Pro Hibernate 3

DAVE MINTER AND JEFF LINWOOD

Apress®

ISBN (pbk): 1-59059-511-4

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

Lead Editor: Steve Anglin
Technical Reviewer: Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Linda Marousek
Production Manager: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Michael Brinkman
Artist: April Milne
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

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 Downloads section.

To our families.

Contents at a Glance

Contents

PART 2 Hibernate 3 Reference

About the Authors

DAVE MINTER is a freelance IT consultant (`http://paperstack.com/cv/`) from rainy London, England. His liking for computers was kicked off when a Wang 2200 minicomputer made a big impression on him at the tender age of six. Since then he has worked for the biggest of blue chips and the smallest of startups. These days he makes his living designing and building multi-tier applications that "just work." Dave is the coauthor with Jeff of *Building Portals with the Java Portlet API* (Apress, 2004). He has a computer studies degree from the University of Glamorgan.

JEFF LINWOOD is a software developer and consultant with the Gossamer Group (`http://www.gossamer-group.com/`) in sunny Austin, Texas. Jeff has been in software programming since he had a 286 in high school. He was caught up with the Internet when he got access to a Unix shell account and it has been downhill ever since. Jeff coauthored *Building Portals with the Java Portlet API* (Apress, 2004) with Dave and *Pro Struts Applications* (Apress, 2003), and was a technical reviewer for *Enterprise Java Development on a Budget* (Apress, 2004) and *Extreme Programming with Ant* (SAMS, 2003). He has a chemical engineering degree from Carnegie Mellon University.

About the Technical Reviewer

DILIP THOMAS is an Open Source enthusiast who keeps a close watch on LAMP technologies, Open Standards, and the full range of Apache Jakarta projects. He is coauthor of *PHP MySQL Website Programming: Problem - Design - Solution* (Apress, 2003) and a technical reviewer/editor on several Open Source/Open Standard book projects. Dilip is an Editorial Director at Software & Support Verlag GmbH.

Dilip resides in Bangalore with his beautiful wife, Indu, and several hundred books and journals. You can reach him via email at `dilip.thomas@gmail.com`.

Acknowledgments

This book would not have been possible without the energy and enthusiasm of the Apress staff: especially Steve Anglin for entrusting us with the project in the first place; Beth Christmas, our Project Manager; Linda Marousek, our Copy Editor; and Katie Stence, our Production Editor. Thank you all.

We are indebted to Dilip Thomas, our Technical Editor, whose timely and pithy commentary helped us to polish the rough edges of this work. The parts of the book that we are particularly proud of are usually the direct result of his notes and enquiries.

Dave would like to thank his parents for entrusting the family ZX81 to his clutches some years past, and his wonderful girlfriend Kalani Seymour for her patience and enthusiasm. He would also like to thank Jeff Verdegan for his assistance with a particularly tricky example that makes an appearance in Chapter 13.

Jeff would like to thank his family, Nancy, Judy, Rob, and Beth, for being supportive during the book-writing process. He'd also like to thank his friends Giuliano, Ronalyn, Roman, Karl, Michael, Jason, Valerie, Micah, and Betsy.

Where this book fails to achieve its aims, we the authors are, of course, entirely responsible.

Introduction

Virtually every application we have worked on has involved a database. While Java and the JDBC standard have significantly reduced the effort involved in carrying out simple database operations, the point of contact between the object-oriented world and the relational world always causes problems.

Hibernate offers a chance to automate most of the hard parts of this particular problem. By writing a simple XML file, you can make the database look object oriented. Along the way, you get some performance improvements. You also get an enormous improvement in the elegance of your code.

We were originally skeptical of yet another object-relational system, but Hibernate is so much easier to work with that we are entirely converted. Neither of us expect to write a database-based system with JDBC and SQL prepared statements again—Hibernate or its successors will form an essential part of the foundations. Hibernate is not just "another" object-relational system, it is the standard to beat.

Speaking of standards, the EJB3 specification has received a lot of input from the Hibernate team—who have, in turn, made efforts to reflect the emerging standard in their API. Therefore, to learn good practices for EJB3, start here with Hibernate.

Who This Book Is For

This book is for Java developers who have at least some basic experience with databases and using JDBC. You will need to have a basic familiarity with SQL and database schemas. While it is not required, it will certainly be helpful if you have an understanding of the aims of database normalization. We do not assume that you have prior experience with Hibernate or any other object-relational technology.

Hibernate is an Open Source project, so it is fitting that all of our examples make use of Open Source technologies. Part of Hibernate's promise is that many of the details of a particular relational database are abstracted away, so most of our examples will work with any database Hibernate supports. When we discuss database features that not all supported databases have, such as stored procedures, we specifically mention which database we used—both of our example databases are Open Source and readily available.

This book is not an academic text and does not generally attempt to describe the internal workings of Hibernate itself. As professional developers, we have tried to impart the understanding of how to use Hibernate rather than how to write it!

How This Book Is Organized

This book is roughly divided into two parts, beginning with a primer on the basics of Hibernate in Chapters 1 through 4 and continuing with more technical or design-oriented content in Chapters 5 through 14.

We encourage readers who are completely new to this technology to read through the primer section as this will give you a grounding in the basic requirements for a Hibernate-based application.

Readers who are familiar with Hibernate 2 should start with Chapter 14, where we discuss the differences between Hibernate 2 and Hibernate 3, and then refer back to appropriate chapters:

- *Chapter 1, An Introduction to Hibernate 3*: A basic introduction to the fundamentals of Hibernate and an overview of some sample code.
- *Chapter 2, Integrating and Configuring Hibernate*: How to integrate Hibernate into your Java application, an overview of the configuration options for Hibernate, and we explain how Hibernate deals with the differences between its supported relational databases.
- *Chapter 3, Building a Simple Application*: Two working-example Hibernate programs in full with extensive commentary on how the source code, mapping files, and database are related.
- *Chapter 4, Using Annotations with Hibernate*: How to use the new EJB3 annotations with Hibernate 3 to create object-relational mappings in your Java source code.
- *Chapter 5, The Persistence Lifecycle*: We explain how Hibernate manages the objects that represent data in tables, and also present an overview of the relevant methods for creating, retrieving, updating, and deleting objects through the Hibernate session.
- *Chapter 6, Creating Mappings*: We discuss mapping files in depth. We cover all of the commonly used elements in detail.
- *Chapter 7, Querying Objects with Criteria*: How to use the Hibernate criteria query API to selectively retrieve objects from the database.
- *Chapter 8, Querying with HQL and SQL*: This chapter contains a discussion of Hibernate's object-oriented query language, HQL, and how to use native SQL with Hibernate.
- *Chapter 9, Using the Session*: The ways in which sessions, transactions, locking, caching, and multi-threading are related. We also present a comprehensive example of a deadlock.
- *Chapter 10, Design Considerations with Hibernate 3:* How the Hibernate persistence layer of an application is designed.
- *Chapter 11, Events and Interceptors*: We discuss these very similar features. Interceptors were available in Hibernate 2 and are relatively unchanged. Events, while similar to Interceptors, were introduced with Hibernate 3.
- *Chapter 12, Hibernate Filters*: Hibernate provides filtering functionality for limiting the data returned by queries to a definable subset. We give an example showing this useful functionality and explain where you might use it in an application.

- *Chapter 13, Fitting Hibernate into the Existing Environment*: We discuss how to go about integrating Hibernate with legacy applications and databases. We also present an example of how to invoke a stored procedure.
- *Chapter 14, Upgrading from Hibernate 2*: For users familiar with Hibernate 2, we discuss the changes and additions in Hibernate 3.

PART 1

Hibernate 3 Primer

CHAPTER 1

An Introduction to Hibernate 3

Most significant development projects involve a relational database. The mainstay of most commercial applications is the large-scale storage of ordered information, such as catalogs, customer lists, contract details, published text, and architectural designs.

With the advent of the World Wide Web, the demand for databases has increased. Though they may not know it, the customers of online bookshops and newspapers are using databases. Somewhere in the guts of the application a database is being queried and a response is offered.

While the demand for such applications has grown, their creation has not become noticeably simpler. Some standardization has occurred—the most successful being the Enterprise JavaBeans (EJBs) standard of Java 2 Enterprise Edition (J2EE), which provides for container and bean-managed persistence of Entity Bean classes. Unfortunately, this and other persistence models all suffer to one degree or another from the mismatch between the relational model and the object-oriented model. In short, database persistence is difficult.

There are solutions for which EJBs are more appropriate, for which some sort of Object Relational Mapping (ORM) like Hibernate is appropriate, and some for which the traditional connected[1] approach of direct access via the JDBC API is most appropriate. Making the call on which to use requires an intimate knowledge of the strengths and weaknesses of them all—but all else being equal, we think that Hibernate offers compelling advantages in terms of ease of use.

To illustrate some of Hibernate's other strengths, in this chapter we will show you a "Hello World" Java database example. This application allows previously stored messages to be displayed using a simple key such as the "Message of the day." We will show the essential implementation details using the connected approach and using Hibernate.

Both of our examples are invoked from the main method in Listing 1-1, with the point at which we invoke the specific implementation marked in bold.

1. There is no standard way of referring to "direct use of JDBC" as opposed to ORM or EJBs, so we have adopted the term "connected" as the least awkward way of expressing this.

Listing 1-1. *The Main Method That Will Invoke Our Hibernate and Connected Implementations*

```
public static void main(String[] args) {
   if (args.length != 1) {
      System.err.println("Nope, enter one message number");
   } else {
      try {
         int messageId = Integer.parseInt(args[0]);
         Motd motd = getMotd(messageId);
         if (motd != null) {
            System.out.println(motd.getMessage());
         } else {
            System.out.println("No such message");
         }
      } catch (NumberFormatException e) {
         System.err.println("You must enter an integer - " + args[0]
               + " won't do.");
      } catch (MotdException e) {
         System.err.println("Couldn't get the message: " + e);
      }
   }
}
```

Plain Old Java Objects (POJOs)

In our ideal world, it would be trivial to take any Java object and persist it to the database. No special coding would be required to achieve this, no performance penalty would ensue, and the result would be totally portable. The common term for the direct persistence of traditional Java objects is Object Relational Mapping. There is a direct and simple correspondence between the POJOs[2], something that the forthcoming changes in the EJB3 standard look set to emulate.

Where Entity Beans have to follow a myriad of awkward naming conventions, POJOs can be any Java object at all. Hibernate allows you to persist POJOs, with very few constraints. Listing 1-2 is an example of a simple POJO to represent our "Message of the day" (`Motd`) announcement.

Listing 1-2. *The POJO That We Are Using in This Chapter's Examples*

```
public class Motd {
   protected Motd() {
   }
```

2. Java objects requiring no special treatment to be stored are often referred to as Plain Old Java Objects, or POJOs for short. This is really just to provide a handy term to differentiate them from Entity Beans and the like, which must conform to complicated and limiting contracts. The name also conveys something of the benefits of Hibernate—you don't have to do anything special to support persistence of a POJO via Hibernate, so you really can reuse your existing Java objects.

```
   public Motd(int messageId, String message) {
      this.messageId = messageId;
      this.message = message;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getMessage() {
      return message;
   }

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

   private int messageId;
   private String message;
}
```

What sort of POJO can be persisted? Practically anything. It has to provide a default constructor (but this can be given package or private scope to avoid pollution of your API), but that's it. Not an especially onerous burden.

Origins of Hibernate and Object Relational Mapping

If Hibernate is the solution, what was the problem? The gargantuan body of code in Listing 1-3 is required to populate our example `Motd` object from the database even when we know the appropriate message identifier:

Listing 1-3. *The Connected Approach to Retrieving the POJO*

```
   public static Motd getMotd(int messageId) throws MotdException {
      Connection c = null;
      PreparedStatement p = null;
      Motd message = null;

      try {

         Class.forName("org.postgresql.Driver");
         c = DriverManager.getConnection(
               "jdbc:postgresql://127.0.0.1/hibernate",
               "hibernate",
               "hibernate");
         p = c.prepareStatement(
               "select message from motd where id = ?");

         p.setInt(1, messageId);
         ResultSet rs = p.executeQuery();

         if (rs.next()) {
            String text = rs.getString(1);
            message = new Motd(messageId, text);

            if (rs.next()) {
               log.warning(
                  "Multiple messages retrieved for message ID: "
                 + messageId);
            }
         }

      } catch (Exception e) {
         log.log(Level.SEVERE, "Could not acquire message", e);
         throw new MotdException(
               "Failed to retrieve message from the database.", e);
      } finally {
         if (p != null) {
            try {
               p.close();
            } catch (SQLException e) {
               log.log(Level.WARNING,
                     "Could not close ostensibly open statement.", e);
            }
         }

         if (c != null) {
            try {
               c.close();
            } catch (SQLException e) {
               log.log(Level.WARNING,
                     "Could not close ostensibly open connection.", e);
            }
         }
      }

      return message;
   }
```

While some of this can be trimmed down—for example, the acquisition of a connection is more likely to be done in a single line from a `DataSource`—there is still a lot of boilerplate code here. Entering this manually is tedious and error prone.

EJBs As a Persistence Solution

So why not just use EJBs to retrieve data? Entity Beans are, after all, designed to represent, store, and retrieve data in the database.

Strictly speaking, an Entity Bean is permitted two types of persistence in an EJB server: Bean-Managed Persistence (BMP) and Container-Managed Persistence (CMP). In BMP, the bean itself is responsible for carrying out all of the Structured Query Language (SQL) associated with storing and retrieving its data—in other words, it requires the author to create the appropriate JDBC logic complete with all the boilerplate in Listing 1-3. CMP, on the other hand, requires the container to carry out the work of storing and retrieving the bean data.
So why doesn't that solve the problem? Here are just a few of the reasons:

- CMP Entity Beans require a one-to-one mapping to database tables.
- They are (by reputation at least) slow.
- Someone has to determine which bean field maps to which table column.
- They require special method names. If these are not followed correctly, they will fail silently.
- Entity Beans have to reside within a J2EE application server environment—they are a heavyweight solution.
- They cannot readily be extracted as "general purpose" components for other applications.
- They can't be serializable.
- They rarely exist as portable components to be dropped into a foreign application—you generally have to roll your own EJB solution.

Hibernate As a Persistence Solution

Hibernate addresses a lot of these points, or alleviates some of the pain where it can't, so we'll address the points in turn.

Hibernate does not require you to map one POJO to one table. A POJO can be constructed out of a selection of table columns, or several POJOs can be persisted into a single table.

Though there is some performance overhead while Hibernate starts up and processes its configuration files, it is generally perceived as being a fast tool. This is very hard to quantify, and, to some extent, the poor reputation of Entity Beans may have been earned more from the mistakes of those designing and deploying such applications than on its own merits. As with all performance questions, you should carry out tests rather than rely on anecdotal evidence.

In Hibernate it is possible, but not necessary, to specify the mappings at deployment time. The EJB solution ensures portability of applications across environments, but the Hibernate approach tends to reduce the pain of deploying an application to a new environment.

Hibernate persistence has no requirement for a J2EE application server or any other special environment. It is, therefore, a much more suitable solution for stand-alone applications, for client-side application storage, and other environments where a J2EE server is not immediately available.

Hibernate uses POJOs that can very easily and naturally be generalized for use in other applications. There is no direct dependency upon the Hibernate libraries so the POJO can be put to any use that does not require persistence—or can be persisted using any other POJO "friendly" mechanism.

Hibernate presents no problems when handling serializable POJOs.

Finally, there is a very large body of preexisting code. Any Java object capable of being persisted to a database is a candidate for Hibernate persistence. Therefore, Hibernate is an excellent solution to choose when adding a standard persistence layer to an application that uses an ad hoc solution, or which has not yet had database persistence incorporated into it. Furthermore, by selecting Hibernate persistence, you are not tying yourself to any particular design decisions for the business objects in your application.

A Hibernate Hello World Example

Listing 1-4 shows how much less boilerplate is required with Hibernate than with the connected approach in Listing 1-3.

Listing 1-4. *The Hibernate Approach to Retrieving Our POJO*

```
public static Motd getMotd(int messageId)
   throws MotdException
{
   SessionFactory sessions =
      new Configuration().configure().buildSessionFactory();
   Session session = sessions.openSession();
   Transaction tx = null;
   try {
      tx = session.beginTransaction();

      Motd motd =
         (Motd)session.get( Motd.class,
                            new Integer(messageId));

      tx.commit();
      tx = null;
      return motd;
   } catch ( HibernateException e ) {
      if ( tx != null ) tx.rollback();
      log.log(Level.SEVERE, "Could not acquire message", e);
      throw new MotdException(
            "Failed to retrieve message from the database.",e);
   } finally {
      session.close();
   }
}
```

Even for this trivial example there would be a further reduction in the amount of code required in a real deployment—particularly in an application-server environment. The `SessionFactory` would normally be created elsewhere and made available to the application as a Java Native Directory Interface (JNDI) resource. It is particularly interesting that because this object is keyed on its identifier, we can carry out all of the SQL of Listing 1-4, along with the population of the object from the result set using the single line

```
Motd motd = (Motd)session.get(Motd.class, new Integer(messageId));
```

When more complex queries that do not directly involve the primary key are required, some SQL (or rather, HQL [the Hibernate Query Language]) is required, but the work of populating objects is permanently eradicated.

Some of the additional code in our Hibernate 3 example actually provides functionality (transactionality and caching) beyond that of the connected example. In addition, we have greatly reduced the amount of error-handling logic required.

Mappings

Of course there is more to it than this—Hibernate needs something to tell it which tables relate to which objects (the information is actually provided in an XML-mapping file). But while some tools inflict vast, poorly documented XML configuration files on their users, Hibernate offers a breath of fresh air; you create and associate a small, clear mapping file with each of the POJOs that you wish to map into the database. You're permitted to use a single monolithic configuration file if you prefer, but it's neither compulsory nor encouraged.

A Document Type Definition (DTD) is provided for all Hibernate's configuration files, so with a good XML editor you should be able to take advantage of autocompletion and autovalidation of the files as you create them. A number of tools allow the automated creation of the mapping files, and Java 5 annotations can be used to replace them entirely.

Listing 1-5 shows the file mapping our `Motd` POJO into the database.

Listing 1-5. *The XML File Mapping Our POJO to the Database*

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

```
<hibernate-mapping>
   <class name="Motd" table="Motd">
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="message" column="message" type="string"/>
   </class>
</hibernate-mapping>
```

It would be reasonable to ask if the complexity has simply been moved from the application code into the XML-mapping file. But, in fact, this isn't really the case for several reasons.

First, the XML file is much easier to edit than a complex population of a POJO from a result set.

Second, we have still done away with the complicated error handling that was required with the connected approach. This is probably the weakest reason, however, since there are various techniques to minimize this without resorting to Hibernate.

Third, finally, and most importantly, if the POJO provides bean-like property access methods and a default constructor, then tools provided with Hibernate allow the direct generation of a mapping file. We will discuss the use of these tools in depth in Chapter 10.

Database Generation

If you are creating a new Hibernate application around an existing database, creating the database schema is not an issue; it is presented to you on a platter. If you are starting a new application, you will need to create the schema, the POJOs, and the mapping directly.

Yet again, there is a tool to make life easy for you. The `SchemaExport` tool allows a database schema to be generated for your database directly from the mapping file. This is even better than it sounds! Hibernate comes with intimate knowledge of various different dialects of SQL, so the schema will be tailored to your particular database software—or can be generated for each different database to which you want to deploy your application.

The generation of a SQL script to create your database schema is similar to the generation of the mapping file from the POJO:

```
java org.hibernate.tool.hbm2ddl.SchemaExport Motd.hbm.xml
```

Bootstrapping Your Configuration

You've probably now realized one of the things we really like about Hibernate—if you have any one of the three things you need (POJOs, schema, mapping), you can generate the other two without having to write them from scratch.

Figure 1-1 shows the various ways in which files and databases can be created:

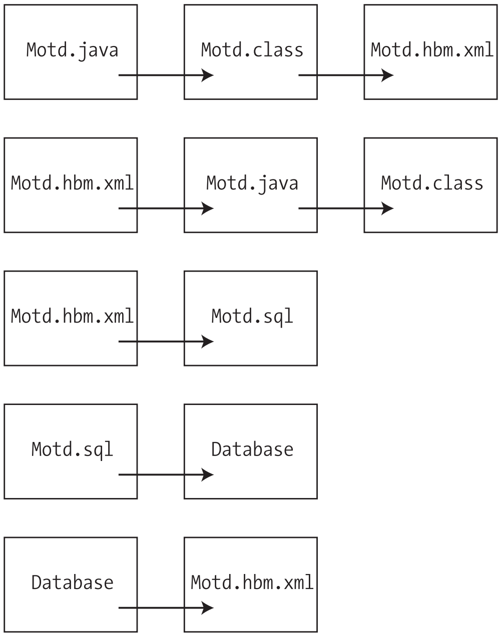

Figure 1-1. *Automated creation of files in Hibernate*

from which a simplified set of the relationships makes it clear that we can get from any one file the other two essentials (see Figure 1-2).

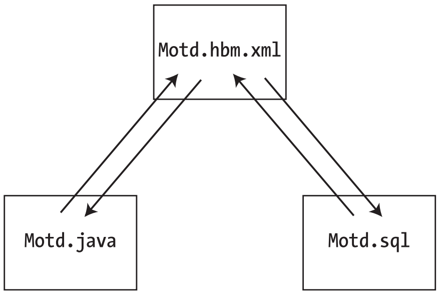

Figure 1-2. *Mutual creation of files in Hibernate*

Once we have the SQL and Java files, we can generate the database and class files in the conventional manner. In practice, if you have a complicated schema or a complicated set of POJOs, you will always need to make changes from the default options produced by these simple tools. But even so, this takes a lot of the legwork out of the process.

If you are creating your application from scratch, you will probably want to drive the creation of POJOs from your mapping file, or take advantage of other tools that will allow you to specify the minutiae of each POJO's relationship with other classes inside the class definition itself.

The Relationship of Hibernate 3 with EJB3

The current version of the EJB standard does not provide an ORM mechanism. However, this will change. The forthcoming standard for the next (3.0) incarnation of EJB is being developed as Java Specification Request (JSR) 220, and will include an EJB ORM standard. Why learn Hibernate when a new standard blessed by Sun is to be released? Because the original developers of Hibernate are closely involved in the creation of the standard and as one of the most successful ORM solutions, Hibernate has heavily influenced the design of EJB3.

EJB3 supports and encourages the use of transparent persistence of Entity Beans, which conform to the same minimal set of requirements as the POJOs supported by Hibernate. As a result of this, Hibernate can be used as the implementation of the `EntityManager` component of an EJB container.

Hibernate's HQL also has a strong correlation with the new EJB Query Language (EJB QL), though this is probably as much an artifact of their common ancestry in SQL as the Hibernate team's participation in the standards effort.

