*Converting XML Documents into Java Objects*

# Java™ & XML Data Binding

O'REILLY®

*Brett McLaughlin*

# Java™ and XML Data Binding

*Brett McLaughlin*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

# CHAPTER 3
# Generating Classes

Now that we're through the formalities, I want to focus specifically on the JAXB data binding framework. In this chapter, I start by discussing how to take a set of XML constraints and convert those constraints to a set of Java source files. In addition to seeing how this work with JAXB, this chapter should give you a solid idea of how class generation works so that when we move to other frameworks (in the second half of this book), you'll already have a handle on class generation and how it works. I also briefly touch on the future of JAXB—specifically, which constraint models are supported and which should be supported in future versions.

Without belaboring the point, I want to be clear that this and other JAXB chapters were written using a prerelease version of Sun's JAXB framework (the 1.0 version was not yet available). Because of this, small inconsistencies may creep in as this book goes to press. If you run across a problem with the examples, consult the JAXB documentation and feel free to contact us. Details of who to send mail to are in the preface of the book, and you can also check the book's web site at *http://www.newInstance.com*.

## Process Flow

First, let's run through the process flow involved with generating constraints. This will help you get an idea of where we're going and how the pieces in this chapter fit together. It should also form a simple mental checklist for you to follow when generating classes; if you skip a step, problems crop up, so be sure to take each in turn. Here's how the steps break down:

1. Create a set of constraints for your XML data.
2. Create a binding schema for converting the constraints into Java.
3. Generate the classes using the binding framework.
4. Compile the classes and ensure they are ready for use.

I'll cover each step in order.

## Constraints

The first step is to create a set of constraints for your XML data. If you followed my advice from Chapter 2, then you are doing this *before* writing your XML documents. That tends, as I mentioned, to produce more organized constraint models. You'll want to ensure that your constraint model is complete, as well; the last thing you want is to have to add an attribute or element that you forgot and then regenerate your source files. As mentioned previously, this can cause conflicts with older versions of generated classes conflicting with your updated ones.

Additionally, now you need to ensure that your constraint model syntax is supported by the binding framework you want to use. In other words, if you go to a lot of trouble to generate a documented XML Schema and then find out that your framework of choice supports only DTDs, expect some yelling and screaming. Take the time *before* writing constraints to verify this, or you can't say that I didn't warn you when things get ugly. As a general rule, you will never go wrong using DTDs right now, as all frameworks support them. I'd guess that a year or two from now, XML Schemas will be just as safe, but the frameworks simply aren't there yet.

Once you've developed your constraints, you need to perform some level of testing before you run your class generation tools on them. This is a crucial step, as it verifies that your data is going to match up with your constraints. Write several XML documents (or use existing ones, if you have them already) and validate them against your new constraints. This can be done with Xerces, your favorite XML parser, or various IDEs available for XML authoring. You'll want to try and test as many different documents as you can, preferably with a variety of data in them. Testing many different documents is the best way to make sure you didn't misname or leave something out, which would cause problems down the line. Once you've got the verified constraint model and are happy with it, you're ready to move on to a binding schema.

You should realize that documentation and comments in your DTD or constraint model will not affect class generation. Hopefully that doesn't urge you to leave documentation out but pushes you to write well-formatted comments. This will help your co-workers and generally make life easier. So please, comment, comment, comment.

## Binding Schema

Once you've got your constraint set ready, you'll need to write a binding schema for most frameworks. There is a lot of variance from the simplest binding schema to the most complex, so don't expect me to cover all the details of binding schemas here, or even in this chapter. I'll explain the basic options in this chapter and then devote Chapter 6 to a complete exploration of the topic. You will get a taste of what's to come in this chapter, though.

You'll notice that I put a qualifier on the first sentence of that last paragraph: *most* frameworks. Some data binding frameworks do not require a binding schema, although they may allow more advanced options through the use of one. Currently, JAXB requires a binding schema, but Castor and Zeus do not. The Coins framework uses a significantly different process, but does employ the idea of a binding schema. So while you may always provide a binding schema for the sake of specifying options, realize that you don't have to in some cases.

