Beginning Hibernate

From Novice to Professional

Dave Minter and
Jeff Linwood

Apress®

Beginning Hibernate: From Novice to Professional

Copyright © 2006 by Dave Minter, Jeff Linwood

ISBN-13 (pbk): 978-1-59059-693-7

ISBN-10 (pbk): 1-59059-693-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Sumit Pal
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Senior Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Damon Larson
Assistant Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: April Eddy
Indexer: Michael Brinkman
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 3

Building a Simple Application

In this chapter, you'll take another look at some of the steps necessary to get the example from Chapter 1 up and running. You'll also build a somewhat larger application from scratch. All of the code in this book is available for download from the Apress site (`www.apress.com`).

Installing the Tools

To run the examples in this chapter, you will need to install a number of tools. You will require a JDK, the Hibernate and Hibernate Tools distributions, the Ant build tool, and the HSQLDB database. Table 3-1 lists the specific tools you will need and where you can find them.

Table 3-1. *The Tools Used in This Book*

Tool	Version	Download Location
Hibernate	3.2.0	`http://hibernate.org`
Hibernate Tools	3.1	`http://hibernate.org`
Ant	1.6.5	`http://ant.apache.org`
HSQLDB	1.8.0.2	`http://hsqldb.org`

Hibernate and Hibernate Tools

The latest version of Hibernate is always available from `http://hibernate.org`, under the left-hand menu link named "Download." Various older versions and additional libraries are available from the resulting page, but you should select Hibernate Core 3.2.0 or a later version. At the time of writing, this is still a release-candidate version, but we expect the final release to be available by the time you read this book—if it is not, and you don't want to use a pre-release version, then most of the examples will work equally well with the previous 3.1.0 release of the Hibernate core. Download the archive and unpack it to a local directory. The unpacked archive contains all the source code for Hibernate itself, a JAR library built from this source, and all the library files that are necessary to run the sample.

You should then download Hibernate Tools from the same site. At the time of writing, it is currently at version 3.1 (again, this is currently in a late beta release, but we recommend using the beta version, rather that its inferior predecessors, if a final 3.1 version has not been released yet). Hibernate Tools provides various plug-ins for the Ant build tool and the free Eclipse IDE. In this chapter, we make use of the Ant plug-ins only, but we discuss the Eclipse features in Appendix B. Again, the archive should be downloaded and unpacked to a local directory. This archive does not include the source code (which is available elsewhere on the `www.hibernate.org` site, if you decide to take a look at it).

HSQLDB 1.8.0

The database we will be using in our examples is the HSQL database. This is written in Java and is freely available open source software. While we used version 1.8.0.2 for our examples, we expect that any later version will be suitable. HSQL is derived from code originally released as "Hypersonic." You may encounter the term in some of the HSQL documentation and should treat it as synonymous with "HSQL." We may also refer to the product as HSQLDB when it might otherwise be mistaken for Hibernate Query Language (HQL), whose acronym is distressingly similar!

Our examples are tailored to HSQL because HSQL will run on any of the platforms that Hibernate will run on, and because HSQL is freely available with minimal installation requirements. However, if you want to run the examples with your own database, then the differences should boil down to the following:

- The Hibernate dialect class
- The JDBC driver
- The connection URL for the database
- The username for the database
- The password for the database

You will see where these can be specified later in this chapter. You will notice that where we specify the URL for connection to the database, we often append a `shutdown=true` attribute. This fixes a minor problem in which HSQLDB does not write its changes to disk until a `Connection` object is closed (something that may never happen when a connection is being managed by Hibernate's own connection pooling logic). This is not necessary on non-embedded databases.

Ant 1.6.5

You will want to install the Ant build tool. We will not attempt to explain the `build.xml` format in detail; if you are familiar with Ant, then the example build script provided in this chapter will be enough to get you started—if not, then Ant is a topic in its own right. We would recommend *Enterprise Java Development on a Budget*, by Christopher M. Judd and Brian Sam-Bodden (Apress, 2004), for good coverage of open source tools such as Ant.

While Ant in general lies outside the scope of this book, we will discuss the use of the Hibernate tasks used by our scripts.

Listing 3-1 provides the Ant script to build the example for this chapter.

Listing 3-1. *An Ant Script to Build the Chapter 3 Examples*

```
<project name="sample">

   <property file="build.properties"/>

   <property name="src" location="src"/>
   <property name="bin" location="bin"/>
   <property name="sql" location="sql"/>
   <property name="hibernate.tools"
      value="${hibernate.tools.home}${hibernate.tools.path}"/>

   <path id="classpath.base">
      <pathelement location="${src}"/>
      <pathelement location="${bin}"/>
      <pathelement location="${hibernate.home}/hibernate3.jar"/>
      <fileset dir="${hibernate.home}/lib" includes="**/*.jar"/>
      <pathelement location="${hsql.home}/lib/hsqldb.jar"/>
   </path>

   <path id="classpath.tools">
      <path refid="classpath.base"/>
      <pathelement
         location="${hibernate.tools}/hibernate-tools.jar"/>
   </path>

   <taskdef name="htools"
      classname="org.hibernate.tool.ant.HibernateToolTask"
      classpathref="classpath.tools"/>

   <target name="exportDDL" depends="compile">
      <htools destdir="${sql}">
         <classpath refid="classpath.tools"/>
         <configuration
            configurationfile="${src}/hibernate.cfg.xml"/>
         <hbm2ddl drop="true" outputfilename="sample.sql"/>
      </htools>
   </target>

   <target name="compile">
      <javac srcdir="${src}" destdir="${bin}" classpathref="classpath.base"/>
   </target>

   <target name="populateMessages" depends="compile">
      <java classname="sample.PopulateMessages" classpathref="classpath.base"/>
   </target>
```

```
<target name="listMessages" depends="compile">
   <java classname="sample.ListMessages" classpathref="classpath.base"/>
</target>

<target name="createUsers" depends="compile">
   <java classname="sample.CreateUser" classpathref="classpath.base">
      <arg value="dave"/>
      <arg value="dodgy"/>
   </java>
   <java classname="sample.CreateUser" classpathref="classpath.base">
      <arg value="jeff"/>
      <arg value="jammy"/>
   </java>
</target>

<target name="createCategories" depends="compile">
   <java classname="sample.CreateCategory" classpathref="classpath.base">
      <arg value="retro"/>
   </java>
   <java classname="sample.CreateCategory" classpathref="classpath.base">
      <arg value="kitsch"/>
   </java>
</target>

<target name="postAdverts" depends="compile">
   <java classname="sample.PostAdvert" classpathref="classpath.base">
      <arg value="dave"/>
      <arg value="retro"/>
      <arg value="Sinclair Spectrum for sale"/>
      <arg value="48k original box and packaging"/>
   </java>
   <java classname="sample.PostAdvert" classpathref="classpath.base">
      <arg value="dave"/>
      <arg value="kitsch"/>
      <arg value="Commemorative Plates"/>
      <arg value="Kitten and puppies design"/>
   </java>
   <java classname="sample.PostAdvert" classpathref="classpath.base">
      <arg value="jeff"/>
      <arg value="retro"/>
      <arg value="Atari 2600 wanted"/>
      <arg value="Must have original joysticks."/>
   </java>
```

```
      <java classname="sample.PostAdvert" classpathref="classpath.base">
         <arg value="jeff"/>
         <arg value="kitsch"/>
         <arg value="Inflatable Sofa"/>
         <arg value="Leopard skin pattern. Nice."/>
      </java>
   </target>

   <target name="listAdverts" depends="compile">
      <java classname="sample.ListAdverts" classpathref="classpath.base"/>
   </target>

</project>
```