EJB3 allows the relationship between Entities/POJOs and the database to be described with a deployment descriptor. Although, at the time of writing, this has not been fully specified, it looks likely to resemble the Hibernate mapping files in both form and purpose. Even more interesting, EJB3 will also allow this relationship to be expressed as Java 5 annotations inline with the Entity/POJO's source code. This is directly supported by Hibernate 3 annotations. The convenience of this approach suggests that it will become the standard way to maintain these mappings as Java 5 deployments become commonplace.

In short, if you code to Hibernate 3 now, the effort involved in the transition to EJB3 later will be minimal if at all necessary. Meanwhile, Hibernate 3 is available now and has a persuasive pedigree.

Summary

In this chapter we have considered the problems and requirements that have driven the development of Hibernate. We have looked at some of the details of a trivial example application written with and without the aid of Hibernate. While we have glossed over some of the implementation details, we will discuss these in depth in Chapter 3. We have also discussed some of the tools that can accelerate your initial steps in creating a Hibernate project.

In the next chapter, we will look at the architecture of Hibernate and how it is integrated into your applications.

CHAPTER 2

Integrating and Configuring Hibernate

Compared to other Java persistence solutions, integrating Hibernate into a Java application is easy. The designers of Hibernate avoided some of the more common pitfalls and problems with the existing Java persistence solutions, and created a clean but powerful architecture. In practice, this means that you do not have to run Hibernate inside any particular J2EE container or framework—Hibernate 3 only requires Java 2 Standard Edition (J2SE) version 1.2 or later, although the new annotations feature requires J2SE 5.0 (or later).

If you already have an application that uses Hibernate 2, the migration path is easy from Hibernate 2 to Hibernate 3. Hibernate 3 is not completely backwards-compatible, but most of the changes are additional features that you can integrate into your existing application as you see fit. The Hibernate developers provided implementations of the core Hibernate 2 objects in Hibernate 3 with the Hibernate 2 methods for backwards compatibility. We cover the changes between Hibernate 2 and Hibernate 3 in more depth in Chapter 14.

One of the key features of Hibernate's design is the principle of least intrusiveness—the Hibernate developers did not want Hibernate to intrude into your application more than was necessary. This leads into several of the architectural decisions made for Hibernate. In Chapter 1, we discussed the problem Hibernate solves, and how your application's business objects map to a relational database. In this chapter, we explain how Hibernate integrates with your Java application, how to configure Hibernate, and how to set up a database connection pool for Hibernate.

Integrating Hibernate with Your Java Application

You can call Hibernate from your Java application directly, or you can access Hibernate through another framework. Hibernate supports the Java Management Extensions (JMX), Java Connector Architecture (JCA), and Java Naming and Directory Interface (JNDI) standards. Using JMX, you can configure Hibernate while it is running. Hibernate may be deployed as a JCA connector, and you can use JNDI to obtain a Hibernate session factory in your application. In addition, Hibernate uses standard Java Database Connectivity (JDBC) database drivers to access the relational database. Hibernate does not replace JDBC as a database connectivity layer—Hibernate sits on a level above JDBC.

In addition to the standard Java APIs, many Java web and application frameworks now integrate with Hibernate. Hibernate's simple, clean API makes it simple for these frameworks to support Hibernate in one way or another. The Spring framework provides excellent

Hibernate integration, including generic support for persistence objects, a generic set of persistence exceptions, and transaction management. You will need to get started with Spring's `HibernateTemplate` class. Spring is outside the scope of this Hibernate book, although it comes recommended by both of us. For more about Spring, see *Pro Spring* by Rob Harrop and Jan Machacek (Apress, 2005) or go to the website `http://www.springframework.org/`.

No matter how you access Hibernate, you will need to create a Hibernate configuration, and then build a Hibernate session factory from the configuration. Once you have a session factory, your application asks the factory for a session. Your application will use Hibernate sessions to manage the persistent objects. In the next section, we discuss how to use the `Configuration` class to provide all of the settings for your Java application's Hibernate instance.

Deploying Hibernate

To integrate Hibernate with your Java application, you will need to use several Java libraries. The first library is the Java Archive (`.jar`) file for your JDBC driver, which you will need to find for your specific relational database. The JDBC driver requirements come from the database-specific classes in Hibernate called *dialects*. These dialects define the SQL variant and the specific database features to use for each vendor's database. We discuss dialects in more detail further on in the chapter. The Hibernate website contains a platform-specific FAQ that offers solutions to several database-specific questions.

If you need Hibernate to work with older JDBC versions, be sure to disable two JDBC 2 specific features: batch update and scrollable result sets. Use the following configuration values for Hibernate (we discuss configuration later in this chapter):

```
hibernate.jdbc.batch_size=0
hibernate.jdbc.use_scrollable_resultsets=false
```

After you configure the JDBC driver, the next step is to deploy `hibernate3.jar` with your application, which comes with the Hibernate 3 binary distribution. The `hibernate3.jar` file contains the `org.hibernate` classes, along with several DTD and XML schema files. Inside the `lib` directory of the Hibernate 3 distribution, you will find many different Java archives, some of which are required for running Hibernate 3, and some of which are only required for optional functionality. After you deploy all of the libraries with your application, you will need to create a configuration file for your application.

Required Libraries for Running Hibernate 3

Hibernate requires several libraries beyond `hibernate3.jar`. These libraries are included in the `lib` directory of your Hibernate 3 installation. The up-to-date list of libraries shipped with Hibernate 3 is in the `lib/README.txt` file in the Hibernate distribution.

There are several optional libraries included with the Hibernate 3 distribution. If you build Hibernate from source, a few of these are necessary for Hibernate to compile. Other libraries provide connection pools, additional caching functionality (the `Session` cache is mandatory), and the JCA API.

Enterprise JavaBeans 3

The Enterprise JavaBeans 3 (EJB3) specification was available in early release form when we wrote this book. Although Hibernate's design influenced many of the changes (lightweight persistence and the query language) in EJB3, Hibernate is not a full implementation of EJB3. Your EJB3 container may use Hibernate as its persistence service for object-relational mapping (ORM), and if it does, you may be able to access the Hibernate session factory through a JNDI lookup or another mechanism.

If you have used EJB 1.x or 2.x in the past, EJB3 should be a much-needed simplification. We strongly suggest using Hibernate instead of EJB3 if all you need is a persistence solution—Hibernate does not require a specific application server or container. Unfortunately, we have seen too many uses of EJB as just an ORM layer, instead of either taking advantage of EJB's transaction and distributed application capabilities or using a lighter-weight ORM solution such as Hibernate.

Hibernate 3's annotations are compatible with the proposed EJB3 annotations, although there are Hibernate-specific extensions. In Chapter 4, we discuss how to use the new annotations and metadata features of Hibernate 3.

Java Management Extensions (JMX) and Hibernate

Java Management Extensions is a standard API for managing Java applications and components.

Hibernate provides two `MBeans` for JMX: `HibernateServiceMBean` and `StatisticsService➥MBean`. Both of these are interfaces that reside in the `org.hibernate.jmx` package. The `HibernateService` and `StatisticsService` classes in the same package implement the interfaces. The `HibernateServiceMBean` provides getter and setter methods for many of the Hibernate configuration properties, including the data source, transaction strategy, caching, dialect, and other database settings. It also provides a mechanism for adding mapping files to the configuration. An administrator could use a JMX management console to change the Hibernate configuration. When the `HibernateServiceMBean` starts, the `MBean` will create a `Configuration` object from its properties and mapping files, and then build a `SessionFactory` object. The `SessionFactory` object binds to the JNDI location specified on the JMX `MBean`, and your Java application can use standard JNDI calls to get that session factory.

The other `MBean` is for statistics. Hibernate can log statistics about the performance of query execution, caching, and object entity operations. Using a JMX console, an administrator can enable statistics and then view up-to-date performance indicators through the console.

The advantage of JMX here is that administrators or other nondevelopers may change properties at runtime with a standardized JMX console that they are already familiar with. JMX also has a loose coupling advantage—your application does not have to worry about configuring Hibernate because JMX is responsible.

If you are not familiar with JMX, the Open Source JBoss application server uses JMX to manage its internal components. You can manage Hibernate or another `MBean` through JBoss's administration console.

Hibernate Configuration

Before you create a session factory, you will need to tell Hibernate where to find the mapping files that define how a Java class maps to a database table. Hibernate also uses a set of configuration settings, which are usually either in a standard Java properties file called `hibernate.properties` or an XML file named `hibernate.cfg.xml`. You may use either the XML or the Java properties file for configuration. We recommend using the XML format, because it has support for configuring your mapping files. If you use `hibernate.properties`, you will have to configure your mappings in Java with the `org.hibernate.cfg.Configuration` class. Create an instance of the `Configuration` class to set the properties and mapping files for your `SessionFactory`. There is only one public constructor for `Configuration`, the default constructor.

Internally, Hibernate uses a somewhat confusing set of classes in the `org.hibernate.cfg` package to configure itself. Although you should be aware of what these classes do for the `Configuration` class, you will typically only interact with the `Configuration` class directly. The `Configuration` constructor creates a new instance of the `org.hibernate.cfg.SettingsFactory` class for use in the `Configuration`.

The settings factory creates a new instance of `org.hibernate.cfg.Settings` out of the Hibernate configuration properties. These properties could be in a `hibernate.properties` file located in the root of the classpath or the Java system properties.

The `buildSessionFactory()` method on the `Configuration` class creates an instance of the `SessionFactory`:

```
public SessionFactory buildSessionFactory() throws HibernateException
```

The `SessionFactory` is a heavyweight object, and your application should use one Hibernate `SessionFactory` object for each database it interacts with. The `SessionFactory` relies on the configuration properties, which we discuss in the next section of this chapter.

After you obtain the `SessionFactory`, you can retrieve Hibernate `org.hibernate.Session` objects. These `Session` objects are lightweight, as opposed to the `SessionFactory` object. You perform your persistence operations with the `Session` object. In Chapter 3, we present an example that walks you through a simple application that uses Hibernate for data persistence.

Hibernate Properties

Typically, you will specify your Hibernate configuration in a properties file called `hibernate.properties` in the root directory of your application's classpath. Hibernate has an extensive list of properties for configuration (see Table 2-1)—not all of them are required, of course, because of sensible defaults. You will need to configure a JDBC connection here, provide one programmatically in your application, or specify the JNDI name of a container-provided JDBC connection. You should also configure the SQL dialect specific to your deployment database that Hibernate should use to generate SQL. We discuss SQL dialects in more detail later in the chapter.

Table 2-1. *Hibernate Configuration Property Names and Descriptions*

Property Name	Description
`hibernate.connection.provider_class`	Class that implements Hibernate's `ConnectionProvider` interface.
`hibernate.connection.driver_class`	The JDBC driver class.
`hibernate.connection.isolation`	The transaction isolation level for the JDBC connection.
`hibernate.connection.url`	The JDBC URL to the database instance.
`hibernate.connection.username`	Database username.
`hibernate.connection.password`	Database password.
`hibernate.connection.autocommit`	Uses `autocommit` for the JDBC connection.
`hibernate.connection.pool_size`	Limits the number of connections waiting in the Hibernate database connection pool.
`hibernate.connection.datasource`	`Datasource` name for a container-managed data source.
`hibernate.connection.`*`<JDBCpropertyname>`*	Passes any JDBC property you would like to the JDBC connection—for instance, `hibernate.connection.debuglevel=info` would pass a JDBC property called `debuglevel`.
`hibernate.jndi.class`	Initial context class for JNDI.
`hibernate.jndi.url`	Provides URL for JNDI.
`hibernate.jndi.`*`<JNDIpropertyname>`*	Passes any JNDI property you would like to the JNDI `InitialContext`.
`hibernate.session_factory_name`	If this property is set, the Hibernate session factory will bind to this JNDI name.
`hibernate.dialect`	SQL dialect to use for Hibernate, varies by database. See section on SQL dialects.
`hibernate.default_schema`	Default database owner name that Hibernate uses to generate SQL for unqualified table names.
`hibernate.default_catalog`	Default database catalog name that Hibernate uses to generate SQL for unqualified table names.
`hibernate.show_sql`	Logs the generated SQL commands.
`hibernate.use_sql_comments`	Generates SQL with comments.
`hibernate.max_fetch_depth`	Determines how deep Hibernate will go to fetch the results of an outer join. Used by Hibernate's outer join loader.
`hibernate.jdbc.use_streams_for_binary`	Determines if binary data is read or written over JDBC as streams.

Continued

Table 2-1. *Continued*

Property Name	Description
`hibernate.jdbc.use_scrollable_resultset`	Determines if Hibernate will use JDBC scrollable `resultsets` for a user-provided JDBC connection.
`hibernate.jdbc.use_get_generated_keys`	If the database driver supports the JDBC 3 autogenerated keys API, Hibernate will retrieve any generated keys from the statement after it executes a SQL query.
`hibernate.jdbc.fetch_size`	Determines how many rows the JDBC connection will try and buffer with every fetch. This is a balance between memory and minimizing database network traffic.
`hibernate.jdbc.batch_size`	The maximum batch size for updates.
`hibernate.jdbc.factory_class`	The class name of a custom implementation of the `org.hibernate.jdbc.Batcher` interface for controlling JDBC prepared statements.
`hibernate.jdbc.batch_versioned_data`	Determines if Hibernate batches versioned data, which will depend on your JDBC driver properly implementing row counts for batch updates. Hibernate uses the row count to determine if the update was successful.
`hibernate.xml.output_stylesheet`	Specifies an XSLT stylesheet for Hibernate's XML data binder. Requires `xalan.jar`.
`hibernate.c3p0.max_size`	Maximum size of the connection pool for C3PO.
`hibernate.c3p0.min_size`	Minimum size of the connection pool for C3PO.
`hibernate.c3p0.timeout`	Timeout for C3PO (in seconds).
`hibernate.c3p0.max_statements`	Upper limit for the SQL statement cache for C3PO.
`hibernate.c3p0.acquire_increment`	After connection pool is completely utilized, determines how many new connections are added to the pool.
`hibernate.c3p0.idle_test_period`	Determines how long to wait before a connection is validated.
`hibernate.proxool`	Prefix for the Proxool database connection pool.
`hibernate.proxool.xml`	Path to a Proxool XML configuration file.
`hibernate.proxool.properties`	Path to a Proxool properties file.
`hibernate.proxool.existing_pool`	Configures Proxool with an existing pool.
`hibernate.proxool.pool_alias`	Alias to use for any of the above configured Proxool pools.
`hibernate.transaction.auto_close_session`	Closes session automatically after a transaction.

Property Name	Description
`hibernate.transaction.flush_before_completion`	Automatically flushes before completion.
`hibernate.transaction.factory_class`	Specifies a class that implements the `org.hibernate.transaction.TransactionFactory` interface.
`hibernate.transaction.manager_lookup_class`	Specifies a class that implements the `org.hibernate.transaction.TransactionManagerLookup` interface.
`jta.UserTransaction`	The JNDI name for the `UserTransaction` object.
`hibernate.cache.provider_class`	Specifies a class that implements the `org.hibernate.cache.CacheProvider` interface.
`hibernate.cache.use_query_cache`	Specifies whether or not to use the query cache.
`hibernate.cache.query_cache_factory`	Specifies a class that implements the `org.hibernate.cache.QueryCacheFactory` interface for getting `QueryCache` objects.
`hibernate.cache.use_second_level_cache`	Determines whether or not to use the Hibernate second level cache.
`hibernate.cache.use_minimal_puts`	Configures the cache to favor minimal puts over minimal gets.
`hibernate.cache.region_prefix`	The prefix to use for the name of the cache.
`hibernate.generate_statistics`	Determines whether statistics are collected.
`hibernate.use_identifier_rollback`	Determines whether Hibernate uses identifier rollback.
`hibernate.cglib.use_reflection_optimizer`	Instead of using slower standard Java reflection, uses the `CGLib` code generation library to optimize access to business object properties. Slower at startup if this is enabled, but faster runtime performance.
`hibernate.query.factory_class`	Specifies an HQL query factory class name.
`hibernate.query.substitutions`	Any possible SQL token substitutions Hibernate should use.
`hibernate.hbm2ddl.auto`	Automatically creates, updates, or drops database schema on startup and shut down. There are three possible values: create, create-drop, and update. Be careful with create-drop!
`hibernate.sql_exception_converter`	Specifies which `SQLExceptionConverter` to use to convert `SQLExceptions` into `JDBCExceptions`.
`hibernate.wrap_result_sets`	Turns on JDBC result set wrapping with column names.
`hibernate.order_updates`	Order SQL update statements by each primary key.

XML Configuration

As we discussed previously, Hibernate offers XML configuration capabilities. The advantage of using the XML configuration is that the XML configuration file provides the ability to configure your Hibernate mapping files. Similar to `hibernate.properties`, create an XML configuration file called `hibernate.cfg.xml` and place it in the root of your application's classpath. The XML configuration file needs to conform to the Hibernate 3 Configuration DTD, which is available from `http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd`.

Listing 2-1 is an example XML configuration for Hibernate.

Listing 2-1. *An XML Configuration for Hibernate*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM ➥
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <mapping jar="hibernate-mappings.jar"/>
        <mapping resource="com/apress/hibernate/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

In addition to specifying the properties listed in Table 2-1 to configure a session factory, you can also configure mapping files, caching, listeners, and the JNDI name for the session factory in the XML configuration. When you use the XML configuration file, you do not need to use the `hibernate` prefix for properties, as illustrated in Listing 2-1, with the `dialect` property. We discuss mapping file configuration in the next section, and caching in Chapter 9.

If you are already using `hibernate.properties`, `hibernate.cfg.xml` will override any settings in `hibernate.properties`.

After placing your XML configuration file in the root directory of the classpath, you will need to call one of the `configure()` methods on your application's `Configuration` object. With the default file name (`hibernate.cfg.xml`), you can call `configure()` with no arguments. If you used a different file name (for instance, because you have production, staging, user acceptance test, and development environments, with different databases), use one of the following methods on a `Configuration` object:

- `public Configuration configure(String) throws HibernateException`: Loads the XML configuration from a resource accessible by the current thread's context class loader.
- `public Configuration configure(URL) throws HibernateException`: Retrieves the XML configuration from a valid URL.
- `public Configuration configure(File) throws HibernateException`: Uses an XML configuration file on the file system.

Mapping Documents

After you create your mapping documents for Hibernate, Hibernate needs to know where to find them. Before you create the session factory, add them to your `Configuration` object, or specify them in the `hibernate.cfg.xml` XML configuration file. For more about Hibernate's mappings, see Chapter 6.

If you choose to add the mapping documents directly to an instance of `Configuration`, use one of the following methods:

- `addFile(String)`: Uses the path to an XML mapping document for Hibernate. An example of this would be `com/hibernatebook/config/Example.hbm.xml`.
- `addFile(File)`: Uses a `File` object that represents an XML mapping document.
- `addClass(Class)`: Translates a Java class name into a file name, which Hibernate then loads as an input stream from the Java class's class loader; for example, Hibernate would look for the file called `com/hibernatebook/config/Example.hbm.xml` for a class named `com.hibernatebook.config.Example`.
- `addJar(File)`: Adds any mapping documents (`*.hbm.xml`) in the specified `.jar` file to the `Configuration` object.
- `addDirectory(File)`: Adds any mapping documents (`*.hbm.xml`) in the specified directory to the `Configuration` object.

The following methods also add mapping documents to the `Configuration` object, although you are less likely to use them unless you have specialized application deployment issues:

- `addXML(String)`: Takes a `String` object that contains the Hibernate mapping XML.
- `addURL(URL)`: Requires a valid URL to the Hibernate mapping XML.
- `addCacheableFile(File)`: Saves time when Hibernate loads XML mapping files at startup by caching XML mapping documents on the file system as serialized DOM `Document` objects. This improves performance after the first load. Takes a `File` object that points to the XML mapping document, not the `.bin` cache file.
- `addCacheableFile(String)`: Same as `addCacheableFile(File)`, except this method takes a path to the file. Hibernate constructs a `File` object out of the path and then passes it to `addCacheableFile(File)`.
- `addDocument(Document)`: Takes a valid DOM `Document` object with the XML.

The `addJar()` and `addDirectory()` methods are the most convenient, because you can load all of your Hibernate mapping documents at one time. Both of these methods also simplify code configuration, layout, and refactoring, because you will not have to maintain code that adds each document separately. We find that it is easy to create a mapping document and then forget to add it to your application's Hibernate initialization code, so both of these methods are handy.

Although specifying mapping documents directly in the code is one approach, another is to use the `<mapping>` element in the `hibernate.cfg.xml` XML configuration file. The `<mapping>` element has three possible attributes: `jar`, `resource`, and `file`, which map to the `addJar()`, `addResource()`, and `addFile()` methods on the `Configuration` object:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM ➥
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <mapping jar="hibernate-mappings.jar"/>
        <mapping resource="com/apress/hibernate/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Whether you use the XML configuration file or directly specify the mapping files in code is up to you—we suggest you use whichever you are less likely to forget to do.

Naming Strategy

If your project has an existing standard for naming database tables or columns, or you would like to specify exactly how Hibernate maps Java class names to database table names, you can use Hibernate's *naming strategy* functionality. Custom naming strategies specify how Hibernate maps Java class names to database table names, properties to column names, and the name of a table used to hold a collection of properties for a Java class. A naming strategy may also alter the table names and column names specified in the Hibernate mapping documents—use this for adding a consistent application-wide prefix to table names, for instance.

Although you could specify the names of all of the tables and columns in the mapping document, if you have clear and consistent rules for naming already, you will save a lot of time and frustration with a custom naming strategy. If you decide later to add a prefix to all database table names, it is easy to do with a naming strategy, but difficult to maintain and keep going forward in every Hibernate mapping document.

■ **Note** Using Hibernate with an existing database often means creating a custom naming strategy for Hibernate. If the database tables have a prefix, it may be cleaner to implement a naming strategy that adds that prefix, instead of specifying the full table name with a prefix in the Hibernate mapping documents.

A custom naming strategy should implement the `org.hibernate.cfg.NamingStrategy` interface or extend one of the two provided naming strategy classes: `org.hibernate.cfg.DefaultNamingStrategy` or `org.hibernate.cfg.ImprovedNamingStrategy`. The default naming strategy simply returns the unqualified Java class name as the database table name. For instance, the table name for the Java class `com.hibernatebook.AccessGroups` would be `AccessGroups`. The column name would be the same as the property name, and the collection table would have the same name as the property.

The improved naming strategy adds underscores in place of uppercase letters in mixed-case table and column names, and then lowercases the name. For instance, the same `com.hibernatebook.AccessGroups` Java class would correspond to a database table named `access_groups`.

Neither of these naming strategies takes into account the case where you have two classes with the same name in different packages in your application. For instance, if you had two classes, `com.hibernatebook.webcast.Group` and `com.hibernatebook.security.Group`, both would map to a table named `Group`, which is not what you want. You would have to set the table name in the class's mapping for at least one (if not both, for clarity). We discuss setting table names in Chapter 6, on mapping.