Binding schemas provide the ability to specify both local and global options, and this concept is important to grasp. For example, specifying the Java package to generate source code within is a global option and affects all generated code. However, supplying a class name of `Employee` for the XML element `person` is a local option and applies only to that element. You'll want to be very careful when setting global options, as every generated class is affected. Of course, some frameworks allow you to override global options for specific elements, so you often get the best of both worlds.

Finally, you need to know the format that your framework uses for binding schemas. As I already mentioned, this is generally some XML-compliant format. The elements and attributes allowed by each framework often varies, though; be sure to use the correct conventions for the correct framework. As JAXB standardizes, expect to see binding schema syntax to converge on what JAXB uses, but for now things are still a bit spread out across various frameworks. Once you've developed your binding schema, though, you can pass it along with your constraints and wait for the magic to happen.

## Generation

At this point, the actual mechanics of class generation kick in. This is generally a sort of "black box," as frameworks each approach this step of the process differently. You supply a set of constraints, usually a binding schema, and out pops a set of source code ready for compilation. Because JAXB is closed source and the code is not available for viewing, I'm not going to get into specifics of how JAXB's black box works. In the chapters for the open source frameworks, I will address these details, but for JAXB, just trust the framework to do that hard work.

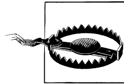
## Source Code

The result of the generation step is one or more Java source files. These files should be ready for compilation, using normal Java approaches (`javac`). At this point, frameworks generally leave you on your own, assuming you can compile these classes to a directory and location of your choice. Be sure to use the `-d` switch (on `javac`) so that any package you specified is built into the output location of your compiled classes.

### What About Multithreading?

This book focuses mainly on how to use data binding APIs and therefore doesn't spend much time on issues like threading, locking, and multiprocessing. However, for those of you who are wondering, here's a short look at how multithreading affects data binding.

It is important to realize that class generation does not make any changes to either your constraint model or your binding schema; these can be used repeatedly without any problem. However, like XML parsers, you'll want to avoid trying to process these documents (the constraints and binding schema) with multiple processes simultaneously. This is a basic I/O principle, but is always worth saying for those of you getting a little overzealous with threading. It also brings up another important concept: compile-time class generation. While it's certainly possible to generate classes from constraints at runtime, it isn't a very good idea unless you're writing a data binding tool. While it's possible to shove the generated source code into a `javac` process and then even hook a Java `ClassLoader` into the resultant classes, this is really not a good idea. I highly recommend generating source at compile time, compiling these files, and then using them at runtime, in the plain-vanilla standard Java approach.



There are a few odd cases in which data binding packages generate source code that will not compile. This is almost always the result of a bug in the data binding implementation, rather than something you have done incorrectly. I'll address some of these cases in the text, but if you see this occurring, you should report your problem to the mailing list for the framework being used.

Keep in mind, though, that this source code may not be in a pretty, formatted, commented state (as all the rest of your code is, right?). This means that Javadoc and other documentation methods on these classes will be terse, if not nonexistent. Hopefully this will change as frameworks get the basics down and move on to finer details like this. Additionally, the generated classes will almost always be dependant on one another, and will need to be compiled at the same time. Once you've got a set of Java classes, simply add them to your classpath, and you are ready to use them.

Once you've put all of this into one coherent process, the result is similar to that shown in Figure 3-1. Even if you are using a framework other than JAXB, this process will be similar for any class generation setup.

## Creating the Constraints

The first step in getting ready for class generation, as you can see from Figure 3-1, is getting a set of constraints ready to generate classes from. As this isn't a book on writing XML (and there are plenty of good ones on the subject already), I'm not going to spend time describing how to formulate constraints.