The properties file imported in the first line provides the paths to your installed libraries, and you should adjust it as appropriate (as shown in Listing 3-2). If you unpack Hibernate 3.2.0, it will create a directory called `hibernate-3.2`, which we have renamed to the full version path; we have done something similar with the HSQL database directory.

The Hibernate Tools archive currently unpacks to two directories (`plugins` and `features`). We have created a parent directory to contain these. The path to the appropriate JAR file (`hibernate-tools.jar`) within the unpacked directory is dependent upon the specific Hibernate Tools version, so we have added the `hibernate.tools.path` property to point our build script at this.

Listing 3-2. *The* `build.properties` *File to Configure the Ant Script*

```
# Path to the hibernate install directory
hibernate.home=/hibernate/hibernate-3.2.0

# Path to the hibernate-tools install directory
hibernate.tools.home=/hibernate/hibernate-tools-3.1

# Path to hibernate-tools.jar relative to hibernate.tools.home
hibernate.tools.path=/plugins/org.hibernate.eclipse_3.1.0/lib/tools

# Path to the HSQL DB install directory
hsql.home=/hsqldb/hsqldb-1.8.0.2
```

Aside from the configuration settings, the only oddity in the `build.xml` file is the configuration and use of a Hibernate-specific Ant task. The `taskdef` (shown in Listing 3-3) makes this task available for use, using the appropriate classes from the `tools.jar` file.

Listing 3-3. *Defining the Hibernate Tools Ant Tasks*

```
<taskdef name="htools"
      classname="org.hibernate.tool.ant.HibernateToolTask"
      classpathref="classpath.tools"/>
```

This task provides several subtasks, but in this chapter we will only make use of the `hbm2ddl` subtask. This reads in the mapping and configuration files and generates Data Definition Language (DDL) scripts to create an appropriate schema in the database to represent our entities.

Table 3-2 shows the basic directories that our build script assumes, relative to the example project's root.

Table 3-2. *The Project Directories*

Directory	Contents
`src`	Source code and configuration files (excluding those directly related to the build)
`bin`	Compiled class files
`sql`	Generated DDL scripts

The root of the project contains the build script and build configuration file; it will also contain the database files generated by HSQL when the `exportDDL` task is run.

The Ant Tasks

Table 3-3 shows the tasks contained in the Ant build script.

Table 3-3. *The Tasks Available in the Example Ant Script*

Task	Action
`exportDDL`	Creates the appropriate database objects. It also generates a script that can be run against an HSQL database to re-create these objects if necessary.
`compile`	Builds the class files. This task is a dependency of all the tasks except `exportDDL` (which does not require the class files), so it is not necessary to invoke it directly.
`populateMessages`	Populates the database with a sample message.
`listMessages`	Lists all messages stored in the database by `populateMessages`.
`createUsers`	Creates a pair of users in the database for the Advert example.
`createCategories`	Creates a pair of categories in the database for the Advert example.
`postAdverts`	Creates several adverts in the database for the Advert example.
`listAdverts`	Lists the adverts in the database for the Advert example.

Enabling Logging

Before going on to run any of the examples in this chapter, you will want to create a `log4j.properties` file in the classpath. A suitable example is provided with the Hibernate tools in the `etc` directory of the unpacked archive.

Our example includes this file in the `src` directory of our project and places that directory itself on the classpath. In some circumstances—such as when building a JAR file for inclusion in other projects—it may be better to copy the appropriate properties file(s) into the target directory with the class files.

Creating a Hibernate Configuration File

There are several ways that Hibernate can be given all of the information that it needs to connect to the database and determine its mappings. For our Message example, we used the configuration file `hibernate.cfg.xml` placed in our project's `src` directory and given in Listing 3-4.

