the design, construction, operation or maintenance of any nuclear facility.



Please
Recycle

1. Introduction to the JavaTM Architecture for XML Binding (JAXB) 5
Why Use JAXB? 5
JAXB Applications Use Java Technology and XML 6
JAXB Applications Guarantee Valid Data 6
JAXB Applications Are Fast 7
JAXB Applications Are Easy to Create and Use 7
JAXB Applications Can Convert Data 7
JAXB Applications Can Be Customized 8
JAXB Applications Are Extensible 8
Uses of JAXB 9
Scenario 1: Balancing a Checkbook 9
Scenario 2: Comparing Price Quotes from Suppliers 9
Getting the Most From this User's Guide 10
2. Before You Begin: XML Basics 13
What is XML? 13
Document Type Definitions 14
Element Declaration 14
Attribute Declaration 15
XML Documents 16
3. How JAXB Works 19
Overview 19
Binding a Schema to Classes 20
Building Data Representations 23
Unmarshalling 24
Validation 24
Marshalling 24
Working with the Data 25
Limitations 26
4. Binding a Schema to Classes 27
The Example DTD: checkbook.dtd 27
Writing the Binding Schema 29
Creating the Minimum-Required Binding Schema 29
Understanding the Default Binding Declarations 30
The Element Binding Declarations 31
The Attribute Binding Declarations 32
The Content Binding Declarations 33
Customizing the Binding Schema 35
Specifying Types 37
Specifying Non-Primitive Types 37
Specifying Primitive Types 39
Creating Enumerated Types 39
Customizing Content Model Binding Declarations 40
Creating Interfaces 42
Managing Schema Evolution 43
Generating the Java Classes 44

The Generated Java Source Files 45
The Checkbook.java File 45
The Transactions.java File 46
The Entry.java File 48
The Check.java File 48
The CheckCategory.java File 50
The Pending.java File 50
5. Building Data Representations 53
The XML Document Instance: march.xml 53
Setting Up Your Application 54
Building a Content Tree 55
Unmarshalling 55
Instantiation 56
Accessing Content 57
Validating 59
Marshalling 60
Appending Content Trees 61
6. Working With The Data 63
The Example XML Document: checkbook.xml 63
Setting Up the CheckbookBalance Class 64
Extending the Derived Classes 65
Unmarshalling 65
Dispatching 65
Unmarshalling the Subclass 66
Adding Functionality 67
Using the New Functionality in Your Application 68
A. The Example DTD, XML Documents, and Binding Schema 71
B. The Application Files 75

CHAPTER 1

Introduction to the Java™ Architecture for XML Binding (JAXB)

JAXB provides a fast, convenient way to create a two-way mapping between XML documents and Java objects. Given a schema, which specifies the structure of XML data, the JAXB compiler generates a set of Java classes containing all the code to parse XML documents based on the schema. An application that uses the generated classes can build a Java object tree representing an XML document, manipulate the content of the tree, and re-generate XML documents from the tree, all without requiring the developer to write complex parsing and processing code.

Why Use JAXB?

Using JAXB for a data-processing application has many benefits because a JAXB application:

- Uses Java Technology and XML
- Guarantees Valid Data
- Is Fast
- Is Easy to Use
- Can Constrain Data
- Is Customizable
- Is Extensible

This section explains all of these JAXB qualities in more detail.

JAXB Applications Use Java Technology and XML

The most important reasons to use JAXB are that JAXB applications are written in the Java programming language and can process XML data. To understand the implications of these features, you first need to understand why XML and Java technology are so important and how they complement each other.

XML is an industry-standard and system-independent way to represent data. Data that is represented using XML can be published in multiple media because, unlike HTML, XML describes the *structure* of the data, not its format. XML data can be passed between applications because the structure of the data can be specified in a schema, which allows a parser to validate and process data that follows the schema. XML does not provide a set of tags like HTML; you use the schema to define your own tags to describe your particular data. XML data is easy to work with because it is written in a simple text format, readable by both humans and text-editing software. For these reasons, XML is quickly becoming a common method for data interchange between applications, especially business-to-business enterprise applications.

Applications written in the Java programming language are portable: Any system with a Java Virtual Machine[1] can run the bytecode produced by compiling a Java application. With the portable code that Java technology provides, XML is even more useful in the context of sharing data between applications. Applications, especially web-based applications, need the support of Java technology to parse and process the data in a platform-independent manner. Likewise, Java applications need the platform-independent data format that XML provides in order to communicate and share information.

Essentially, JAXB provides a bridge between these two complementary technologies. JAXB includes a compiler that maps a schema to a set of Java classes. Once you have your classes, you can build Java object representations of the XML data that follow the rules that the schema defines. Just as an XML document is an instance of a schema, a Java object is an instance of a class. Thus, JAXB allows you to create Java objects at the same conceptual level as the XML data. Representing your data in this way allows you to manipulate it in the same manner you manipulate Java objects, making it easier to create applications to process XML data. Once you have your data in the form of Java objects, it is easy to access it. In addition, after working with the data, you can write the Java objects to a new XML document. With the easy access to XML data that JAXB provides, you only need to write the applications that will actually use the data, rather than spend time writing code to format the data.

JAXB Applications Guarantee Valid Data

Because JAXB maps schemas to classes, you must have a schema to use JAXB. Some XML parsers and processors do not support schemas or do not require schemas. These processors can be more flexible than JAXB, but without a schema they cannot guarantee that your data

1. As used in this guide, the terms “Java Virtual Machine” or “JVM” mean a virtual machine for the Java platform.

is valid. JAXB, on the other hand, can make this guarantee. In fact, it is impossible to use JAXB to create a Java object tree from an XML document that is invalid with respect to the schema used to create the classes.

JAXB Applications Are Fast

Two commonly-used XML parsing APIs are SAX (Simple API for XML) and DOM (Document Object Model). A SAX parser is an event-driven parser, which means that it reacts to pieces of the document as it is parsing it; it does not store any of the document in memory. A DOM parser builds an in-memory data structure of the document whose contents can be manipulated, but it is much slower than a SAX parser. A JAXB application, on the other hand, has the speed of a SAX parser and the data-storage capability of a DOM parser.

Although SAX parsers are fast, early prototyping of JAXB has shown that JAXB can be faster than SAX parsers. JAXB has faster parsing because the generated Java classes are precompiled and contain the schema logic, thereby avoiding the dynamic interpretation that a SAX parser must perform.

A JAXB application can build an in-memory data structure like a DOM parser. However, unlike DOM, it does not include a lot of extra functionality for tree-manipulation. Unlike a DOM application, a JAXB application is specific to one schema: You cannot use it to process XML documents that are based on another shema. For these reasons, a JAXB application uses memory much more efficiently than DOM.

JAXB Applications Are Easy to Create and Use

Since all the processing code is generated for you, JAXB is easier to use than most XML parsers: You can just input a stream to access the content. In addition, most XML parsers are limited to the data-typing offered by a DTD. A DTD is one kind of XML schema language. You still need to provide the conversion code, which can be error-prone and difficult to maintain. JAXB automatically generates code that you can customize to perform content conversion for you.

If you know how to program with the Java programming language and have minimal knowledge of XML, you will be able to use JAXB. Furthermore, the generated classes conform to standard Java API conventions, so it's even easier to start working with JAXB.

JAXB Applications Can Convert Data

Although an XML document is specified by a schema, at this time, a schema is limited in how tightly it can specify the content of an XML document. Data-interchange applications need formal data-typing. XML 1.0 does not explicitly provide data-typing beyond expressing

types as attribute values; these attribute values must then be interpreted by parsing code that you provide. In other words, you can enter any type of data you want between two tags, such as integers or strings, as long as the structure of the document conforms to the DTD specification. But frequently you'll want is to be able to convert the data, for example, to specify that only an integer can be contained between two `<quantity>` tags. The JAXB facility provides both structure and content validation in the generated code, which you can customize. More importantly, since JAXB generates Java code, you can assign types exclusively from the Java programming language, such as `Date` and `BigDecimal`, to your elements. For instructions on how to perform type conversions, see *Specifying Types* in the *Binding a Schema to Classes* chapter.

JAXB Applications Can Be Customized

Before generating Java classes from your DTD, you write what is called a *binding schema*, which contains instructions on how to generate the classes. The binding schema is written in an XML-based binding language, whose constructs you use to write the binding schema so that you can specify how your classes are generated. One of the more useful customizations you can make is data-type conversions. For example, as the previous section mentions, you can specify in the binding schema that the quantity element must only contain an integer. In addition to data-type conversions, you can use the binding schema to control the names of classes, packages, and types; and you can generate custom constructors, interfaces, and enumerations.

The binding schema also allows you to manage schema evolution. If you anticipate that your schema will change, the binding schema provides special constructs that define looser bindings that allow more flexibility. When the schema evolves, all you need to do is edit your binding schema and run the schema compiler again to create classes that reflect the changes. If you tried to change your classes instead, once you ran the schema compiler again, your changes would be overwritten. Because your binding instructions are specified in the binding schema--separate from your schema and code--when your schema evolves, you will have a much easier time maintaining your application. See the section *Managing Schema Evolution* for more information.

JAXB Applications Are Extensible

Once you have generated your Java classes, you can use them without change, or you can subclass them to provide additional functionality. The developers of JAXB designed the binding process to make subclassing derived classes easy. See the *Working With The Data* chapter for more information.

Uses of JAXB

JAXB has wide-ranging uses, especially with the advent of Web-based, business-to-business enterprise applications. However, you don't have to be a Web developer to appreciate JAXB because JAXB provides an easy way to work with data, whether or not you intend to share it. This section describes two scenarios to demonstrate how JAXB can be used in the real world.

Scenario 1: Balancing a Checkbook

You can use JAXB to create a simple desktop application for balancing checkbooks. A schema that represents a checkbook could contain a set of transactions and a balance. With the classes generated from the transactions schema you can create XML data for a set of monthly transactions. Each month you could:

1. Create an object representation of the checkbook XML data.
2. Create an object representation of the transactions for the month.
3. Calculate the new balance with the objects.
4. Append the object data of the transactions to the object data of the checkbook.
5. Write out the updated checkbook to a new XML file.

Scenario 2: Comparing Price Quotes from Suppliers

Suppose that you are a shoe manufacturer and would like to find the shoelace supplier with the best prices. In a Web-services environment, suppliers can do business over the Internet, representing data such as price lists in XML. With standard schemas for representing data shared through a repository, businesses can share this data. A customer could access the standard schema from the repository and build a JAXB application from it. Once the application is built, the customer can request the price lists from the various suppliers. These price lists are in the form of XML and will be valid against the standard price list schema. When the JAXB application retrieves the XML data, it creates separate Java object representations of the data. With the objects, the JAXB application can compare the prices for the products which interest the customer and can generate new XML data, which contains only those items that she wants to purchase. If the customer also built a JAXB application with a standard order form schema, she could edit her new price list XML data and add it to the order form XML data, which she can send to the supplier with the lowest prices.

To implement the second scenario, in addition to JAXB, you would need to use other technologies, such as the Java™ API for XML Messaging (JAXM) to send the data and the Java™ API for XML Registries (JAXR) to use the repository. The first scenario can be implemented using only JAXB. Since this guide focuses on JAXB, the checkbook application scenario is used as the example in this user guide. Starting with chapter 4, *Binding a Schema to Classes*, this guide shows you how to build a JAXB application like the checkbook example.

Getting the Most From this User's Guide

This user's guide teaches you everything you need to know to build a simple JAXB application.

If you need a refresher on XML, you can read Chapter 2, *Before You Begin: XML Basics*. Chapter 3 describes how JAXB works and provides more details on the architecture. Chapter 4, 5, and 6 provide step-by-step guides on building a JAXB application. These chapters use the checkbook example described in the previous section.

While building the checkbook application, you will learn how to:

- Write a binding schema, which defines how your schema is bound to Java classes.
- Generate the Java classes from the provided schema and the binding schema.
- Build a Java object representation of XML data based on the schema.
- Generate new XML documents based on the schema.
- Work with the data.

This guide assumes that you know how to program with the Java™ programming language and that you have installed the JAXB implementation correctly. If you have not performed the installation, see the release notes located in the docs directory of your download bundle.

To build the checkbook application, you will need the following files, which are located in the `examples/checkbook` directory of your installation[1] and the appendices of this guide:

- `checkbook.dtd`: A DTD specifying a list of checking account transactions and a balance.
- `march.xml`: The list of transactions for the month of March.
- `checkbook.xml`: A checkbook with a list of transactions and a balance.

The files that you will write with this guide are also available in `examples/checkbook` and the appendices of this guide for your reference:

- `checkbook.xjs`: the binding schema.
- `CheckbookApp.java`: The main application.
- `CheckbookBalance.java`: The subclass of `Checkbook`, which is a class generated from `checkbook.dtd`.

1. Note that the march.xml and the checkbook.xml files in the download bundle have bugs: The march.xml file has April dates and the checkbook.xml file has March dates. This bug will be fixed for FCS. The files shown in Appendix A and in chapters 5 and 6 of this guide are correct.