After you create a naming strategy, pass an instance of the naming strategy to the `Configuration` object's `setNamingStrategy()` method:

```
public Configuration setNamingStrategy(NamingStrategy namingStrategy)
```

Be sure to call this method before building the session factory from the `Configuration`. An example of this call, using the `ImprovedNamingStrategy` naming strategy, would be

```
Configuration conf = new Configuration()
conf.setNamingStrategy(ImprovedNamingStrategy.INSTANCE);
```

As you can see, you could also implement the `NamingStrategy` interface yourself and pass it to the `Configuration` object using this method.

Using a Container-Managed Data Source

If your application server manages data sources, Hibernate can obtain a data source through a JNDI lookup. You will need to specify the name of the data source in JNDI, along with several optional properties. Use the `hibernate.connection.datasource` property to specify the JNDI name, and the `hibernate.jndi.url` and `hibernate.jndi.class` properties to specify the location of the container's JNDI provider and the class name of the container's implementation of the JNDI `InitialContextFactory` interface. You may also use the `hibernate.connection.username` and `hibernate.connection.password` properties to specify the database user your application uses, just like a normal JDBC connection. For example, your `hibernate.properties` file may have these lines for a WebLogic 7.0 managed data source:

```
hibernate.connection.datasource=java:/comp/env/jdbc/TestDB
hibernate.connection.username=dbuser
hibernate.connection.password=dbpassword
hibernate.jndi.url= t3://localhost:7001
hibernate.jndi.class= weblogic.jndi.WLInitialContextFactory
```

If you use a container-managed data source, the container will also manage your transactions.

The Session Factory

The Hibernate session factory creates `Session` objects for a database connection and manages connection data, caching, and mappings behind the scenes. After you create and populate a

`Configuration` object, you will need to build a session factory from the `Configuration`. Your application is responsible for managing the session factory. You should only have one session factory, unless you are using Hibernate to connect to two or more databases with different settings, in which case you would have one session factory for each database.

To maintain backwards compatibility with Hibernate 2, the Hibernate 3 session factory creates `org.hibernate.classic.Session` session objects. The classic session objects implement all of the Hibernate 3 session functionality, in addition to the deprecated Hibernate 2 session methods. We cover the changes in core functionality between Hibernate 2 and 3 in Chapter 14.

Obtain a session from the `SessionFactory` object with one of the `openSession()` methods. There are four `openSession()` methods (see Listing 2-2). The no-argument `openSession()` method opens a session with the database connection and interceptor specified in the `SessionFactory`'s original configuration. You can pass a JDBC connection, a Hibernate interceptor, or both as arguments to the other `openSession()` methods.

Listing 2-2. *openSession() Methods*

```
public org.hibernate.classic.Session openSession() throws HibernateException
public org.hibernate.classic.Session openSession(Interceptor interceptor) ➥
    throws HibernateException
public org.hibernate.classic.Session openSession(Connection connection, ➥
Interceptor interceptor)
public org.hibernate.classic.Session openSession() throws HibernateException
```

We discuss Hibernate interceptors in Chapter 11. You can also retrieve metadata and statistics from the `SessionFactory`. Metadata is covered in Chapter 4.

The other important method on the session factory is `close()`. The `close()` method cleans up after the session factory, so it is important to be completely through with all Hibernate session operations before closing down the session factory. When the session factory closes, it destroys the cache for the entity persisters, the collection persisters, the query cache, and the timestamp cache. Then the session factory closes the JDBC connection provider and removes the current instance from its JNDI object factory binding. The session factory also auto-drops the database schema if you set the `hibernate.hbm2ddl.auto` property to create-drop.

```
public void close() throws HibernateException
```

The Hibernate developers designed their implementation of the `SessionFactory` interface to be scalable in a multithreaded application—because there is only one session factory in the application for each database.

SQL Dialects

JDBC abstracts away many of the underlying connection details for each relational database, but every relational database supports a different set of features and uses a slightly different version of SQL. Some of the features that differ between relational databases are the syntax for marking identity columns, column data types, available SQL functions, foreign key constraint syntax, limits, GUID support, and support for cascade deletes.

Hibernate abstracts away all of these changes into SQL dialects. Each supported database has its own dialect. When Hibernate constructs an SQL query, it uses the dialect for valid syntax for the current database. Hibernate 3 comes with over 20 different dialects. All of the dialects are in the `org.hibernate.dialect` package. Table 2-2 shows the supported databases in Hibernate 3 and their corresponding dialect classes.

Table 2-2. *Supported Databases and Dialect Class Names for Hibernate 3*

Database Name	Dialect Class Name
DB2/390	DB2390Dialect
DB2/400	DB2400Dialect
DB2	DB2Dialect
Derby	DerbyDialect
Firebird	FirebirdDialect
FrontBase	FrontBaseDialect
HSQLDB	HSQLDialect
Informix	InformixDialect
Ingres	IngresDialect
Interbase	InterbaseDialect
Mckoi	MckoiDialect
MySQL	MySQLDialect
MySQL with InnoDB tables	MySQLInnoDBDialect
MySQL with MyISAM tables	MySQLMyISAMDialect
Oracle 9	Oracle9Dialect
Oracle	OracleDialect
PointBase	PointbaseDialect
PostgreSQL	PostgreSQLDialect
Progress	ProgressDialect
SAP DB	SAPDBDialect
SQL Server	SQLServerDialect
Sybase 11	Sybase11Dialect
Sybase Anywhere	SybaseAnywhereDialect
Sybase	SybaseDialect

Configure your chosen dialect with the `hibernate.dialect` property. In Chapter 8, we discuss how to write queries with HQL. HQL provides object-querying functionality that is database-independent. Hibernate translates HQL into database-specific SQL using hints provided by the SQL dialect classes. In addition, Hibernate provides a native SQL facility, which is especially useful for porting existing JDBC applications to Hibernate or for improving the performance of complicated queries.

Database Independence

Choosing your level of database independence is an important design decision. Typically, Java developers prefer to keep logic in the Java application layer as opposed to the database layer (such as triggers or stored procedures). With Hibernate, this makes it easy to create a database-independent Java application. Database administrators generally prefer to use the database for logic. For referential integrity, modeling the foreign key constraints in the database (as opposed to using Java code) is much simpler, so there is little controversy on that subject. Some of the arguments for maintaining the logic at the Java application layer are that switching databases is easier, and all code is maintained in one language. One argument for maintaining logic at the database level is that more than one application may be using the database, and maintaining the logic in one place guarantees consistency. Another argument is that the programming style is different between Java (object-oriented) and SQL (set-oriented).

Summary

In this chapter, we explained how to integrate Hibernate into your Java application. We also detailed the configuration options for Hibernate, including the available Hibernate property settings. We recommended using the `hibernate.cfg.xml` XML configuration for Hibernate because it offers the ability to configure mapping files instead of configuring the mapping files in Java code. Naming strategies allow you to create a consistent company or application-wide database table-naming convention, and they also give you the ability to easily map your Hibernate classes to databases with existing naming conventions. Hibernate uses dialects to define different behaviors for different databases—each database has its own feature set and SQL compatibility quirks, and Hibernate provides built-in support for most common relational databases.

In the next chapter, we build and examine a sample Hibernate application that illustrates the core Hibernate concepts.

CHAPTER 3

Building a Simple Application

In this chapter, we'll take another look at some of the steps necessary to get the example from Chapter 1 up and running. We'll also build a somewhat larger application from scratch.

Install the Tools

Assuming you already have a JDK installed (version 1.3 or later) and available on your workstation, you will need Hibernate and a database. You may also want to install the Ant build tool.

Hibernate 3

The latest version of Hibernate is always available from `http://hibernate.org/` under the left-hand menu named "Download." Various older versions and additional libraries are available from the resulting page, but you will need to select Hibernate 3.0.2 or a later version. Download the archive and unpack that to a local directory. For the purpose of our examples, this will be the directory `C:\home\hibernate-3.0`.

The unpacked archive contains all of the source code for Hibernate itself, a `Jar` library built from this source, and all of the library files that are necessary to run our sample.

HSQL 1.7.3.3

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. The homepage of HSQL is `http://hsqldb.sourceforge.net/` and we will be using version 1.7.3.3. In practice, we expect that any later version will also be suitable. HSQL is derived from a 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 assume in our examples that you will be unpacking HSQL to `C:\home\hsqldb\` and that you will be using a database called Hibernate. First, create a directory to contain the Hibernate files, for example, `C:\home\hsqldb\hibernate\`, and then place the following file named `server.properties` (see Listing 3-1) into that directory:

Listing 3-1. *Configuration File for the Hibernate Example Database*

```
# Filename: C:\home\hsqldb\server.properties
#
# Hibernate examples database - create a
# database on the default port.

# Specifies the path to the database
# files – note that the trailing slash
# IS required.
server.database.0=file:/home/hsqldb/hibernate/

# Specifies the name of the database
server.dbname.0=hibernate
```

You can then manually run the following command from the new Hibernate directory to start up the database server:

```
java -classpath ..\lib\hsqldb.jar org.hsqldb.Server
```

To shut down the database in an orderly fashion, you will want to create and compile the simple Java program shown in Listing 3-2 (this will require `hsqldb.jar` in the classpath for both compilation and runtime).

Listing 3-2. *Simple Shutdown Logic for the HSQL Server*

```
import java.sql.*;

public class Shutdown {
   public static void main(String[] argv)
      throws Exception
   {
      Class.forName("org.hsqldb.jdbcDriver" );
      Connection c =
         DriverManager.getConnection(
            "jdbc:hsqldb:hsql://localhost/hibernate",
            "sa",
            "");
      Statement s = c.createStatement();
      s.execute("SHUTDOWN");
      c.close();
   }
}
```

The alternative to this is to use the `DatabaseManager` tool provided with HSQL. This and many other features are comprehensively documented in the files you unpacked into the HSQL directory and on the HSQL website. Sadly, the details of HSQL are outside the scope of this book, but we think you will find it extremely useful to be aware of this excellent tool.

Most of 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 must run the examples with your own database, then the differences mostly boil down to

- 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.

Ant 1.6.2

Optionally, you may want to install the Ant build tool. Ant is available from `http://ant.apache.org/`. Our examples are built with Ant 1.6.2. 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—and if not, then Ant is a topic in its own right. We 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.

Listing 3-3 provides the Ant script to build the example for Chapter 3, and we provide a (simpler) script to build the example from Chapter 1 later in the chapter. Note that our script here includes an Ant task to generate the schema directly from the mapping files.

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

```
<project default="all">

   <property name="hibernate" location="/home/hibernate-3.0" />
   <property name="jdbc" location="/home/hsqldb/hsqldb.jar" />

   <property name="src" location="src" />
   <property name="config" location="." />
   <property name="dist" location="dist" />
   <property name="bin" location="${dist}/bin" />
   <property name="lib" location="${dist}/lib" />
   <property name="name" value="chapter03" />

   <path id="classpath.base">
      <pathelement location="." />
         <pathelement location="${bin}" />
      <pathelement location="${hibernate}/hibernate3.jar" />
      <fileset dir="${hibernate}/lib" includes="**/*.jar" />
      <pathelement location="${jdbc}" />
   </path>
   <target name="init">
      <mkdir dir="${dist}" />
      <mkdir dir="${bin}" />
      <mkdir dir="${lib}" />
   </target>

   <target name="schema">
      <taskdef
         name="schemaexport"
         classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
         classpathref="classpath.base" />

      <schemaexport
         properties="hibernate.properties"
         quiet="no"
         text="yes"
         drop="no"
         output="export.sql">
            <fileset dir="${src}">
               <include name="**/*.hbm.xml"/>
            </fileset>
      </schemaexport>
   </target>

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

   <target name="dist" depends="compile">
      <jar
         destfile="${lib}/${name}.jar"
         basedir="."
         includes="build.xml,src/**,cfg/**,bin/**"
         excludes=".classpath,.project,.cvsignore" />
   </target>

   <target name="clean">
      <delete dir="${dist}" />
      <delete dir="${bin}" />
      <delete file="${lib}/${name}.jar" />
   </target>

   <target name="all" depends="dist" />
</project>
```

Tip Although we would prefer to use a single configuration file for both running the examples and generating the schema, the `SchemaExport` task in the current release of Hibernate 3 cannot correctly import the configuration from a `hibernate-configuration` XML file. We, therefore, have to use a properties file and explicitly import the `hibernate-mapping` files, as these cannot be specified in the properties file. It is worth checking the Hibernate website to see if the problem has been fixed by the time you are reading this.

Create a Hibernate Configuration File

Hibernate can be given all of the information it needs to connect to the database and determine its mappings in several ways. For our Message of the day[1] example, we used the configuration file in Listing 3-4.

Listing 3-4. *The "Message of the day" Application's Mapping File*

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

<hibernate-configuration>
   <session-factory>
      <property name="connection.driver_class">
         org.hsqldb.jdbcDriver
      </property>
      <property name="connection.url">
         jdbc:hsqldb:hsql://localhost/hibernate
      </property>
      <property name="connection.username">sa</property>
      <property name="connection.password"> </property>
      <property name="pool_size">5</property>
      <property name="show_sql">false</property>
      <property name="dialect">
         org.hibernate.dialect.HSQLDialect
      </property>

      <mapping resource=" Motd.hbm.xml"/>

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

1. In our choice of name, we are consciously emulating the Unix "Message of the day" program, which issued announcements to users as they logged in. This was commonly abbreviated to "Motd" and we're using the two names somewhat interchangeably.

Most of the tags in this file should be pretty familiar from setting up JDBC connections. The `show_sql` tag, 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 the `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 optimal correct SQL to use when creating and querying the database.

■ **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. This is true of the Hibernate tools as well, so when you provide the `--config` flag to `SchemaExport`, for example, you are providing the path relative to the classpath, not the full path to the file.

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

```
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc:hsqldb:hsql://localhost/hibernate
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.pool_size=5
hibernate.show_sql=false
hibernate.dialect= org.hibernate.dialect.HSQLDialect
```

As you will notice, this does not contain the resource mapping from the XML file and, in fact, you cannot include this information in a properties file; you'll need to directly map your classes into the Hibernate `Configuration` at runtime. Here is how this can be done:

```
Configuration config = new Configuration();
config.addClass( Motd.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 our class is `book.hibernate.gettingstarted.Motd`, we should see the following pair of files from the root of the classpath:

```
/book/hibernate/gettingstarted/Motd.class
/book/hibernate/gettingstarted/Motd.hbm.xml
```

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( "config/Motd.hbm.xml" );
config.setProperties( System.getProperties() );
SessionFactory sessions = config.buildSessionFactory();
```

You may have as many or as few mapping files as you wish—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, `Motd.hbm.xml` in the default package to map the `Motd` 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. For example:

```
java -classpath ...
   -Dhibernate.connection.driver_class=org.hsqldb.jdbcDriver
   -Dhibernate.connection.url=jdbc:hsqldb:hsql://localhost/hibernate
   -Dhibernate.connection.username=sa
   -Dhibernate.connection.password=
   -Dhibernate.pool_size=5
   -Dhibernate.show_sql=false
   -Dhibernate.dialect= org.hibernate.dialect.HSQLDialect
      org.hibernate.tool.hbm2ddl.SchemaExport
      --output=advert.sql
      /book/hibernate/gettingstarted/Motd.hbm.xml
```

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

Run the Message of the Day Example

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

Listing 3-5. *The Motd POJO Class*

```
public class Motd {
   protected Motd() {
   }

   public Motd(int id, String message) {
      this.id = id;
      this.message = message;
   }
   public int getId() {
      return id;
   }

   public String getMessage() {
      return message;
   }

   public void setId(int id) {
      this.id = id;
   }

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

   private int id;
   private String message;
}
```

Now, the Exception class (see Listing 3-6), which was not shown—extends Exception and is used as a container for any exceptions thrown in the persistence layer of the application.

Listing 3-6. *The MotdException Class*

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

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

Finally, we have the full text of the main part of the application in Listing 3-7.

Listing 3-7. *The Motd Application*

```
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.Transaction;
import org.hibernate.cfg.Configuration;
```

```
public class HibernateMotd {

   public static void main(String[] args) {
      if (args.length != 1) {
         System.err.println("Nope, enter one message number");
      } else {
         try {
            int messageId = Integer.parseInt(args[0]);
            Motd motd = getMotd(messageId);
            if (motd != null) {
               System.out.println(motd.getMessage());
            } else {
               System.out.println("No such message");
            }
         } catch (NumberFormatException e) {
            System.err.println("You must enter an integer - " + args[0]
                  + " won't do.");
         } catch (MotdException e) {
            System.err.println("Couldn't get the message: " + e);
         }
      }
   }

   public static Motd getMotd(int messageId)
      throws MotdException
   {
      SessionFactory sessions =
         new Configuration().configure().buildSessionFactory();
      Session session = sessions.openSession();
      Transaction tx = null;
      try {
         tx = session.beginTransaction();

         Motd motd = (Motd)session.get(Motd.class, new Integer(messageId));

         tx.commit();
         tx = null;
         return motd;
      } catch ( HibernateException e ) {
         if ( tx != null ) tx.rollback();
         log.log(Level.SEVERE, "Could not acquire message", e);
         throw new MotdException(
               "Failed to retrieve message from the database.",e);
      } finally {
         session.close();
      }
   }

   private static final Logger log = Logger.getAnonymousLogger();
}
```

The Ant build script in Listing 3-8 will build these class files into a suitable `.jar` file.

Listing 3-8. *The Motd Ant Build Script*

```
<project default="all">

  <property name="hibernate" location="/home/hibernate-3.0" />
  <property name="jdbc"      location="/home/hsqldb/lib/hsqldb.jar " />

  <property name="src"     location="." />
  <property name="config"  location="." />
  <property name="bin"     location="bin" />
  <property name="name"    value="motd"/>

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

  <path id="classpath.run">
    <pathelement location="${config}" />
    <path refid="classpath.base" />
  </path>

  <target name="init">
    <mkdir dir="${bin}" />
  </target>

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

  <target name="dist">
    <jar destfile="${name}.jar"
        basedir="${bin}"
         />
  </target>
  <target name="clean">
    <delete dir="${bin}" />
    <delete file="${name}.jar"/>
  </target>

  <target name="all" depends="dist" />
</project>
```

Run Ant and a `.jar` file will be built representing the Message of the day example application.

The next step is to build the database. This can be done manually, of course, but even if you prefer to do this, generating the SQL to create the database is a useful check to make sure that the mappings and the data model correspond correctly.

To keep things readable, we will set the classpath environment variable for these examples, although we would recommend either setting it with the classpath flag `--classpath` or adding a suitable task to the Ant script. The classpath should include

```
hibernate3.jar
```

Available from the root of the Hibernate directory is

```
commons-logging-1.0.4.jar
cglib-full-2.0.2.jar
commons-collections-2.1.1.jar
antlr-2.7.4.jar
dom4j-1.5.2.jar
ehcache-1.1.jar
jta.jar
```

These are all provided in the `lib` subdirectory of the Hibernate directory (along with a `README.txt` file that briefly describes each of the provided `.jar` files).

You need to include the JDBC driver for your database. For example:

```
hsqldb.jar
```

You will also need to include the `.jar` built from the Ant script, `motd.jar`.

The command to build the schema is then

```
java org.hibernate.tool.hbm2ddl.SchemaExport --text --output=motd.sql ➥
--config=hibernate.cfg.xml
```

This will display, amongst the logging messages, the SQL being written to the `motd.sql` file. If you want to inject this directly into the database, remove the `text` and `output` flags.

■ **Caution** You can lose all of your data if you run the `SchemaExport` tool with the wrong flags. Although you can use the `text` flag to prevent it running against the database directly, it is too easy to omit this. We recommend that you *never* run the tool directly on a machine containing data that you value, because the selection of the wrong flags will result in your tables being dropped—along with any data they may have contained—and recreated pristine, but empty.

Once you have run the generated SQL script to create the database, you can run the sample application, although there is not yet any data in the database:

```
java HibernateMotd 1
```

This produces (after some logging information)

```
No such message
```

Add some data to the database directly. For example, run the following SQL through `DatabaseManager` if you're using HSQL:

```
insert into motd (id,message) values (1,'Hello World')
```

Run the `HibernateMotd` command again, and you should now see the Hello World message after the other logging messages.

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)

Creating the Hibernate configuration file is generally a matter of copying and editing a configuration file from an existing project.

Writing the POJOs can be a time-consuming task. For new projects, these can be generated directly from the mapping files, while, for older projects, a large part of Hibernate's appeal may be the chance to map our existing data classes without much extra effort!

The mapping file contains the relationship between the application and the database. Obviously, this must be expressed somewhere, and the provision of comprehensive options with sensible default values helps to keep the creation of simple mappings brief without constraining Hibernate's flexibility in more complicated situations.

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.

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, multiple 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.

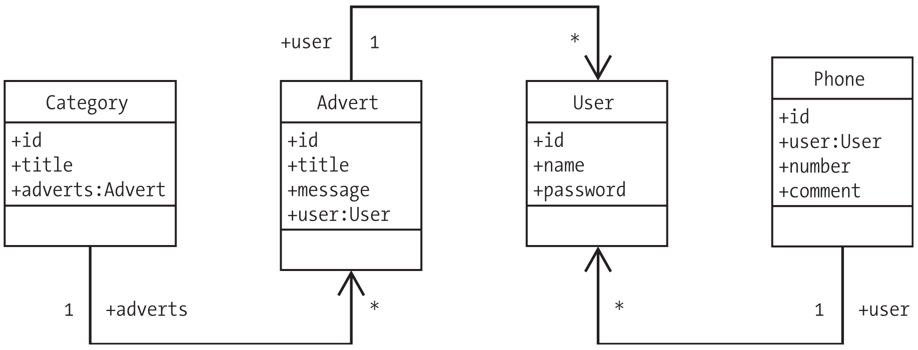

Figure 3-1. *The online billboard classes*

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 creation of the mapping files.