Listing 3-4. *The Message Application's Mapping File*

```
<?xml version='1.0' encoding='utf-8'?>
  <session-factory>
    <property name="hibernate.connection.url">
       jdbc:hsqldb:file:testdb;shutdown=true
    </property>
    <property name="hibernate.connection.driver_class">
       org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">0</property>
    <property name="hibernate.dialect">
       org.hibernate.dialect.HSQLDialect
    </property>
    <property name="hibernate.show_sql">false</property>

    <!-- "Import" the mapping resources here -->
    <mapping resource="sample/entity/Message.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

The various database-related fields (`hibernate.connection.*`) should look pretty familiar from setting up JDBC connections, with the exception of the `hibernate.connection.pool` property, which is used to disable a feature (connection pooling) that causes problems when using the HSQL database. The `show_sql` value, set to `false` in our example, is extremely useful when debugging problems with your programs—when set to `true`, all of the SQL prepared by Hibernate is logged to the standard output stream (i.e., the console).

The SQL dialects, discussed in Chapter 2, allow you to select the database type that Hibernate will be talking to. You must select a dialect, even if it is `GenericDialect`—most database platforms accept a common subset of SQL, but there are inconsistencies and extensions specific to each. Hibernate uses the dialect class to determine the appropriate SQL to use when creating and querying the database. If you elect to use `GenericDialect`, then Hibernate will only be able to use a common subset of SQL to perform its operations, and will be unable to take advantage of various database-specific features to improve performance.

■ **Caution** Hibernate looks in the classpath for the configuration file. If you place it anywhere else, Hibernate will complain that you haven't provided necessary configuration details.

Hibernate does not require you to use an XML configuration file. You have two other options. First, you can provide a normal Java properties file. The equivalent properties file to Listing 3-4 would be as follows:

```
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc:hsqldb:file:testdb;shutdown=true
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.connection.pool_size=0
hibernate.show_sql=false
hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

As you'll notice, this does not contain the resource mapping from the XML file—and in fact, you cannot include this information in a properties file; if you want to configure Hibernate this way, you'll need to directly map your classes into the Hibernate `Configuration` at run time. Here's how this can be done:

```
Configuration config = new Configuration();
config.addClass( sample.entity.Message.class );
config.setProperties( System.getProperties() );
SessionFactory sessions = config.buildSessionFactory();
```

Note that the `Configuration` object will look in the classpath for a mapping file *in the same package* as the class it has been passed. So, in this example, where the fully qualified name of the class is `sample.entity.Message`, you should see the following pair of files from the root of the classpath:

```
/sample/entity/Message.class
/sample/entity/Message.hbm.xml
```

Here, `Message.class` is the compiled output from the `Message.java` code given in Listing 3-5 (and briefly discussed in Chapter 1), and `Message.hbm.xml` is the XML mapping file provided in Chapter 1 as Listing 1-5. If for some reason you want to keep your mapping files in a different directory, you can alternatively provide them as resources like this (note that this resource path must still be relative to the classpath):

```
Configuration config = new Configuration();
config.addResource( "Message.hbm.xml" );
config.setProperties( System.getProperties() );
SessionFactory sessions = config.buildSessionFactory();
```

You may have as many or as few mapping files as you wish, given any names you like—however, it is conventional to have one mapping file for each class that you are mapping, placed in the same directory as the class itself, and named similarly (for example, `Message.hbm.xml` in the default package to map the `Message` class also in the default package). This allows you to find any given class mapping quickly, and keeps the mapping files easily readable.

If you don't want to provide the configuration properties in a file, you can apply them directly using the `-D` flag. Here's an example:

```
java -classpath ...
   -Dhibernate.connection.driver_class=org.hsqldb.jdbcDriver
   -Dhibernate.connection.url= jdbc:hsqldb:file:testdb;shutdown=true
   -Dhibernate.connection.username=sa
   -Dhibernate.connection.password=
   -Dhibernate.connection.pool_size=0
   -Dhibernate.show_sql=false
   -Dhibernate.dialect=org.hibernate.dialect.HSQLDialect
    ...
```

Given its verbosity, this is probably the least convenient of the three methods, but it is occasionally useful when running tools and utilities on an ad hoc basis. For most other purposes, we think that the XML configuration file is the best choice.

Running the Message Example

With Hibernate and a database installed, and our configuration file created, all we need to do now is create the classes in full, and then build and run everything. Chapter 1 omitted the trivial parts of the required classes, so we provide them in full in Listings 3-5 through 3-7, after which we'll look at some of the details of what's being invoked.

Listing 3-5. *The* `Message` *POJO Class*

```
package sample.entity;

public class Message {
   private String message;

   public Message(String message) {
      this.message = message;
   }

   Message() {
   }

   public String getMessage() {
      return this.message;
   }

   public void setMessage(String message) {
      this.message = message;
   }
}
```

Listing 3-6 shows a simple application to populate the messages table with examples.

Listing 3-6. *The Code to Create a Sample Message*

```
package sample;

import java.util.Date;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import sample.entity.Message;

public class PopulateMessages {
   public static void main(String[] args) {
         SessionFactory factory =
            new Configuration().configure().buildSessionFactory();
         Session session = factory.openSession();
         session.beginTransaction();

         Message m1 = new Message(
            "Hibernated a message on " + new Date());

         session.save(m1);
         session.getTransaction().commit();
         session.close();
   }
}
```

Finally, Listing 3-7 shows the full text of the application to list all the messages in the database.

Listing 3-7. *The Message Application*

```
package sample;

import java.util.Iterator;
import java.util.List;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import sample.entity.Message;
```

```
public class ListMessages {
   public static void main(String[] args)
   {
      SessionFactory factory =
         new Configuration().configure().buildSessionFactory();
      Session session = factory.openSession();

      List messages = session.createQuery("from Message").list();
      System.out.println("Found " + messages.size() + " message(s):");

      Iterator i = messages.iterator();
      while(i.hasNext()) {
         Message msg = (Message)i.next();
         System.out.println(msg.getMessage());
      }

      session.close();
   }
}
```

The Ant target `exportDDL` will create an appropriate schema in the HSQLDB database files. Running the `populateMessages` task will create a message entry (this can be invoked multiple times). Running the `listMessages` task will list the messages that have been entered into the database so far.

Caution Because we have selected the `drop="true"` option for the `hbm2ddl` subtask of our `exportDDL` target, running this script will effectively delete any data in the named tables. It is rarely a good idea to run such a script from a machine that has database access to the production environment because of the risk of accidentally deleting your production data!

The appropriate classpath entries have been set up in the Ant build script. To run a Hibernate application, you need the `hibernate.jar` file from the root of the Hibernate distribution, and a subset of the libraries provided in the `lib` subdirectory. The origin, purpose, and optionality of each of these libraries is explained in a `README` text file provided in the `lib` directory.