For more information on JAXB, see Java Specification Request, JSR-31, and the XML Data Binding Draft Specification, version 0.2 at java.sun.com/xml/jaxb.

CHAPTER 2

Before You Begin: XML Basics

This chapter explains the basic features of XML as used in this guide. If you already understand simple XML, you can skip to the next chapter.

What is XML?

XML stands for “eXtensible Markup Language”, a language developed by the World Wide Web Consortium (W3C). XML is actually a *meta-language*; a language used to describe other languages. XML allows you to describe other languages through the use of extensible markup tags, which add structure and meaning to documents. Although XML markup tags look like HTML tags, they describe the content rather than the format of the text they contain. More importantly, XML tags are extensible, which means you can define your own tags to better describe your particular content.

The two pieces of a typical XML application are the document type definition (DTD) and a set of XML document instances, which are specified by the DTD. The DTD is a schema that contains the definitions of tags you use in your XML documents by specifying what a set of tags can contain. DTDs define and declare tags and specify their contents; therefore, each DTD is essentially a language specification, and the XML documents that a DTD specifies are written in that DTD’s language. The XML 1.0 specification does not require a DTD; parsers can recognize tags in an XML document and process the data, but without a DTD cannot verify their validity.

Unlike the XML 1.0 Specification, the JAXB facility requires that you supply a DTD to build a JAXB application. The JAXB schema compiler uses the constraints specified in the DTD to build the Java classes. This section discusses only the basic features of DTDs and XML documents as used in the examples of this guide. For more information about XML, see The World Wide Web Consortium website.

Document Type Definitions

A document type definition is more commonly known as a DTD. A DTD defines the structure and content of XML documents that it specifies. A DTD consists of a list of declarations, each of which defines a building block of a document. The basic declarations are the element declarations and attribute declarations. Element declarations define what a particular set of tags in an XML document can contain. Attribute declarations accompany element declarations and provide additional information about the element.

Element Declaration

An element declaration begins with <!ELEMENT and specifies the content of the tags defined by this element. In XML, a tag is the realization of an element in a document. For example, if book is an element, the book element tag is <book>.

All documents must have at least one root element, which is not contained by any other elements declared in the document's DTD. For example, suppose you have a DTD that specifies a book. The DTD contains a declaration like this:

```
<!ELEMENT book
    (titlepage, (prologue | preface), toc, chapter+, epilogue?, appendix*) >
```

The part of the declaration in the outer parentheses specifies what a book can contain and is called the *content model*. The commas are called *sequence connectors;* they dictate that the elements must appear in the XML document in the order listed. In this case, a book contains in this order: a title page, either a prologue or a preface, a table of contents, one or more chapters, an optional epilogue, and zero or more appendices. The part of the *content model* in the inner set of parentheses represents a *model group*. This *model group* uses a *choice connector*, |, dictating that only one of the elements in the group can appear in an instance of the parent element in the XML document. In this case, any book element instance can only contain either a prologue or a preface, but not both. The +, *, and ? are called *occurrence indicators*. The + after chapter indicates that one or more chapters are allowed. The ? after epilogue means that the epilogue is optional. The * after appendix indicates that zero or more of these elements are allowed. Elements that have a * or + are sometimes called repeatable.

Each of these pieces of content in a book element are also elements, which must also be declared:

```
<!ELEMENT titlepage (title, author) >
<!ELEMENT prologue (#PCDATA) >
<!ELEMENT preface (#PCDATA) >
<!ELEMENT toc (chaptitle+, appendixtitle*) >
<!ELEMENT chaptitle (#PCDATA) >
<!ELEMENT appendixtitle (#PCDATA) >
```

```
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA)>
<!ELEMENT chapter (chaptitle, body) >
<!ELEMENT body (#PCDATA) >
<!ELEMENT epilogue (#PCDATA) >
<!ELEMENT appendix (appendixtitle, body) >
```

The notation `#PCDATA` stands for "Parseable Character DATA". PCDATA represents zero or more characters. The content of any element either contains PCDATA, other elements, or a combination of both. An element whose content is defined to have only PCDATA can only contain text. Such an element is essentially a lowest common denominator of a DTD: it cannot be broken down any farther.

As you can see from these example element declarations, specifying the content of an element can be a complicated and error-prone task. You must be careful not to specify content that might be ambiguous and confuse the parser. For example, if you must include the choice connectors and the sequence connectors in one model group, you need to use parentheses to separate the content using one connector rule from the rest of the group that uses the other connector. This case is shown in the following element declaration in which the `(prologue | preface)` model group uses a choice connector, but the rest of the model group uses a sequence connector:

```
<!ELEMENT book
    (titlepage, (prologue | preface), toc, chapter+, epilogue?, appendix*) >
```

See the *XML Documents* section to see how these element declarations are realized in an XML document.

Attribute Declaration

Unlike element declarations, attribute declarations are optional in a DTD. Attribute declarations accompany element declarations and provide additional information about the element. For example, you can add an attribute to your book element that describes the book's type:

```
<!ELEMENT book
    (titlepage, (prologue | preface), toc, chapter+, epiloque?, appendix*) >
<!ATTLIST book
    type ( fiction | travel  | history | biography ) #REQUIRED >
```

An attribute declaration includes the element name to which it applies. After the element name, a list of attribute names are declared. In this case, there is only one attribute, which is called `type`. Following the attribute name are the possible values to which the attribute can be set. In this case, the type of book can be one of four types. The `#REQUIRED` keyword

indicates that the type attribute must always be used when the book element is used in an XML document. If the type attribute were optional, `#REQUIRED` would be replaced with `#IMPLIED`.

The next section explains how this attribute declaration is realized in an XML document.

XML Documents

An XML document is a text file containing XML markup tags, which are pieces of text surrounded by a start-tag and an end-tag. The start-tag, enclosed text, and end-tag comprise an element, which is declared in a DTD. For example, consider our `chaptitle` element declaration from the *Element Declaration*section:

```
<!ELEMENT chaptitle (#PCDATA) >
```

In an XML document, this declaration could be represented as:

```
<chaptitle>The Early Years</chaptitle>
```

These XML tags look similar to HTML tags, but they are different in significant ways: XML tags are extensible and indicate the meaning of the text enclosed within them. For example, you could not define a tag `<chaptitle>` in HTML; you must use tags already defined in the HTML specification, and a set of HTML tags can only define the format of the text it encloses. The `<chaptitle>` tag can indicate to a parser that the information contained within it is a chapter title, which allows you to do much more with your data, including searching and archiving. If you have a DTD that delares a `chaptitle` element, you can also restrict what kind of data a `chaptitle` can contain.

This XML document is a realization of the DTD described in the *Document Type Definitions* section:

```
<!DOCTYPE Duke SYSTEM "Book.DTD">
<book type="biography">
    <titlepage>
        <title>Duke: My Life and Times</title>
        <author>Duke</author>
    </titlepage>
    <prologue>I dedicate this book to ...</prologue>
    <toc>
        <chaptitle>The Early Years</chaptitle>
        <chaptitle>The Later Years</chaptitle>
    </toc>
    <chapter>
        <chaptitle>The Early Years</chaptitle>
        <body>Blah blah blah</body>
    </chapter>
    <chapter>
```

```
        <chaptitle>The Later Years</chaptitle>
        <body>Blah blah</body>
    </chapter>
</book>
```

The DOCTYPE declaration at the top of the document tells a validating parser that the document instance must adhere to the rules defined in the Book.DTD. The book element has a type of biography, indicating that this XML document represents a book that is a biography. As the root element, the book element must enclose the entire document, which means that the document must end with </book>, and no other elements can be contained outside of the book element's tags. Notice that the document contains more than one chapter element. In the Book.DTD, the plus sign following chapter in the book element declaration indicates that a book can contain one or more chapters. Also notice that this document contains no epilogue. The question mark after epilogue in the book declaration specifies that an epilogue is optional.

After this brief XML lesson, you should now be able to build a simple JAXB application.

CHAPTER 3

How JAXB Works

This chapter briefly describes how you use JAXB to:

- Bind the schema to Java classes.
- Build representations of data that follow the rules defined in the schema.
- Use the data in an application.

Overview

To start building a JAXB application all you need is an XML schema. JAXB version 1.0 requires that the schema be a DTD, as defined in the XML 1.0 specification, but later versions will likely support other schema languages.

After you obtain your DTD, you build and use a JAXB application with these steps:

1. Write the *binding schema*, an XML document containing instructions on how to bind a schema to classes. For example, the binding schema might contain an instruction on what primitive type an attribute value should be bound in the generated class.

2. Generate the Java source files using the *schema compiler*, which takes both the DTD and the binding schema as input. After compiling the source code, you can write an application based on the resulting classes.

3. With your application, build a tree of Java objects representing XML data that is valid against the DTD by either:

   a. instantiating the generated classes, or

   b. invoking the `unmarshal` method of a generated class and passing in the document. The `unmarshal` method takes a valid XML document and builds an object-tree representation of it. The object tree is called a *content tree*.

4. Use your application to access the data of the content tree and modify the data of the tree.

5. You can also generate an XML document from the content tree by invoking the marshal method on the root object of the tree.

The next three sections go into more detail on building and using a JAXB application. The *Binding a Schema to Classes* section describes steps 1 and 2. The *Building Data Representations* section describes step 3. The section *Working with the Data* describes steps 4 and 5.

Binding a Schema to Classes

JAXB includes the ***schema compiler***, which generates a set of Java source files from your DTD and ***binding schema***. The binding schema must be written in the binding language that JAXB defines. After the Java source files are generated, you can compile them using the Java compiler just as you would with any Java application.

You don't need to provide a binding instruction for every declaration in your DTD to generate Java classes. The schema compiler makes certain assumptions based on the DTD if your binding schema does not completely specify how every declaration in your DTD should be bound to the code. For example, the schema compiler uses a general name-mapping algorithm to bind XML names to names that are acceptable in the Java programming language. In this case, you can use the binding schema to cause the schema compiler to generate different names. There are many other customizations you can make with the binding schema, including:

- Name the package, derived classes, and methods.
- Assign types to the methods within the derived classes.
- Choose which elements to bind to classes.
- Decide how to bind each attribute and element declaration to a property in the appropriate content class.
- Create custom constructors, interfaces, and enumerations.
- Choose the type of each attribute-value or content specification.

See *Binding a Schema to Classes* for more information on writing a binding schema. FIGURE 3-1 illustrates the process of generating classes.

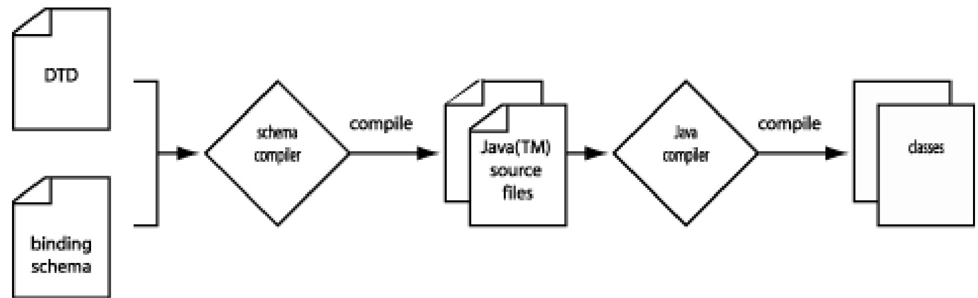


FIGURE 3-1 Generating Classes

Unless you specify otherwise in your binding schema, the schema compiler generates a class for every element whose content contains other elements. Inside a class, the schema compiler generates *properties*, which are methods that you use to access the content of child elements and the values of attributes. These methods return and accept different types, depending on the kind of declaration in the schema. For example, elements with `#PCDATA` content are bound to properties that accept and return `String` values. As an example of binding a schema to classes, consider a simplified version of the DTD from the previous chapter:

```
<!ELEMENT book (title, author, chapter+) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA)>
<!ELEMENT chapter (#PCDATA) >
```

In many cases, the schema compiler can generate an appropriate binding even when the binding schema does not include a binding instruction for a particular declaration in the DTD. In fact, to generate classes from the book DTD, all you need in your binding schema is:

```
<xml-java-binding-schema>
    <element name="book" type="class" root="true" />
</xml-java-binding-schema>
```

From this DTD and binding schema, the schema compiler generates a class `Book` with this constructor and these properties:

```
public Book();
public String getTitle();
public void setTitle(String x);
public String getAuthor();
public void setAuthor(String x);
public List getChapter();
public void deleteChapter();
public void emptyChapter();
```

Recall that the chapter element instance in the book content model has a + occurrence indicator:

```
<!ELEMENT book (title, author, chapter+) >
```

Notice that this chapter element instance in the book element's content is bound to a List property. Earlier we said that all simple elements are bound to String properties. This statement is still true for the chapter element. The List that is returned from getChapter is a List of String values, each of which represents a different chapter element instance. Also notice that the binding schema did not make reference to the occurrence indicator after the chapter element instance. This is an example of how the schema compiler considers the specifications in the DTD as well as the binding instructions in the binding schema when generating classes.