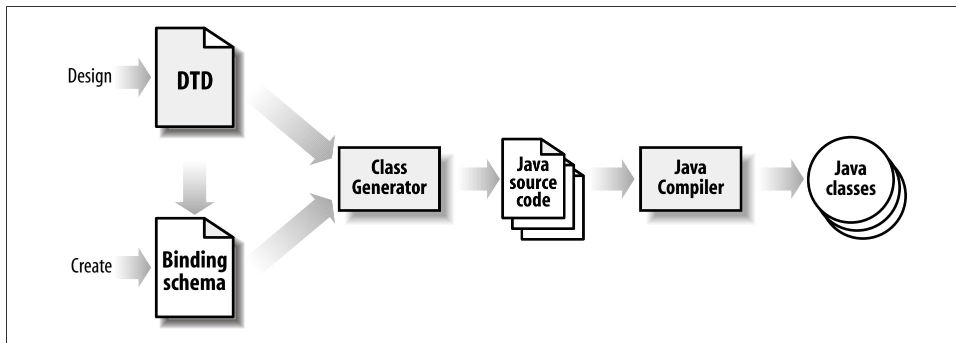



*Figure 3-1. Class generation process flow*

I'll assume that you're capable of figuring out how you want your data represented and then using DTDs or schemas or your constraint model of choice to describe that data. I do want to touch on a few points relevant to data binding, and JAXB specifically, though, and then provide several DTDs for working through the examples.

## JAXB and DTDs

First, as I've mentioned several times, JAXB currently supports only DTDs. From what I can gather from the specification, newsgroups, and mailing lists, this is the plan all the way through the 1.0 final version of the specification and framework. There is a lot of momentum to follow up this release with a "version.next" that does support XML Schema, though.* JAXB does support all the features of DTDs, so you should be able to use any DTDs you've already developed for your data binding needs.

To get started, I want to present a simple DTD that I'll use as a starting point for most of the rest of this chapter. Example 3-1 shows that DTD, which represents a simple movie database.†

*Example 3-1. Movie database DTD*

```
<!ELEMENT movies (movie+)>
<!ATTLIST movies
        version    CDATA    #REQUIRED
>
```

* I realize that for some of you, this may seem contradictory to what you've heard. Early on in the JAXB effort (back in the "codename: Adelard" days, there was a lot of talk about XML Schema support in the first version. That talk died off, though, as getting out even a DTD version began to take more time. In other words, deadlines slipped and things changed.

† Occasionally, folks ask me why I don't use more realistic examples like a telecommunications PoP configuration file, a financial planning package (in XML), or something similar. These examples rarely make sense unless you're in those particular industries, so I chose examples that don't require special knowledge of a specific industry.

*Example 3-1. Movie database DTD (continued)*

```
<!ELEMENT movie (title, cast, director?, producer*)>

<!ELEMENT cast (actor+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT producer (#PCDATA)>

<!ELEMENT actor (#PCDATA)>
<!ATTLIST actor
                headliner    (true | false)    'false'
>
```

> **Do I Really Have to Type This in?**
>
> Since most of you are busy writing your own code and don't want to type the examples in by hand, they are all available for download from this book's web site, *http://www.newInstance.com*. Navigate to the Writing link, click the cover for this book, and you'll be able to read updates on the book; download the DTDs, XML documents, binding schemas, and Java classes from the examples, and find other supplemental material. You'll also learn about new editions, extra goodies found only online, and more, so check it out.

This is pretty basic stuff; just so you get an idea of how this looks when presented as data, Example 3-2 shows an XML document that conforms to this DTD.

*Example 3-2. Sample movie database*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE movies SYSTEM "movies.dtd">

<movies version="1.1">
  <movie>
    <title>Pitch Black</title>
    <cast>
      <actor headliner="true">Vin Diesel</actor>
      <actor headliner="true">Radha Mitchell</actor>
      <actor>Vic Wilson</actor>
    </cast>
    <producer>Tom Engelman</producer>
  </movie>
  <movie>
    <title>Memento</title>
    <cast>
      <actor headliner="true">Guy Pearce</actor>
      <actor headliner="true">Carrie-Anne Moss</actor>
    </cast>
    <director>Christopher Nolan</director>
```

*Example 3-2. Sample movie database (continued)*

```
    <producer>Suzanne Todd</producer>
    <producer>Jennifer Todd</producer>
  </movie>
</movies>
```

There isn't anything remarkable here; I've simply illustrated what XML looks like in relation to its constraints. Before moving on to binding schemas, though, there are a few more things to point out.

## Deterministic Modeling

First on the list of important considerations is *determinism* in your models. I know that sounds like something you'd hear in a political speech, but it is pretty important. Determinism is a fancy word for unambiguous and basically means that your constraint cannot be misinterpreted or interpreted as more than one possibility. If a particular constraint cannot be interpreted without looking ahead or could also fulfill another model, it is *nondeterministic*. For example (from the XML recommendation):