Most of the work required to get this example running is the sort of basic configuration trivia that any application requires (writing Ant scripts, setting classpaths, and so on). The real work consists of these steps:

1. Creating the Hibernate configuration file
2. Creating the mapping file
3. Writing the POJOs (introduced in Chapter 1)

Persisting Multiple Objects

Our example in Chapter 1 was as simple a persistence scenario as you can imagine. In the next few sections of this chapter, we will look at a slightly more complicated scenario.

Our example application will provide the persistence technology for an online billboard application, as shown in Figure 3-1.

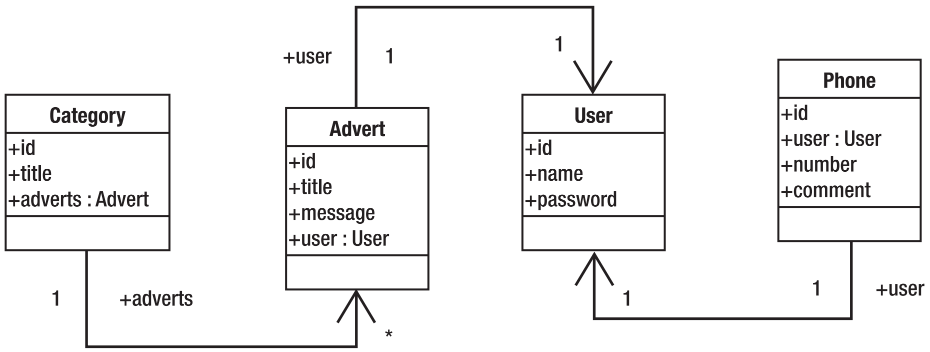

Figure 3-1. *The online billboard classes*

This is a gross simplification of the sort of classes that would be required in a production application. For example, we make no distinction between the roles of users of the application, but it should suffice to show some of the simpler relationships between classes.

Particularly interesting is the many-to-many relationship between categories and advertisements. We would like to be able to have multiple categories and adverts, and place any given advert in more than one category. For example, an electric piano should be listed in the "Instruments" category as well as the "Electronics" category.

Creating Persistence Classes

We will begin by creating the POJOs for the application. This is not strictly necessary in a new application, as they can be generated directly from the mapping files, but since this will be familiar territory, it should help to provide some context for our subsequent creation of the mapping files.

From the class diagram, we know that three classes will be persisted into the database (see Listings 3-8, 3-9, and 3-10). Each class that will be persisted by Hibernate is required to have a default constructor with at least package scope. They should have get and set methods for all of the attributes that are to be persisted. We will provide each with an `id` field, allowing this to be the primary key in our database (we prefer the use of surrogate keys, as changes to business rules can make the use of direct keys risky).

Note A surrogate key is an arbitrary value (usually numeric), with the data type depending on the number of objects expected (e.g., 32-bit, 64-bit, etc.). The surrogate key has no meaning outside the database—it is not a customer number, a phone number, or anything else. As such, if a business decision causes previously unique business data to be duplicated, this will not cause problems since the business data does not form the primary key.

As well as the default constructor for each class, we provide a constructor that allows the fields other than the primary key to be assigned directly. This allows us to create and populate an object in one step instead of several, but we let Hibernate take care of the allocation of our primary keys.

The classes shown in Figure 3-1 are our POJOs. Their implementation is shown in Listings 3-8, 3-9, and 3-10.

Listing 3-8. *The Class Representing Users*

```
package sample.entity;

public class User {
   private long id;
   private String name;
   private String password;

   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }

   User() {
   }

   public String getName() {
      return name;
   }

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

   public String getPassword() {
      return password;
   }
   public void setPassword(String password) {
      this.password = password;
   }

   protected long getId() {
      return id;
   }

   protected void setId(long id) {
      this.id = id;
   }
}
```

Listing 3-9. *The Class Representing Categories (Each Having an Associated Set of* `Advert` *Objects)*

```
package sample.entity;

import java.util.HashSet;
import java.util.Set;

public class Category {
   private long id;
   private String title;
   private Set adverts = new HashSet();

   public Category(String title) {
      this.title = title;
      this.adverts = new HashSet();
   }

   Category() {
   }

   public Set getAdverts() {
      return adverts;
   }

   void setAdverts(Set adverts) {
      this.adverts = adverts;
   }

   public void addAdvert(Advert advert) {
      getAdverts().add(advert);
   }
   public String getTitle() {
      return title;
   }

   public void setTitle(String title) {
      this.title = title;
   }

   protected long getId() {
      return id;
   }

   protected void setId(long id) {
      this.id = id;
   }
}
```

Listing 3-10. *The Class Representing Adverts (Each Instance Has an Associated User Who Placed the Advert)*

```
package sample.entity;

public class Advert {
   private long id;
   private String title;
   private String message;
   private User user;

   public Advert(String title, String message, User user) {
      this.title = title;
      this.message = message;
      this.user = user;
   }

   Advert() {
   }

   public String getMessage() {
      return message;
   }

   public void setMessage(String message) {
      this.message = message;
   }
   public String getTitle() {
      return title;
   }

   public void setTitle(String title) {
      this.title = title;
   }

   public User getUser() {
      return user;
   }

   public void setUser(User user) {
      this.user = user;
   }

   protected long getId() {
      return id;
   }

   protected void setId(long id) {
      this.id = id;
   }
}
```

We have not had to add any unusual features to these classes in order to support the Hibernate tool, though we have chosen to provide package-scoped default constructors to support use of the (optional) lazy-loading feature of Hibernate. Most existing applications will contain POJOs "out of the box" that are compatible with Hibernate.

Creating the Object Mappings

Now that we have our POJOs, we need to map them to the database, representing the fields of each directly or indirectly as values in the columns of the associated tables. We take each in turn.

The fully qualified name of the type that we are mapping is specified, and the table in which we would like to store it is specified (we used `aduser` because `user` is a keyword in many databases).

The class has three fields, as follows:

The `id` *field*: Corresponds to the surrogate key to be used in, and generated by, the database. This special field is handled by the `<id>` element. The name of the field is specified by the `name` attribute (so that `name="id"` corresponds as it must with the method name of `"getId"`). It is identified as being of `long` type, and we would like to store its values in the database in the `long` column. We specify that it should be generated by the database, rather than by Hibernate.