The default bindings that the schema compiler assumes are usually adequate for simple DTDs like the book example. More complicated DTDs will most likely require more complete binding instructions. For example, consider the book DTD with an additional choice model containing the simple elements prologue and preface:

```
<!ELEMENT book (title, author, (prologue | preface), chapter+) >
...
<!ELEMENT prologue (#PCDATA) >
<!ELEMENT preface (#PCDATA) >
```

Using the same binding schema with this new DTD, the schema compiler would produce this constructor and this property:

```
public void Book();
public List getContent();
public void deleteContent();
public void emptyContent();
```

In this example, the property represents the entire content model of the book element. This kind of property is not very useful if you want to access a particular piece of the content. This is why you write a binding schema. With the binding schema, you can make many customizations to your classes, including defining the way model groups are bound to classes, creating interfaces, and converting types. The *Binding a Schema to Classes* chapter shows you how to specify these customizations in the binding schema.

One binding schema that you could write for this DTD is:

```
<xml-java-binding-schema>
    <element name="book" type="class" root="true">
        <content>
                <element-ref name="title" />
                <element-ref name="author" />
                <choice property="prologue-or-preface" />
        </content>
    </element>
</xml-java-binding-schema>
```

The `choice` declaration binds the choice model group to an object property within the `Book` class. The `prologue` and `preface` elements are each bound to separate classes. The `element-ref` declarations will bind the elements to properties within the `Book` class.

Based on both the schema and the DTD, the schema compiler assumes that the desired class name is `Book` and that the simple elements are bound according to these binding declarations:

```
<element name="title" type="value" />
<element name="author" type="value" />
<element name="prologue" type="class" />
<element name="preface" type="class" />
<element name="chapter" type="value" />
```

From the DTD and the binding schema, the schema compiler generates a class `Book` with this constructor and these properties:

```
public void Book();
public String getTitle();
public void setTitle(String x);
public String getAuthor();
public void setAuthor(String x);
public List getChapter();
public void deleteChapter();
public void emptyChapter();
public MarshallableObject getPrologueOrPreface();
public void setPrologueOrPreface(MarshallableObject x);
```

The `prologue-or-preface` property returns and accepts a `MarshallableObject`, which represents objects that can be marshalled and unmarshalled. The reason the property type is not `String` is because you cannot determine if the `String` is a `prologue` or `preface`; with `MarshallableObject`, you can because it will be either a `Prologue` or a `Preface` object.

The root element's model groups in these example DTDs are simpler than the one shown in the example in the previous chapter. The XML 1.0 model group specification is very complex, reflecting the infinite number of ways that data can be ordered and represented. The next chapter explains in more detail how to bind more complicated model groups.

Building Data Representations

The Java classes that the schema compiler generates implement and extend the classes and interfaces of the ***binding framework***. The binding framework is the runtime API that the generated classes use to support three primary operations:

- **Unmarshalling**: the process of producing a content tree from an XML document.
- **Validation**: the process of verifying that the Java object representation conforms to the rules specified in the DTD.

- **Marshalling**: the process of producing an XML document from Java objects.

To perform these operations, each generated class contains methods for unmarshalling data and validating content, and extends the methods of the binding framework that perform marshalling.

Unmarshalling

With the `unmarshal` methods, you can build a Java object tree from XML documents that are instances of the schema used to generate the classes. The object tree built with JAXB is called a *content tree*. Each object in the tree corresponds to an element in the XML document. Similarly, each object in the tree is an instance of a class from the set of generated classes. You can also build a content tree by instantiating objects from the classes because the content tree binds to both the document and the classes. The chapter *Building Data Representations* demonstrates how to use unmarshalling and instantiation to build a content tree.

Validation

The unmarshalling process performs validation while it is building the content tree, so it is impossible to unmarshal an XML document to a content tree that is invalid with respect to the DTD. You can perform validation at any time after you have built your content tree by using the `validate` or `validateThis` methods in each generated class. The `validate` method validates the entire subtree rooted at the root object on which you invoke the `validate` method; the `validateThis` method validates only one object in the tree.

Marshalling

Whether you built the content tree using unmarshalling or instantiation, you can marshal the tree to a new XML document using the `marshal` methods. This means that JAXB also allows you to create new XML documents that are valid with respect to the source DTD. The marshalling process tests if the content tree has been validated before marshalling in case you have made changes to the objects in the tree. So, just as it is impossible to unmarshal an invalid document, it is impossible to marshal an invalid content tree.

FIGURE 3-2 illustrates the two ways to build data representations.

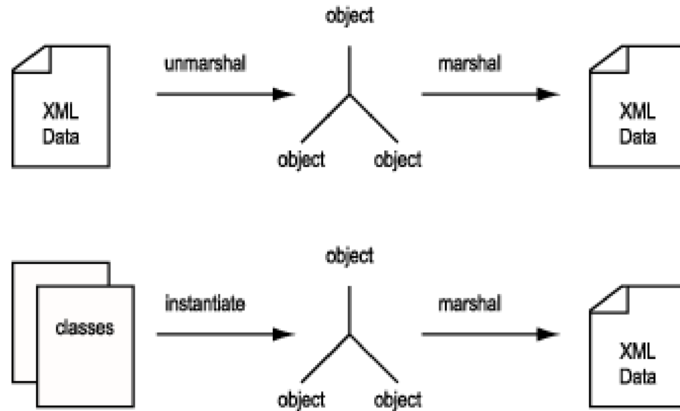


FIGURE 3-2 Building data representations

Continuing with the book example, you can unmarshal this document using the Book class you generated:

```
<book>
    <title>Duke: My Life and Times</title>
    <author>Duke</author>
    <chapter>The First Six Years ... </chapter>
</book>
```

The unmarshal method will return a Book object, say dukeBook, which is the root of the content tree. Once you have the tree, you can begin working with the data.

Working with the Data

You can work with the objects in the content tree just as you would with any Java objects. In this way, JAXB provides a Java programming interface of XML data, which you can use to seamlessly integrate XML data into Java applications.

To access the content of the tree, you use the properties in the generated classes. For example, to get the name of the author of the book Duke: My Life and Times, you simply call getAuthor on the dukeBook object. Let's say Duke's book was actually written by a disgruntled ghost writer. To smooth things over, you could call setAuthor("Duke et. al.") on the dukeBook object. To validate the modified tree, you can call validate on dukeBook, and to marshal it to a new XML document, call dukeBook.marshal().

To provide application-specific functionality, you can extend the classes rather than only use them directly. For example, in addition to accessing a piece of data, you might also want to perform a calculation with the data or, in the `dukeBook` case, add the data to a catalog. You can provide this functionality in a subclass of a derived class. The *Working With The Data* chapter uses the checkbook example previously described in *Scenario 1: Balancing a Checkbook* section to show you how to use the generated classes directly to access your transaction data and how to extend them to balance your checkbook.

Limitations

Because this release of JAXB is an early-access release, it has some limitations, which are likely to be addressed in future releases. Some of these limitations include:

- **Support for only one schema language**:
  As more schema languages are developed and existing schema language specifications become more defined, the developers of JAXB will try to support more schema languages in later releases. For now, the DTD sublanguage of XML 1.0 is a natural choice for a schema language because it is the mostly widely-used schema language and allows JAXB to serve a greater number of developers.
- **No support for XML Namespaces**:
  XML DTDs and XML Namespaces do not work very well together. Because this release of JAXB requires a DTD, Namespaces are not supported.
- **No support for Internal subsets, NOTATIONs, and the ENTITY, ENTITIES, and enumerated NOTATION types from the XML DTD 1.0 sublanguage**:
  These constructs do not often appear in DTDs, so the developers chose not to support them to simplify the binding language specification.

CHAPTER 4

Binding a Schema to Classes

This chapter demonstrates how to use JAXB to bind a DTD to a set of Java classes. To generate the classes, you perform these steps:

1. Write a binding schema, which contains instructions on how to bind a DTD to classes.
2. Run the schema compiler with the DTD and binding schema as input to generate the source code.
3. Compile the source code to generate the classes.

The next two chapters show you how to use the classes to build data representations from XML documents and work with the data. The example that these chapters use is a simple checkbook application described in *Scenario 1: Balancing a Checkbook*. With this application, you will be able to record transactions in a checkbook and determine the balance of the checking account.

The checkbook example uses the `checkbook.dtd`, which is located in the `examples/checkbook` directory of your installation. The first section of this chapter explains the checkbook DTD.

The Example DTD: `checkbook.dtd`

Before creating the checkbook application, you should understand the DTD on which it is based. This section briefly explains the `checkbook.dtd` shown here:

```
<!ELEMENT checkbook ( transactions, balance ) >
<!ELEMENT transactions ( deposit | check | withdrawal )* >
<!ELEMENT deposit ( date, name, amount )>
<!ATTLIST deposit
          category ( salary | interest-income | other ) #IMPLIED >
<!ELEMENT check ( date, name, amount, ( pending | void | cleared ), memo? ) >
<!ATTLIST check
          number CDATA #REQUIRED
```

```
          category ( rent | groceries | other ) #IMPLIED  >
<!ELEMENT withdrawal  ( date, amount ) >
<!ELEMENT balance (#PCDATA) >
<!ELEMENT date (#PCDATA ) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT amount (#PCDATA) >
<!ELEMENT memo (#PCDATA) >
<!ELEMENT pending EMPTY >
<!ELEMENT void EMPTY >
<!ELEMENT cleared EMPTY >
```

An XML document must have exactly one root element, but the document can select which element to use as its root element from its DTD. For our checkbook example, both `checkbook` and `transactions` can be used as root elements by XML document instances of this DTD. The *Creating the Minimum-Required Binding Schema* section shows you how to specify in the binding schema which elements a valid XML document can use as root elements.

According to the DTD, a checkbook contains transactions that are made against the account and a balance representing the amount of money that the account contains. The set of transactions, according to the `transactions` element definition, contains zero or more deposits, checks, or withdrawals.

A deposit transaction consists of:

- The date the deposit was made.
- The name of the person or business entity that provided the money for the deposit.
- The amount of the deposit.

The deposit element also has an attribute, called `category`, which describes the reason for the deposit.

A check transaction consists of:

- The date the check was written.
- The name of the person or business entity receiving the check.
- The amount of the check.
- The check status: cleared, void, or pending (it has not been cashed).
- An optional memo.

The check also has two attributes: `number` and `category`. The `number` attribute represents the number of the check. The `category` attribute represents the reason for writing the check.

The withdrawal transaction consists of only the date the withdrawal was made and the amount of the withdrawal.

The `balance`, `date`, `name`, `amount`, and `memo` elements are defined to have character content. This chapter shows you how to generate various types for the content of the balance, data and amount elements.

The `pending`, `void`, and `cleared` elements do not have content, so they are defined to be EMPTY.

Writing the Binding Schema

You do not need to provide a binding declaration for each component of your DTD. The schema compiler assumes default binding declarations if a particular binding is not provided. If you are not satisfied with the default bindings, however, you need to provide binding declarations in the binding schema to generate the classes that you want.

This section shows you:

1. How to write the minimal binding schema, which is a binding schema that contains the minimal binding declarations that will still allow the schema compiler to generate classes.
2. What code the schema compiler will generate based on the minimal binding schema and the DTD.
3. How to add binding declarations to the minimal binding schema so that the schema compiler will generate the code you want, not just the default code.

This guide explains in detail how to write the binding schema. The binding schema is what you use to control what kind of code the schema compiler generates. Therefore, it's important that you understand how the schema compiler interprets the declarations that you provide in the binding schema and what the schema compiler assumes if you do not provide binding declarations for a declaration in the DTD. In addition, the binding language and the schema compiler that interprets the binding schema are powerful enough to make reasonable assumptions for you, but still allow you much flexibility in defining how your DTD is bound to classes.

Although the schema compiler can produce a reasonable binding in the simple case, when you see what kind of code the schema compiler generates based on the checkbook DTD and a minimal binding schema, you will understand the importance of providing binding declarations so that you can generate the code that is appropriate for your application.

Creating the Minimum-Required Binding Schema

Whether or not you want to accept the default binding declarations assumed by the schema compiler, you need to provide a binding schema. To create the binding schema for the checkbook DTD:

1. Create a new text file called `checkbook.xjs`.

2. In `checkbook.xjs`, enter:

   ```
   <xml-java-binding-schema version="1.0ea">
   ```

   This tag identifies the file as a binding schema.

3. All binding schemas must declare at least one root element. In our example, we want to declare two root elements: `checkbook` and `transactions`. To declare the root elements, enter:

   ```
   <element name=″checkbook″ type=″class″ root=″true″ />
   <element name=″transactions″ type=″class″ root=″true″ />
   ```

   To declare these root elements, you use the `element` binding declaration to bind an element type to a class. The `name` attribute value must be the name of the element as it appears in the DTD. The `type` attribute value is `class` in this case because these are root elements and must be bound to classes. The `root` attribute equals `true` because you want XML document instances of the checkbook DTD to be able to declare `checkbook` or `transactions` as root elements.