From the class diagram, we know that four classes will be persisted into the database (see Listings 3-9 through 3-12). **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 a numeric integer value, with the data type depending on the number of objects expected (i.e., 32 bit, 16 bit, 64 bit, whatever). 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 become nonunique, 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 follows:

Listing 3-9. *The Class Representing Users*

```
package com.hibernatebook.chapter3;

public class User {
   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;
   }

   private long id;
   private String name;
   private String password;
}
```

Listing 3-10. *The Class Representing Phone Numbers (Associated with Users)*

```
package com.hibernatebook.chapter3;

public class Phone {
   public Phone(User user, String number, String comment) {
      this.user = user;
      this.number = number;
      this.comment = comment;
   }

   Phone() {
   }

   public String getComment() {
      return comment;
   }

   public void setComment(String comment) {
      this.comment = comment;
   }

   public String getNumber() {
      return number;
   }

   public void setNumber(String number) {
      this.number = number;
   }

   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;
   }

   private long id;
   private User user;
   private String number;
   private String comment;
}
```

Listing 3-11. *The Class Representing Categories (Each Having an Associated Set of Advert Objects)*

```
package com.hibernatebook.chapter3;

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

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

   Category() {
   }

   public Set getAdverts() {
      return adverts;
   }

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

   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;
   }

   private long id;
   private String title;
   private Set adverts;
}
```

Listing 3-12. *The Class Representing Advert (Each with an Associated User Who Placed the Advert)*

```
package com.hibernatebook.chapter3;

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

We have not had to add any unusual features to these classes in order to support the Hibernate tool. 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, the table in which we would like to store it is specified (we used `aduser` because `user` is a keyword in many databases), and we specify that we would like to use lazy loading—this class should only be loaded from the database if the application needs to manipulate one of its values.

The class has three fields.

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>` tag. 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`, and 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 3-13 should be extremely easy to follow.

Listing 3-13. *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" lazy="true">

      <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 Phone class represents a user's phone numbers. Because any given user may have more than one phone number, there is a many-to-one relationship between the user table and the phone table. From our class diagram we can see that this relationship is directional—the User class does not know what it's phone numbers are, but the Phone class knows which User it is owned by, so the mapping between the two is required for Phone, but not for User.

In the many-to-one tag, we represent the name of the field, user. We specify the column that will store the relationship, aduser, the fully qualified name of the associated class, and we specify that the column cannot contain nulls (i.e., a Phone object must be associated with a User object, it cannot be unowned).

If that seems complicated, all we are really doing is specifying that the Phone table's aduser column is a foreign key onto the User table, and this will be expressed in the SQL generated from this mapping.

The other fields are similar to those in the User mapping (see Listing 3-14).

Listing 3-14. *The Mapping of the Phone 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.Phone" table="phone">

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

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

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

      <many-to-one
         name="user"
         column="aduser"
         class="com.hibernatebook.chapter3.User"
         not-null="true"/>

   </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 tag indicates that the field in question has 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 in Listing 3-14, the need for each field in the mapping becomes clear (see Listing 3-15).

Listing 3-15. *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="com.hibernatebook.chapter3.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="id"/>
         <many-to-many class="com.hibernatebook.chapter3.Advert" column="advert"/>
      </set>

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

Finally, we represent the Advert class (see Listing 3-16). This class does not introduce anything that we have not encountered in the first three mappings.

Listing 3-16. *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="com.hibernatebook.chapter3.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="com.hibernatebook.chapter3.User"
         not-null="true"/>

   </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 with the Chapter 1 example, the simplest thing to do is include them directly within this.

For our example, take the configuration file used for Chapter 1 and substitute for this line

```
<mapping resource="book/hibernate/gettingstarted/Motd.hbm.xml"/>
```

the following four mapping resource entries:

```
<mapping resource="com/hibernate/chapter3/Advert.hbm.xml"/>
<mapping resource="com/hibernate/chapter3/Category.hbm.xml"/>
<mapping resource="com/hibernate/chapter3/Phone.hbm.xml"/>
<mapping resource="com/hibernate/chapter3/User.hbm.xml"/>
```

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 `SchemaExport` tool.

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—but if not, your mappings may be wrong, or there may be a subtle error in the way that you have related your data types.

The script in Listing 3-17 is generated by the `SchemaExport`. Aside from the generated foreign key names, it could easily have been written by hand, and it is easy to compare it against your prior expectations of the database schema.

Listing 3-17. *The Script Generated by Our Run of the SchemaExport Tool in HSQL's Dialect of SQL*

```
alter table phone drop constraint FK65B3D6EAB3E2E6E
alter table link_category_advert drop constraint FKA7C387F0D1B
alter table link_category_advert drop constraint FKA7C387F0AB3E6FD4
alter table advert drop constraint FKAB3E6FD4AB3E2E6E

drop table phone if exists
drop table link_category_advert if exists
drop table aduser if exists
drop table category if exists
drop table advert if exists

create table phone (
   id bigint generated by default as identity (start with 1),
   number varchar(255),
   comment varchar(255),
   aduser bigint not null,
   primary key (id)
)

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

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

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)
)

alter table phone add constraint FK65B3D6EAB3E2E6E
   foreign key (aduser) references aduser

alter table link_category_advert add constraint FKA7C387F0D1B
   foreign key (id) references category

alter table link_category_advert add constraint FKA7C387F0AB3E6FD4
   foreign key (advert) references advert

alter table advert add constraint FKAB3E6FD4AB3E2E6E
   foreign key (aduser) references aduser
```

Note the foreign key constraints and the link table representing the `many-to-one` and `many-to-many` relationships.

Sessions

In Chapter 6, we will look at the full lifecycle of persistence objects in detail, but we do need to understand the basics of the relationship between the session and the persistence objects if we 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 the `Phone`'s `getUser()` method, even if the associated `User` object has not yet been loaded (as a result of lazy loading).

An object which 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. Unlike JDBC, however, Hibernate can 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 where possible.

We will discuss all of this in much greater detail in Chapter 8, but for now it suffices that we need to maintain a single `SessionFactory` for the entire application, but that the session itself contains information that pertains only to the current thread of execution. Although a session is lightweight, it is a stateful object. An object that has been loaded using a session should be manipulated through the same session where possible. In a multi-threaded environment, such as a Servlet, we will therefore want to retain each thread's session for the duration of that thread. The pattern in Listing 3-18 provides an efficient way in which a thread can retrieve and (if necessary) create its session with minimal impact on the clarity of the code doing so.

Listing 3-18. *The HibernateHelper Used to Manage the Session in Our Example*

```
package com.hibernatebook.chapter3.dao;

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

public class HibernateHelper {

   private HibernateHelper() {
   }

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

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

Using the Session

The most common use case for our POJOs will be to create them, and to 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 3-19.

Listing 3-19. *Creating a User Object and Reflecting This in the Database*

```
Transaction tx = null;
try {
   tx = HibernateHelper.getSession().beginTransaction();
   User user = new User(username, password);
   HibernateHelper.getSession().save(user);
   tx.commit();
} catch( HibernateException e ) {
   tx.rollback();
}
```

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` with that username has already been created in the database), then a Hibernate exception will be thrown, and the entire transaction is 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

```
from User u where u.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 u where user.name = ?
```

The complete code to retrieve a user for a specific username is shown in Listing 3-20.

Listing 3-20. *Retrieving a User Object from the Database*

```
Transaction tx = null;
User user = null;
String username="dminter";
try {
   tx = HibernateHelper.getSession().beginTransaction();
   Query q = HibernateHelper.getSession().createQuery(
      "from User u where u.name= :username");
   q.setString("username", username);
   List results = q.list();

   if (results.size() == 1) {
      user = (User) results.get(0);
   }
   tx.commit();
} catch( HibernateException e ) {
   tx.rollback();
}
```

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

The logic to delete a user from the database (Listing 3-21) is no more complicated than the code to save one.

Listing 3-21. *Deleting a User Object and Reflecting This in the Database*

```
User user = getUser(username);
Transaction tx = null;
try {
   tx = HibernateHelper.getSession().beginTransaction();
   HibernateHelper.getSession().update(user);
   HibernateHelper.getSession().delete(user);
   tx.commit();
} catch (HibernateException e) {
   tx.rollback();
}
```

We acquire the `User` object that we want to delete, begin a transaction, and then update the user from the session. This will reassociate the `User` object with the session, so that subsequent changes to it are reflected in the database, and it will update the object with the current state of the data in the database. We then 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 will want to perform on our data, so now we will take a look at the architecture we are going to use to do this.

Building Data Access Objects (DAOs)

The Data Access Object 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.

First, we will provide a base class. This is very simple, but in some circumstances you will want to populate this with logic to acquire the session, or perform other housekeeping tasks, as the `DAO`s are instantiated. For our example in Listing 3-22, we are merely using it to instantiate a `Logger` object.

Listing 3-22. *The Base DAO Class for Our Example*

```
package com.hibernatebook.chapter3.dao;

import java.util.logging.Logger;


protected class DAO {
   public DAO() {
   }

   public static final Logger log = Logger.getAnonymousLogger();
}
```

To help encapsulate the specifics of the database operations that are being carried out, we catch any Hibernate exception that is thrown and wrap it in a business `AdException` instance, as shown in Listing 3-23.

Listing 3-23. *The AdException Class for Our Example*

```
package com.hibernatebook.chapter3;

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 we require to retrieve an existing `User` object, delete an existing `User` object, or create a new `User` object (see Listing 3-24). Changes to the object in question will be persisted to the database at the end of the transaction.

Listing 3-24. *The UserDAO Class for Our Example*

```
package com.hibernatebook.chapter3.dao;

import java.util.List;
import java.util.logging.Level;

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

import com.hibernatebook.chapter3.AdException;
import com.hibernatebook.chapter3.User;
```

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

   public User getUser(String username) throws AdException {
      try {
         Session session = HibernateHelper.getSession();

         Query q = session.createQuery("from User u where u.name= :username");
         q.setString("username", username);
         List results = q.list();

         User user = null;
         if (results.size() == 1) {
            user = (User) results.get(0);
         }

         return user;
      } catch (HibernateException e) {
         log.log(Level.SEVERE,
            "The DAO could not retrieve the named user", e);
         throw new AdException(
            "The DAO could not retrieve the named user", e);
      }
   }

   public User createUser(String username, String password) throws AdException {
      try {
         User user = new User(username, password);
         Session session = HibernateHelper.getSession();
         session.save(user);
         return user;
      } catch (HibernateException e) {
         log.log(Level.SEVERE,
            "The DAO Could not create the user", e);
         throw new AdException(
            "The DAO Could not create the user", e);
      }
   }

   public void deleteUser(String username) throws AdException {
      try {
         User user = getUser(username);
         HibernateHelper.getSession().update(user);
         HibernateHelper.getSession().delete(user);
      } catch (HibernateException e) {
         log.log(Level.SEVERE,
            "The DAO could not delete the named user", e);
         throw new AdException(
            "The DAO could not delete the named user", e);
      }
   }
}
```

The PhoneDAO provides all the methods we require to retrieve all of the Phone objects associated with a specific User, delete an existing Phone object, or create a new Phone object (see Listing 3-25). Changes to the object in question will be persisted to the database at the end of the transaction.

Listing 3-25. *The PhoneDAO Class for Our Example*

```
package com.hibernatebook.chapter3.dao;

import java.util.List;
import java.util.logging.Level;

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

import com.hibernatebook.chapter3.AdException;
import com.hibernatebook.chapter3.Phone;
import com.hibernatebook.chapter3.User;

public class PhoneDAO extends DAO {
   public PhoneDAO() {
   }

   public List getPhone(User user)
      throws AdException
   {
      try {
         Session session = HibernateHelper.getSession();
         Query q = session.createQuery("from Phone p where p.user= :user");
         q.setEntity("user",user);
         List results = q.list();
         return results;
      } catch ( HibernateException e ) {
         log.log(Level.SEVERE, "…", e);
         throw new AdException("…",e);
      }
   }

   public Phone createPhone(String comment, String number, User user)
      throws AdException
   {
      try {
         Phone phone = new Phone(user,number,comment);
         HibernateHelper.getSession().save(phone);
         return phone;
      } catch ( HibernateException e ) {
         log.log(Level.SEVERE, "…", e);
         throw new AdException("…",e);
      }
   }

   public void deletePhone(Phone phone)
      throws AdException
   {
      try {
         HibernateHelper.getSession().update(phone);
         HibernateHelper.getSession().delete(phone);
      } catch ( HibernateException e ) {
         log.log(Level.SEVERE, "…", e);
         throw new AdException("…",e);
      }
   }
}
```

The CategoryDAO provides all the methods we require to retrieve all of the Category objects, delete an existing Category object, or create a new Category object (see Listing 3-26). Changes to the object in question will be persisted to the database at the end of the transaction.

Listing 3-26. *The CategoryDAO Class for Our Example*

```
package com.hibernatebook.chapter3.dao;

import java.util.List;
import java.util.logging.Level;

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

import com.hibernatebook.chapter3.AdException;
import com.hibernatebook.chapter3.Category;


public class CategoryDAO extends DAO {
   public CategoryDAO() {
   }

   public List getAllCategories()
      throws AdException
```

```
    {
       try {
          Query q = HibernateHelper.getSession().createQuery("from Category");
          List list = q.list();
          return list;
       } catch ( HibernateException e ) {
          log.log(Level.SEVERE, "…", e);
          throw new AdException("…",e);
       }
    }

    public Category createCategory(String title)
       throws AdException
    {
       try {
          Category category = new Category(title);
          HibernateHelper.getSession().save(category);
          return category;
       } catch ( HibernateException e ) {
          log.log(Level.SEVERE, "…", e);
          throw new AdException("…",e);
       }
    }

    public void deleteCategory(Category category)
       throws AdException
    {
       try {
          HibernateHelper.getSession().update(category);
          HibernateHelper.getSession().delete(category);
       } catch ( HibernateException e ) {
          log.log(Level.SEVERE, "…", e);
          throw new AdException("…",e);
       }
    }

    public void update(Category category)
       throws AdException
    {
       try {
          HibernateHelper.getSession().update(category); //Attach user
       } catch ( HibernateException e ) {
          log.log(Level.SEVERE, "…", e);
          throw new AdException("…",e);
       }
    }
 }
```

The AdvertDAO provides all the methods we require 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 3-27).

Listing 3-27. *The AdvertDAO Class for Our Example*

```
package com.hibernatebook.chapter3.dao;

import java.util.logging.Level;

import org.hibernate.HibernateException;

import com.hibernatebook.chapter3.AdException;
import com.hibernatebook.chapter3.Advert;
import com.hibernatebook.chapter3.User;


public class AdvertDAO extends DAO {
   public AdvertDAO() {
   }

   public Advert createAdvert(String title, String message, User user)
      throws AdException
   {
      try {
         Advert advert = new Advert(title,message,user);
         HibernateHelper.getSession().save(advert);
         return advert;
      } catch ( HibernateException e ) {
         log.log(Level.SEVERE, "…", e);
         throw new AdException("…",e);
      }
   }

   public void deleteAdvert(Advert advert)
      throws AdException
   {
      try {
         HibernateHelper.getSession().update(advert);
         HibernateHelper.getSession().delete(advert);
      } catch ( HibernateException e ) {
         log.log(Level.SEVERE, "…", e);
         throw new AdException("…",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 connected approach, you will see that Hibernate's logic is admirably compact.

The Example Client

Listing 3-28 shows the example code tying this together. Of course, this isn't a full application, but you now have all of the DAOs necessary to manage the advertisement database, and this example gives a flavor of how they can be used.

The code should first be run with the single parameter `init` to initialize the tables with some data. It should then be run without the parameter to extract and display all of this information from the tables.

Listing 3-28. *The Test Harness for Our Example*

```
package com.hibernatebook.chapter3;

import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Transaction;

import com.hibernatebook.chapter3.AdException;
import com.hibernatebook.chapter3.Advert;
import com.hibernatebook.chapter3.Category;
import com.hibernatebook.chapter3.Phone;
import com.hibernatebook.chapter3.User;
import com.hibernatebook.chapter3.dao.AdvertDAO;
import com.hibernatebook.chapter3.dao.CategoryDAO;
import com.hibernatebook.chapter3.dao.HibernateHelper;
import com.hibernatebook.chapter3.dao.PhoneDAO;
import com.hibernatebook.chapter3.dao.UserDAO;

public class TestChapter3 {

   public static void main(String[] args) {
      Transaction tx = HibernateHelper.getSession().beginTransaction();
      TestChapter3 tc3 = new TestChapter3();
      try {
         if ((args.length == 1) && args[0].equals("init")) {
            tc3.populateDatabase();
            tx.commit();
            System.out.println("Test data created and committed");
         } else {
            tc3.displayDatabase();
            tx.commit();
         }
```

```
        } catch (AdException e) {
            tx.rollback();
        } catch (HibernateException e) {
            tx.rollback();
        }
    }

    public void displayDatabase() throws AdException {
        Iterator ci = cdao.getAllCategories().iterator();
        while (ci.hasNext()) {
            Category category = (Category) ci.next();

            System.out.println("========");
            System.out.println("Category: " + category.getTitle());

            Iterator ai = category.getAdverts().iterator();
            while (ai.hasNext()) {
                Advert ad = (Advert) ai.next();
                System.out.println("  " + ad.getTitle());
                System.out.println("  " + ad.getMessage());
                System.out.println(" Contact: " + ad.getUser().getName());
                Iterator pi = pdao.getPhone(ad.getUser()).iterator();
                while (pi.hasNext()) {
                    Phone phone = (Phone) pi.next();
                    System.out.println(" " + phone.getComment() + ": "
                            + phone.getNumber());
                }
                System.out.println("--------");
            }
        }
    }

    public void populateDatabase() throws AdException {
        Category cat1 = cdao.createCategory("Computing");
        Category cat2 = cdao.createCategory("Instruments");

        User dave = udao.createUser("dminter", "london");
        pdao.createPhone("Mobile", "07973 000 000", dave);
        pdao.createPhone("Home", "0208 000 000", dave);
        pdao.createPhone("Work", "0207 000 000", dave);

        User jeff = udao.createUser("jlinwood", "austin");
        pdao.createPhone("Cell", "555 000 001", jeff);
        pdao.createPhone("Home", "555 000 002", jeff);
        pdao.createPhone("Work", "555 000 003", jeff);
```

```
cat1.getAdverts().add(
      adao.createAdvert(
            "Sinclair Spectrum for Sale",
            "48k, original box and packaging.",
            dave));
cat1.getAdverts().add(
      adao.createAdvert(
            "IBM PC for sale",
            "Original, not clone. 640Kb.",
            dave));
cat1.getAdverts().add(
      adao.createAdvert(
            "Apple II for sale",
            "Complete with paddles. Call after 5pm",
            dave));
cat2.getAdverts().add(
      adao.createAdvert(
            "Elderly baby Grand Piano for sale",
            "Overstrung. Badly out of tune.",
            dave));
cat2.getAdverts().add(
      adao.createAdvert(
            "Trombone for sale",
            "Slide missing. £1 + £30 p&p",
            dave));
cat2.getAdverts().add(
      adao.createAdvert(
            "Marimba wanted",
            "Will offer up to £100",
            dave));

cat1.getAdverts().add(
      adao.createAdvert(
            "Atari 2600 wanted",
            "Will pay up to $10",
            jeff));
cat1.getAdverts().add(
      adao.createAdvert(
            "Timex 2000 for sale",
            "Some games, $30",
            jeff));
cat1.getAdverts().add(
      adao.createAdvert(
            "Laptop",
            "Unwanted gift from disgruntled author. Otherwise good condition",
            jeff));
      cat2.getAdverts().add(
            adao.createAdvert(
                  "Piccolo",
                  "Tarnished but good sound.",
                  jeff));
      cat2.getAdverts().add(
            adao.createAdvert(
                  "Slightly used triangle",
                  "Not quite triangular anymore. $1",
                  jeff));
      cat2.getAdverts().add(
            adao.createAdvert(
                  "Timpani set",
                  "$30 total, call for postage",
                  jeff));

      cdao.update(cat1);
      cdao.update(cat2);
   }

   private final UserDAO udao = new UserDAO();
   private final PhoneDAO pdao = new PhoneDAO();
   private final AdvertDAO adao = new AdvertDAO();
   private final CategoryDAO cdao = new CategoryDAO();
}
```

When you run our example with the `init` parameter you should see the following (after the initial logging output):

```
Test data created and committed
```

Since we have successfully committed the transaction, we know that the database has been updated successfully. If you were to run it again with the `init` parameter, the transaction will fail as soon as we try to add a user called "dminter" again—the unique constraint on the `aduser` table prevents this:

```
SEVERE: ERROR: duplicate key violates unique constraint "category_title_key"
Jan 30, 2005 2:43:44 PM
   org.hibernate.event.AbstractFlushingEventListener performExecutions
SEVERE: Could not synchronize database state with session
org.hibernate.exception.ConstraintViolationException:
   Could not execute JDBC batch update
```

When you run our example without the `init` parameter, the output should look like this (after the initial logging output):