The `name` *field*: Represents the name of the user. It should be stored in a column called `name`. It has type `String`. We do not permit duplicate names to be stored in the table.

The `password` *field*: Represents a given user's password. It should be stored in a column called `password`. It has type `String`.

Bearing these features in mind, the mapping file in Listing 3-11 should be extremely easy to follow.

Listing 3-11. *The Mapping of the* `User` *Class into the Database*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="com.hibernatebook.chapter3.User" table="aduser">

      <id name="id" type="long" column="id">
         <generator class="native"/>
      </id>

      <property name="name" column="name" type="string" unique="true"/>

      <property name="password" column="password" type="string"/>

   </class>
</hibernate-mapping>
```

The `Category` mapping presents another type of relationship: many-to-many. Each `Category` object is associated with a set of adverts, while any given advert can be associated with multiple categories.

The `<set>` element indicates that the field in question has a `java.util.Set` type with the name `adverts`. This sort of relationship requires the creation of an additional link table, so we specify the name of the table containing that information.

We state that the primary key (used to retrieve items) for the objects contained in the `link` table is represented by the `id` column, and provide the fully qualified name of the class type contained in the table. We specify the column in the `link` table representing the adverts associated with each category.

Again, this is complicated when described, but if you look at the example table from Listing 3-14, the need for each field in the mapping becomes clear (see Listing 3-12).

Listing 3-12. *The Mapping of the* `Category` *Class into the Database*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
   <class name="sample.entity.Category" table="category">

      <id name="id" type="long" column="id">
         <generator class="native"/>
      </id>

      <property
            name="title"
            column="title"
            type="string"
            unique="true"/>

      <set name="adverts" table="link_category_advert" >
         <key column="category" foreign-key="fk_advert_category"/>
         <many-to-many class="sample.entity.Advert"
            column="advert"
            foreign-key="fk_category_advert"/>
      </set>

   </class>
</hibernate-mapping>
```

Finally, we represent the `Advert` class (see Listing 3-13). This class introduces the many-to-one association, in this case with the `User` class. Any given advertisement must belong to a single user, but any given user can place many different advertisements.

Listing 3-13. *The Mapping of the* `Advert` *Class into the Database*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="sample.entity.Advert" table="advert">

      <id name="id" type="long" column="id">
         <generator class="native"/>
      </id>

      <property name="message" column="message" type="string"/>
      <property name="title" column="title" type="string"/>
      <many-to-one
         name="user"
         column="aduser"
         class="sample.entity.User"
         not-null="true"
         foreign-key="fk_advert_user"/>

   </class>
</hibernate-mapping>
```

Once you have created the individual mapping files, you need to tell Hibernate where to find them. If you're using a Hibernate configuration file, as in the Chapter 1 example, the simplest thing to do is include links to the mapping files directly within this.

For our example, take the configuration file described for Chapter 1 (Listing 1-5) and add the following three mapping resource entries:

```
<mapping resource="sample/entity/Advert.hbm.xml"/>
<mapping resource="sample/entity/Category.hbm.xml"/>
<mapping resource="sample/entity/User.hbm.xml"/>
```

after the following line:

```
<mapping resource="sample/entity/Message.hbm.xml"/>
```

This section may seem confusing, as it is something of a flying visit to the subject of mappings and some of their whys and wherefores. We provide a more in-depth discussion of mapping in later chapters—specifically, general mapping concepts in Chapter 5, and XML-based mapping files in Chapter 7. We also discuss how you can use the new Java 5 Annotations features to represent mappings directly in your source code in Chapter 6.

Creating the Tables

With the object mapping in place and our Hibernate configuration file set up correctly, we have everything we need to generate a script to create the database for our application by invoking the `exportDDL` task. This builds the entities shown in Figure 3-2.

Even though we can generate the database directly, we also recommend taking some time to work out what schema you would expect your mappings to generate. This allows you to "sanity check" the script to make sure it corresponds with your expectations. If you and the tool both agree on what things should look like, then all is well and good; if not, your mappings may be wrong or there may be a subtle error in the way that you have related your data types.

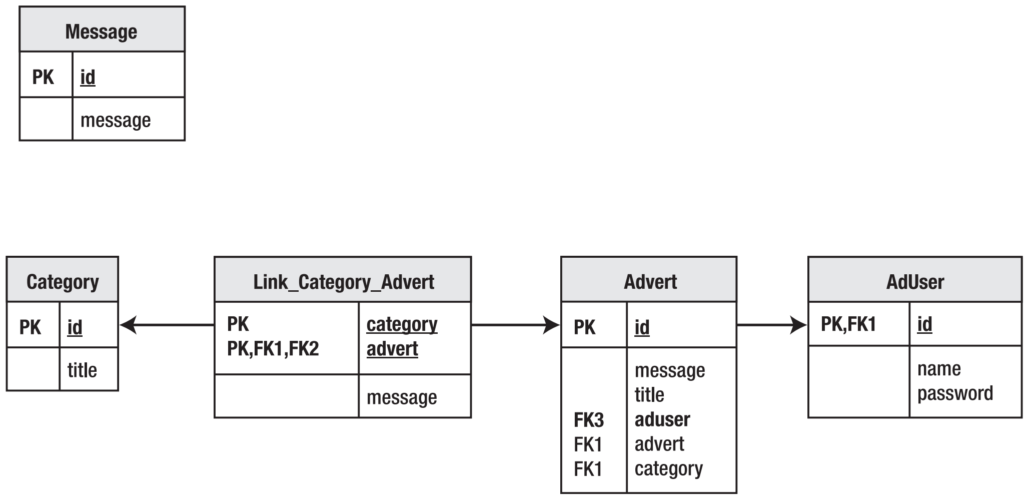

Figure 3-2. *The database entity relationships*

The script in Listing 3-14 is generated by the `exportDDL` task. It could easily have been written by hand, and it is easy to compare it against your prior expectations of the database schema (we have changed the formatting slightly, but otherwise this is identical to the output of the task).

Listing 3-14. *The Script Generated by the* `exportDDL` *Task*