4. Enter the end tag for the `xml-java-binding-schema` element:

   ```
   </xml-java-binding-schema>
   ```

You now have a legal binding schema from which the schema compiler can generate classes. Based on the DTD, the schema compiler makes assumptions regarding how to bind the other DTD declarations for which you did not provide binding declarations. The next section explains the binding declarations assumed by the schema compiler based on the checkbook DTD and the current binding schema. Once you understand what kind of code is produced from a minimal binding schema, writing the rest of the binding schema is easy. The *Writing the Binding Schema* section shows you how to add custom bindings to this minimal binding schema to produce the code you want for your checkbook application.

Understanding the Default Binding Declarations

The binding declarations that the schema compiler assumes based on the checkbook DTD and the minimal binding schema are:

```
<element name="checkbook" type="class" root=″true″>
    <content>
      <element-ref name="transactions"/>
      <element-ref name="balance"/></content></element>
<element name="transactions" type="class" root=″true″>
    <content property="content"/></element>
<element name="deposit" type="class">
    <attribute name="category"/>
```

```
    <content>
      <element-ref name="date"/>
      <element-ref name="name"/>
      <element-ref name="amount"/></content></element>
  <element name="check" type="class">
    <attribute name="number"/>
    <attribute name="category"/>
    <content property="content"/></element>
  <element name="withdrawal" type="class">
    <content>
      <element-ref name="date"/>
      <element-ref name="amount"/></content></element>
  <element name="balance" type="value"/>
  <element name="date" type="value"/>
  <element name="name" type="value"/>
  <element name="amount" type="value"/>
  <element name="memo" type="value"/>
  <element name="pending" type="class"/>
  <element name="void" type="class"/>
  <element name="cleared" type="class"/>
```

The rest of this section explains each of these binding declarations.

The Element Binding Declarations

The schema compiler assumes different binding declarations for elements depending on what kind of content they have or if they have attributes. For simple elements, which have only character content and no attributes, the schema compiler assumes that the elements are bound to properties within the class of their parent element. The `balance`, `date`, `name`, and `amount` elements are simple elements and so the schema compiler assumes these bindings for them:

```
<element name="balance" type="value"/>
<element name="date" type="value"/>
<element name="name" type="value"/>
<element name="amount" type="value"/>
<element name="memo" type="value"/>
```

The `name` attribute's value must be the name of the element as it appears in the DTD. The `type` attribute is `value` in these cases because these elements are bound to properties, not classes. These binding declarations will cause the schema compiler to generate these properties:

```
String getBalance();
void setBalance(String x);
String getDate();
void setDate(String x);
String getName();
void setName(String x);
String getAmount();
void setAmount(String x);
```

```
String getMemo();
void setMemo(String x);
```

A String property is fine for the name and the memo elements. However, for balance and amount, you need a type that better represents currency values. Similarly, the date element should be bound to a property that accepts and returns some kind of type that better represents a date. The *Specifying Types* section will show you how to generate properties that accept and return different types.

For all other kinds of elements, the schema compiler assumes that the elements are bound to classes. If an element contains anything other than character content or has attributes, the schema compiler will bind it to a class. The deposit, check, and withdrawal elements all have element content, and deposit and check have attributes. These elements are also bound to classes. The default bindings for these elements are:

```
<element name="deposit" type="class" >
<element name="check" type="class" >
<element name="withdrawal" type="class" >
```

Notice that you do not need to specify the root attribute as you did in the *Creating the Minimum-Required Binding Schema* section. The schema compiler assumes that the root attribute value is false. You only need to specify the value of the root attribute as true if you want the XML document instance to have the ability to use the element as a root element. From the deposit element binding declaration the schema compiler generates this class definition and constructor:

```
public class Deposit extends MarshallableObject {
  public void Deposit();
```

The Check, and Withdrawal classes will look similar to this one.

The pending, void, and cleared elements have EMPTY content and are included in a choice model group, and so they are bound to classes, as specified in their type attribute declarations:

```
<element name="pending" type="class"/>
<element name="void" type="class"/>
<element name="cleared" type="class"/>
```

The Attribute Binding Declarations

The checkbook DTD defines three attributes. The deposit element has a category attribute, and the check element has a number attribute and also has a category attribute:

```
<!ELEMENT deposit ...
<!ATTLIST deposit
          category ( salary | interest-income | other ) #IMPLIED >
<!ELEMENT check ...
<!ATTLIST check
```

```
        number CDATA #REQUIRED
        category ( rent | groceries | other ) #IMPLIED  >
```

All of these attributes take atomic values, rather than compound values, and so the schema compiler assumes that these attributes are bound to `String` properties, which these default binding declarations specify:

```
<element name="deposit" type="class" >
 <attribute name="category" />
...
<element name="check" type="class">
 <attribute name="number" />
 <attribute name="category" />
...
```

Within the `Deposit` class, the schema compiler generates this property to represent the `category` attribute:

```
void setCategory(String x);
String getCategory();
```

Within the `Check` class, the schema compiler generates these properties to represent the `number` and `category` attributes:

```
void setNumber(String x);
String getNumber();
void setCategory(String x);
String getCategory();
```

Notice that the `number` property accepts and returns a `String`. The *Specifying Types* section will show you how to customize your binding schema so that the schema compiler will generate a `number` property that accepts and returns an `int`. Likewise, the *Creating Enumerated Types* section will show you how to generate an enumerated type for the `category` property.

The Content Binding Declarations

The `content` binding declaration is the most complicated, reflecting the infinite number of ways you can specify content in XML, but JAXB makes it easy for you. The most common kind of content model is a simple, non-repeating sequence, such as `(a, b, c, d)`. If you use this kind of content model, most likely you won't need to specify a binding declaration for it because the schema compiler generates a separate property for each element in the sequence, which is usually what you want.

Most of the elements in `checkbook.dtd` have simple, non-repeating sequence content. For those elements, the schema compiler assumes these binding declarations shown in bold:

```
<element name="checkbook" type="class" root="true">
    <content>
```

```
      <element-ref name="transactions"/>
      <element-ref name="balance"/></content></element>
<element name="deposit" type="class">
    <attribute name="category"/>
    <content>
      <element-ref name="date"/>
      <element-ref name="name"/>
      <element-ref name="amount"/></content></element>
<element name="withdrawal" type="class">
    <content>
      <element-ref name="date"/>
      <element-ref name="amount"/></content></element>
```

The `element-ref` binding declaration is used to bind an instance of an element in a content model to a property in the parent element's class. The `element` declaration corresponding to the element instance binds the element itself, including the binding of the element to its type. By default, the schema compiler generates `String` properties from all `element-ref` binding declarations that refer to simple elements. The default property for the `name` element is:

```
public class Deposit {
    ...
    String getName();
    void setName(String x);
```

The schema compiler also generates a `String` property for `balance`, `date`, and `amount`. The `Checkbook` class contains a property for the `balance` element. The `Deposit` and `Withdrawal` classes both have properties for the `date` and `amount` elements. In the *Specifying Types* section, you'll see that the types of these properties change when you use the `convert` attribute in the corresponding `element` binding declarations.

Because you specified that the `transactions` element is bound to a class, the `element-ref` binding declaration for the transactions element will cause the schema compiler to generate this property in the `Checkbook` class:

```
Transactions getTransactions();
void setTransactions(Transactions x);
```

If an element's content is anything other than a simple, non-repeating sequence, the schema compiler will assume the *general-content property* binding declaration, which is:

```
<content property="content" />
```

The `transactions` and `check` elements do not have simple, non-repeating sequences as their content, and so the schema compiler assumes these binding declarations for them:

```
<element name="transactions" type="class" root="true">
    <content property="content"/></element>
...
<element name="check" type="class">
    <attribute name="number"/>
    <attribute name="category"/>
    <content property="content"/></element>
```

The *general-content declaration* is used to bind an entire model group, including nested model groups, to one property. This declaration is not very useful for our purposes because we want to access the individual elements in these content models. This declaration is useful for defining more flexible bindings if you anticipate that your DTD will change in the future. For more information on this declaration, see the *Managing Schema Evolution* section.

From these binding declarations, the schema compiler generates this property in both the `Transactions` class and the `Check` class:

```
List getContent();
void emptyContent();
void deleteContent();
```

The `getContent` method returns a mutable list containing the property's current value. The `emptyContent` method discards the values in the list and creates a new, empty list. The `deleteContent` method discards the list.

Customizing the Binding Schema

Now that you understand what binding declarations the schema compiler will assume based on your DTD and minimal binding schema, it's easy to write the binding schema: All you need to write are the binding declarations that are not assumed by the schema compiler. This section explains each of the customizations you can make to the binding schema to get the classes that you want.

The binding schema that you will use for the checkbook application is:

```
<xml-java-binding-schema version="1.0ea">
<element name="checkbook" type="class" root="true" />
<element name="transactions" type="class" root="true">
 <content>
     <choice property="entries" collection="list" supertype="Entry" />
 </content>
</element>
<element name="balance" type="value" convert="BigDecimal"/>
<element name="amount" type="value" convert="BigDecimal" />
<element name="date" type="value" convert="TransDate" />
<element name="deposit" type="class" >
 <attribute name="category" convert="DepCategory" />
</element>
<element name="check" type="class" >
   <content>
     <element-ref name="date"/>
     <element-ref name="name" />
     <element-ref name="amount" />
     <choice property="pend-void-clrd"/>
   </content>
    <attribute name="number" convert="int" />
    <attribute name="category" convert="CheckCategory" />
```

```
    </element>
    <conversion name="BigDecimal" type="java.math.BigDecimal" />
    <conversion name="TransDate" type="java.util.Date"
      parse="TransDate.parseDate" print="TransDate.printDate" />
    <enumeration name="DepCategory" members="salary interest-income other"/>
    <enumeration name="CheckCategory" members="rent groceries other"/>
    <interface name="Entry" members="Deposit Check Withdrawal"
      properties="date amount" />
  </xml-java-binding-schema>
```

This section steps you through this binding schema as if you were writing it from scratch. You already entered the root element bindings in the *Creating the Minimum-Required Binding Schema* section, so let's start with that.

1. Replace the empty element tag (/>) with a right-angle bracket (>) and add an end-tag for the transactions root element binding declaration because you will be adding content declarations to it:

```
<xml-java-binding-schema version="1.0ea">
   <element name="checkbook" type="class" root="true" />
   <element name="transactions" type="class" root="true" >
   </element>
```

2. Before you start writing your custom binding declarations, the first declarations you need in your binding schema are the element binding declarations for the deposit and check elements because you will be specifying types for their attributes and customizing the content model of the check element. The schema compiler needs these declarations so that it can generate the properties for the attributes and the choice content in the correct class and for the correct elements.

   Within the root element of your binding schema, after the root element binding declarations, enter:

```
<element name="deposit" type="class" >

</element>
...
<element name="check" type="class" >

</element>
```

   You'll add the custom attribute and content binding declarations within these element binding declarations later.

Specifying Types

By default, the schema compiler generates `get` methods that return a `String` and `set` methods that accept a `String` for all simple elements and attributes. For example, consider these default binding declarations:

```
<element-ref name="amount" />
...
<element name="amount" type="value" />
```

From these binding declarations, the schema compiler generates these two methods:

```
public String getAmount();
public void setAmount(String amount);
```

If you want to perform some calculations with the amount, you need to convert it from a `String` to some other type that will allow you to use it in a calculation. For calculations involving currency values, the `BigDecimal` type is a good choice because it represents arbitrary-precision signed decimal numbers, and the `BigDecimal` class provides methods for basic arithmetic.

To specify a type, you use the `conversion` declaration to define the conversion and the `convert` attribute of the `element` or `attribute` declaration, depending on whether you are converting the type of an element property or an attribute property, to reference the `conversion` declaration.

Specifying Non-Primitive Types

To define a conversion from `String` to `BigDecimal`:

1. Add this `conversion` declaration anywhere at the top-level (within `<xml-java-binding-schema version="1.0ea">` tags) of your binding schema, perhaps after the `check` element binding declaration:

   ```
   <conversion name="BigDecimal" type="java.math.BigDecimal" />
   ```

   Any `element` or `attribute` binding declaration that uses this conversion refers to it by the name `BigDecimal`, as specified by the `name` attribute. The `type` attribute value is the actual type to which a property is converted.

2. To instruct the schema compiler to generate an `amount` property with a `BigDecimal` type:

   a. Add an element binding declaration for the `amount` element to the top level of your binding schema:

   ```
   <element name="amount" type="value" />
   ```

b. Add a convert attribute to the amount element binding and set its value to BigDecimal:

```
<element name="amount" type="value" convert="BigDecimal"/>
```

These binding declarations produce these method signatures:

```
public java.math.BigDecimal getAmount();
public void setAmount(java.math.BigDecimal amount);
```

Declaring the conversion separately from an element binding allows you to reuse a conversion with other element bindings. You can do this with the balance element, which also needs to be bound to a BigDecimal property

To instruct the schema compiler to generate a balance property with a BigDecimal type:

1. Add an element binding declaration for the balance element to the top level of your binding schema and assign "BigDecimal" to its convert attribute:

```
<element name="balance" type="value" convert="BigDecimal" />
```

You can also convert the date element's property type to a java.util.Date. Since a date can be written so many different ways, you need to specify how the date should be parsed when unmarshalled and printed when marshalled. The conversion declaration includes parse and print attributes for this purpose.