```
========
Category: Computing
  Apple II for sale
  Complete with paddles. Call after 5pm
 Contact: dminter
 Mobile: 07973 000 000
 Home: 0208 000 000
 Work: 0207 000 000
--------
  Atari 2600 wanted
  Will pay up to $10
 Contact: jlinwood
 Cell: 555 000 001
 Home: 555 000 002
 Work: 555 000 003
--------
```

. . .and so forth. It is easy to see how this logic could now be incorporated into a JSP, Servlet, or even an EJB Session Bean.

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 `SchemaExport` tool.

In the next chapter, we look at how the new Annotations feature of J2SE 5.0 can be used to add mapping metadata to your Java source files.

CHAPTER 4

Using Annotations with Hibernate

In this chapter, we are going to discuss a common set of circumstances for Hibernate development. One scenario is that you already have a Java object model. This could be because you used a requirements-driven development process that modeled all of the Java objects out, or it could be because you used an agile development process and wrote most of your code without a database backend. In either case, you will need to create Hibernate mappings and a database schema for your application's persistence. Using Hibernate 3's new annotations feature, we can add the metadata Hibernate requires for object-relational mapping directly to our Java classes to maintain our object model and our mapping information in one set of source code. This helps ensure that our development process keeps the object-relational mappings up to date with the object model.

A less likely scenario would be if you already have a relational database schema and a Java object model—in this case, you would need to create the Hibernate mapping documents by hand. If you are just replacing an existing Java persistence layer with Hibernate (for instance, because it is homegrown and difficult to maintain), then this should be an easy project because you already understand which database tables and columns map to your Java objects and properties. Another scenario is that one team developed the Java object model and another team developed the database schema, and now you have to create Hibernate mapping documents that mesh the two. For scenarios where you are creating a mapping layer for a Java object model that already exists and a database schema that already exists, refer to our chapter on mapping files (Chapter 6) for directions on how to do this manually.

The last scenario is to build the Java object model for an existing database schema. The Hibernate tools project contains a reverse-engineering wizard that will connect to a database over JDBC and create Hibernate mapping documents, a Hibernate XML configuration file, and the Java classes for your object model. At the time of writing, these tools were still in an alpha state, so check the Hibernate tools web page (`http://www.hibernate.org/255.html`) for documentation and downloads.

Creating Hibernate Mappings with Annotations

The first scenario we discuss begins with a Java object model. Say your development team created a Java object model and you are ready to add persistence to your project. As you know from previous chapters, Hibernate requires metadata to describe the object-relational mapping for your application.

Previous versions of Hibernate used XML files to describe these mappings, and this model continues to work with Hibernate 3. If you wanted to create mapping documents from your POJO source code, you could annotate your Java classes with special Hibernate-specific Java doclet tags and use the Open Source XDoclet tool to generate XML mapping documents. Typically, you would run XDoclet through an Ant build file during your build process, and then include the generated Hibernate XML mapping files in your target distribution. The XDoclet solution works well for many developers. If you are interested in learning more about XDoclet, you should visit the XDoclet web page (`http://xdoclet.sourceforge.net/xdoclet/index.html`). In this book, we are going to cover the Hibernate 3 Java annotations instead of XDoclet, but both use the same attribute-oriented style for providing Hibernate mappings. One point to consider is that using XDoclet requires an extra step (besides compilation) to generate Hibernate XML mapping documents, whereas using Hibernate 3's annotations only requires compilation. Hibernate 3 works directly with the annotated compiled Java class files, there is no intermediate step to generate XML mappings.

■**Note** At the time of this writing, Hibernate 3's annotations support is still in beta. Consult the Hibernate Annotations web page (`http://www.hibernate.org/247.html`) for any changes or improvements from the beta to the production release.

The J2SE 5.0 platform now supports annotations directly inside your Java source code or compiled classes. You do not need to run a third-party tool like XDoclet to take advantage of source-level metadata, and Java code can access annotations at runtime through the reflection API, which opens up many more possibilities. To learn more about J2SE 5.0 annotations, see Sun's Java Language website (`http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`) or *Pro Java Programming, Second Edition* by Brett Spell from Apress.

The Enterprise JavaBeans 3 (EJB3) expert group used the J2SE 5.0 annotations to create a new metadata model for persistence. The new EJB3 specification uses annotations and POJO, similar to the Hibernate 2/XDoclet solution. Although you could use XDoclet to mark up POJO for EJB2 Entity Beans, the model was still a maintenance nightmare. The Hibernate team provided design input for a much cleaner EJB3. Hibernate 3 uses the standard EJB3 annotations, along with several Hibernate-specific annotations. Beyond Hibernate, the EJB3 annotations specification goes further allowing you to specify the transaction settings, security attributes, JNDI resources, and anything else defined through deployment descriptors or interfaces in EJB 2.1. This should lead to a much cleaner development process for EJB3 applications.

In short, with Hibernate 3 there are three different development styles for creating mappings:

- Creating the XML mapping documents by hand
- Annotating the POJO with Hibernate XDoclet tags and generating XML mapping documents at buildtime
- Annotating the POJO with Hibernate 3/EJB3 annotations (J2SE 5.0 and higher)

You can mix and match these if you absolutely need to, although we strongly believe that this will lead to a lot of confusion if different developers on a team are using different ways to do the same thing. You will have to determine which of these works best for you—some developers prefer to have all of the information about an object in one source file, and others prefer to keep their Java source code plain and uncluttered.

At the time of this writing, the annotations support in Hibernate 3 was not completely mature, because the EJB3 specification had not been finalized. The support for annotations should improve in future releases of the Hibernate 3 annotations toolset. One limitation you will run into with configuration is that there is no way to override annotations through XML mapping documents at the present time, although the Hibernate team (or other developers) will probably add this functionality in a future release of the annotations toolset. Another limitation is that the currently released version of the database schema export Ant task does not support annotations yet.

Using Annotations in Your Application

You will need to install the Hibernate 3 annotations toolset, available from the Hibernate home page (`http://www.hibernate.org`). If you do not already use JDK 5.0, you will need to upgrade to take advantage of the native support for annotations. Use XDoclet instead of annotations for metadata support if you do not want to upgrade to JDK 5.0 yet. Your application needs the `hibernate-annotations.jar` from the toolset and the `ejb-3.0-edr2.jar` EJB3 `.jar` file, although your application server may already provide its own EJB3 `.jar` file. At the time of this writing, the Hibernate 3 annotations toolset was still in beta, and the EJB3 specification was not final—specific `.jar` file names may have changed. Read the release notes for the Hibernate 3 annotations toolset to determine the exact deployment requirements.

After you set up the annotations in your Java objects, the only thing you need to change in your application is the way you add mappings to the configuration. With Hibernate annotations, you can either add annotated classes to the `org.hibernate.cfg.AnnotationConfiguration` object, or you can specify annotated classes in the Hibernate XML configuration file. Both accomplish the same thing, as we show in the next example.

Introducing the Hibernate Annotations

In Chapter 3, we walked through an example that used Hibernate's XML mapping documents to provide metadata for our POJOs. In this section, we use the same mapping concepts from Chapter 3, but we use annotations instead of an XML mapping file. For a more in-depth look at mapping, read Chapter 6.

When you develop using annotations, you start with a Java class and then annotate the source code listing with metadata notations. In J2SE 5.0, the Java Runtime Environment (JRE) parses these annotations. Hibernate uses Java reflection to read the annotations and treat them like a mapping. In this section, we are going to introduce the basic annotations, starting with a single class, `Book`, that has no annotations or mapping information. For this example's purposes, we do not have an existing database schema to work with, so we need to define our relational database schema as we go.

At the beginning of our example, the `Book` class is very simple. It has two fields, `title` and `pages`, and an identifier `id`, which is an integer. The title is a `String` object, and `pages` is an integer. As we go through this example, we will add annotations, fields, and methods to the `Book` class. The complete source code listing for the completed `Book.java` is at the end of this chapter.

Listing 4-1 is the source code for `Book.java`, in its un-annotated form, as a starting point for our example.

Listing 4-1. *The Book Class, Un-annotated*

```
package com.hibernatebook.annotations;

public class Book
{
    protected String title;
    protected int pages;
    protected int id;

    public int getPages()
    {
        return pages;
    }
    public void setPages(int pages)
    {
        this.pages = pages;
    }
    public String getTitle()
    {
        return title;
    }
    public void setTitle(String title)
    {
        this.title = title;
    }
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
}
```

As you can see, this is a POJO, just like our `AnnotationConfiguration` previous examples. We are going to annotate this class as we go along, explaining the concepts behind annotation.

Entity Beans with @Entity

The first step is to annotate the `Book` class as an EJB3 Entity Bean. Instead of implementing an EJB interface to mark the class as an Entity Bean, we add the `@Entity` annotation to the `Book` class:

```
package com.hibernatebook.annotations;
import javax.persistence.*;

@Entity
public class Book
```

Notice that we also imported the classes from the `javax.persistence` package. Part of the J2SE 5.0 annotations specification requires that we import the annotations into the source code, so the compiler knows which annotations our markers refer to. After we add the annotation to mark this class as an Entity Bean, we need to map the identifier for the class.

The `@Entity` annotation marks this class as an Entity Bean, so it has to have a no-argument constructor that is at least protected scope. Other EJB3 rules for an Entity Bean class are that the class may not be final, and the Entity Bean class has to be concrete. Many of the rules for EJB3 Entity Bean classes and Hibernate 3 persistent objects are the same—this is because the Hibernate team had much input into the EJB3 design process, and there are only so many ways to design a relatively unobtrusive object-relational persistence solution.

The instance variables on the POJO need to be hidden behind getter/setter accessor methods, and not exposed as public fields on the class. For Entity Beans, the EJB3 specification provides two different ways for the persistence container (in this case, Hibernate) to access the contents of the instance variables for persistence. The container can either go through the accessor methods you created or it can access the instance variables directly. For each Entity Bean, you may choose one access strategy or the other by providing a value named `access` for the `@Entity` annotation. The access strategy you choose also determines where in your class the annotations go. For accessor methods, use the value `AccessType.PROPERTY` for `access`, and annotate the getter methods for instance variables. To make the container use the fields on the Entity Bean directly, first give all of your persistent instance variables protected access. The value of the `access` attribute will need to be `AccessType.FIELD`, and you should annotate the instance variables in the class. For our `Book` class, we will explicitly choose to have the persistence container use the accessor methods:

```
package com.hibernatebook.annotations;
import javax.persistence.*;

@Entity (access=AccessType.PROPERTY)
public class Book
```

As you can see, although we did have to add the import statement and the annotations, this is still just a POJO that we could use for any purpose we want.

Primary Keys with @Id

Each Entity Bean has to have a primary key, which you annotate on the class with the `@Id` annotation. The primary key may either be a single field or a composite field. We discuss primary key mapping in more detail in Chapter 6 if you want to brush up on the fundamental concepts behind identity generation and primary keys.

The `@Id` annotation takes two attribute values: `generate` and `generator` (which is not very useful). The `generate` attribute value defines the generator type to use, from the javax.persistence.GeneratorType enumeration. If you do not specify a generator type, the default is NONE, and your application should provide functionality for primary key generation. There are five different types of primary key generators for EJB3 on GeneratorType:

- NONE: No primary key gets generated; the application is responsible for creating unique primary keys.
- AUTO: Hibernate decides which generator type to use, based on the database's support for primary key generation.
- IDENTITY: Database is responsible for determining the next primary key.
- SEQUENCE: Some databases support a SEQUENCE column type, which you can name with the optional generator attribute value.
- TABLE: Keeps a separate table with the primary key values. The name of this table also comes from the optional generator attribute value.

You will notice that the available values for the generate attribute do not exactly match the values for Hibernate's primary key generators for XML mapping.

For our Book class, we are going to use the most portable form of primary key generation, AUTO. Letting Hibernate determine which generator type to use makes our code portable between different databases. Remember that because we used the accessor methods for our container access, we need to annotate the getter method for the identifier and any other instance variables:

```
@Id (generate=GeneratorType.AUTO)
public int getId()
{
    return id;
}
```

In addition to defining the class as an Entity Bean and annotating the identifier, we can define which database tables to use for storage.

Database Table Mapping with @Table and @SecondaryTable

The @Table annotation provides the name of the table in the database for the Entity Bean. This annotation is entirely optional; like XML mapping, Hibernate will default to using the same name for the database table as the class. To use this annotation, provide a value for the name attribute, for an annotation that looks like @Table(name="RS_ORDER_HISTORY"). The other attributes are catalog, schema, and uniqueConstraints. The catalog and schema attributes describe which database catalog and schema to use, respectively. The uniqueConstraints attribute should be an array of javax.persistence.UniqueConstraints objects for setting the unique constraints during generation, but it is not supported in Hibernate 3's annotations at the time of this writing.

The @SecondaryTable annotation provides a way to model an Entity Bean that consists of data from several different database tables. In addition to providing an @Table annotation for the primary database table, your Entity Bean can have zero or more @SecondaryTable annotations. The @SecondaryTable annotation has the same attributes as the @Table annotation, plus the join attribute. The join attribute is the join column for the primary database table, and it accepts an array of javax.persistence.JoinColumn objects.

To use secondary tables, you will need to use the secondaryTable attribute for the @Column annotation on each instance variable that gets mapped to a secondary database table. The value of the secondaryTable attribute should be the name of that secondary database table.

Field Persistence with @Basic and @Transient

Each instance variable in your POJO is persistent by default—Hibernate will store the variable for you. Depending on the type of instance variable, Hibernate will choose a sensible default. Some of your objects may have fields that do not need to be stored to the database because they are transient. The EJB3 specification provides the @Transient annotation for these transient fields. The @Transient annotation does not have any attributes, and you would simply add it to the instance variable or the getter method, depending on the Entity Bean's property access strategy.

The @Basic annotation is the default annotation for instance variables. The only attribute for @Basic is the fetch type, fetch. The only two possible fetch types are eager and lazy—the values are on the javax.persistence.FetchType enumeration. Variables with basic annotation persist to the database using object-relational mapping information from the @Column annotation.

For our example, we can add a Date field named publicationDate to our Book class that we would not want to store in the database. We would mark this field transient with the @Transient annotation:

```
@Transient
public Date getPublicationDate()
{
    return publicationDate;
}
```

We put the @Transient annotation on the getter method for our Book class because we are using the property access strategy for our Entity Bean. We do not have to explicitly define @Basic annotations for the rest of the fields to make the fields persistent.

Object-Relational Mapping Details with @Column

By default, persistent instance variables will use sensible defaults for the mapping of columns in database tables to properties in an object. These defaults work well if you create the database schema at the same time as the object model; but if you map your object model to an existing database, or you need to override Hibernate's defaults, use the @Column annotation to provide specifics about the object-relational mapping for that instance variable.

The name attribute for the @Column annotation provides the name of the column on the database table, if it is different from the name of the instance variable. The length attribute controls the size of the database column. Set the primaryKey attribute to true if this column is the primary key—otherwise, its default value of false does not need to be specified. If the column has to be unique, set the unique attribute to true. The default value for unique is false. By default, the database column may contain nulls—if it cannot have any nulls stored in it, then set the nullable attribute to false. As we discussed in the “Database Table Mapping with @Table and @SecondaryTable” section, you may use the secondaryTable attribute to store this instance variable in another table.

We can specify the database column used to store the title of the book. We will use a database column named working_title, which will have a length of 200 and will not allow null values. The relevant excerpt from our Book class is the following:

```
@Column(name="working_title",length=200,nullable=false)
public String getTitle()
{
    return title;
}
```

You only need to use the @Column annotation if Hibernate’s default database mapping does not work for your application.

Modelling Relationships with Annotations

Using annotations, we can model associations between Entity Beans. EJB3 supports one-to-one, one-to-many, many-to-one, and many-to-many associations. Each of these has a corresponding annotation.

The @ManyToOne annotation defines a many-to-one relationship between two Entity Beans. When you use a many-to-one relationship, annotate the field or getter method for the owner with the @ManyToOne annotation. For instance, in our example, we will add a Publisher class that has a one-to-many relationshop to the Book class. On the Book class, we add a publisher field and then annotate it with @ManyToOne. The @ManyToOne annotation has several attributes, which are targetEntity, cascade, fetch, and optional. The target entity is the name of the Entity Bean, but the default is the class of the annotated field, so the targetEntity attribute is optional. The cascade attribute has several values, ALL, PERSIST, MERGE, REMOVE, and REFRESH. The javax.persistence.CascadeType enumeration contains these values if you would like to use them. You can use more than one cascade type for the attribute. These cascade types are slightly different from the Hibernate 3 cascade types. Not every Hibernate 3 cascade type is available (delete-orphan, evict, lock). The fetch attribute takes the same two possible fetch values as the @Basic annotations, eager or lazy. The default type for fetch is eager. The optional attribute defaults to true.

For many-to-one associations, the database needs a column to contain the foreign key. The EJB3 specification defines an @JoinColumn annotation that you may use to specify the foreign key column. If you do not provide an @JoinColumn annotation for your relationships, Hibernate uses a default naming strategy for the foreign key column: the owner’s entity name, an underscore, and the owner’s primary key database column name. In our example, this would be publisher_id. You may specify a new name for the foreign key column with the name attribute. The EJB3 specification defines a referencedColumnName attribute to allow you to specify another column (besides the primary key) to join on with the owner’s Entity Bean. The rest of the attributes on @JoinColumn work the same way as they do on the @Column annotation (unique, nullable, primaryKey, insertable, updatable, columnDefinition, and secondaryTable).

For our example, we have added the @ManyToOne annotation and the @JoinColumn annotation. We did not need to add the @JoinColumn annotation for our example to work, the defaults would have worked fine. The relevant excerpt from the Book class is as follows:

```
    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="publisher_id")
    public Publisher getPublisher()
    {
        return publisher;
    }
```

Notice that we used the `CascadeType` enumeration to add the `ALL` value to the `cascade` attribute. We are going to add the corresponding `one-to-many` relationship to the `Publisher` class to make the relationship bidirectional.

The `one-to-many` association annotation is `@OneToMany`. From the EJB3 specification, the association needs to be on a `Collection` or a `Set`. There are several attributes for the `@OneToMany` annotation. The first is the target entity for the `one-to-many` relationship, `targetEntity`. The target entity should be the name of the class on the other side of the entity relationship. Using the Java generics feature, we can specify a class for our collection; however, in this case we do not need to use the `targetEntity` attribute. Additionally, we can specify the `cascade` and `fetch` attributes, like the `@ManyToOne` annotation. The default fetch type is lazy. The last attribute is `mappedBy`. If the `one-to-many` relationship is bidirectional (as our example is), we can point Hibernate to the `many-to-one` relationship on the other side with the `mappedBy` attribute. We would specify the name of the property on the child class that points to the owner. In our example, that would be `publisher`. Listing 4-2 contains the whole `Publisher` class. Notice the `@OneToMany` annotation, bolded in this example.

Listing 4-2. *The Publisher Class, Showing the @OneToMany Annotation*

```
package com.hibernatebook.annotations;

import javax.persistence.*;
import java.util.*;

@Entity
public class Publisher
{
    protected String name;
    protected Set <Book> books = new HashSet<Book>();
    protected int id;

    @OneToMany(mappedBy="publisher")
    public Set<Book> getBooks()
    {
        return books;
    }
    public void setBooks(Set<Book> books)
    {
        this.books = books;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    @Id (generate=GeneratorType.AUTO)
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
}
```

Also notice that we used the Java generics feature on the collection for the `Publisher` class. Because we used generics, we did not have to specify an entity type for the `@OneToMany` annotation.

For `one-to-many` relationships that are unidirectional, the Hibernate development team recommends using an association table, similar to a `many-to-many` relationship.

Model `many-to-many` relationships with the `@ManyToMany` annotation. The EJB3 specification specifies that there are two sides of the `many-to-many` relationship: an owning side and a non-owning, or inverse, side. We use an association table annotation, `@AssociationTable`, on one side of the relationship to define the mapping between the two. The non-owning Entity Bean should have a `@ManyToMany` annotation with a `mappedBy` attribute (like `one-to-many`) pointing to the owning Entity Bean. The defaults for the `many-to-many` relationship are usually acceptable, so you do not have to specify an `@AssociationTable` annotation. The attributes for `@ManyToMany` are the same as `@OneToMany`, and they work the same way.

To show a `many-to-many` relationship, we added an `Author` class. A book can have more than one author, and an author can write more than one book. We used generics again to keep our mappings simple. Here is an excerpt from our `Book` class:

```
    @ManyToMany
    public Set<Author> getAuthors()
    {
        return authors;
    }
```

The corresponding code in the `Author` class is as follows:

```
    @ManyToMany(mappedBy="authors")
    public Set<Book> getBooks()
    {
        return books;
    }
```

The @AssociationTable annotation has three optional attributes. The first, table, is an @Table annotation that describes the database table. The joinColumns attribute needs to contain at least one @JoinColumn annotation that describes the foreign key columns owning Entity Bean. Its counterpart is the inverseJoinColumns attribute that also uses @JoinColumn annotations, but to describe the foreign key columns for the non-owning Entity Bean.

Last, we have one-to-one relationships. We model those with the @OneToOne annotation. This annotation has the same attributes as @OneToMany, plus a usePKasFK attribute. The usePKasFK attribute controls whether both Entity Beans have the same primary key values—if they do, this value should be true. It defaults to false where one side has a foreign key that points to the other side's primary key. This annotation can also have a @JoinColumn annotation that describes the database column for the foreign key, just like the other relationship annotations.

Inheritance

The EJB3 Entity Bean model supports inheritance, along with polymorphism. As we discuss in the "Mapping Inheritance Relationships" section of Chapter 6, there are three different ways to map database tables to a hierarchy of Java classes. EJB3 supports all three of these methods (one table for the hierarchy, one table for each class, joins on tables for subclasses), as does Hibernate's implementation of the EJB3 specification. The classes in the hierarchy have to use the same types for their primary keys, according to the EJB3 specification.

The @Inheritance annotation on an Entity Bean marks this class as part of an inheritance hierarchy. The strategy attribute defines which object-relational inheritance mapping strategy to use. From the javax.persistence.InheritanceType enumeration, the three possible strategies are JOINED, SINGLE_TABLE, and TABLE_PER_CLASS.

For the joined inheritance strategy, we would not have to change the base class, but we would need to add the @Inheritance annotation to the subclasses. We could specify the column we use to join the subclass to the base class with the @InheritanceJoinColumn annotation. This annotation has name, referencedColumnName, and columnDefinition attributes. Our ComputerBook class extends our Book class, using the joined inheritance strategy (see Listing 4-3).

Listing 4-3. *The ComputerBook Class Extends the Book Class with the @Inheritance Annotation.*

```
package com.hibernatebook.annotations;

import javax.persistence.*;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@InheritanceJoinColumn(name="BOOK_ID")
public class ComputerBook extends Book
{
    protected String softwareName;

    public String getSoftwareName()
    {
        return softwareName;
    }
    public void setSoftwareName(String softwareName)
    {
        this.softwareName = softwareName;
    }
}
```

Notice that we used the @InheritanceJoinColumn annotation, although we did not have to.

The second type of inheritance is the single table inheritance, where the data for the base class and its subclasses is in one table. What differentiates the data in the table is a discriminator column that contains a unique value for the base class and for each subclass. For instance, if we used this strategy for our Book and ComputerBook example, we could have a column on our Book table that contained either “BOOK” or “COMPUTERBOOK” for each entry, denoting which class it belonged to. We would specify this inheritance strategy with this annotation on the Book class:

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE, discriminator_value="BOOK")
```

Our ComputerBook class would contain this annotation:

```
@Inheritance(discriminator_value="COMPBOOK")
```

The last type of inheritance is the table per class inheritance. We would simply specify the inheritance strategy as TABLE_PER_CLASS.

Using the Annotated Classes in Your Hibernate Configuration

Once you have an annotated class, you will need to provide the class to your application’s Hibernate configuration, just as if it were an XML mapping. With annotations, we can use either the declarative configuration in the hibernate.cfg.xml XML configuration document, or we can programmatically add annotated classes to Hibernate’s org.hibernate.cfg.AnnotationConfiguration object. Your configuration may use both annotated classes and XML mappings in the same configuration as you choose.

To provide declarative mapping, we use a normal hibernate.cfg.xml XML configuration file, and add the annotated classes to the mapping (see Listing 4-4). Notice that we specified the name of the annotated class as a mapping. We could have more than one mapping here, and they could be either annotated classes or XML mappings.