```
alter table advert
   drop constraint fk_advert_user;

alter table link_category_advert
   drop constraint fk_advert_category;

alter table link_category_advert
   drop constraint fk_category_advert;

drop table aduser if exists;
drop table advert if exists;
drop table category if exists;
drop table link_category_advert if exists;
drop table message if exists;

create table aduser (
   id bigint generated by default as identity (start with 1),
   name varchar(255),
   password varchar(255),
   primary key (id),
   unique (name));
```

```
create table advert (
   id bigint generated by default as identity (start with 1),
   message varchar(255),
   title varchar(255),
   aduser bigint not null,
   primary key (id));

create table category (
   id bigint generated by default as identity (start with 1),
   title varchar(255),
   primary key (id),
   unique (title));

create table link_category_advert (
   category bigint not null,
   advert bigint not null,
   primary key (category, advert));

create table message (
   id bigint generated by default as identity (start with 1),
   message varchar(255),
   primary key (id));

alter table advert
   add constraint fk_advert_user
   foreign key (aduser) references aduser;

alter table link_category_advert
   add constraint fk_advert_category
   foreign key (category) references category;

alter table link_category_advert
   add constraint fk_category_advert
   foreign key (advert) references advert;
```

Note the foreign key constraints and the link table representing the many-to-many relationship.

Sessions

Chapter 4 will discuss the full life cycle of persistence objects in detail—but you need to understand the basics of the relationship between the session and the persistence objects if you are to build even a trivial application in Hibernate.

The Session and Related Objects

The session is always created from a SessionFactory. The SessionFactory is a heavyweight object, and there would normally be a single instance per application. In some ways, it is a little like a connection pool in a connected application. In a J2EE application, it would typically be retrieved as a JNDI resource. It is created from a Configuration object, which in turn acquires the Hibernate configuration information and uses this to generate an appropriate SessionFactory instance.

The session itself has a certain amount in common with a JDBC Connection object. To read an object from the database, you must use a session directly or indirectly. An example of a direct use of the session to do this would be, as in Chapter 1, calling the session.get() method, or creating a Query object from the session (a Query is very much like a PreparedStatement).

An indirect use of the session would be using an object itself associated with the session. For example, if we have retrieved a Phone object from the database using a session directly, we can retrieve a User object by calling Phone's getUser() method, even if the associated User object has not yet been loaded (as a result of lazy loading).

An object that has not been loaded via the session can be explicitly associated with the session in several ways, the simplest of which is to call the session.update() method passing in the object in question.

The session does a lot more than this, however, as it provides some caching functionality, manages the lazy loading of objects, and watches for changes to associated objects (so that the changes can be persisted to the database).

A Hibernate transaction is typically used in much the same way as a JDBC transaction. It is used to batch together mutually dependent Hibernate operations, allowing them to be completed or rolled back atomically, and to isolate operations from external changes to the database. Hibernate can also take advantage of a transaction's scope to limit unnecessary JDBC "chatter," queuing SQL to be transmitted in a batch at the end of the transaction when possible.

We will discuss all of this in much greater detail in Chapter 4, but for now it suffices that we need to maintain a single SessionFactory for the entire application. However, a session should only be accessed within a single thread of execution. Because a session also represents information cached from the database, it is desirable to retain it for use within the thread until anything (specifically any Hibernate exception) causes it to become invalid.

We present in Listing 3-15 a pattern from which Data Access Objects (DAOs) can be derived, providing an efficient way for a thread to retrieve and (if necessary) create its sessions with a minimal impact on the clarity of the code.

Listing 3-15. *The Base Class Used to Manage the Session in the Example*

```
package sample.dao;

import java.util.logging.Level;
import java.util.logging.Logger;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class DAO {

   protected DAO() {
   }

   public static Session getSession() {
      Session session = (Session) DAO.session.get();
      if (session == null) {
         session = sessionFactory.openSession();
         DAO.session.set(session);
      }
      return session;
   }

   protected void begin() {
      getSession().beginTransaction();
   }

   protected void commit() {
      getSession().getTransaction().commit();
   }

   protected void rollback() {
      try {
      getSession().getTransaction().rollback();
      } catch( HibernateException e ) {
         log.log(Level.WARNING,"Cannot rollback",e);
      }

      try {
         getSession().close();
      } catch( HibernateException e ) {
         log.log(Level.WARNING,"Cannot close",e);
      }
      DAO.session.set(null);
   }

   public static void close() {
      getSession().close();
      DAO.session.set(null);
   }
   private static final Logger log = Logger.getAnonymousLogger();

   private static final ThreadLocal session = new ThreadLocal();

   private static final SessionFactory sessionFactory =
      new Configuration().configure().buildSessionFactory();
}
```

Using the Session

The most common use cases for our POJOs will be to create them and delete them. In both cases, we want the change to be reflected in the database.

For example, we want to be able to create a user, specifying the username and password, and have this information stored in the database when we are done.

The logic to create a user (and reflect this in the database) is incredibly simple, as shown in Listing 3-16.

Listing 3-16. *Creating a* `User` *Object and Reflecting This in the Database*

```
try {
   begin();
   User user = new User(username,password);
   getSession().save(user);
   commit();
   return user;
} catch( HibernateException e ) {
   rollback();
   throw new AdException("Could not create user " + username,e);
}
```

We begin a transaction, create the new `User` object, ask the session to save the object, and then commit the transaction. If a problem is encountered (if, for example, a `User` entity with that username has already been created in the database), then a Hibernate exception will be thrown, and the entire transaction will be rolled back.

To retrieve the `User` object from the database, we will make our first excursion into HQL. HQL is somewhat similar to SQL, but you should bear in mind that it refers to the names used in the mapping files, rather than the table names and columns of the underlying database.

The appropriate HQL query to retrieve the users having a given name field is as follows:

```
from User where name= :username
```

where `User` is the class name and `:username` is the HQL named parameter that our code will populate when we carry out the query. This is remarkably similar to the SQL for a prepared statement to achieve the same end:

```
select * from user where name = ?
```

The complete code to retrieve a user for a specific username is shown in Listing 3-17.

Listing 3-17. *Retrieving a* `User` *Object from the Database*