To convert the date element's property to a java.util.Date:

1. Add this conversion declaration to the binding schema:

```
<conversion name="TransDate" type="java.util.Date"
          parse="TransDate.parseDate" print="TransDate.printDate"/>
```

   The TransDate name refers to a Java class that you need to provide. This class contains a static parseDate method specifying how to parse the date and a static printDate method specifying how to print the date. The TransDate class is included in the examples/checkbook directory of your installation.

2. To instruct the schema compiler to generate a date property with a the TransDate class, add a convert attribute to the date element binding declaration and set its value to TransDate:

```
<element name="date" type="value" convert="TransDate"/>
```

These binding declarations produce these method signatures:

```
public java.util.Date getDate();
public void setDate(java.util.Date x);
```

In the `BigDecimal` conversion example, you don't need to specify a `parse` or `print` method because the `java.math.BigDecimal` class already specifies a constructor that takes a `String` and returns a `BigDecimal` and a `toString` method that takes a `BigDecimal` and returns a `String`. To apply the conversion, the schema compiler generates code to invoke the constructor and the `toString` method.

Specifying Primitive Types

When you specify primitive types, such as `int`, you don't need to provide a separate `conversion` binding declaration; you simply add the `convert` attribute to the `element` or `attribute` binding declaration and assign the value to the primitive type.

Within the `check` element binding declaration, add an `attribute` binding declaration for the `number` attribute and specify an `int` type for its property:

```
<element name="check" type="class" >
  <attribute name="number" convert="int" />
</element>
```

Creating Enumerated Types

The `convert` attribute can also be used to specify an enumerated type. In the Java programming language, you represent enumerated types with a typesafe enum, which is a class whose instances represent a fixed set of values.

An attribute whose value can only be set to one of a fixed set of values is a good candidate for an enumerated type. The checkbook DTD has two such attributes: the `category` attribute of the `deposit` element and the `category` attribute of the `check` element:

```
<!ATTLIST deposit ...
    category ( salary | interest-income | other ) #IMPLIED >

<!ATTLIST check ...
    category ( rent | groceries | other ) #IMPLIED >
```

To generate typesafe enums for these attributes:

1. Enter two enumeration tags for each conversion at the top level of your binding schema, perhaps after the `TransDate` conversion declaration:

   ```
   <conversion name="TransDate" ...
   <enumeration
   <enumeration
   ```

2. For the `enumeration` tag's `name` attribute, you need to enter a unique name for each enumeration because both of the enumerations are at the top level of the binding schema. Enter `DepCategory` for the `deposit` category and `CheckCategory` for the `check` category.

```
<enumeration name="DepCategory"
<enumeration name="CheckCategory"
```

These names will be the names of the classes to represent the typesafe enums.

3. Add a `members` attribute to each `enumeration` declaration, and assign to it the possible values of each attribute:

```
<enumeration name="DepCategory" members="salary interest-income other" />
<enumeration name="CheckCategory" members="rent groceries other" />
```

4. Add `attribute` binding declarations within the `deposit` and `check` element binding declarations and assign the appropriate `enumeration` declaration name to each `convert` attribute of the `attribute` binding declaration:

```
<element name="deposit" ...
    <attribute name="category" convert="DepCategory" />
...
<element name="check" ...
    <attribute name="category" convert="CheckCategory" />
```

The schema compiler will generate this class from the binding of the check category attribute:

```
public final class CheckCategory {
 public final static CheckCategory RENT;
 public final static CheckCategory GROCERIES;
 public final static CheckCategory OTHER;
 public static CheckCategory parse(String x);
 public String toString();
}
```

You'll see how to work with this class in the *Building Data Representations* chapter.

Customizing Content Model Binding Declarations

Content models can be very complex, and so the binding language defines many different binding declarations to handle different kinds of content models. You define these bindings with the `content` binding declaration.

You can use the `content` binding declaration to define two kinds of content model declarations: the *general-content property* and the *model-based content property*. A *general-content property* is used to bind an entire content model to one property. This declaration is not used to bind anything in the checkbook DTD, but it is useful for defining more flexible bindings if you anticipate that your DTD will change in the future. For more information on this declaration, see the *Managing Schema Evolution* section.

A *model-based content property* declaration can contain four types of declarations to specify different kinds of bindings for model groups:

- `element-ref`, which specifies the binding of one element instance within another element's content. The `element` construct specifies the binding of the element itself.
- `choice`, which specifies the binding of a nested choice model group to a property.
- `sequence`, which specifies the binding of a nested sequence model group to a property.
- `rest`, which is a more flexible binding declaration that you can use to specify any kind of content. See the *Managing Schema Evolution* section for more information on the `rest` binding declaration.

Remember from the *Understanding the Default Binding Declarations* section that the schema compiler assumes a `general-content property` binding declaration for the `check` and `transactions` element content because these elements do not have simple, non-repeating sequence content models. Instead, these elements have choice model groups in their `content` declarations:

```
<!ELEMENT transactions ( deposit | check | withdrawal )* >
...
<!ELEMENT check ( date, name, amount, ( pending | void | cleared ), memo? ) >
...
```

This section shows you how to use the `choice` binding declaration to cause the schema compiler to generate more useful properties for the content of these elements.

In the case of the `transactions` element content, we want to assign the entire group to one choice property. To specify the binding of the `transactions` element content:

1. Inside the `transactions` element binding declaration, enter a `content` tag and a `choice` binding declaration, and assign the value "entries" to the property attribute: :

   ```
   <element name="transactions" type="class" root="true" >
     <content>
         <choice property="entries"
   ```

   The schema compiler will generate a property called `Entries`. For example, the `get` method will be called `getEntries`. The reason we use the name "entries" is because the name refers to an interface declaration, which the section *Creating Interfaces* will show you how to create.

2. Because the content model has a * occurrence indicator, we need to bind this content to a collection property. Enter the `collection` attribute and give it a value of "list":

   ```
   <choice property="entries" collection="list"/>
   ```

   A collection property can represent either an array or a `List`. In this case, you should bind the content to a `List` because a `List`, unlike an array, allows you to add more entries at runtime.

3. Enter the end-tag for the content binding declaration:

```
</content>
```

These binding declarations will produce this property in the Transactions class:

```
public List getEntries();
public void deleteEntries();
public void emptyEntries();
```

The getEntry method returns the entire list of entries. Once you get the list, you can iterate through the list as you would with any List to get a particular Entry. The List returned from getEntries is mutable: if you change an Entry in this list, it changes the Entry in the content tree. The emptyEntries method discards the values in the list and creates a new, empty list. The deleteEntries method discards the list.

To specify the binding of the nested choice model group in the check element content model, within the check element binding declaration:

1. Enter the element-ref binding declarations for the date, name, and amount elements, and insert the choice binding declaration as shown in bold within the check element's content binding declaration:

```
<element name="check" type="class" >
   <attribute name="number" convert="int" />
   <attribute name="category" convert="CheckCategory" />
   <content>
     <element-ref name="date" />
     <element-ref name="name" />
     <element-ref name="amount" />
     <choice property="pend-void-clrd" />
   </content>
</element>
```

The property attribute value is the name of the generated property. For example, the get method will be called getPendVoidClrd. If the content also contains a choice, sequence, or rest content binding declaration, you need to specify the default element-ref binding declarations for the elements preceding this content; otherwise, the schema compiler doesn't know which elements are intended for which binding declaration.

Creating Interfaces

You might have noticed that the deposit, check, and withdrawal elements have some common content. You might also have noticed that each represents an entry in a list of transactions. When you have a group of classes that provide similar functionality and have

some common behavior and properties, you can use an interface to capture the similarities between the classes. In the case of `deposit`, `check`, and `withdrawal`, they all have `date` and `amount` elements. The `date` and `amount` will be the common properties in the interface.

To cause the schema compiler to generate an interface, which `Deposit`, `Check`, and `Withdrawal` will implement:

1. Anywhere at the top level of your binding schema, perhaps after the `enumeration` declarations, enter this `interface` declaration:

```
<interface name="Entry" members="Deposit Check Withdrawal"
           properties="date amount" />
```

   The `members` attribute represents all of the classes that implement the interface. The `properties` attribute represents the common content shared by the members of the interface.

Because the `deposit`, `check`, and `withdrawal` elements occur in the `transactions` element's content model as a choice group, you need to reference `Entry` from the binding of the choice group. You previously assigned the name of the interface to the `property` attribute and the value `"list"` to the collection attribute. Assign the name of the interface to the `supertype` attribute:

```
<element name="transactions" type="class" class="Transactions >
<content>
   <choice property="entries" collection="list" supertype="Entry"/>
   </content>
</element>
```

The `supertype` attribute indicates a class or interface declared in the binding schema that each of the element classes included in the choice property implements.

These binding declarations will produce an interface called `Entry`, which will include the date and amount properties:

```
public interface Entry {
 ...
 public int getAmount();
 public void setAmount(int x);
 public Date getDate();
 public void setDate(Date d);
```

Managing Schema Evolution

As with any facility that generates code, a developer writing applications based on the code needs to ensure that newly generated code does not break the applications. If you anticipate that your DTD will change, you can use the more flexible binding declarations in the binding language to protect the integrity of your applications.

The easiest way to manage schema evolution is to accept the default bindings that the schema compiler produces. These bindings are very loose definitions of the DTD declarations, and thus are more flexible to changes in the DTD. For example, any model groups that do not consist of distinct elements in a non-repeating sequence are bound using a `general content property` declaration, which binds the entire content to one property. If you were to add elements to this model group, these elements would still be represented by the property, and the classes would not change.

You can use the *general content property* declaration for any model group. To bind content using this declaration, you use the content construct:

```
<content property="mygroup" />
```

This binding will generate this property:

```
public List getMygroup();
public void deleteMygroup();
public void emptyMygroup();
```

Another binding declaration you can use to manage schema evolution is the `rest` binding declaration. This binding declaration is similar to the `general content property` declaration in that it can represent any kind of content. By appending a `rest` declaration onto an element content's binding declaration, you can add other elements and groups to the DTD's content model in the future without affecting the generated classes. For example, you can add a `rest` construct to the content binding of the `withdrawal` element:

```
<content>
    ...
<rest property="rest" />

</content>
```

Because you have the `rest` property, you can add other content to the `withdrawal` element without breaking your application. In addition, you can still individually access the old content with the properties generated by the other declarations within the `content` declaration. Only the new content, defined by the `rest` property, will be represented by a `List` property.

Generating the Java Classes

Now that you've completed the binding schema, you can run the schema compiler to generates the Java classes. This guide assumes that you have followed the instructions in the release notes, located in the `docs` directory of your installation and have set your classpaths correctly.

To generate the Java classes:

1. Run the schema compiler with checkbook.dtd and checkbook.xjs, the binding schema that you created:

   ```
   xjc checkbook.dtd checkbook.xjs
   ```

   You should now see the files Checkbook.java and Entry.java in your current directory.

2. Compile the source files into Java classes:

   ```
   javac *.java
   ```

The rest of this section explains the code generated in the files. If you don't need an explanation of the code, go to the next chapter, *Building Data Representations* to build content trees using the classes.

The Generated Java Source Files

This section briefly explains some of the public methods and classes generated by the schema compiler based on the checkbook DTD and the binding schema that you created in the *Writing the Binding Schema* section. Since the `Deposit`, `Check`, and `Withdrawal` classes are so similar, among these classes, this section only explains the `Check` class. Likewise, this section only explains the `CheckCategory` enumerated class, and out of the `Pending`, `Void`, and `Cleared` classes, this section only explains the `Pending` class.

The `Checkbook.java` File

The `checkbook` element from transactions.dtd is bound to the `Checkbook` class whose signature is:

```
public class Checkbook extends MarshallableRootElement implements RootElement
```

Because `checkbook` is a root element, the `Checkbook` class extends `MarshallableRootElement`, which is the class representing root element objects that can be marshalled and unmarshalled, and implements `RootElement`.