Listing 4-4. *Hibernate XML Configuration File with an Annotated Class*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">➥
org.hsqldb.jdbcDriver</property>
        <property name="hibernate.connection.url">➥
jdbc:hsqldb:file:annotationsdb</property>
```

```
        <property name="hibernate.username">user</property>
        <property name="hibernate.password">user</property>
        <property name="hibernate.show_sql">true</property>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Mapping files -->
        <mapping class="com.hibernatebook.annotations.Book"/>
    </session-factory>
</hibernate-configuration>
```

You can also add an annotated class to your Hibernate configuration programmatically. The annotations toolset comes with an `org.hibernate.cfg.AnnotationConfiguration` object that extends the base Hibernate `Configuration` object for adding mappings we described in Chapter 2. The methods on `AnnotationConfiguration` for adding annotated classes to the configuration are

```
addAnnotatedClass(Class persistentClass) throws MappingException
addAnnotatedClasses(List<Class> classes)
addPackage(String packageName) throws MappingException
```

Using these methods, you can add one annotated class, a list of annotated classes, or an entire package (by name) of annotated classes. As with the Hibernate XML configuration file, the annotated classes work with XML mapping files, and you can use both in an application without problems (as long as the application runs on J2SE 5.0 or higher, of course).

Code Listings

Listing 4-5 concludes the chapter with the source code listings we did not fully include for the `Book` and `Author` classes. The database schema also follows.

Listing 4-5. *The Book Class, Fully Annotated*

```
package com.hibernatebook.annotations;

import java.util.*;

import javax.persistence.*;

@Entity (access=AccessType.PROPERTY)
public class Book
{
    protected String title;

    protected Publisher publisher;
    protected Set <Author> authors = new HashSet<Author>();

    protected int pages;
    protected int id;
    protected Date publicationDate;

    public int getPages()
    {
        return pages;
    }
    public void setPages(int pages)
    {
        this.pages = pages;
    }
    @Column(name="working_title",length=200,nullable=false)
    public String getTitle()
    {
        return title;
    }
    public void setTitle(String title)
    {
        this.title = title;
    }

    @Id (generate=GeneratorType.AUTO)
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }

    @Transient
    public Date getPublicationDate()
    {
        return publicationDate;
    }
    public void setPublicationDate(Date publicationDate)
    {
        this.publicationDate = publicationDate;
    }

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="publisher_id")
    public Publisher getPublisher()
    {
        return publisher;
    }
    public void setPublisher(Publisher publisher)
    {
        this.publisher = publisher;
    }

    @ManyToMany
    public Set<Author> getAuthors()
    {
        return authors;
    }
    public void setAuthors(Set<Author> authors)
    {
        this.authors = authors;
    }
}
```

Listing 4-6 demonstrates the `Author` class having a `many-to-many` relationship with the `Book` class.

Listing 4-6. *The Author Class, Showing a many-to-many Relationship*

```
package com.hibernatebook.annotations;

import java.util.*;
import javax.persistence.*;

@Entity
public class Author
{
    protected String name;
    protected String email;

    protected Set <Book> books = new HashSet<Book>();
    protected int id;

    @ManyToMany(mappedBy="authors")
    public Set<Book> getBooks()
    {
        return books;
    }
    public void setBooks(Set<Book> books)
    {
        this.books = books;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }

    public String getEmail()
    {
        return email;
    }
    public void setEmail(String email)
    {
        this.email = email;
    }

    @Id (generate=GeneratorType.AUTO)
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
}
```

Last, Listing 4-7 shows the database schema for the classes in this chapter, as generated by the `SchemaExport` class for the HSQL database. Notice the foreign keys, some of which we specified in the annotations. Also notice the `Book_Author` `many-to-many` association table, where we relied on the defaults for its creation.

Listing 4-7. *The Database Schema for the Example*

```
create table Author (
   id integer generated by default as identity (start with 1),
   name varchar(255),
   email varchar(255),
   primary key (id)
)
create table Book (
   id integer generated by default as identity (start with 1),
   working_title varchar(200) not null,
   pages integer,
   publisher_id integer,
   primary key (id)
)
```

```
create table Book_Author (
   books_id integer,
   authors_id integer
)

create table ComputerBook (
   BOOK_ID integer not null,
   softwareName varchar(255),
   primary key (BOOK_ID)
)

create table Publisher (
   id integer generated by default as identity (start with 1),
   name varchar(255),
   primary key (id)
)

alter table Book add constraint FK1FAF0990BF1C70
    foreign key (publisher_id) references Publisher

alter table Book_Author add constraint FK1A9A0FA1B629DD87
    foreign key (authors_id) references Author

alter table Book_Author add constraint FK1A9A0FA1D3BA8BC3
    foreign key (books_id) references Book

alter table ComputerBook add constraint FK98D97CC4600B1724
    foreign key (BOOK_ID) references Book
```

Summary

In this chapter, we used EJB3 annotations to add metadata to our POJOs for Hibernate. Instead of using XML mapping documents, we can keep all of the metadata in the Java code. Your development preferences will determine whether you like this method better than using XML mapping documents. The EJB3 specification does not completely match Hibernate's capabilities. We demonstrated how to model `one-to-many` relationships with annotations. We also explained how inheritance works with annotations.

PART 2

Hibernate 3 Reference

CHAPTER 5

The Persistence Lifecycle

In this chapter, we discuss the lifecyle of persistent objects in Hibernate. These persistent objects are Plain Old Java Objects (POJOs) without any special marker interfaces or inheritance related to Hibernate. Part of Hibernate's popularity comes from its ability to work with a normal object model. We also discuss the methods used for creating, retrieving, updating, and deleting persistent objects from Hibernate.

Introduction to the Lifecycle

After adding Hibernate to your application, you do not need to change your existing Java object model to add persistence marker interfaces or any other type of hint for Hibernate. Instead, Hibernate works with normal Java objects that your application creates with the `new` operator, or that other objects create. For Hibernate's purposes, these can be drawn up into two categories: objects Hibernate has entity mappings for and objects which are not recognized by Hibernate. Any object has the potential to be persisted by Hibernate if it contains instance variables that could be stored in a relational database, and if a mapping exists.

Given an instance of an object that is mapped to Hibernate, we have to describe the different states it could be in. *Transient* objects exist in memory. Hibernate does not manage transient objects or persist changes to transient objects. You will have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier. We discuss saving in more detail in the "Saving Objects" section of this chapter.

Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects. If properties change on a persistent object, Hibernate will keep the database representation up to date.

The last state an object could be in is detached. *Detatched* objects have a representation in the database, but the application closes the Hibernate `Session` object that the detached object is associated with. The application will need to reattach a detatched object to a valid Hibernate session to ensure that the database reflects changes to the detatched object. A detached instance can be associated with another Hibernate session when your application calls one of the `load`, `refresh`, `merge`, `update`, or `save` methods on the other session with a reference to the detached object. After the call, the detached object would be a persistent object managed by the new Hibernate session.

Commonly, detached objects are stored in a web application's session. A typical web application would only keep a Hibernate session open while processing one user action at a time. Some objects (such as a user object or an order object) would be stored in the user's web application session, and thus need to be reattached to a Hibernate session if the user makes a change with the next page load.

Prior versions to Hibernate 3 had support for the `Lifecycle` and `Validatable` interfaces. These allowed your objects to listen for `save`, `update`, `delete`, `load`, and `validate` events using methods on the object. In Hibernate 3, this functionality moved into events and interceptors, and these interfaces were removed.

Saving Objects

Creating an instance of a class you mapped with a Hibernate mapping does not automatically persist the object to the database. Until you explicitly save the object with a valid Hibernate session, the object is transient, like any other Java object. In Hibernate, we use one of the `save()` methods on the `Session` interface to store a transient object in the database:

```
public Serializable save(Object object) throws HibernateException
public void save(Object object, Serializable id) throws HibernateException
public Serializable save(String entityName,Object object) throws HibernateException
```

All of the `save()` methods take a transient object (which cannot be null) as an argument. Hibernate expects to find a mapping for the transient object's class—Hibernate cannot persist arbitrary objects to a generalized persistence store. One of the `save()` methods also takes an object `id` as an argument. This `id` also cannot be null. The first `save()` method passes `null` as the `entityName` argument to the last `save()` method. The `save()` methods all create a new `org.hibernate.event.SaveOrUpdateEvent` event. We discuss events in more detail in Chapter 11, although you do not have to worry about these implementation details to use Hibernate effectively.

At its most simple, we create a new object in Java, set a few of its properties, and then save it through the session:

```
Supplier superCorp = new Supplier();
superCorp.setName("SuperCorp");
session.save(superCorp);
```

Referring back to our discussion of the various states that objects may be in, Hibernate provides a way to save any object if you are unaware of whether it is in the transient or the detached state with the `saveOrUpdate()` method. Hibernate looks at the identifier on the object to determine whether to save or update the object to the database. The method signature is

```
public void saveOrUpdate(Object object) throws HibernateException
```

If an object is in the persistent state, Hibernate manages the updates to the database itself when you change a property on the object. We discuss updating in the "Updating Objects" section of this chapter.

Object Equality and Identity

When we discuss persistent objects in Hibernate, we also need to consider the role that object equality and identity plays with Hibernate. When we have a persistent object in Hibernate, that object represents both an instance of a class in a particular Java Virtual Machine (JVM) and a row (or rows) in a database table.

Requesting a persistent object again in the same Hibernate session returns the same Java instance of a class, which means that you can compare the objects using the standard Java `==` equality syntax. If you request a persistent object in more than one Hibernate session, Hibernate will provide an instance for each session, and the `==` operator will return `false` if you compare these object instances.

Taking this into account, if you are comparing objects in two different sessions, you will need to implement the `equals()` method on your Java persistence objects. Like any other Java application where you would create an `equals()` method, you should compare the values of the object's properties.

Loading Objects

Hibernate's `Session` interface provides several `load()` methods for loading objects from your database. Each `load()` method requires the object's primary key as an identifier, which must be `Serializable`. Because of this requirement, you should convert any primitive identifiers to objects. In addition to the `id`, Hibernate also needs to know which class or entity name to use to find the object with that `id`. Last, you will need to cast the `Object` returned by `load()` to the class you desire. The basic `load()` methods are as follows:

```
public Object load(Class theClass, Serializable id) throws HibernateException
public Object load(String entityName, Serializable id) throws HibernateException
public void load(Object object, Serializable id) throws HibernateException
```

The last `load()` method takes an object as an argument. The object should be of the same class as the object you would like loaded and should be empty. Hibernate will populate that object with the object you requested. We find this syntax to be somewhat confusing when put into applications, so we do not tend to use it ourselves.

The other `load()` methods take a lock mode as an argument. The lock mode specifies whether Hibernate should look into cache for the object, and which database lock level Hibernate should use for the row (or rows) of data that represent this object. The Hibernate developers claim that Hibernate will usually pick the correct lock mode for you, although we have seen instances where it is important to manually choose the correct lock. In addition, your database may choose its own locking strategy, for instance locking down an entire table rather than multiple rows within a table. In order, least restrictive to most restrictive, the various lock modes you can use are the following:

- `NONE`: No row-level locking, and use a cached object if available. This is the Hibernate default.
- `READ`: Data that is in the middle of a transaction (and possibly invalid) should not be read by any other `SELECT` queries until it is committed.
- `UPGRADE`: Uses the `SELECT FOR UPDATE` SQL syntax to lock the data until the transaction is finished.
- `UPGRADE_NOWAIT`: Similar to `UPGRADE`, but uses the `NOWAIT` keyword (for Oracle), which returns an error immediately if there is another thread using that row.

All of these lock modes are static fields on the `org.hibernate.LockMode` class. We discuss locking and deadlocks with respect to transactions in more detail in Chapter 9. The `load()` methods that use lock modes are as follows:

```
public Object load(Class theClass, Serializable id, LockMode lockMode)
            throws HibernateException
public Object load(String entityName, Serializable id, LockMode lockMode)
            throws HibernateException
```

Only use the `load()` method if you are sure that the object exists. If you are not sure that the object exists, then use one of the `get()` methods. The `load()` methods will throw an exception if the unique `id` is not found in the database. Much like `load()`, the `get()` methods require an identifier and either an entity name or a class. There are also two `get()` methods that take a lock mode as an argument. The difference is that the `get()` method will return `null` if the unique `id` is not found. The `get()` methods are

```
public Object get(Class clazz, Serializable id)
           throws HibernateException
public Object get(String entityName, Serializable id)
           throws HibernateException
public Object get(Class clazz, Serializable id, LockMode lockMode)
           throws HibernateException
public Object get(String entityName, Serializable id, LockMode lockMode)
           throws HibernateException
```

If you would like to retrieve the entity name for a given object, you can use the `getEntityName()` method on the `Session` interface:

```
public String getEntityName(Object object) throws HibernateException
```

Using the `get()` methods and the `load()` methods is straightforward. For the following code sample, we would be getting the `Supplier` id from another Java class; for instance through a web application, someone may select a `Supplier` details page for the supplier with the id `1`. If we are not sure that the supplier exists, we use the `get()` method, where we could check for null:

```
// get an id from some other Java class, for instance, through a web application
Supplier supplier = (Supplier) session.get(Supplier.class,id);
if (supplier == null)
{
    System.out.println("Supplier not found for id " + id);
    return;
}
```

We can also retrieve the entity name from Hibernate and use it with either the `get()` or the `load()` methods. Remember that the `load()` method will throw an exception if an object with that `id` cannot be found:

```
String entityName = session.getEntityName(supplier);
 Supplier secondarySupplier = (Supplier) session.load(entityName,id);
```

Refreshing Objects

Hibernate provides a mechanism to refresh persistent objects from their database representations. Typically, you will not have to use this, but it could come into use for one of three scenarios.

The first scenario is that your Hibernate application is not the only application working with this data. Rather than using locks to control access, you only need to load the properties into the object for reading at this time.

Another scenario is that your application executed some SQL directly against the database, and now the data in memory is out of sync with the database representation.

The last scenario is that your database uses triggers to populate properties on the object—for instance, some database designs include a "created" timestamp as a column that gets populated by a trigger when the record is first stored in the database. Use one of the `refresh()` methods on the `Session` interface to refresh an instance of a persistent object:

```
public void refresh(Object object) throws HibernateException
public void refresh(Object object, LockMode lockMode)
          throws HibernateException
```

Although you may be used to the idea of refreshing objects from using other Java persistence systems or tools, we have found that in Hibernate, you typically do not have to worry about using these refresh methods.

Updating Objects

Hibernate automatically manages any changes made to persistent objects. If a property changes on a persistent object, the associated Hibernate session will queue the change for persistence to the database using SQL. From a developer's perspective, you do not have to do any work to store these changes, unless you would like to force Hibernate to commit all of its changes in the queue. You can also determine if the session is dirty and changes need to be committed. When you commit the Hibernate transaction, Hibernate will take care of these details for you.

The `flush()` method forces Hibernate to flush the session:

```
public void flush() throws HibernateException
```

You can determine if the session is dirty with the `isDirty()` method:

```
public boolean isDirty() throws HibernateException
```

You can also instruct Hibernate to use a flushing mode for the session with the `setFlushMode()` method. The `getFlushMode()` method returns the flush mode for the current session:

```
public void setFlushMode(FlushMode flushMode)
public FlushMode getFlushMode()
```

The possible flush modes are the following:

- `ALWAYS`: Every query flushes the session before the query is executed.
- `AUTO`: Hibernate manages the query flushing to guarantee that the data returned by a query is up to date.
- `COMMIT`: Hibernate flushes the session on transaction commits.
- `NEVER`: Your application needs to manage the session flushing with the `flush()` method. Hibernate never flushes the session itself.

By default, Hibernate uses the `AUTO` flush mode. Typically, you will not have to change the flush mode for your application because Hibernate does an excellent job of managing the database synchronization itself.

Deleting Objects

Removing an object from the database is straightforward. The `Session` interface provides a `delete()` method that takes a persistent object as an argument. The argument could also be a transient object with the identifier set to the id of the object that needs to be erased:

```
public void delete(Object object) throws HibernateException
```

In the simplest form, where you are simply deleting an object with no associations to other objects, this is straightforward. Many objects do have associations with other objects. In Hibernate, deletes may cascade from one object to its associated objects.

For instance, consider the situation where you have a parent with a collection of child objects, and you would like to delete them all. The easiest way to handle this is to use the `cascade` attribute on the collection's element in the Hibernate mapping. If you set the `cascade` attribute to `delete` or `all`, the `delete` will be cascaded to all of the associated objects. Hibernate will take care of deleting these for you—deleting the parent erases the associated objects.

In some cases, your application will need to execute business logic if an object is deleted. Rather than try and add that logic to every place where `delete()` gets called for that object, you can add crosscutting functionality that should help keep your application from becoming spaghetti code. With Hibernate, you may use either events or interceptors to do this, and we discuss both of them in Chapter 11. The `delete()` method fires a `DeleteEvent`, which is responsible for deleting the object.

■**Note** In Hibernate 3, the `delete(String hql)` method on the `Session` interface was deprecated. Instead of using the `delete(String hql)` method for HQL deletes, you should create a delete query and execute it. See Chapter 8 for more about HQL and bulk deletes.

Hibernate 3 also supports bulk deletes, where your application executes a `DELETE` HQL statement against the database. These are very useful for deleting more than one object at a time because each object does not need to be loaded into memory just to be deleted. Network traffic is greatly reduced, as are the memory requirements.

There is another type of cascading operation that has to do with deletes, named `delete-orphan`. If you remove a child object from a collection, you can ask Hibernate to automatically delete the child object. This only works if there is a `one-to-many` relationship between the parent and the child. We discuss `delete-orphan` in the next section.

Cascading Operations

With Hibernate, the operations we discussed so far in this chapter may be cascaded through an object's collections. Hibernate provides several different types of cascading that correspond to the different Hibernate operations:

- `create`
- `merge`
- `delete`
- `save-update`
- `evict`
- `replicate`
- `lock`
- `refresh`

You may combine these values in a comma-separated list to allow cascading for several different operations. Rather than add all of these in a comma-separated list, Hibernate provides a shortcut value named `all` that tells Hibernate to cascade all of these operations from the parent to each child object (for that relationship).

As part of the Hibernate mapping process, you can tell Hibernate to use one of these cascading types for a relationship between two objects (the parent and the child). On the collection or property element in the mapping file, set the `cascade` attribute to the type (or types) you would like to use. Hibernate also provides an optional way to set a default cascading type, using the `default-cascade` attribute on the `<hibernate-mapping>` XML element. Use the default cascading to specify how properties and collections (that do not have a `cascade` attribute in their mapping) perform cascading. The `default-cascade` attribute accepts the same values as the `cascade` attribute. You may override the `default-cascade` attribute by providing a `cascade` attribute for those properties or collections.

The last possible cascading type is `delete-orphan`. Use `delete-orphan` to remove a child object from the database when you remove the child from the parent's collection. This cascading type only works on `one-to-many` associations. The `all` cascading type does not include `delete-orphan`—you will have to use `"all,delete-orphan"`, such as in the following excerpt from a Hibernate mapping file:

```
<bag name="products" inverse="true" cascade="all,delete-orphan">
    <key column="supplierId"/>
    <one-to-many class="Product"/>
</bag>
```

Simply remove a child object from a parent object's collection after you have added the `delete-orphan` cascading type. Hibernate will remove the child object from the database itself, without any additional calls. For instance, here is an example that removes a child object from the collection:

```
supplier.getProducts().remove(product);
```

We discuss Hibernate mapping in more detail in Chapter 6.

Querying Objects

Hibernate provides several different ways to query for objects stored in the database. The criteria query API is a Java API for constructing a query as an object. HQL is an object-oriented query language, similar to SQL, that you may use to retrieve objects that match the query. Hibernate provides a way to execute SQL directly against the database to retrieve objects, if you have legacy applications that use SQL or if you need to use SQL features that are not supported through HQL and the criteria query API. Chapter 7 covers the criteria query API and Chapter 8 discusses HQL and the native SQL facility.

EJB3/JSR 220 Persistence API

The Hibernate persistence concepts we talked about in this chapter also largely map to the EJB3 specification. At the time of this writing, the EJB3 specification is not yet released, and Hibernate 3 does not include direct support for the EJB3 `javax.persistence.EntityManager` interface, which is similar to the Hibernate `Session` interface. If you would like to code directly to the EJB3 interfaces, support will most likely be found in a version of the JBoss application server.

The JBoss application server team worked with the Hibernate team to implement EJB3 using Hibernate as the persistence mechanism in JBoss. For more information about the JBoss/Hibernate/EJB3 integration, visit the JBoss product web page for EJB3 at `http://www.jboss.org/products/ejb3`.

Summary

Hibernate provides a simple API for creating, retrieving, updating, and deleting objects from the relational database through the `Session` interface. Understanding the difference between transient, persistent, and detached objects in Hibernate allows us to understand how changes to the objects update the database table.

After learning how to manage objects with Hibernate in this chapter, the following chapter discusses how to create mapping files for Hibernate's object-relational mapping. We go into depth for querying objects with the criteria query API in Chapter 7, and how to use the Hibernate Query Language in Chapter 8.

CHAPTER 6

Creating Mappings

In our simple example programs in Chapter 1 and Chapter 3, we demonstrated how a mapping file could be used to establish the relationship between the object model and the database schema. A mapping file can map a single class or multiple classes to the database. The mapping can also describe standard queries (in HQL and in SQL) and filters.

Hibernate Types

Although we have referred to the Hibernate types in passing, we have not discussed the terminology in any depth. In order to express the behavior of the mapping file elements, we need to make these fine distinctions explicit.

Hibernate types fall into three broad categories: entities, components, and values.

Entities

Generally, an entity is a POJO class that has been mapped into the database using the `<class>` or `<subclass>` elements.

An entity can also be a dynamic map (actually a `Map` of `Maps`). These are mapped against the database in the same way as a POJO, but with the default entity mode of the `SessionFactory` set to `dynamic-map`.

The advantage of POJOs over the `dynamic-map` approach is that compile-time type safety is retained. Conversely, `dynamic-maps` are a quick way to get up and running when building prototypes. We recommend that you use the standard entity mode unless you need to sacrifice accuracy for timeliness.

We have decided to omit discussion of some features pertaining to the XML mapping metadata. This permits an alternative representation of data as the contents of an XML document, instead of a POJO or `Map`. This is a somewhat experimental feature in Hibernate 3 and, at the time of this writing, we do not recommend that you use it. For more information, consult the Hibernate website (`http://www.hibernate.org`).

Components

Lying somewhere between entities and values are component types. When the class representation is simple and its instances have a strong `one-to-one` relationship with instances of another class, then it is a good candidate to become a component of that other class.

The component will normally be mapped as columns in the same table that represents most of the other attributes of the owning class, so the strength of the relationship must justify this inclusion.

The advantage of this approach is that it allows us to dispense with the primary key of the component and the join to its containing table. If a poor choice of component is made (for example, where a `many-to-one` relationship actually holds), then data will be duplicated unnecessarily in the component columns.

Values

Everything that is not an entity or a component is a value. Generally these correspond to the data types supported by your database, the collection types, and, optionally, some user-defined types.

The details of these mappings will be vendor specific, so Hibernate provides its own value type names and the Java types are defined in terms of these (see Table 6-1).

Table 6-1. *The Standard Hibernate 3 Value Names*

Primitives and Wrappers

Hibernate 3 Type	Corresponding Java Type
`integer`	`int`, `java.lang.Integer`
`long`	`long`, `java.lang.Long`
`short`	`short`, `java.lang.Short`
`float`	`float`, `java.lang.Float`
`double`	`double`, `java.lang.Double`
`character`	`char`, `java.lang.Character`
`byte`	`byte`, `java.lang.Byte`
`boolean`, `yes_no`, `true_false`	`boolean`, `java.lang.Boolean`

Other Classes

Hibernate 3 Type	Corresponding Java Type
`string`	`java.lang.String`
`date`, `time`, `timestamp`	`java.util.Date`
`calendar`, `calendar_date`	`java.util.Calendar`
`big_decimal`	`java.math.BigDecimal`
`big_integer`	`java.math.BigInteger`
`locale`	`java.util.Locale`
`timezone`	`java.util.TimeZone`
`currency`	`java.util.Currency`
`class`	`java.lang.Class`
`binary`	`byte[]`
`text`	`java.lang.String`
`serializable`	`java.io.Serializable`

Other Classes

Hibernate 3 Type	Corresponding Java Type
`clob`	`java.sql.Clob`
`blob`	`java.sql.Blob`

In addition to these standard types, you can create your own. Your user-type class should implement the `org.hibernate.foo.UserType` or the `org.hibernate.foo.CompositeUserType` interfaces. Once implemented, a custom type can behave identically to the standard types, though depending on your requirements it may be necessary to specify multiple column names to contain its values, or to provide initialization parameters for your implementation.

For one-off cases, we recommend that you use a component, which has nearly identical semantics. Unless you propose to make substantial use of a custom type throughout your application, it will not be worth the effort. We do not discuss this feature further in this text.

The Anatomy of a Mapping File

A mapping file is a normal XML file. It is validated against a DTD, which can be downloaded from `http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd` or you can look through the annotated version on the companion site for this book (`http://hibernatebook.com/`).

The terminology used in the naming of elements and attributes is somewhat confusing at first because it is the point of contact between the jargon of the object-oriented and the relational worlds.

The Hibernate-Mapping Element

The root element of any mapping file is `<hibernate-mapping>`. As the top-level element, the attributes of this are mostly defining default behaviors and settings to apply to the child elements (see Table 6-2).

Table 6-2. *The <hibernate-mapping> Attributes*

Attribute	Values	Default	Description
`auto-import`	`true`, `false`	`true`	By default, this allows you to use the unqualified class names in Hibernate queries. You would only normally set this to `false` if the class name would otherwise be ambiguous.
`catalog`			The database catalog against which queries should apply.
`default-access`		`property`	The default access type. If set to `property`, then `get` and `set` methods are used to access the data. If set to `field`, then the data is accessed directly. Alternatively, you can provide the class name of a `PropertyAccessor` implementation defining any other access mechanism.

Continued

Table 6-2. *Continued*

Attribute	Values	Default	Description
`default-cascade`		none	Defines how (and if) direct changes to data should affect dependent data by default.
`default-lazy`	`true`, `false`	`true`	Defines whether lazy instantiation is used by default. Generally, the performance benefits are such that you will want to use lazy instantiation wherever possible.
`package`			The package from which all implicit imports are considered to occur.
`schema`			The database schema against which queries should apply.

The default cascade modes available for the `default-cascade` attribute (and indeed for the cascade attributes in all other elements) are