```
try {
   begin();
   Query q = getSession().createQuery("from User where name = :username");
   q.setString("username",username);
   User user = (User)q.uniqueResult();
   commit();
   return user;
} catch( HibernateException e ) {
   rollback();
   throw new AdException("Could not get user " + username,e);
}
```

We begin a transaction, create a `Query` object (similar in purpose to `PreparedStatement` in connected applications), populate the parameter of the query with the appropriate username, and then list the results of the query. We extract the user (if one has been retrieved successfully) and commit the transaction. If there is a problem reading the data, the transaction will be rolled back.

The key line used to obtain the `User` entity is:

```
User user = (User)q.uniqueResult();
```

We use the `uniqueResult()`method because it is guaranteed to throw an exception if somehow our query identifies more than one `User` object for the given username. In principle, this could happen if the underlying database's constraints don't match our mapping constraint for a unique username field, and an exception is an appropriate way to handle the failure.

The logic to delete a user from the database (Listing 3-18) is even more trivial than that required to create one.

Listing 3-18. *Deleting a* `User` *Object and Reflecting This in the Database*

```
try {
   begin();
   getSession().delete(user);
   commit();
} catch( HibernateException e ) {
   rollback();
   throw new AdException("Could not delete user " + user.getName(),e);
}
```

We simply instruct the session to delete the `User` object from the database, and commit the transaction. The transaction will roll back if there is a problem—for example, if the user has already been deleted.

You have now seen all the basic operations that we want to perform on our data, so we will now take a look at the architecture we are going to use to do this.

Building DAOs

The DAO pattern is well known to most developers. The idea is to separate out the POJOs from the logic used to persist them into, and retrieve them from, the database. The specifics of the implementation vary—at one extreme, they can be provided as interfaces instantiated from a factory class, allowing a completely pluggable database layer. For our example, we have selected a compromise of concrete DAO classes. Each DAO class represents the operations that can be performed on a POJO type.

We have already described the base class `DAO` in Listing 3-15, and the preceding examples made use of this.

To help encapsulate the specifics of the database operations that are being carried out, we catch any `HibernateException` that is thrown and wrap it in a business `AdException` instance, as shown in Listing 3-19.

Listing 3-19. *The* `AdException` *Class for the Example*

```
package sample;

public class AdException extends Exception {
   public AdException(String message) {
      super(message);
   }

   public AdException(String message, Throwable cause) {
      super(message,cause);
   }
}
```

The `UserDAO` provides all the methods required to retrieve an existing `User` object, delete an existing `User` object, or create a new `User` object (see Listing 3-20). Changes to the object in question will be persisted to the database at the end of the transaction.

Listing 3-20. *The* `UserDAO` *Class for the Example*

```
package sample.dao;

import org.hibernate.HibernateException;
import org.hibernate.Query;

import sample.AdException;
import sample.entity.User;

public class UserDAO extends DAO {
   public UserDAO() {
   }
```

```
    public User get(String username)
       throws AdException
    {
       try {
          begin();
          Query q = getSession().createQuery("from User where name = :username");
          q.setString("username",username);
          User user = (User)q.uniqueResult();
          commit();
          return user;
       } catch( HibernateException e ) {
          rollback();
          throw new AdException("Could not get user " + username,e);
       }
    }

    public User create(String username,String password)
       throws AdException
    {
       try {
          begin();
          User user = new User(username,password);
          getSession().save(user);
          commit();
          return user;
       } catch( HibernateException e ) {
          rollback();
          throw new AdException("Could not create user " + username,e);
       }
    }

    public void delete(User user)
       throws AdException
    {
       try {
          begin();
          getSession().delete(user);
          commit();
       } catch( HibernateException e ) {
          rollback();
          throw new AdException("Could not delete user " + user.getName(),e);
       }
    }
 }
```

CategoryDAO provides all the methods required to retrieve all of the Category objects, delete an existing Category object, or create a new Category object (see Listing 3-21). Changes to the object in question will be persisted to the database at the end of the transaction.

Listing 3-21. *The* CategoryDAO *Class for the Example*

```
package sample.dao;

import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;

import sample.AdException;
import sample.entity.Category;

public class CategoryDAO extends DAO {
   public Category get(String title) throws AdException {
      try {
         begin();
         Query q = getSession().createQuery(
               "from Category where title = :title");
         q.setString("title", title);
         Category category = (Category) q.uniqueResult();
         commit();
         return category;
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not obtain the named category " + title, e);
      }
   }

   public List list() throws AdException {
      try {
         begin();
         Query q = getSession().createQuery("from Category");
         List list = q.list();
         commit();
         return list;
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not list the categories", e);
      }
   }
   public Category create(String title) throws AdException {
      try {
         begin();
         Category cat = new Category(title);
         getSession().save(cat);
         commit();
         return null;
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not create the category", e);
      }
   }

   public void save(Category category) throws AdException {
      try {
         begin();
         getSession().update(category);
         commit();
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not save the category", e);
      }
   }

   public void delete(Category category) throws AdException {
      try {
         begin();
         getSession().delete(category);
         commit();
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not delete the category", e);
      }
   }
}
```

`AdvertDAO` provides all the methods required to delete an existing `Advert` object or create a new `Advert` object (adverts are always retrieved by selecting them from a category, and are thus indirectly loaded by the `CategoryDAO` class). Changes to the object in question will be persisted to the database at the end of the transaction (see Listing 3-22).

Listing 3-22. *The* `AdvertDAO` *Class for the Example*