```
<!ELEMENT non-deterministic ((b, c) | (b, d))>
```

Here, if a `b` element is encountered, it's not clear whether a parser should expect a `c` or a `d` element to follow it. This would require the parser to read ahead and therefore is nondeterministic. To fix this problem, you would collapse the declaration to:

```
<!ELEMENT deterministic (b, (c | d))>
```

This also illustrates an important point: generally, changing a nondeterministic model into a deterministic one. Nondeterminism is a pain to deal with when you're trying to validate XML; it's flat-out impossible to deal with in data binding. The class generation tools will either completely choke or produce all sorts of wild results (try it sometime; it's actually sort of fun!). Generally, XML IDEs will catch this, but you'll want to watch for this problem, as it creates uncertain results and is a nonobvious problem for constraints to have.

## Simple Elements

Another thing to think about when defining your constraints is *simple element* definitions. A simple element is an element that has only textual content. Its model looks like this:

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT director (#PCDATA)>
```

Both elements are simple and contain only `PCDATA` (parsed character data). So, in generated classes, you might expect to do something like this:

```
String movieTitle = movie.getTitle( );
String director = movie.getDirector( );
```

However, this isn't the case. JAXB and other data binding frameworks are going to generate classes for your elements in the general case. There are ways to get around this, and I'll cover them in the chapter on binding schemas, but in the simplest case, you will need to write code that looks more like this:

```
Title titleObject = movie.getTitle( );
String movieTitle = titleObject.getValue( );

Director directorObject = movie.getDirector( );
String director = directorObject.getValue( );
```

As you can see, more of an object hierarchy is built than you might have expected. Of course, you could use the first version of the code if you changed the constraints to look like this:

```
<!ELEMENT movies (movie+)>
<!ATTLIST movies
          version    CDATA    #REQUIRED
>

<!ELEMENT movie (cast, producer*)>
<!ATTLIST movie
          title    CDATA    #REQUIRED
          director CDATA    #IMPLIED
>

<!ELEMENT cast (actor+)>
<!ELEMENT producer (#PCDATA)>

<!ELEMENT actor (#PCDATA)>
<!ATTLIST actor
                headliner    (true | false)    'false'
>
```

Here, I've collapsed these two simple elements into attributes on the `movie` element; the result is that they are generated as simple Java Strings and available through accessor methods on the generated `Movie` object. This can be extended into a more general principle: elements are turned into Java objects and attributes are turned into Java member variables (usually Java primitives like `int`, `float`, `String`, etc.). Here's the resultant object:

```
public class Movie {
    private String title;
    private String director;
    // Other variables

    public String getTitle( ) {
        return title;
    }

    public String getDirector( ) {
        return director;
    }
```

```
    // Other accessor and mutator methods...
}
```

The result is a much easier object to use. The moral of this little tale is that well-designed constraints can result in cleaner and easier-to-use generated objects. It also results in better XML design, as now single values are stored as XML attributes, with multiple values stored in XML elements.

However, you should be careful not to get too overzealous in this collapsing of simple elements. For example, you might look at the `producer` element and thinking you can collapse it into the `movie` element as an attribute as well. However, you'd end up with a different constraint model; you would be able to specify only one producer, instead of more than one, as desired. In this case, it's appropriate to have a separate `producer` element, since that element can occur multiple times within the `movie` element. You're going to end up with a list of producers in your code:

```
// Using Java collections...
List producerList = movie.getProducerList( );

// or, possibly...
Producer[] producerArray = movie.getProducer( );
```

Be careful not to go crazy with this approach, or you'll end up changing the constraint set itself, rather than just "enhancing" the one that you already have.

## Constraint Naming

A final consideration in constraint modeling: be careful of the names that you use. Remember that in data binding, your generated classes are going to use names defined in your DTD. Take this DTD fragment for part of a role-playing game's descriptor, for example:

```
<!ELEMENT characters (character+)>

<!ELEMENT character EMPTY>
<!ATTLIST character
          name    CDATA    #REQUIRED
          race    (human | elf | dwarf | orc) #REQUIRED
          class   (paladin | knight | mage | archer) #REQUIRED
>
```

This looks pretty innocent, until you run JAXB's class generation tool and end up with this source file fragment:

```
public class Character {
    // Normal variables and methods

    public String getClass( ) {
        return _Class;
    }
}
```

Obviously, I've simplified things a bit, but you can see immediately that this is not a class that will compile; if you know much about Java, you'll realize that `getClass()` is a method on `java.lang.Object` that cannot be overridden (it's declared `final`).