Like every generated class, the Checkbook class contains a zero-argument constructor:

```
public Checkbook();
```

Recall that the `checkbook` element contains a `transactions` element and a `balance` element:

```
<!ELEMENT checkbook ( transactions, balance ) >
```

The `transactions` and `balance` elements are bound to these properties in the `Checkbook` class:

```
// the transactions property
public void setTransactions(Transactions x);
public Transactions getTransactions();

// the balance property
public void setBalance(java.math.BigDecimal x);
public java.math.BigDecimal getBalance();
```

The `transactions` property accepts and returns a `Transactions` object because the `transactions` element is also represented by a class. The `balance` property accepts and returns a `java.math.BigDecimal` because of these binding declarations that you specified in the *Specifying Types* section:

```
<element name="balance" type="value" convert="BigDecimal"/>
...
<conversion name="BigDecimal" type="java.math.BigDecimal" />
```

Although the `MarshallableRootElement` defines `marshal` methods, which the `Checkbook` class uses by extension, it does not define any `unmarshal` methods. Thus, the schema compiler generates these static `unmarshal` methods in the `Checkbook` class:

```
public static Checkbook unmarshal(InputStream in)
public static Checkbook unmarshal(XMLScanner xs)
public static Checkbook unmarshal(XMLScanner xs,
Dispatcher d)
```

When you unmarshal an XML document, you can invoke either `unmarshal(InputStream)` or `unmarshal(XMLScanner)` in which the `InputStream` or the `XMLScanner` represents your XML document.

Although unmarshalling performs validation for you, you need to perform validation after you edit the content tree and before you marshal the tree back to an XML document. For these purposes, the schema compiler generates these methods:

```
public void validateThis();
public void validate();
```

After you edit a piece of the content tree, you can use `validateThis` to validate the edited object. Before you marshal the content tree to an XML document, you must use `validate` to validate the entire content tree.

The `Transactions.java` File

The `transactions` element from the checkbook.dtd is bound to the `Transactions` class whose signature is:

```
public class Transactions extends MarshallableRootElement implements RootElement
```

Like `checkbook`, `transactions` is a root element, and so the `Transactions` class extends `MarshallableRootElement`, and implements `RootElement`.

The `Transactions` class contains a zero-argument constructor:

```
public Transactions();
```

The `transactions` element contains zero or more `deposit`, `check`, or `withdrawal` elements:

```
<!ELEMENT transactions (deposit | check | withdrawal)* >
```

When you followed the instructions in *Customizing Content Model Binding Declarations*, you specified that the schema compiler bind the `transaction` element content to a `List` collection property called "entries". In the section *Creating Interfaces*, you specified that the supertype of the `entries` property is the interface `Entry`:

```
<element name="transactions" type="class" class="Transactions >
<content>
   <choice property="entries" collection="list" supertype="Entry"/>
   </content>
</element>
```

The `interface` binding declaration caused the schema compiler to bind the transactions content to a list of deposits, checks, and withdrawals. The `interface` binding declaration caused the schema compiler to generate an `Entry` interface, which the `Deposit`, `Check`, and `Withdrawal` classes implement. The `Entry` interface is explained in the next section.

The `entries` property consists of three methods that you use to access the `transactions` element content:

```
public List getEntries();
public void deleteEntries();
public void emptyEntries();
```

The `getEntry` method returns the entire list of entries. Once you get the list, you can iterate through the list as you would with any `List` to get a particular `Entry`. The `List` returned from `getEntries` is mutable: If you change an `Entry` in this list, it changes the `Entry` in the content tree. The `deleteEntries` method deletes the current list of entries. The `emptyEntries` method deletes the list's values.

Any content of `transactions` (whether it's two deposits or one deposit and five withdrawals) implements `Entry`. Therefore, you do not need to perform `instanceof` tests or type casting on the items in the `List` representing the entries, unless you are working with an element contained in `deposit`, `check`, or `withdrawal` that is not one a member of the `Entry` interface.

Like the `Checkbook` class, the `Transactions` class contain the `marshal`, `unmarshal` and `validate` methods that a user should invoke.

The Entry.java File

The supertype of the transactions element content is the Entry interface, as specified in this interface binding declaration:

```
<element name="transactions" type="class" class="Transactions >
  <content>
    <choice property="entries" collection="list" supertype="Entry"/>
  </content>
</element>
```

The Entry interface signature is:

```
public interface Entry {
```

The only common element content between the deposit, check, and withdrawal elements is date and amount, and so you assigned these elements to the properties attribute of the interface construct In the *Creating Interfaces* section:

```
<interface name="Entry" members="Deposit Check Withdrawal"
                 properties="date amount" />
```

The Entry interface, therefore, includes the properties for the date and amount elements:

```
public int getAmount();
public void setAmount(int x);
public Date getDate();
public void setDate(Date d);
```

The amount property returns and accepts an int because you wrote a conversion binding declaration for converting from String to BigDecimal and used the convert attribute in the amount element binding declaration and assigned to it the value "BigDecimal":

```
<element name="amount" type="value" convert="BigDecimal" />
<conversion name="BigDecimal" type="java.math.BigDecimal" />
```

The Check.java File

The check element is bound to the Check class, which has this signature:

```
public interface Check extends MarshallableObject
               implements Element, Entry{
```

The Check class extends MarshallableObject, which is the abstract class that represents any object that can be marshalled or unmarshalled but is not necessarily a root element. The Check class must implement Element because, unlike MarshallableRootElement, a

MarshallableObject does not have to be an object derived from an element and therefore does not implement Element itself. Finally, Check implements Entry because you specified Entry as the supertype of the transaction element content in the section *Creating Interfaces*.

The check element has two attributes and contains six elements:

```
<!ELEMENT check ( date, name, (pending | void | cleared), memo? ) >
<!ATTLIST check
          number CDATA #IMPLIED
          category ( rent | groceries | other ) #IMPLIED >
```

The date element is bound to a property that returns a Date object:

```
public Date getDate();
public void setDate(Date x);
```

The name, and memo elements, which only contain text are bound to these properties:

```
public String getName();
public void setName(String x);
public String getMemo();
public void setMemo(String x);
```

In the *Specifying Types* section, you specified a type of int for the number attribute, producing this property:

```
public int getNumber();
public void setNumber(int x);
```

In the *Customizing Content Model Binding Declarations* section you specified that the choice model group, (pending | void | cleared), is bound to a property, producing these methods:

```
public MarshallableObject getPendVoidClrd();
public void setPendVoidClrd(MarshallableObject x);
```

The pend-void-clrd property returns and accepts a MarshallableObject, which represents objects that can be marshalled and unmarshalled. The reason the property type is not String is because you cannot determine if a String is supposed to be a pending, void, or cleared element; with MarshallableObject, you can because the MarshallableObject will be either a Pending, Void, or Cleared object.

The Check class also contains a property for the checkCategory enumeration, which you specified in the *Creating Enumerated Types* section:

```
<enumeration name="CheckCategory" members="rent groceries other" />
...
<attribute name="category" convert="CheckCategory" />
```

The property that this binding declaration generates is:

```
public CheckCategory getCheckCategory();
```

```
public void setCheckCategory(CheckCategory x);
```

This property accepts and returns a `CheckCategory` object, which is an instance of the typesafe enum class, `CheckCategory`. This class is discussed in the next section.

The `CheckCategory.java` File

The `category` attribute takes one of a fixed set of values represented by a choice group in the attribute definition:

```
<!ATTLIST check ...
                category ( rent | groceries | other ) #IMPLIED >
```

When following the instructions in the *Creating Enumerated Types* section, you specified a binding of this attribute to a typesafe enum:

```
<enumeration name="CheckCategory" members="rent groceries other" />
...
<attribute name="category" convert="CheckCategory" />
```

A typesafe enum is a class that represents an enumeration consisting of an element or attribute and its list of possible values, only one of which can be assigned to the class. The name of the class corresponds to the element or attribute, and static fields represent the values. Typesafe enum classes have many advantages, including compile-time type checking. The typesafe enum generated from the category attribute is:

```
public final class CheckCategory {
 public final static CheckCategory RENT;
 public final static CheckCategory GROCERIES;
 public final static CheckCategory OTHER;
 public static CheckCategory parse(String x);
 public String toString();
}
```

The `parse` method attempts to map a `String` argument to one of the accepted values. The `toString` method returns the current value of a `CheckCategory` as a `String`.

The `Pending.java` File

The `Pending` class represents one of the members of the choice model group contained in the check element declaration:

```
<!ELEMENT check ( date, name, (pending | void | cleared), memo? ) >
```

This element, as well as the `void` and `cleared` elements, are bound to their own classes because they are part of this choice model group. As explained in the *The Check.java File* section, the property that this group is bound to must return a `MarshallableObject`, which must also be either a `Pending`, `Void`, or `Cleared` object, so that your application knows which element it will encounter during unmarshalling or marshalling. The `Pending`, `Void` and `Cleared` classes each have a zero-argument constructor.

The next chapter shows you how to work with these classes to build data representations.

CHAPTER **5**

Building Data Representations

This chapter demonstrates how to use the classes you generated in the previous chapter to:

- Unmarshal an XML document into a content tree.
- Instantiate the classes to build a content tree.
- Validate your content tree against the DTD.
- Marshal a content tree to a new XML document.
- Append a content tree onto another content tree.

The XML Document Instance: `march.xml`

The `march.xml` document is valid against `checkbook.dtd`. Remember from *Before You Begin: XML Basics* that an XML document must have a root element that encloses all other elements in the document. The root element of `march.xml` document is the `transactions` element because `march.xml` represents a set of transactions for the month of March. As shown here in `march.xml`, you only had one deposit, one check, and one withdrawal transaction for the month of March:

```
<?xml version="1.0" encoding="US-ASCII"?>
<transactions>
  <deposit category="salary" >
    <date>03-14-2001</date>
    <name>Me</name>
    <amount>3000.00</amount>
  </deposit>
  <check number="2" category="groceries">
    <date>03-15-2001</date>
    <name>Conglomerate Foods</name>
    <amount>34.95</amount>
    <pending/>
    <memo>food</memo>
  </check>
  <withdrawal>
```

```
      <date>03-16-2001</date>
      <amount>20.00</amount>
    </withdrawal>
  </transactions>
```

This document is included in the `examples/checkbook` directory of your installation[1]. This chapter shows you how to unmarshal this document into a content tree.

Setting Up Your Application

Before you can use JAXB to build data representations or work with the data, you need to first create a Java application that will perform these functions. To set up your JAXB application:

1. Create a file called `CheckbookApp.java`
2. Import these packages:

   ```
   import java.io.*;
   import java.util.*;
   import javax.xml.bind.*;
   import javax.xml.marshal.*;
   ```

   The last two packages are part of the binding framework, which defines the `unmarshal`, `marshal`, and `validate` methods.

3. Declare the `CheckbookApp` class:

   ```
   public class CheckbookApp {
   }
   ```

4. Initialize two `Transactions` objects:

   ```
   public static Transactions marchTrans = new Transactions();
   public static Transactions aprilTrans = new Transactions();
   ```

   You'll need to reuse these objects in your application.

5. Inside `CheckbookApp`, create your `main` method:

   ```
   public class CheckbookApp {
   ```

1. Note that the march.xml file in your installation has April dates instead of March dates. This bug will be fixed for FCS. The file shown in this chapter and in the appendix is correct.

```
    public static void main(String args[]) throws Exception {
    }
}
```

The completed CheckbookApp.java file is located in your examples/checkbook directory.

Building a Content Tree

JAXB allows you to build a content tree in one of two ways: by unmarshalling an XML document or by instantiating the generated classes.

Unmarshalling

Once you have generated the classes from the DTD that specifies an XML document, you can unmarshal the document into a content tree.

In your docs/examples/checkbook directory, find the file called march.xml, which contains the transactions written in the month of March.

To unmarshal this XML document into a content tree in the CheckbookApp.java file:

1. Create a method called buildTrees:

```
public static void buildTrees() throws Exception {
}
```

2. In your new method read the XML file into a FileInputStream:

```
File march = new File("march.xml");
FileInputStream fIn = new FileInputStream(march);
```

3. Call the unmarshal method of Transactions, which is the class that represents the transactions root element of checkbook.dtd:

```
try {
    marchTrans = marchTrans.unmarshal(fIn);
} finally {
    fIn.close();
}
```

4. Invoke the buildTrees method from your main method:

```
buildTrees();
```

At this point, `CheckbookApp` generates a content tree from `march.xml`. The *Accessing Content* section will show you how to manipulate the contents of the tree. The next section shows you how to build a content tree by instantiating the generated classes.

Instantiation

If you have an XML DTD but no valid XML instance documents specified by the DTD, you can create a valid XML document by building a content tree from the derived classes and marshalling the tree to an XML document.

Suppose that you want to create a content tree representing a list of transactions for the month of April.

To build this content tree with instantiation:

1. In the `buildTrees` method, after the call to unmarshal the `march.xml` file, get the list of entries from the `aprilTrans` object, create a new `Check` object, representing the rent check for the month of April:

   ```
   List aprilEntries = aprilTrans.getEntries();
   Check aprilRentCheck = new Check();
   CheckCategory aprilRent = CheckCategory.RENT;
   aprilRentCheck.setCategory(aprilRent);
   ```

2. Set the name of the entity to receive the check:

   ```
   aprilRentCheck.setName("Gilchrest Gardens Manor");
   ```

3. Set the check number:

   ```
   aprilRentCheck.setNumber(51);
   ```

   You can pass an integer to the `setNumber` method because you specified in the binding schema that the `number` property accepts and returns an `int`.