```
create, merge, delete, save-update, evict, replicate, lock, refresh
```

These correspond to the various possible changes in the lifestyle of the parent object. When set (you can include combinations of them as comma-separated values), the relevant changes to the parent will be cascaded to the relation. Typically, for example, you may want to apply the `save-update` cascade option to a class that includes `Set` attributes, so that when new persistent classes are added to these they will not have to be saved explicitly in the session.

There are also three special options:

```
all, delete-orphan, none
```

`all` and `none` specify, respectively, that all changes to the parent should be propagated to the relation, and that none should. `delete-orphan` applies only to `one-to-many` associations, and specifies that the relation should be deleted when it is no longer referenced by the parent.

The required order and cardinality of the child elements of `hibernate-mapping` are the following:

```
(meta*,
 typedef*,
 import*,
 (class | subclass | joined-subclass | union-subclass)*,
 (query | sql-query)*,
 filter-def*)
```

Throughout this book we have assumed that the mappings are defined in one mapping file for each significant class that is to be mapped to the database. We suggest that you follow this practice in general, but there are some exceptions to this rule. You may, for instance, find it useful to place `query` and `sql-query` entries into an independent mapping class, particularly when they do not fall clearly into the context of a single class.

THE ORDER AND CARDINALITY INFORMATION FROM THE DTD

The mapping files used by Hibernate have a great many elements and are somewhat self-referential. For example, the `component` element permits you to include within it further `component` elements, and within those, further `component` elements—and so on, *ad infinitum*.

While we do not quote exhaustively from the mapping file's DTD document, we will sometimes quote that part of it which specifies the permitted ordering and cardinality (number of occurrences) of the child elements of a given element.

The cardinality is expressed by a symbol after the end of the name of the element: * means "zero or more occurrences," ? means "zero or one occurrences," and no trailing symbol means "exactly one occurrence."

The elements can be grouped using brackets, and where the elements are interchangeable the vertical bar | means "or."

In practical terms, this allows us to tell from the order and cardinality information quoted for the `hibernate-mapping` file that all of the elements immediately below it are, in fact, optional. We can also see that there is no limit to the number of `class` elements that you can include if you wish to.

You can look up this information in this format in the DTD for the mapping file for all the elements, including the ones we have omitted from this chapter. You will also find within the DTD the specification of which attributes are permitted to each element, the values they may take (where these are constrained), and their default values where one is provided. We recommend that you look at the DTD for enlightenment whenever you are trying to work out if a specific mapping file should be syntactically valid.

The Class Element

The child element that you will use most often—indeed in nearly all of your mapping files—will be `<class>`. As you have seen in earlier chapters, this is where we generally describe the relationships between Java objects and database entities. The `<class>` element permits the following attributes to be defined (see Table 6-3).

Table 6-3. *The <class> Attributes*

Attribute	Values	Default	Description
`abstract`	`true`, `false`	`false`	This flag should be set if the class being mapped is abstract.
`batch-size`		`1`	This is the number of items that can be batched together when retrieving instances of the class by identifier.
`catalog`			The database catalog against which the queries should apply.
`check`			SQL to create a multirow check constraint for schema generation.

Continued

Table 6-3. *Continued*

Attribute	Values	Default	Description
`discriminator-value`			A value used to distinguish between otherwise identical subclasses of a common type persisted to the same table. `is null` and `is not null` are permissible values. To distinguish between a Cat and a Dog derivative of the Mammal abstract class, you might use discriminator values of "C" and "D," respectively.
`dynamic-insert`	`true`, `false`	`false`	If set to `true`, null columns will also appear in generated `INSERT` commands.
`dynamic-update`	`true`, `false`	`false`	If set to `true`, unchanged columns will also appear in generated `UPDATE` commands.
`entity-name`			The name of the entity to use in place of the class name (therefore required if dynamic mapping is used).
`lazy`	`true`, `false`		Can be used to disable or enable lazy fetching against the enclosing mapping's default.
`mutable`	`true`, `false`	`true`	Can be used to flag that a class is mutable (allowing Hibernate to make some performance optimizations when dealing with these classes).
`name`			The fully qualified Java name of the class (or interface) that is to be made persistent.
`optimistic-lock`	`none`, `version`, `dirty`, `all`	`version`	Specifies the optimistic locking strategy to use. This applies at a class level, but in Hibernate 3 can also be specified (or overridden) at an attribute level.
`persister`			Allows a custom `ClassPersister` object to be used when persisting this class.
`polymorphism`	`implicit`, `explicit`	`implicit`	Determines how polymorphism is to be used. The default implicit behavior will return instances of the class if superclasses or implemented interfaces are named in the query, and will return subclasses if the class itself is named in the query.
`proxy`			Specifies a class or interface to use as the proxy for lazy initialization.
`rowid`			Flags that row ids should be used (a database implementation detail allowing Hibernate to optimize updates).

Attribute	Values	Default	Description
schema			Optionally overrides the schema specified by the hibernate-mapping element.
select-before-update	true, false	false	If set to true, Hibernate will perform a select before performing an update to ensure that an update is actually required (i.e., that columns have been modified). While this is often likely to be less efficient, it can prevent database triggers from being invoked unnecessarily.
subselect			A subselect of the contents of the underlying table. A class can only use a subselect if it is immutable and read-only (because the SQL defined here cannot be reversed). Generally, the use of a database view is preferable.
table			The table name associated with the class (or, if unspecified, the unqualified class name will be used).
where			An arbitrary SQL where condition to be used when retrieving objects of this class from the table.

Many of these attributes in the class element are designed to support preexisting database schemas. In practice, the name attribute is very often the only one set.

The required order and cardinality of the child elements of class are

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 (id | composite-id),
 discriminator?,
 (version | timestamp)?,
 (property | many-to-one | one-to-one | component | dynamic-component |
  properties | any | map | set | list | bag | idbag |
  array | primitive-array | query-list)*,
 ((join*, subclass*) | joined-subclass* | union-subclass*),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 filter*)
```

The Id Element

All entities need to define their primary key in some way. Any class directly defined by the `class` element (not derived or component classes) must therefore have an `id` or `composite-id` element to define this (see Table 6-4). Note that while it is not a requirement that your class implementation itself should implement the primary key attribute, it is certainly advisable. If you cannot alter your class design to accommodate this, however, then the session provides the `getIdentifier()` method to determine the identifier of a persistent class independently.

Table 6-4. *The <id> Attributes*

Attribute	Values	Default	Description
`access`			How the properties should be accessed: `field` (directly), `property` (calling the `get`/`set` methods), or the name of a `PropertyAccessor` class to be used. The value from the `hibernate-mapping` will be inherited if this is not specified.
`column`			The name of the column in the table containing the primary key. The value given in the `name` attribute will be used if this is not specified.
`length`			The column length to be used.
`name`			The name of the attribute in the class representing this primary key. If this is omitted, it is assumed that the class does not have an attribute directly representing this primary key. Naturally, the `column` attribute must be provided if the `name` attribute is omitted.
`type`			The Hibernate type of the column.
`unsaved-value`			The value that the attribute should take when an instance of the class has been created, but not yet persisted to the database. This attribute is mandatory.

The `<id>` element requires a `<generator>` element to be specified, which defines how to generate a new primary key for a new instance of the class. The generator takes a `class` attribute, which defines the mechanism to be used. The class should be an implementation of `IdentifierGenerator`. Optional `<param>` elements can be provided if the identifier needs additional configuration information, each having the form as follows:

```
<param name="parameter name">parameter value</param>
```

Hibernate provides several default `IdentifierGenerator` implementations, and these can be referenced by convenient short names, as shown in Table 6-5.

Table 6-5. *The Default IdentiferGenerator Implementations*

Short Name	Description
`guid`	Uses a database-generated "Globally" Unique identifier. This is not portable to databases that do not have a `guid` type, and the specific implementation, and hence the quality of the uniqueness of this key, may vary from vendor to vendor.
`hilo`	Uses a database table and column to efficiently and portably maintain and generate identifiers that are unique to that database. The Hibernate `int`, `short`, or `long` types are supported.

Short Name	Description
identity	Supports the identity column type available in some, but not all, databases. This is, therefore, not a fully portable option. The Hibernate int, short, or long types are supported.
increment	Generated by adding 1 to the current highest key value. Can apply to int, short, or long hibernate types. This only works if other processes are not permitted to update the table at the same time. If multiple processes are running, then depending on the constraints enforced by the database the result will be an error in the application(s) or data corruption!
native	Selects one of sequence, identity, and hilo as appropriate. This is a good compromise option since it uses the innate features of the database and is portable to most platforms. This is particularly appropriate if your code is likely to be deployed to a number of database implementations with differing capabilities.
seqhilo	Where the dialect supports database sequences, this uses a sequence to efficiently (but not portably) generate identifiers that are unique to that database. The Hibernate int, short, or long types are supported.
sequence	Supports the sequence column type (essentially a database-enforced increment) available in some, but not all, databases. This is, therefore, not a fully portable option. The Hibernate int, short, or long types are supported.
uuid	Attempts to portably generate a (cross-database) unique primary key. The key is composed of the local IP address, the startup time of the JVM (accurate to a quarter second), the system time, and a counter value (unique within the JVM). This cannot guarantee absolutely that a given key is unique, but it will be good enough for most clustering purposes.

The child elements of the <id> element are as follows:

```
(meta*, column*, generator?)
```

While this is all rather complex, Listing 6-1 shows a typical <id> element from Chapter 3 illustrating the simplicity of the usual case.

Listing 6-1. *A Typical <id> Element*

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

■Note Where the <id> element cannot be defined, a compound key can instead be defined using the <composite-id> element. This is provided purely to support existing database schemas. A new Hibernate project with a clean database design does not require this. We will look at connecting to legacy databases in more depth in Chapter 13.

The Property Element

While it is not absolutely essential, almost all classes will also maintain a set of properties in the database in addition to the primary key. These are defined by `<property>` elements (see Table 6-6).

Table 6-6. *The <property> Attributes*

Attribute	Values	Default	Description
`access`			How the properties should be accessed: `field` (directly), `property` (calling the `get`/`set` methods), or the name of a `PropertyAccessor` class to be used. The value from the class, or `hibernate-mapping`, will be inherited if this is not specified.
`column`			The column in which the property will be maintained. If omitted this will default to the name of the attriute—or it can be specified with nested `column` elements (see Listing 6-2).
`formula`			An arbitrary SQL query representing a computed property (i.e., one which is calculated dynamically, rather than being represented in a column).
`index`			The name of an index maintained for this column.
`insert`	`true`, `false`	`true`	Specifies whether creation of an instance of the class should result in the column associated with this attribute being included in insert statements.
`lazy`	`true`, `false`	`false`	Defines whether lazy instantiation is used by default for this column.
`length`			The column length to be used.
`name`			The (mandatory) name of the attribute. This should start with a lowercase letter.
`not-null`	`true`, `false`	`false`	Specifies if the column is permitted to contain `null` values.
`optimistic-lock`	`true`, `false`	`true`	Determines if optimistic locking should be used when the attribute has been updated.
`precision`			Allows the precision to be specified for numeric data (the number of digits).
`scale`			Allows the scale to be specified for numeric data (the number of digits to the right of the decimal point).
`type`			The Hibernate type of the column.
`unique`	`true`, `false`	`false`	Indicates if duplicate values are permitted for this column/attribute.
`update`	`true`, `false`	`true`	Specifies whether changes to this attribute in instances of the class should result in the column associated with this attribute being included in update statements.

The child elements of the property element are as follows:

```
(meta*, (column | formula)*, type?)
```

Any element accepting a column attribute, as is the case for the property element, will also accept column elements in its place. For example, see Listing 6-2.

Listing 6-2. *Using the <column> Element*

```
<property name="message"/>
   <column name="message" type="string"/>
</property>
```

This particular example does not really give us anything over and above the use of the column attribute directly, but the column element comes into its own with custom types and the use of some of the more complex mappings that we will be looking into later in the chapter.

The Component Element

The component element is used to map classes that will be represented as extra columns within a table describing some other class. We have already discussed how components fit in as a compromise between full entity types and mere value types.

The <component> element can take the attributes listed in Table 6-7.

Table 6-7. *The <component> Attributes*

Attribute	Values	Default	Description
access			How the properties should be accessed: field (directly), property (calling the get/set methods), or the name of a PropertyAccessor class to be used.
class			The class that the parent class incorporates by composition.
insert	true, false	true	Specifies whether creation of an instance of the class should result in the column associated with this attribute being included in insert statements.
lazy	true, false	false	Defines whether lazy instantiation is used by default for this mapped entity.
name			The name of the attribute (component) to be persisted.
optimistic-lock	true, false	true	Specifies the optimistic locking strategy to use.
unique	true, false	false	An indication that the values that represent the component must be unique within the table.
update	true, false	true	Specifies whether changes to this attribute in instances of the class should result in the column associated with this attribute being included in update statements.

The child elements of the `component` element are as follows:

```
(meta*,
  parent?,
  (property | many-to-one | one-to-one |
   component | dynamic-component | any |
   map | set | list | bag |
   array | primitive-array)* )
```

We provide a full example of the use of the `component` element in the "Mapping Composition" section later in this chapter.

The One-to-One Element

The `one-to-one` element expresses the relationship between two classes where each instance of the first class is related to a single instance of the second and vice versa. Such a `one-to-one` relationship can be expressed either by giving each of the respective tables the same primary key values, or by using a foreign key constraint from one table onto a unique identifier column of the other. Table 6-8 shows the attributes that apply to the `one-to-one` element.

Table 6-8. *The <one-to-one> Attributes*

Attribute	Values	Default	Description
`access`			Specifies how the class member should be accessed: `field` for direct field access, or `attribute` for access via the `get` and `set` methods.
`batch-size`			When in lazy-load mode, this specifies the number of occurrences to acquire in each access.
`cascade`			Determines how changes to the parent entity will affect the linked relation.
`catalog`			The database catalog against which queries should apply.
`check`			SQL to create a multirow check constraint for schema generation.
`fetch`	`join`, `select`, `subselect`		The mode in which the element will be retrieved (`outer-join`, a series of `selects`, or a series of `subselects`). Only one member of the enclosing class can be retrieved by `outer-join`.
`inverse`	`true`, `false`	`false`	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping.
`lazy`	`true`, `false`		Overrides the entity loading mode.
`name`			Assigns a name to the entity (required in dynamic mappings).
`optimistic-lock`	`true`, `false`	`true`	Specifies whether optimistic locking should be used.

Attribute	Values	Default	Description
`outer-join`	`true`, `false`, `auto`		Specifies whether an `outer-join` should be used.
`persister`			Specifies a custom `ClassPersister` class that can be used to override Hibernate's default handling of object persistence. You would not normally use this facility.
`schema`			The database schema against which queries should apply
`subselect`			A `subselect` of the contents of the underlying table. A class can only use a `subselect` if it is immutable and read-only (because the SQL defined here cannot be reversed). Generally, the use of a database view is preferable.
`table`			The name of the table in which the associated entity is stored.
`where`			An arbitrary SQL `where` clause limiting the linked entities.

You would select a primary key association when you do not want an additional table column to relate the two entities. The "master" of the two entites takes a normal primary key generator and its one-to-one mapping entry will typically have the attribute name and associated class specified only. The "slave" entity will be mapped similarly, but must have the `constrained` attribute setting applied to ensure that the relationship is recognized.

Because the slave class's primary key must be identical to that allocated to the master, it is given the special `id generator` type of `foreign`. On the slave end, the `id` and `one-to-one` elements will, therefore, look like this:

```
<id name="id" column="product">
   <generator class="foreign">
      <param name="property">campaign</param>
   </generator>
</id>

<one-to-one name="campaign"
   class="com.hibernatebook.chapter06.Campaign"
   constrained="true"/>
```

There are some limitations to this approach: it cannot be used on the receiving end of a `many-to-one` relationship (even when the `many` end of the association is limited by a unique constraint), and the slave entity cannot be the slave of more than one entity.

In these circumstances, you will need to declare the master end of the association as a uniquely constrained `one-to-many` association. The slave entity's table will then need to take a foreign key column associating it with the master's primary key, and the `property-ref` attribute setting is used to declare this relationship, like so:

```
<one-to-one
   name="campaign"
   class="com.hibernatebook.chapter06.Campaign"
   property-ref="product"/>
```

The format used in this example is the most common. The body of the element consists of an infrequently used optional element:

```
(meta* | formula*)
```

We discuss the `many-to-many` element and the alternative approach of composition in some detail in the "Mapping Collections" section later in this chapter.

The Many-to-One Element

The `many-to-many` association describes the relationship where multiple instances of this one class can reference a single instance of another class. This enforces a relational rule where the "many" class has a foreign key into the (usually primary) unique key of the "one" class. Table 6-9 shows the attributes permissible for the `many-to-one` element.

Table 6-9. *The <many-to-one> Attributes*

Attribute	Values	Default	Description
`access`			Specifies how the class member should be accessed: `field` for direct field access, or `attribute` for access via the `get` and `set` methods.
`cascade`			Determines how changes to the parent entity will affect the linked relation.
`class`			The property type of the `attribute` or `field` (if omitted this will be determined by reflection).
`column`			The column containing the identifier of the target entity (i.e., the foreign key from this entity into the mapped one).
`entity-name`			The name of the associated entity.
`fetch`	`join`, `select`		The mode in which the element will be retrieved (`outer-join`, a series of `selects`, or a series of `subselects`). Only one member of the enclosing class can be retrieved by `outer-join`.
`foreign-key`			The name of the foreign key constraint to generate for this association.
`formula`			An arbitrary SQL expression to use in place of the normal primary key relationship between the entities.
`index`			The column which will specify the ordering of these entities when retrieved by the inverse of this mapping.

Attribute	Values	Default	Description
insert	true, false	true	When set to false, this prevents inserts if the field has already been mapped as part of a composite identifier or some other attribute.
lazy	true, false	false	Overrides the entity loading mode.
name			The (mandatory) name of the attribute. This should start with a lowercase letter.
not-found	exception, ignore	exception	The behavior to exhibit if the related entity does not exist—either throw an exception, or ignore the problem.
not-null	true, false	false	Specifies whether a not-null constraint should be applied to this column.
optimistic-lock	true, false	true	Specifies whether optimistic locking should be used.
outer-join	true, false, auto		Specifies whether an outer-join should be used.
property-ref			If the referenced table's foreign key is not onto the primary key of the many end of the relationship, then property-ref can be used to specify the table that it references. This should only be the case for legacy designs—when creating a new schema, your foreign keys should always reference the primary key of the related table.
unique	true, false	false	Specifies whether a unique constraint should be applied to this column.
update	true, false	true	When set to false, this prevents updates if the field has already been mapped elsewhere.

By specifying a unique constraint on a many-to-one relationship, it is effectively converted into a one-to-one relationship, and this is the more common form for that, both because it results in a simpler mapping and because it requires less intrusive changes to the database should it become desirable to relax the one-to-one association into a many-to-one. This is discussed in more depth in the "Mapping Other Associations" section later in this chapter.

This element has a small number of optional daughter elements—the column element will be required when a composite key has to be specified:

```
(meta*, (column | formula)*)
```

The Collection Elements

These are the elements that are required for you to include an attribute in your class that represents any of the collection classes. For example, if you have an attribute of type Set, then you will need to use a bag or set element to represent its relationship with the database.

Because of the simplicity of the object-oriented relationship involved, where one object has an attribute capable of containing many objects, it is a common fallacy to assume that the relationship must be expressed as a `one-to-many`. In practice, however, this will almost always be easiest to express as a `many-to-many` relationship, where an additional link table closely corresponds with the role of the collection itself. See the "Mapping Collections" section later in this chapter for a more detailed illustration of this.

All of the collection mapping elements share the attributes shown in Table 6-10.

Table 6-10. *The Attributes Common to the Collection Elements*

Attribute	Values	Default	Description
`access`			Specifies how the class member should be accessed: `field` for direct field access, or `attribute` for access via the `get` and `set` methods.
`batch-size`			This number of items can be batched together when retrieving instances of the class by identifier.
`cascade`			Determines how changes to the parent entity will affect the linked relation.
`catalog`			The database catalog against which the queries should apply.
`check`			SQL to create a multirow check constraint for schema generation.
`fetch`	`join`, `select`, `subselect`		The mode in which the element will be retrieved (`outer-join`, a series of `selects`, or a series of `subselects`). Only one member of the enclosing class can be retrieved by `outer-join`.
`lazy`	`true`, `false`		Can be used to disable or enable lazy fetching against the enclosing mapping's default.
`name`			The (mandatory) name of the attribute. This should start with a lowercase letter.
`optimistic-lock`	`true`, `false`	`true`	Specifies the optimistic locking strategy to use.
`outer-join`	`true`, `false`, `auto`		Specifies whether an `outer-join` should be used.
`persister`			Allows a custom `ClassPersister` object to be used when persisting this class.
`schema`			The database schema against which queries should apply.
`subselect`			A `subselect` of the contents of the underlying table. A class can only use a `subselect` if it is immutable and read-only (because the SQL defined here cannot be reversed). Generally, the use of a database view is preferable.