If you tried to compile the resultant classes, you'd get an error like this:

```
Character.java:51: getClass() in Character cannot override getClass() in
    java.lang.Object; overridden method is final
    public String getClass() {
                  ^
1 error
```

You would either need to rename the attribute in your DTD or use a binding schema to map the `class` attribute to a different variable name in Java.

Now, I want to issue a warning here, before you change all of our DTDs to use Java-compliant names. If the data you are describing is best named `class`, `string`, or any other reserved word in Java, *leave the name alone!* However, if you can more accurately name a piece of data by using a nonreserved word, then it's a good idea to take these steps now before doing any class generation.

The point I'm trying to make is that you should use the best names possible for your constraints, but you should *not* make decisions about your data based on the possibility that the data may be used by JAXB or any other data binding framework. You'll just want to make a note to yourself of any names that could cause trouble and be sure to map those names to legal Java ones (I'll cover this in detail in Chapter 6).

# Binding Schema Basics

Once you've got your constraints (I saved my movie database DTD as *movies.dtd*), you're ready to create a binding schema for your classes. This will instruct the class generation tool to generate classes, to use a specific Java package, to use collections, and a variety of other options. Although I won't spend a lot of time on the schemas in this chapter, I'll give you some basics that will get us through some simple examples. Specifically, I'll deal with global options here and leave the local options, as well as more advanced features, to Chapter 6.

## The Minimum Binding Schema

The first thing that you'll want to get a handle on is the "minimum binding schema." This is the least-amount-of-work principle; often, you'll want to generate classes from your DTD without any changes. To do this, you'll need to create a binding schema that provides very minimal information to the JAXB schema compiler tool.

The JAXB binding schema is an XML document, and the root element must be `xml-java-binding-schema`. It must also have a single attribute, `version`, and currently the only allowed value for this attribute is `1.0-ea`.*

The JAXB download comes with the DTD for this schema. It's located in the *[jaxb-root]/doc/* directory and called *xjs.dtd*.

For a minimal binding schema, you must specify the root element of the DTD being passed in; this allows JAXB to determine which generated object (in source code) is the "top-level" one. This is accomplished through the `element` element (yup, you read that right). By supplying the `root` attribute and giving it a value of `true`, you've given JAXB what it needs. Add to this the `name` attribute, which identifies the element you're working on and, finally, the `type` attribute, which tells JAXB what type of Java construct to create from the element. For the `movies` element, you want a Java class, so use the `class` value for this attribute.

That idea took a paragraph to explain, but requires only three or four lines to put into action. Example 3-3 shows a binding schema for the movie database DTD.

*Example 3-3. Binding schema for movie database*

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

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

Save this schema as *movies.xjs*. The standard extension for binding schemas in JAXB is *xjs*, and I'd recommend you use it as well. With this fairly small XML file, you're ready for basic data binding.

It is possible to perform basic class generation without a binding schema. The JAXB schema compiler allows you to specify the root element (or elements) on the command line to the compiler. I'm not a big fan of this approach, though, as it's impossible for another developer to know what you provided. In other words, the binding schema provides documentation about what options were used in class generation. For that reason, I encourage you to use the simple binding schema shown above, rather than the command-line options, for generating classes.

## Global Options

In addition to specifying the root element, a few other basic options are worth pointing out now. These are all global options, meaning that they affect all generated

* Presumably, other values will be allowed when subsequent versions of the binding schema are released.

classes. You will need to use the options element, which is a child of the top-level xml-java-binding-schema element, to specify these. Each option has an attribute on that element, and you give a value for the property you want to set. These global options and the attributes used to set them are summarized in Table 3-1.