4. Set the date for the check:

   ```
   aprilRentCheck.setDate(TransDate.parseDate("04-12-2001"));
   ```

   You use the `parseDate` method from the `TransDate` class that you provided in the previous chapter because you specified your own format for the date, which is: MM-dd-yyyy.

5. Set the amount for the check:

   ```
   aprilRentCheck.setAmount(new java.math.BigDecimal("1500.00"));
   ```

   You can pass a `java.math.BigDecimal` to the `setAmount` method because you

specified in the binding schema that the amount property accepts and returns a `java.math.BigDecimal`. When you start calculating the balance for the checkbook in the next chapter, you will see the advantage of performing these type conversions.

6. Set the check status to pending:

```
Pending pending = new Pending();
aprilRentCheck.setPendVoidClrd(pending);
```

7. Add the `Check` to the list of entries in the `aprilTrans` content tree:

```
aprilEntries.add(aprilRentCheck);
```

   The `Entry` object represents a list of transactions, which includes any number of deposits, checks, and withdrawals. The `Deposit`, `Check`, and `Withdrawal` classes implement `Entry`, which represents the common functionality of these three classes. After you create a `Check`, `Deposit`, or `Withdrawal`, you add it to the Entries list. Since the list is mutable, the transactions you add to it are automatically added to the content tree.

You now have two content trees: one for the March transactions, the other for the April transactions. The next section demonstrates how to access content from the trees.

Accessing Content

Whether you built a content tree by unmarshalling an XML document or by instantiating your classes, you access the content in the same way. This section will demonstrate accessing the content of the content trees you created in the previous section.

In the `march.xml` file, you have a grocery check made out to Conglomerate Foods. You now realize that you shopped at Mom and Pop Foods instead. You need to change the name of the recipient on the grocery check to “Mom and Pop Foods.”

1. Create a new method called `accessContent`:

```
public static void accessContent() {}
```

2. In your `accessContent` method, invoke `getEntries` on the `marchTrans` object:

```
List entryList = marchTrans.getEntries();
```

   The `entryList` contains all of the transactions in the content tree repesenting the data from `march.xml`.

3. Iterate through the list to find check transactions:

```
for(ListIterator i = entryList.listIterator(); i.hasNext(); ) {
    Entry entry = (Entry)i.next();
    if ( entry instanceof Check ){
```

4. Get the category of each check entry you find to determine if it is the groceries check:

```
CheckCategory category = ((Check) entry).getCategory();
if(category.equals(CheckCategory.GROCERIES)){
```

5. If the check is the groceries check, set the name of the recipient to “Mom and Pop Foods” and add the closing curly braces:

```
            ((Check)entry).setName("Mom and Pop Foods");
            break;
        }
    }
}
```

6. Invoke the `accessContent` method from the main method:

```
accessContent();
```

After you created your content tree for the April transactions, your landlord informs you that he has increased the rent to $2000. So, you need to change the rent check amount in the content tree representing your transactions for the month of April.

To change the amount of the rent check:

1. In the `accessContent` method, get the list of entries, but this time invoke `getEntries` on the `aprilTrans` object:

```
List aprilEntries = aprilTrans.getEntries();
```

2. Iterate through the list to find the rent check, and set the amount to $2000:

```
for(ListIterator i = aprilEntries.listIterator(); i.hasNext(); ) {
    Entry entry = (Entry)i.next();
    if ( entry instanceof Check ){
        CheckCategory category = ((Check) entry).getCategory();
        if(category.equals(CheckCategory.RENT)){
            entry.setAmount(new java.math.BigDecimal("2000.00"));
        }
    }
}
```

Notice in the final step that you did not have to cast entry to a `Check` to set the amount of the check. This is because you specified in the binding schema that `amount` is one of the members of the `Entry` interface because it is one element that `check`, `deposit`, and `withdrawal` all share. So, any `Entry` instance will have an amount. In step 4 of the first set of steps, you had to cast `entry` to a `Check` to set the name because `name` is not one of the members of the `Entry` interface. The reason you did not specify `Entry` to contain a name is because `withdrawal` does not include `name` as one of its elements.

Because you have made changes to the content trees, you should make sure that your tree is still valid before marshalling it. The next section explains how to validate the content trees.

Validating

Before marshalling a content tree to an XML document, you must ensure that the content tree is valid with respect to the DTD. If you used unmarshalling rather than instantiation to build your content tree, and you have not changed the tree, you do not need to validate before marshalling because the unmarshalling process incorporates validation. If you used instantiation to build the tree, you will always need to explicitly perform validation before marshalling.

You have changed both content trees in the previous section, and so you must validate them before marshalling.

To validate both content trees:

1. Create a method called validateTrees:

```
public static void validateTrees() throws Exception {}
```

2. Within the method, call validate on both `marchTrans` and `aprilTrans`:

```
    marchTrans.validate();
    aprilTrans.validate();
```

3. Invoke `validateTrees` from the `main` method:

```
validateTrees();
```

Marshalling

After validating your content trees, you are ready to marshal them to new XML documents. Whether you built a content tree using unmarshalling or instantiation, you marshal the tree in the same manner.

To marshal the content trees:

1. Create a new method called `marshalTrees`:

   ```
   public static void marshalTrees() throws Exception {}
   ```

2. In the `marshalTrees` method, create new files to contain the updated content for both trees:

   ```
   File march_new = new File("march_new.xml");
   File april_new = new File("april_new.xml");
   ```

3. Create the `OutputStream` objects to send to the `marshal` method:

   ```
   FileOutputStream fMOut = new FileOutputStream(march_new);
   FileOutputStream fAOut = new FileOutputStream(april_new);
   ```

4. Invoke the `marshal` method on each tree:

   ```
   try {
      marchTrans.marshal(fMOut);
      aprilTrans.marshal(fAOut);
   } finally {
      fMOut.close();
      fAOut.close();
   }
   ```

5. Invoke `marshalTrees` from the `main` method:

   ```
   marshalTrees();
   ```

After you recompile your classes and run `CheckbookApp`, you will see the files `march_new.xml` and `april_new.xml` in your directory. If you compare `march.xml` with `march_new.xml`, you will find that the only difference between the two files is the name of the groceries check, which you changed. JAXB preserves the equivalence between an XML document and the same XML document marshalled from its content tree.

The next section shows you how to add the content tree representing the April transactions to the content tree representing the March transactions.

Appending Content Trees

Since an object in a content tree can have more than one parent, you can append content trees together. We can employ this technique to append the April transactions to the March transactions before adding the transactions to the checkbook as shown in the next chapter.

To append `AprilTrans` to `Trans`:

1. Create a new method called `appendTrees`:

   ```
   public static void appendTrees() {}
   ```

1. In the new method, get the list of entries from each `Transactions` object:

   ```
   List mEntries = marchTrans.getEntries();
   List aEntries = aprilTrans.getEntries();
   ```

2. Use the `List` method, `addAll`, to add the entire list of April transactions to the list of March transactions:

   ```
   mEntries.addAll(aEntries);
   ```

3. Invoke `appendTrees` from the `main` method:

   ```
   appendTrees();
   ```

The next chapter shows you how to extend the `Checkbook` class to provide functionality for adding your March and April transactions to your checkbook and balancing the checkbook.

CHAPTER 6

Working With The Data

This chapter shows you how to use extension to add application-specific functionality to your JAXB application. We will continue with the CheckbookApp you created in the previous chapter. In this chapter, we'll create a new class, called CheckbookBalance. This new class will extend the generated class Checkbook and will contain a method that adds the transactions to the checkbook and computes the new balance. The CheckbookApp class calls this new method to add the transactions from March and April (represented by the content trees you created in the previous chapter) to the checkbook. In this chapter, we will work with the checkbook.xml document, which represents the checkbook, the list of transactions and the account balance.

The Example XML Document: checkbook.xml

The checkbook.xml document defines a checkbook, which contains a set of transactions and a balance, so instead of defining transactions as the root element, this document defines the checkbook element as the root element, which contains the list of transactions and the balance:

```
<?xml version="1.0" encoding="US-ASCII"?>
<checkbook>
  <transactions>
    <deposit category="salary">
      <date>02-09-2001</date>
      <name>Me</name>
      <amount>2500.00</amount>
    </deposit>
    <check number="90" category="other">
      <date>02-12-2001</date>
      <name>My Local Bookstore</name>
      <amount>34.95</amount>
```

```
      <pending/>
      <memo>Duke's Book</memo>
    </check>
    <check number="91" category="rent">
      <date>02-28-2001</date>
      <name>Landlord</name>
      <amount>1500.00</amount>
      <void/>
      <memo>February</memo>
    </check>
  </transactions>
  <balance>50000.00</balance>
</checkbook>
```

Notice that your checkbook only has the February transactions in it. This chapter will show you how to add the March and April transactions to the checkbook and update the balance. The `checkbook.xml` file is also located in the `examples/checkbook` directory of your installation.[1]

Setting Up the `CheckbookBalance` Class

To create the `CheckbookBalance` class:

1. Create a file called `CheckbookBalance.java`
2. Import these packages:

```
import java.io.*;
import java.util.*;
import java.math.*;
```

3. Declare the `CheckbookBalance` class so that it extends `Checkbook`:

```
public class CheckbookBalance extends Checkbook {
}
```

4. Inside `CheckbookBalance`, create the `balanceCheckbook` method:

```
public class CheckbookBalance extends Checkbook {
    void balanceCheckbook(Transactions trans) throws Exception {
    }
}
```

1. Note that the checkbook.xml file located in your examples/checkbook directory has a bug. The dates are supposed to be February dates, not March dates. This bug will be fixed for FCS. The file shown in this chapter and in the appendix is correct.

This method will contain all the code to add transactions to the checkbook and compute the new balance. The `CheckbookApp` class will pass the entire list of transactions that you created in the *Appending Content Trees* section to the `balanceCheckbook` method.

The completed `CheckbookBalance.java` file is located in your `docs` directory.

Extending the Derived Classes

In the *Building Data Representations* chapter, you learned how to use the derived classes directly. Another way of using the derived classes is through extension. Extension involves subclassing a derived class to provide application-specific functionality. This section demonstrates extension by showing you how to balance your checkbook and add your transaction entries to the checkbook.

Unmarshalling

Before you can perform the calculations, you need to unmarshal the file containing your transactions into a content tree, just as you did in the *Unmarshalling* section of the previous chapter. However, this time you are extending `Checkbook` rather than using it directly, which means that you unmarshal a `CheckbookBalance` object, not a `Checkbook` object. If you do not specify that `CheckbookBalance` must be unmarshalled, the unmarshalling process will return a `Checkbook` object, not a `CheckbookBalance` object. To solve this problem, you need to register `CheckbookBalance` with a `Dispatcher`.

Dispatching

A *dispatcher* maps element names to class names and sublass names to class names. The unmarshalling process begins with the invocation of a dispatcher's `unmarshal` methods. A default dispatcher unmarshals XML content into instances of generated classes. For our example, this means that the `unmarshal` method would return a `Checkbook`, not a `CheckbookBalance`. To return a `CheckbookBalance`, you need to register `CheckbookBalance` with the `Dispatcher`.

To register `CheckbookBalance` with a `Dispatcher`:

1. At the top of `CheckbookApp.java`, initialize a `CheckbookBalance` object:

```
public static CheckbookBalance chBook = new CheckbookBalance();
```

2. Create a new method in CheckbookApp.java called unmarshalSubclass:

```
public static void unmarshalSubclass() throws Exception{}
```

3. In the unmarshalSubclass method in CheckbookApp.java, acquire the default Dispatcher from Checkbook:

```
Dispatcher d = Checkbook.newDispatcher();
```

4. Register CheckbookBalance with the returned Dispatcher:

```
d.register(Checkbook.class, CheckbookBalance.class);
```

This method registers the CheckbookBalance subclass with the Dispatcher so that it unmarshalls a CheckbookBalance instead of a Checkbook.

5. Invoke unmarshalSubclass from the main method:

```
unmarshalSubclass();
```

Unmarshalling the Subclass

Because you registered CheckbookBalance with a dispatcher, you need to call the dispatcher's unmarshal method, not the CheckbookBalance object's unmarshal method.

To unmarshal CheckbookBalance:

1. In the unmarshalSubclass method, read checkbook.xml into a FileInputStream:

```
File checkbookNew = new File("checkbook.xml");
FileInputStream fNewIn = new FileInputStream(checkbookNew);
```

2. Cast the object returned by the unmarshal method to a CheckbookBalance object:

```
try{
    chBook = (CheckbookBalance)(d.unmarshal(fNewIn));
} finally {
    fNewIn.close();
}
```

The next section demonstrates how to implement the checkbook-balancing calculations in CheckbookBalance.java.

Adding Functionality

The CheckbookBalance.java file contains only one method: balanceCheckbook. This section shows you how to implement balanceCheckbook to calculate your new balance based on the previous month's balance and add transactions from March and April.