```
package sample.dao;

import org.hibernate.HibernateException;
import sample.AdException;
import sample.entity.Advert;
import sample.entity.User;

public class AdvertDAO extends DAO {
   public Advert create(String title, String message, User user)
         throws AdException {
      try {
         begin();
         Advert advert = new Advert(title, message, user);
         getSession().save(advert);
         commit();
         return advert;
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not create advert", e);
      }
   }

   public void delete(Advert advert)
      throws AdException
   {
      try {
         begin();
         getSession().delete(advert);
         commit();
      } catch (HibernateException e) {
         rollback();
         throw new AdException("Could not delete advert", e);
      }
   }
}
```

If you compare the amount of code required to create our DAO classes here with the amount of code that would be required to implement them using the usual JDBC approach, you will see that Hibernate's logic is admirably compact.

The Example Client

Listing 3-23 shows the example code tying this together. Of course, this isn't a full application, but you now have all the DAOs necessary to manage the advertisement database. This example gives a flavor of how they can be used.

The code should be run with the tasks in the Ant script delivered in Listing 3-1. After running the `exportDDL` task to create the empty database, you should run the `createUsers` and `createCategories` tasks to provide initial users and categories, and then the `postAdverts` task to place advertisements in the database. Finally, run the `listAdverts` task to display the saved data.

The code invoking the DAOs to perform the tasks in question is shown in Listing 3-23.

Listing 3-23. *The Class to Create the Example Users*

```
package sample;

import sample.dao.DAO;
import sample.dao.UserDAO;

public class CreateUser {
   public static void main(String[] args) {

      if (args.length != 2) {
         System.out.println("params required: username, password");
         return;
      }
      String username = args[0];
      String password = args[1];

      try {
         UserDAO userDao = new UserDAO();

         System.out.println("Creating user " + username);
         userDao.create(username, password);
         System.out.println("Created user");
         DAO.close();
      } catch (AdException e) {
         System.out.println(e.getMessage());
      }

   }
}
```

The `CreateUser` class uses the `UserDAO` class to create and persist an appropriate `User` object. The specifics of the (two) users created are drawn from the command-line parameters provided in the `createUsers` Ant task.

In Listing 3-24, we create `Category` objects via the `CategoryDAO` class—and again we draw the specific details from the command line provided by the Ant script.

Listing 3-24. *The Class to Create the Example Categories*

```
package sample;

import sample.dao.CategoryDAO;
import sample.dao.DAO;

public class CreateCategory {
   public static void main(String[] args) {
      if (args.length != 1) {
         System.out.println("param required: categoryTitle");
         return;
      }

      CategoryDAO categories = new CategoryDAO();
      String title = args[0];
      try {
         System.out.println("Creating category " + title);
         categories.create(title);
         System.out.println("Created category");
         DAO.close();
      } catch (AdException e) {
         System.out.println(e.getMessage());
      }

   }
}
```

The code in Listing 3-25 allows us to create an advert for a preexisting user in a preexisting category. Note our use of `UserDAO` and `CategoryDAO` to obtain `User` and `Category` objects from the database. As with the user and category, the advert details are supplied by the Ant task.

Listing 3-25. *The Class to Create the Example Adverts*

```
package sample;

import sample.dao.AdvertDAO;
import sample.dao.CategoryDAO;
import sample.dao.DAO;
import sample.dao.UserDAO;
import sample.entity.Advert;
import sample.entity.Category;
import sample.entity.User;

public class PostAdvert {
   public static void main(String[] args) {

      if (args.length != 4) {
System.out.println("params required: username, categoryTitle, title, message");
         return;
      }
      String username = args[0];
      String categoryTitle = args[1];
      String title = args[2];
      String message = args[3];

      try {
         UserDAO users = new UserDAO();
         CategoryDAO categories = new CategoryDAO();
         AdvertDAO adverts = new AdvertDAO();

         User user = users.get(username);
         Category category = categories.get(categoryTitle);
         Advert advert = adverts.create(title, message, user);

         category.addAdvert(advert);
         categories.save(category);

         DAO.close();
      } catch (AdException e) {
         e.printStackTrace();
      }

   }
}
```

Finally, in Listing 3-26, we make use of `CategoryDAO` to iterate over the categories, and within these, the adverts drawn from the database. It is easy to see how this logic could now be incorporated into a JSP file or servlet. It could even be used from within an EJB session bean.

Listing 3-26. *The Class to Display the Contents of the Database*

```
package sample;

import java.util.Iterator;
import java.util.List;

import sample.dao.CategoryDAO;
import sample.dao.DAO;
import sample.entity.Advert;
import sample.entity.Category;

public class ListAdverts {
   public static void main(String[] args) {
      try {
         List categories = new CategoryDAO().list();
```

```
            Iterator ci = categories.iterator();
            while(ci.hasNext()) {
               Category category = (Category)ci.next();
               System.out.println("Category: " + category.getTitle());
               System.out.println();
               Iterator ai = category.getAdverts().iterator();
               while(ai.hasNext()) {
                  Advert advert = (Advert)ai.next();
                  System.out.println();
                  System.out.println("Title: " + advert.getTitle());
                  System.out.println(advert.getMessage());
                  System.out.println(" posted by " + advert.getUser().getName());
               }
            }

            DAO.close();
         } catch( AdException e ) {
            System.out.println(e.getMessage());
         }


   }
}
```

A large part of the logic here is either output information, or concerned with accessing the collections themselves. Java 5 devotees will see an obvious opportunity to make use of generics and enhanced `for` loops in this example. A quick taste of the simplified version of this code might look like Listing 3-27.

Listing 3-27. *Enhancing Your DAOs with Java 5 Features*

```
List<Category> categories = new CategoryDAO().list();
for( Category category : categories ) {
   // ...
   for( Advert advert : category.getAdverts() ) {
      // ...
   }
}
DAO.close();
```

When you run the example applications, you will see a considerable amount of "chatter" from the logging API, and from the Ant tool when you run these tasks, much of which can be controlled or eliminated in a production application.

You will also notice that because you are starting each of these applications as new tasks (several times in the case of the tasks to create data), the tasks proceed relatively slowly. This is an artifact of the repeated creation of `SessionFactory`—a heavyweight object—from each invocation of the JVM from the Ant `java` task, and is not a problem in "real" applications.

Summary

In this chapter, we've shown how to acquire the Hibernate tools, how to create and run the example from Chapter 1, and how to create a slightly larger application from scratch, driving the database table generation from the `hbm2ddl` Ant task. All of the files described in this chapter and the others can be downloaded from the Apress web site (`www.apress.com`).

In the next chapter, we will look at the architecture of Hibernate and the lifecycle of a Hibernate-based application.