*Table 3-1. Global binding schema options*

| Attribute name | Allowed values | Default | Purpose |
|---|---|---|---|
| package | Any legal package name | N/A | Sets the Java package that source files use (e.g., com.oreilly.jaxb) |
| default-reference-collection-type | array, list | list | Sets the default collection type for multiple-valued properties |
| property-get-set-prefixes | true, false | true | Indicates if the accessor and mutator methods generated have a get and set prefix (e.g., getTitle( ) versus title( )) |
| marshallable | true, false | true | Indicates whether this class should have a marshal( ) method generated |
| unmarshallable | true, false | true | Indicates whether this class should have an unmarshal( ) method generated |

As you can see, these options are generally pretty self-explanatory. For example, to generate the movies database classes within the javajaxb.generated.movies package, with all other options set to the default values, you'd use the binding schema shown in Example 3-4.

*Example 3-4. Modified binding schema for movies database*

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

<xml-java-binding-schema version="1.0-ea">
  <options package="javajaxb.generated.movies" />

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

Pretty simple, isn't it? The resultant classes are all in the specified package. In this example, I've added in the specification to generate multiple-valued properties as arrays instead of Java lists:

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

<xml-java-binding-schema version="1.0-ea">
  <options package="javajaxb.generated.movies"
           default-reference-collection-type="array" />

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

The result of this addition is apparent in the `Movies` class, which has multiple `Movie` subobjects. The methods generated look like this (using arrays):

```
public Movie[] getMovie() {
    // implementation
}

public void setMovie(Movie[] _Movie) {
    // implementation
}
```

I realize that to many of you, the name `getMovie()` may seem a bit odd. This is true for almost all programmers getting into data binding. While you'll learn how to change this method name in Chapter 6, you should be aware that many frameworks (including some covered in this book) use this same sort of naming schema. It's not pretty, but you might want to start getting used to it.

Without using this property, Java collection classes are used, and the same method looks like this in the generated source code:

```
public List getMovie() {
    // implementation
}

public void deleteMovie() {
    // implementation
}

public void emptyMovie() {
    // implementation
}
```

As you can see, there is both a different return value from the `getMovie()` method, as well as a few new methods added, specific to Java `List` types. One other thing to notice is that there isn't a `setMovie(List movie)` method. To change the movies list, you'll need to write code like this:

```
// Obtain the current list
List movieList = movies.getMovie();

// The list is live, so we can operate upon it directly
movieList.add(newMovie);
movieList.add(anotherNewMovie);
```

As you can see, the Java `List` returned is live, so you can simply operate upon it rather than continuing to work with the `Movies` object. You should also take care with the types that you add to this list, as Java collections are not type-safe; you could just as easily add strings, dates, or other objects that would cause problems later on when converting the objects back to XML.

I also want to advise you against ever using the `property-get-set-prefixes` option. The result is a pair of methods like this:

```
public String title( ) {
    // implementation
}

public void title(String title) {
    // implementation
}
```

Here, the accessor (for retrieving values) and the mutator (for setting them) have the same method name since the prefixes have been removed. With only the return type and parameters different, this is extremely confusing. Because it doesn't help in any situation, results in confusing code, and requires extra work in the binding schema, I'd urge you to simply stay away from the option.

I realize that I've rushed through most of these details; we'll revisit all of this in detail in the chapter on binding schemas, so don't worry if you're a little dizzy. However, with the basics introduced here, you're ready to get to the actual source code generation and see these options in action for yourself.

# Generating Java Source Files

At this point, you've got all of the required components to generate source code from the movie database constraint set. In this section, I detail the actual process of using the command-line tools in JAXB to generate classes. You'll find out how to get set up with the JAXB framework, use the provided scripts, and actually generate classes.

## Getting Set Up

The first thing you need to do, if you haven't already, is download the JAXB release. Visit *http://java.sun.com/xml/jaxb* and follow the links to download the reference implementation of JAXB. I also recommend that you download the PDF specification for reference. Once you've got the release (named something like *jaxb-1_0-bin.zip*), you'll want to extract this to a directory on your hard drive. On my Windows machine, I used *c:\dev\javajaxb\jaxb-1.0*, and it's extracted at */dev/javajaxb/jaxb-1.0* on my Mac (running OS X).