To implement the balanceCheckbook method in the CheckbookBalance class:

1. Get the current balance recorded in your checkbook:

   ```
   BigDecimal balance = this.getBalance();
   ```

2. Get the list of entries from the Transactions object that is passed into this method:

   ```
   List tEntries = trans.getEntries();
   ```

3. Initialize a BigDecimal to keep track of the amount of each transaction:

   ```
   BigDecimal amt;
   ```

4. Create a loop to iterate through the list of transactions and get the amount of each transaction:

   ```
   for (ListIterator i = tEntries.listIterator(); i.hasNext(); ) {
     Entry entry = (Entry)i.next();
     amt = entry.getAmount();
   ```

5. If the entry is a Deposit, add the amount of the deposit to the checkbook balance; otherwise, subtract the amount of the transaction from the balance:

   ```
    if (entry instanceof Deposit){
         balance = balance.add(amt);
    } else {
         balance = balance.subtract(amt);
    }
   ```

6. After re-calculating the balance based on a transaction, add the transaction to the list of transactions in the checkbook:

   ```
     this.getTransactions().getEntries().add(entry);
   }
   ```

7. After you have looped through the list of transactions, check if the balance is negative. If it is, warn the account holder that the account is overdrawn:

   ```
   if(balance.compareTo(new BigDecimal(0.00)) == -1){
         System.out.println("You are overdrawn.");
   }
   ```

8. Print out the new balance and set the balance in the checkbook to the new balance:

```
System.out.println("Your balance is: "+balance);
this.setBalance(balance);
```

The next section shows you how to invoke the new functionality in `CheckbookBalance` from `CheckbookApp.java`.

Using the New Functionality in Your Application

This section shows you how to use the `balanceCheckbook` method from `CheckbookApp` to balance your checkbook and add the transactions from March and April to the checkbook.

The `CheckbookBalance` class is written in a generic way so that it can take any `Transactions` object and add the transactions to a checkbook and calculate the new balance. All you need to do from your application is call the `balanceCheckbook` method with your particular `Transactions` object and marshal the result to a new checkbook file.

To update your checkbook with the new transactions and balance:

1. In the main method of the `CheckbookApp` class, call the `balanceCheckbook` method with the marchTrans object:

   ```
   chBook.balanceCheckbook(marchTrans);
   ```

   The `marchTrans` object contains the list of transactions from March and April that you created in the *Appending Content Trees* section.

   The `chBook` object now contains the updated checkbook transactions and balance.

2. Create a new method in `CheckbookApp` called `validateAndMarshalCheckbook`:

   ```
   public static void validateAndMarshalCheckbook() throws Exception {}
   ```

3. Since you edited the checkbook content tree, perform validation on `chBook` within the new method you created:

   ```
   chBook.validate();
   ```

4. Create a new XML file for the updated checkbook:

   ```
   File checkbook_new = new File("checkbook_new.xml");
   FileOutputStream fCOut = new FileOutputStream(checkbook_new);
   ```

5. Marshal the updated checkbook to the new XML file:

```
try {
    chBook.marshal(fCOut);
} finally {
    fCOut.close();
}
```

6. Invoke `validateAndMarshalCheckbook` from the main method of `CheckbookApp`:

```
validateAndMarshalCheckbook();
```

7. Save `CheckbookBalance.java` and `CheckbookApp.java` and recompile:

```
javac *.java
```

8. Run `CheckbookApp` again:

```
java CheckbookApp
```

Your `checkbook_new.xml` file has the updated balance of $48065.72 and contains your February, March, and April transactions.

APPENDIX **A**

The Example DTD, XML Documents, and Binding Schema

The DTD: checkbook.dtd

```
<!ELEMENT checkbook ( transactions, balance ) >
<!ELEMENT transactions ( deposit | check | withdrawal )* >
<!ELEMENT deposit ( date, name, amount )>
<!ATTLIST deposit
            category ( salary | interest-income | other ) #IMPLIED >
<!ELEMENT check ( date, name, amount, ( pending | void | cleared ), memo? )
>
<!ATTLIST check
            number CDATA #REQUIRED
            category ( rent | groceries | other ) #IMPLIED  >
<!ELEMENT withdrawal  ( date, amount ) >
<!ELEMENT balance (#PCDATA) >
<!ELEMENT date (#PCDATA ) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT amount (#PCDATA) >
<!ELEMENT memo (#PCDATA) >
<!ELEMENT pending EMPTY >
<!ELEMENT void EMPTY >
<!ELEMENT cleared EMPTY >
```

The March Transactions: march.xml

```
<?xml version="1.0" encoding="US-ASCII"?>

<transactions>
  <deposit>
    <date>04-14-2001</date>
    <name>Me</name>
    <amount>101.01</amount>
  </deposit>
  <check number="2" category="groceries">
    <date>03-15-2001</date>
    <name>Conglomerate Foods</name>
    <amount>34.95</amount>
    <pending/>
    <memo>food</memo>
  </check>
  <withdrawal>
    <date>03-16-2001</date>
    <amount>0.34</amount>
  </withdrawal>
</transactions>
```

The Checkbook: checkbook.xml

```
<?xml version="1.0" encoding="US-ASCII"?>

<checkbook>
  <transactions>
    <deposit>
      <date>02-09-2001</date>
      <name>Me</name>
      <amount>1500.00</amount>
    </deposit>
    <check number="90" category="other">
      <date>02-12-2001</date>
      <name>Faberge</name>
      <amount>34.95</amount>
      <pending/>
      <memo>Faberge Eggs</memo>
    </check>
    <withdrawal>
      <date>02-27-2001</date>
      <amount>20.00</amount>
```

```
    </withdrawal>
    <check number="91" category="rent">
      <date>02-29-2001</date>
      <name>Landlord</name>
      <amount>1500.00</amount>
      <void/>
      <memo>February</memo>
    </check>
  </transactions>
  <balance>50000.00</balance>
</checkbook>
```

The Binding Schema: checkbook.xjs

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<xml-java-binding-schema version="1.0ea">
  <element name="checkbook" type="class" root="true" />
  <element name="transactions" type="class" root="true">
    <content>
      <choice property="entries" collection="list" supertype="Entry" />
    </content>
  </element>
  <element name="balance" type="value" convert="BigDecimal"/>
  <element name="amount" type="value" convert="BigDecimal" />
  <element name="date" type="value" convert="TransDate" />
  <element name="deposit" type="class" >
    <attribute name="category" convert="DepCategory" />
  </element>
  <element name="check" type="class" >
    <content>
      <element-ref name="date"/>
      <element-ref name="name" />
      <element-ref name="amount" />
      <choice property="pend-void-clrd"/>
    </content>
    <attribute name="number" convert="int" />
    <attribute name="category" convert="CheckCategory" />
  </element>
  <conversion name="BigDecimal" type="java.math.BigDecimal" />
  <conversion name="TransDate" type="java.util.Date"
              parse="TransDate.parseDate" print="TransDate.printDate" />
  <enumeration name="DepCategory" members="salary interest-income other"/>
  <enumeration name="CheckCategory" members="rent groceries other"/>
  <interface name="Entry" members="Deposit Check Withdrawal"
              properties="date amount" />
</xml-java-binding-schema>
```

APPENDIX **B**

The Application Files

The Main Application File: CheckbookApp.java

```
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import javax.xml.marshal.*;

public class CheckbookApp {

    public static Transactions marchTrans = new Transactions();
    public static Transactions aprilTrans = new Transactions();
    public static CheckbookBalance chBook = new CheckbookBalance();

    public static void main(String[] args) throws Exception{

        // Build the content trees
        buildTrees();

        // Access content of trees
        accessContent();

        // Validate the trees
        validateTrees();

        // Marshal the trees
        marshalTrees();

        // Append the april transactions to the march transactions
        appendTrees();
```

```
    // Unmarshal the checkbook subclass
    unmarshalSubclass();

    // Add the transactions to the checkbook and update the balance
    chBook.balanceCheckbook(marchTrans);

    // Validate the updated checkbook and marshal it
    validateAndMarshalCheckbook();


 }

// Building the content trees
public static void buildTrees() throws Exception{

        // Unmarshall the march.xml file
        File march = new File("march.xml");
        FileInputStream fIn = new FileInputStream(march);
        try {
         marchTrans = marchTrans.unmarshal(fIn);
        } finally {
         fIn.close();
        }

       // Instantiate a content tree for the April transactions
       List aprilEntries = aprilTrans.getEntries();
       Check aprilRentCheck = new Check();
       CheckCategory aprilRent = CheckCategory.RENT;
       aprilRentCheck.setCategory(aprilRent);
       aprilRentCheck.setName("Me");
       aprilRentCheck.setNumber(51);
       aprilRentCheck.setDate(TransDate.parseDate("04-12-2001"));
       aprilRentCheck.setAmount(new java.math.BigDecimal("1500.00"));
       Pending pending = new Pending();
       aprilRentCheck.setPendVoidClrd(pending);
       aprilEntries.add(aprilRentCheck);
}

// Access content of trees
public static void accessContent() {

       // Edit the name on the groceries check in the Trans contenttree
      List entryList = marchTrans.getEntries();
      Entry entry;
      for (ListIterator i = entryList.listIterator(); i.hasNext(); ) {
         entry = (Entry)i.next();
         if( entry instanceof Check ) {
            CheckCategory category = ((Check)entry).getCategory();
            if (category.equals(CheckCategory.GROCERIES)) {
                ((Check)entry).setName("Mom & Pop Foods");
                break;
```

```
            }
          }
        }

        // Edit the rent check in the aprilTrans content tree
        List aprilEntries = aprilTrans.getEntries();
        for (ListIterator i = aprilEntries.listIterator(); i.hasNext();
) {
          entry = (Entry)i.next();
          if ( entry instanceof Check ) {
              CheckCategory category = ((Check)entry).getCategory();
              if (category.equals(CheckCategory.RENT)) {
                  entry.setAmount(new java.math.BigDecimal("2000.00"));
                  break;
              }
          }
        }

    }

    // Validate the trees
    public static void validateTrees() throws Exception{
        // Validate the two content trees
        marchTrans.validate();
        aprilTrans.validate();
    }

    // Marshal the trees
    public static void marshalTrees() throws Exception {

        // Create output files for the two content trees
        File march_new = new File("march_new.xml");
        File april_new = new File("april_new.xml");
        FileOutputStream fMOut = new FileOutputStream(march_new);
        FileOutputStream fAOut = new FileOutputStream(april_new);

        // Marshal the two content trees to new XML documents
        try {
          marchTrans.marshal(fMOut);
          aprilTrans.marshal(fAOut);
        } finally {
          fAOut.close();
        }
    }

    // Append the april transactions to the march transactions
    public static void appendTrees() {

        // Append the aprilTrans content tree to the Trans content tree
        List mEntries = marchTrans.getEntries();
        List aEntries = aprilTrans.getEntries();
```

```
            mEntries.addAll(aEntries);
        }

        // Unmarshal the checkbook subclass
        public static void unmarshalSubclass() throws Exception{

            // Register the subclass of Checkbook with a dispatcher
            Dispatcher d = Checkbook.newDispatcher();
            d.register(Checkbook.class, CheckbookBalance.class);

            // Unmarshal the checkbook.xml file
            File checkbookNew = new File(“checkbook.xml”);
            FileInputStream fNewIn = new FileInputStream(checkbookNew);

            // Unmarshal the checkbook file to a CheckbookBalance
            try {
                chBook = (CheckbookBalance) (d.unmarshal(fNewIn));
            } finally {
                fNewIn.close();
            }
        }

        // Validate the updated checkbook and marshal it
        public static void validateAndMarshalCheckbook() throws Exception{

           chBook.validate();

           // Create an output file for the updated checkbook
           File checkbook_new = new File(“checkbook_new.xml”);
           FileOutputStream fCOut = new FileOutputStream(checkbook_new);

            // Marshal the updated checkbook
           try {
            chBook.marshal(fCOut);
           } finally {
            fCOut.close();
           }
        }
}
```

The Subclass: CheckbookBalance.java

```
import java.util.*;
import java.io.*;
import java.math.*;

public class CheckbookBalance extends Checkbook {
```

```
void balanceCheckbook(Transactions trans) throws Exception {

          // Get the current balance of the checkbook
          BigDecimal balance = this.getBalance();

          // Get the list of transactions from the Trans object
          List tEntries = trans.getEntries();

          // Initialize a BigDecimal to track the amount of each transaction
          BigDecimal amt;

          // Iterate through the transaction list, recalculate the balance,
          // and add the transaction to the checkbook
          for (ListIterator i = tEntries.listIterator(); i.hasNext(); ) {
            Entry entry = (Entry)i.next();
            amt = entry.getAmount();
            if (entry instanceof Deposit){
              balance = balance.add(amt);
            } else {
               balance = balance.subtract(amt);
            }
            this.getTransactions().getEntries().add(entry);
          }

          // Check if the balance is negative.
          if(balance.compareTo(new BigDecimal(0.00)) == -1){
              System.out.println("You are overdrawn.");
          }

          // Output the new balance
          System.out.println("Your balance is: "+balance);

          // Update the balance in the checkbook.
          this.setBalance(balance);

  }
}
```