You'll want to note the two *jar* files in the *lib/* directory, *jaxb-rt-1.0-ea.jar* and *jaxb-xjc-1.0-ea.jar*. The first is used for JAXB classes at runtime (indicated by the *rt*), and the second contains the classes used in schema compilation. In other words, you'll want the first in your classpath for your applications using generated classes and the second in your classpath when generating those classes.

Additionally, JAXB comes with a script in the *bin/* directory, used for invoking the Java class that starts the schema compiler for class generation. However, at least in

the version I've got, this script works only on Unix-based systems. Instructions for invoking the JAXB schema compiler on Windows are available, but they are pretty poor. To help Windows users, Example 3-5 shows a batch file that invokes the schema compiler (and report errors usefully) for Windows systems. I've saved it as *xjc.bat*, also in my *bin/* directory.

*Example 3-5. Batch file for class generation using JAXB*

```
@echo off

if "%JAVA_HOME%" == "" goto java_home_error
if "%JAXB_HOME%" == "" goto jaxb_home_error

set LOCALCLASSPATH=%JAVA_HOME%\lib\tools.jar;%JAXB_HOME%\lib\jaxb-xjc-1.0-ea.jar

echo Starting JAXB Schema Compiler...

rem The next two lines of text are ONE line in the batch file!!
"%JAVA_HOME%\bin\java.exe" -classpath "%LOCALCLASSPATH%"
    com.sun.tools.xjc.Main %1 %2 %3 %4 %5 %6 %7 %8 %9

goto end

:java_home_error

echo ERROR: JAVA_HOME not found in your environment.
echo Please, set the JAVA_HOME variable in your environment to match the
echo location of the Java Virtual Machine you want to use, like this:
echo   set JAVA_HOME=c:\java\jdk1.3.1

goto end

:jaxb_home_error

echo ERROR: JAXB_HOME not found in your environment.
echo Please, set the JAXB_HOME variable in your environment to match the
echo location of the JAXB installation, like this:
echo   set JAXB_HOME=c:\dev\javajaxb\jaxb-1.0-ea

goto end

:end

set LOCALCLASSPATH=
```

You'll need to set two environment variables before running this batch file: `JAVA_HOME` (to your JDK installation) and `JAXB_HOME` (to your JAXB installation). In other words, use the script like this:

```
Microsoft Windows XP [Version 5.1.2526]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Brett McLaughlin>cd \dev\javajaxb
```

```
C:\dev\javajaxb>set JAVA_HOME=c:\java\jdk1.3.1

C:\dev\javajaxb>set JAXB_HOME=c:\dev\javajaxb\jaxb-1.0

C:\dev\javajaxb>set PATH=%PATH%;%JAXB_HOME%\bin

C:\dev\javajaxb>xjc
```

I've set the two required environment variables and set my PATH to include the binary directory with the *xjc.bat* script. At this point, you're ready to generate some code.

## Supplying Output

Finally, you can get down to the actual fun part. For the sake of these examples, I'm using my Windows system. I'll try to alternate between Windows and the Unix box to give you a sample of both operating systems. Here's my directory layout, so you'll understand how my commands relate to the filesystem I've got set up. Figure 3-2 shows the basic setup, which I'll use for the future chapters as well.

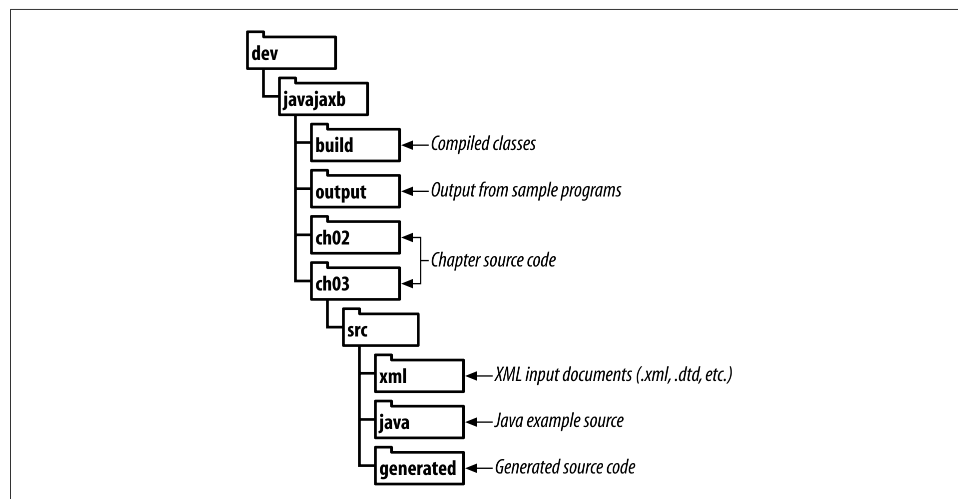



*Figure 3-2. Filesystem layout*

The movie database DTD is saved as *movies.dtd* in the *xml/* directory, and the *movies.xjs* binding schema is stored in the *bindingSchema/* directory. I've created a *generated/* directory in which to put the source code that JAXB generates. Finally, I'm running the schema compiler from the *javajaxb/ch03/* directory. You can execute the schema compiler script like this:

```
C:\dev\javajaxb\ch03\src>xjc xml/movies.dtd bindingSchema/movies.xjs -d generated
Starting JAXB Schema Compiler...
generated\javajaxb\generated\movies\Actor.java
generated\javajaxb\generated\movies\Cast.java
generated\javajaxb\generated\movies\Movie.java
generated\javajaxb\generated\movies\Movies.java
```

As you can see, the output is almost disappointing after all the work it took to get it going. Each element in the XML document resulted in a single, generated Java source file. Additionally, the package supplied in the binding schema is used to determine the directory in which to place the source files, as well as the package declaration for the source files.

If you change into the *generated/* directory and look at the source files, you'll see that they are pretty complex. In addition to the methods you would expect (`getMovie()`, `setTitle()`, etc.), you'll see several other methods, like `validate()`, `marshal()`, and `unmarshal()`. I'll look at these methods more closely in the next two chapters on marshalling and unmarshalling, so don't worry about them now. Before getting to that discussion, though, you need to verify your output and make sure it's ready for use.

## Verifying Output

If you're expecting a lot of manual inspection, use of tools, and other fancy inspection instructions, I'm happy to report that you're wrong. To ensure that the generated classes work, all you need to do is ensure that they compile:

```
C:\dev\javajaxb\ch03\src>cd generated

C:\dev\javajaxb>javac -d build ch03\src\generated\javajaxb\generated\*.java
```

They all do, and you can verify that the classes were created with a simple directory listing:

```
C:\dev\javajaxb>dir build\javajaxb\generated\movies
 Volume in drive C has no label.
 Volume Serial Number is 3050-C7C5

 Directory of C:\dev\javajaxb\build\javajaxb\generated\movies

11/07/2001  09:54a      <DIR>          .
11/07/2001  09:54a      <DIR>          ..
11/07/2001  09:54a               5,202 Actor.class
11/07/2001  09:54a                 187 Cast$1.class
11/07/2001  09:54a               1,290 Cast$ActorPredicate.class
11/07/2001  09:54a               4,789 Cast.class
11/07/2001  09:54a                 190 Movie$1.class
11/07/2001  09:54a               1,256 Movie$ProducerPredicate.class
11/07/2001  09:54a               6,686 Movie.class
11/07/2001  09:54a                 193 Movies$1.class
11/07/2001  09:54a               1,300 Movies$MoviePredicate.class
11/07/2001  09:54a               5,580 Movies.class
              10 File(s)         26,673 bytes
               2 Dir(s)   8,806,182,912 bytes free
```

These commands are similar for Unix users. You'll see several classes that resulted, and since things compiled, the JAXB schema compiler obviously did its job. Next, you can add these commands to your classpath and use them in an application.

I realize that you may have expected more; it took quite a few pages to get to the point of schema compilation and then only about a paragraph to make something happen. That's the beauty of data binding; the actual class generation is generally a piece of cake. In the next chapter, I'll show you how to use these classes, converting XML to Java, using the Java objects, and working with the data in an application. For now, just make sure that your classes are all in place, and get ready for some actual action.

Before moving on, you also should take some time to perform this process on your own XML constraints. If you've got DTDs that you are using, or want to use, for data binding, I highly recommend playing around with them. There's simply no substitute for good old trial and error. Once you feel comfortable with the schema compiler and the various global options for binding schemas, you're ready to go on to the next chapter.