Attribute	Values	Default	Description
table			The name of the table in which the associated entity is stored.
where			An arbitrary SQL `where` clause limiting the linked entities.

Set

A `set` collection allows collection attributes derived from the `Set` interface to be persisted.

In addition to the common collection mappings, the `set` element offers the `inverse`, `order-by`, and `sort` attributes, as shown in Table 6-11.

Table 6-11. *The Additional <set> Attributes*

Attribute	Values	Default	Description
inverse	true, false	false	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping.
order-by			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the `Set`.
sort			Specifies the `Collection` class sorting to be used. The value can be `unsorted`, `natural`, or any `Comparator` class.

The child elements of the `set` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 key,
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of a set mapping is

```
<set name="set" table="nameset">
   <key column="fooid"/>
   <element type="string" column="name" not-null="true"/>
</set>
```

List

A list collection allows collection attributes derived from the List interface to be persisted.

In addition to the common collection mappings, the list element offers the inverse attribute, as seen in Table 6-12.

Table 6-12. *The Additional <list> Attributes*

Attribute	Values	Default	Description
inverse	true, false	false	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping.

The child elements of the list element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 key,
 (index | list-index),
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of a list mapping is

```
<list name="list" table="namelist">
   <key column="fooid"/>
   <index column="position"/>
   <element type="string" column="name" not-null="true"/>
</list>
```

Idbag

An idbag collection allows collection attributes derived from the List interface. A bag data structure permits unordered storage of unordered items and permits duplicates. Because the collection classes do not provide a native bag implementation, List tends to be used as a substitute. The imposition of ordering imposed by a list is not of itself a problem, but implementation code can become dependent upon the ordering information.

The idbag usually maps to a List, but by managing its database representation with a surrogate key, the performance of updates and deletions of items in the idbag is made dramatically better than the unkeyed bag (described at the end of this section). Hibernate does not provide a mechanism for identifying the identifier of a row in the bag.

In addition to the common collection mappings, the `idbag` element offers the `order-by` element, as seen in Table 6-13.

Table 6-13. *The Additional <idbag> Attribute*

Attribute	Values	Default	Description
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the collection.

The child elements of the `idbag` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 collection-id,
 key,
 (element | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of an `idbag` mapping is

```
<idbag name="idbag" table="nameidbag">
   <collection-id column="id" type="int">
      <generator class="native"/>
   </collection-id>

   <key column="fooid"/>
   <element type="string" column="name" not-null="true"/>
</idbag>
```

Map

A `map` collection allows `collection` attributes derived from the `Map` interface to be persisted.

In addition to the common collection mappings, the `bag` element offers the `inverse`, `order-by`, and `sort` attributes, as seen in Table 6-14.

Table 6-14. *The Additional <map> Attributes*

Attribute	Values	Default	Description
`inverse`	`true`, `false`	`false`	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping.
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the map.
`sort`			Specifies the `Collection` class sorting to be used. The value can be `unsorted`, `natural`, or any `Comparator` class.

The child elements of the `map` element are as follows:

```
(meta*,
  subselect?,
  cache?,
  synchronize*,
  key,
  (map-key | composite-map-key | map-key-many-to-many |
   index | composite-index | index-many-to-many |
   index-many-to-any),
  (element | one-to-many | many-to-many | composite-element |
   many-to-any),
  loader?,
  sql-insert?,
  sql-update?,
  sql-delete?,
  sql-delete-all?,
  filter*)
```

A typical implementation of the mapping is

```
<map name="map" table="namemap">
   <key column="fooid"/>
   <index column="name" type="string"/>
   <element column="value" type="string" not-null="true"/>
</map>
```

Bag

If your class represents data as a `List`, but you do not want to maintain an index column to keep track of the order of items, you can optionally use a `bag` object to achieve this. The order in which the items are stored and retrieved from a `bag` is completely ignored.

Although the `bag`'s table does not contain enough information to determine the order of its contents prior to persistence into the table, it *is* possible to apply an `order by` clause to the SQL used to obtain the contents of the `bag` so that it has a `natural` sorted order as it is acquired. This will not be honored at other times during the lifetime of the object.

There is a performance impact to the lack of a proper key for the `bag` elements, and this will manifest itself when performing `update` or `delete` operations on the contents of the `bag`.

In addition to the common collection mappings, the `bag` element, therefore, offers the `order-by` as well as the `inverse` attribute, as seen in Table 6-15.

Table 6-15. *The Additional <bag> Attributes*

Attribute	Values	Default	Description
`inverse`	`true`, `false`	`false`	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping.
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the collection.

The child elements of the `bag` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 key,
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of a `bag` mapping is

```
<bag name="bag" table="namebag">
   <key column="fooid"/>
   <element column="value" type="string" not-null="true"/>
</bag>
```

Mapping Simple Classes

Figure 6-1 shows the class diagram and entity relationship diagram for a simple class. They are as straightforward as you would expect.

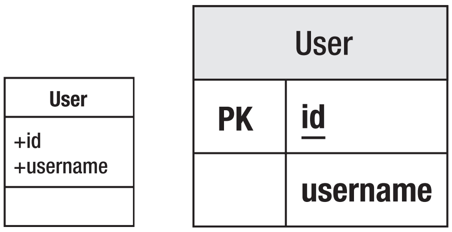

Figure 6-1. *Representing a simple class*

The elements that we have discussed so far are sufficient to map a basic class into a single table, as shown on Listing 6-3.

Listing 6-3. *A Simple Class to Represent a User*

```
package com.hibernatebook.chapter06;

public class User {

   public User(String username) {
      this.username = username;
   }

   User() {
   }

   public int getId() {
      return id;
   }

   public String getUsername() {
      return username;
   }

   public void setId(int id) {
      this.id = id;
   }

   public void setUsername(String username) {
      this.username = username;
   }

   // We will map the id to the table's primary key
   private int id = -1;

   // We will map the username into a column in the table
   private String username;
}
```

It's pretty easy to see that we might want to represent the class in Listing 6-3 in a table with the following format (see Table 6-16).

Table 6-16. *Mapping a Simple Class to a Simple Table*

Column	Type
Id	Integer
Username	Varchar(32)

The mapping between the two is, thus, similarly straightforward:

```
<?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="book.hibernatebook.chapter06.User">

      <id name="id" type="int">
         <generator class="native"/>
      </id>

      <property name="username" type="string" length="32"/>

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

Aside from the very limited number of properties maintained by the class, this is a pretty common mapping type, so it is reassuring to see that it can be managed with a minimal number of elements (`<hibernate-mapping>`, `<class>`, `<id>`, `<generator>`, and `<property>`).

Mapping Composition

Figure 6-2 shows the class diagram and the entity relationship diagram for a composition relationship between two classes. Here the `Advert` class is composed of a `Picture` class in addition to its normal value types.

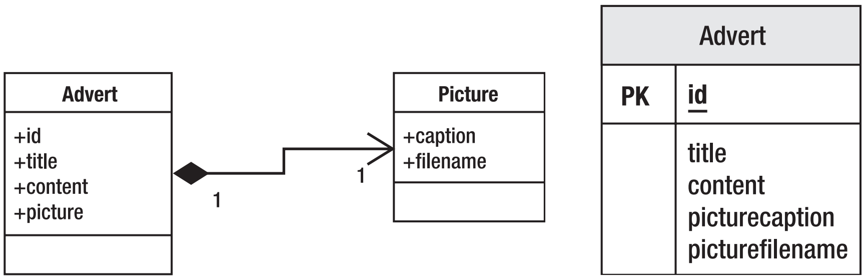

Figure 6-2. *Representing composition*

Composition is the strongest form of aggregation—in which the lifecycle of each object is dependent upon the lifecycle of the whole. Although Java does not make the distinction between other types of aggregation and composition, it becomes relevant when we choose to store the components in the database, because the most efficient and natural way to do this is to store them in the same table.

In our example, we will look at an Advert class that has this relationship with a Picture class. The idea is that our Advert is always going to be associated with an illustration (see Listings 6-4 and 6-5). In these circumstances, there is a clear 1:1 relationship that could be represented between two distinct tables, but which is more efficiently represented with one.

Listing 6-4. *The Class Representing the Illustration*

```
package com.hibernatebook.chapter06;

public class Picture {
   public Picture(String caption, String filename) {
      this.caption = caption;
      this.filename = filename;
   }

   Picture() {
   }

   public String getCaption() {
      return this.caption;
   }

   public String getFilename() {
      return this.filename;
   }

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

   public void setFilename(String filename) {
      this.filename = filename;
   }

   private String caption;
   private String filename;
}
```

Listing 6-5. *The Class Representing the Advert*

```
package com.hibernatebook.chapter06;

public class Advert {
   public Advert(String title, String content, Picture picture) {
      this.title = title;
      this.content = content;
      this.picture = picture;
   }
   Advert() {
   }

   public int getId() {
      return id;
   }

   public String getTitle() {
      return this.title;
   }

   public String getContent() {
      return this.content;
   }

   public Picture getPicture() {
      return this.picture;
   }

   public void setId(int id) {
      this.id = id;
   }

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

   public void setContent(String content) {
      this.content = content;
   }

   public void setPicture(Picture picture) {
      this.picture = picture;
   }

   private int id = -1;
   private String title;
   private String content;
   private Picture picture;
}
```

Again, Hibernate manages to express this simple relationship with a correspondingly simple mapping file. We introduce the `component` entity for this association. Here it is in use:

```
<?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">
```

```
<class name="com.hibernatebook.chapter06.Advert">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <property name="title" type="string" length="255"/>
    <property name="content" type="text"/>
    <component name="picture" class="com.hibernatebook.chapter06.Picture">
        <property name="caption" type="string" length="255"/>
        <property name="filename" type="string" length="32"/>
    </component>
</class>
```

In our example, we use the `<property>` element to describe the relationship between the `Picture` and its attributes. In fact, this is true of all of the rest of the elements of `<class>`—a `<component>` element can even contain more `<component>` elements. Of course, this makes perfect sense, since a component usually corresponds with a Java class.

Mapping Other Associations

In Figure 6-3, the `Advert` class includes an instance of a `Picture` class. The relationship in the tables is represented with the `Picture` table having a foreign key onto the `Advert` table.

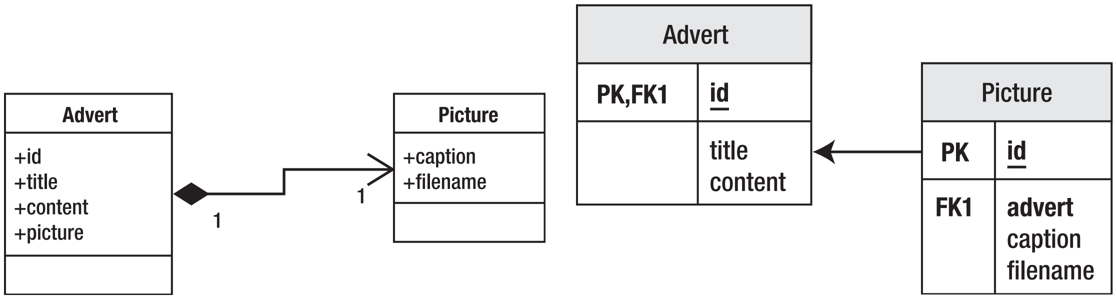

Figure 6-3. *Mapping an aggregation or composition relationship*

A `one-to-one` correspondence does not absolutely require you to incorporate both parties into the same table. There are often good reasons not to. For example, in our `Picture` example, it is entirely possible that while the initial implementation will permit only one `Picture` per `Advert`, a future implementation will relax this relationship. Consider this scenario from the perspective of the database for a moment (see Table 6-17).

Table 6-17. *The Advert Table*

Id	Title	Contents	PictureCaption	PictureFilename
1	Bike	Bicycle for sale	My bike (you can ride it if you like)	`advert001.jpg`
2	Sofa	Sofa, comfy but used	Chesterfield Sofa	`advert002.jpg`
3	Car	Shabby MGF for sale	MGF VVC (BRG)	`advert003.jpg`

If we want to allow the advert for the sofa to include another picture, we would have to duplicate some of the data, or include null columns. What would probably be preferable is to set up a pair of tables, one to represent the adverts, and a one to represent the distinct tables (as shown in Tables 6-18 and 6-19).

Table 6-18. *The Refined Advert Table*

Id	Title	Contents
1	Bike	Bicycle for sale
2	Sofa	Sofa, comfy but used
3	Car	Shabby MGF for sale

Table 6-19. *The Picture Table*

Id	Advert	Caption	Filename
1	1	My bike (you can ride it if you like)	`advert001.jpg`
2	2	Chesterfield Sofa	`advert002.jpg`
3	3	MGF VVC (BRG)	`advert003.jpg`

If we decide (considering the database only) to allow additional pictures, we can then include extra rows in the `Picture` table without duplicating any data unnecessarily (see Table 6-20).

Table 6-20. *The Picture Table with Multiple Pictures per Advert*

Id	Advert	Caption	Filename
1	1	My bike (you can ride it if you like)	`advert001.jpg`
2	2	Chesterfield Sofa	`advert002.jpg`
3	**2**	**Back of Sofa**	`advert003.jpg`
4	3	MGF VVC (BRG)	`advert004.jpg`

With the single `Advert` table, the query to extract the data necessary to materialize an instance of the `Advert` consists of something like this:

```
select id,title,contents,picturecaption,picturefilename from advert where id = 1
```

It is obvious here that a single row will be returned, since we are carrying out the selection on the primary key.

Once we split things into two tables, we have a slightly more ambiguous pair of queries:

```
select id,title,contents from advert where id = 1
select id,caption,filename from picture where advert = 1
```

While Hibernate is not under any particular obligation to use this pair of SQL instructions to retrieve the data (it could reduce it to a join on the table pair), it is the easiest way of thinking about the data we are going to retrieve. While the first query of the two is required to return a single row, this is not true for the second query—if we have added multiple pictures, we will get multiple rows back.

In these circumstances, there is very little difference between the `one-to-one` relationship and a `one-to-many` relationship, except from a business perspective. That is to say, we don't associate an advert with multiple pictures because we *choose* not to, not because we cannot.

This, perhaps, explains why the expression of a `one-to-one` relationship in Hibernate is usually carried out via a `many-to-one` mapping. If you do not find that persuasive, remember that a foreign key relationship, which is the relationship that the advert column in the picture table has with the id column in the advert table, is a M:1 relationship between the entities.

In our example, the `Picture` table will be maintaining the "advert" column as a foreign key into the `Advert` table, so this must be expressed as a `many-to-one` relationship with the `Advert` object (see Listing 6-6).

Listing 6-6. *The New Picture Mapping*

```
<?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">

<class name="com.hibernatebook.chapter06.Picture">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <many-to-one
      name="advert"
      class="com.hibernatebook.chapter06.Advert"
      column="advert"/>
    <property name="caption" type="string" length="255"/>
    <property name="filename" type="string" length="32"/>
</class>
```

If you still object to the `many-to-one` relationship, you will probably find it cathartic to note that we have explicitly constrained this relationship with the "unique" attribute. You will also find it reassuring that in order to make navigation possible directly from the `Advert` to its associated `Picture`, we can in fact use a `one-to-one` mapping entry. We need to be able to navigate in this direction because we expect to retrieve adverts from the database, and then display their associated pictures (see Listing 6-7).

Listing 6-7. *The Revised Advert Mapping*

```
<?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">
```

```
<class name="com.hibernatebook.chapter06.Advert">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <property name="title" type="string" length="255"/>
    <property name="content" type="text"/>
    <one-to-one name="picture"
                class="com.hibernatebook.chapter06.Picture"
                property-ref="picture">
</class>
```

Now that we have seen how `one-to-one` and `many-to-one` relationships are expressed, we will see how a `many-to-many` relationship can be expressed.

Mapping Collections

In Figure 6-4, we show the `User` objects as having an unknown number of `Advert` instances. In the database, this is then represented with three tables, one of which is a link table between the two entity tables.

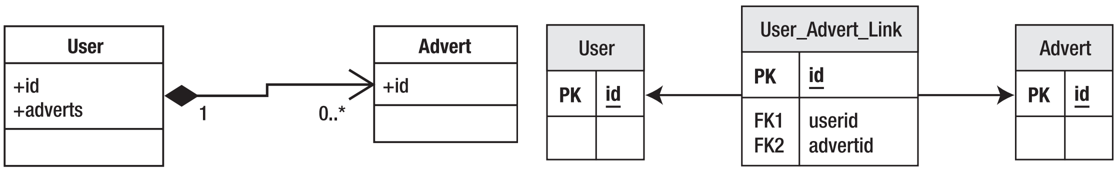

Figure 6-4. *Mapping collections*

The Java `Collection` classes provide the most elegant mechanism for expressing the `many` end of a `many-to-many` relationship in our own classes:

```
public Set getAdverts();
```

If we use generics, we can give an even more precise speecification:

```
public Set<Advert> getAdverts();
```

■ **Note** A lot of legacy code will not use generics. However, if you have the opportunity you should do so, as it allows you to make this sort of distinction clear at the API level, instead of at the documentation level. Hibernate 3 is compatible with Java 5 Generics.

Of course we can place values (of `Object` type) into collections as well as entities, and Java 5 introduced autoboxing so that we have the illusion of being able to place primitives into them as well:

```
List<Integer> ages = getAges();
int first = ages.get(0);
```

The only catch with collection mapping is that an additional table is required to correctly express the relationship between the owning table and the collection. Here is how it should be done; the entity table contains only its own attributes (see Table 6-21).

Table 6-21. *EntityTable*

Id	Name
1	`Entity 1`

while a separate collection table contains the actual values (see Table 6-22). In this case, we are linking a `List` to the owning entity, so we need to include a column to represent the position of the values in the list as well as the foreign key into the owning entity, and the column for the actual values that are contained within the collection.

Table 6-22. *ListTable*

entityid	positionInList	listValue
1	1	`Good`
1	2	`Bad`
1	3	`Indifferent`

In a legacy schema, you may quite often encounter a situation where all of the values have been retained within a single table (see Table 6-23).

Table 6-23. *EntityTable*

Id	Name	positionInList	listValue
1	`Entity 1`	1	`Good`
1	`Entity 1`	2	`Bad`
1	`Entity 1`	3	`Indifferent`

However, it should be obvious that this is not just poor design from Hibernate's perspective, but is also bad relational design; the values in the entity's `name` attribute have been duplicated needlessly, so this is not a properly normalized table. We also break the foreign key of the table, and need to form a compound key of `id` and `positionInList`. Overall this is a poor design and we encourage you to use a second table if this is at all possible. If you must work with such an existing design, see Chapter 13 for some techniques for approaching this type of problem.

If your collection is going to contain entity types instead of value types, the approach is essentially the same, but your second table will contain keys into the second entity table instead of value types. This changes the combination of tables into the situation shown in the entity relationship diagram (see Figure 6-4) where we have a link table joining two major tables into a `many-to-many` relationship. This is a very familiar pattern in properly normalized relational schemas.

The following code represents a mapping of a `Set` attribute representing the `Adverts` with which the `User` class is associated:

```
<set name="adverts"
     table="user_advert_link"
     cascade="save-update">
   <key column="userid"/>
   <many-to-many
      class="com.hibernatebook.chapter06.Advert"
      column="advertid"/>
</set>
```

Hibernate's use of collections tends to expose the lazy-loading issues more than most other mappings. If you enable lazy loading, the `Collection` that you retrieve from the session will be a proxy implementing the relevant collection interface (in our example, `Set`) rather than one of the usual Java concrete collection implementations.

This allows Hibernate to retrieve the contents of the collection only as they are required by the user. If you load an entity, consult a single item from the collection, and then discard it, often only a handful of SQL operations will be required. If the collection in question represents hundreds of entity instances, the performance advantages of lazy loading (compared with the massive task of reading in *all* of the entities concerned) are massive.

However, you will need to be careful to ensure that you do not try to access the contents of a lazily loaded collection at a time when it is no longer associated with the session unless you can be certain that the entity that you are accessing has already been loaded.

Mapping Inheritance Relationships

In Figure 6-5, we show a simple class hierarchy. The superclass is `Advert` and there are two classes derived from this: a `Personal` class to represent personal advertisements and a `Property` class to represent property advertisements.

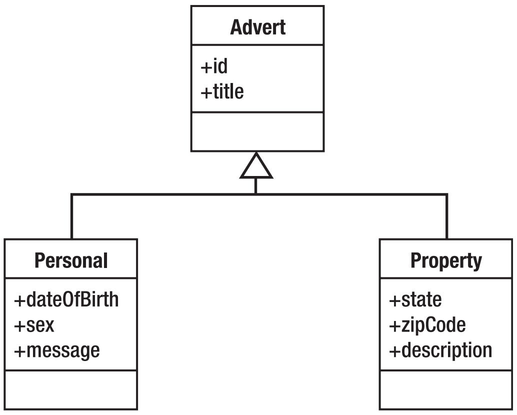

Figure 6-5. *A simple inheritance hierarchy*

Hibernate can represent inheritance relationships in a relational schema in three ways, each mapped in a slightly different way. These are

- One table for each concrete class implementation
- One table for each subclass (including interfaces and abstract classes)
- One table for each class hierarchy

Each of these techniques has different costs and benefits, so we will show you an example mapping from each and discuss some of these issues.

One Table per Concrete Class

This approach is the easiest to implement. You map each of the concrete classes as normal, writing mapping elements for each of its persistent properties (including those that are inherited). No mapping files are required for interfaces and abstract classes.

Figure 6-6 shows the schema required to represent the hierarchy from Figure 6-5 using this technique.

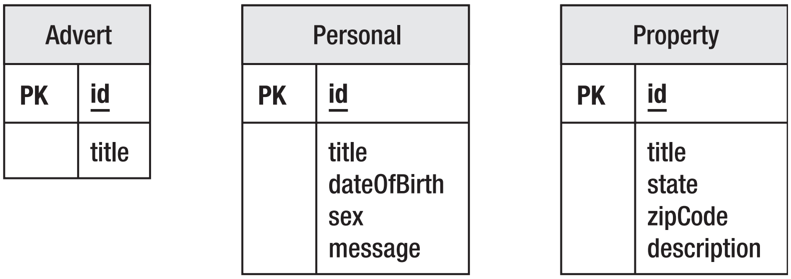

Figure 6-6. *Mapping one table per concrete class*

While this is easy to create, there are several disadvantages; the data belonging to a parent class is scattered across a number of different tables, so a query couched in terms of the parent class is likely to cause a large number of select operations. It also means that changes to a parent class can touch an awful lot of tables. We suggest you file this approach under "quick and dirty solutions."

In Listing 6-8, we demonstrate how a derived class (`Property`) can be mapped to a single table independently of its superclass (`Advert`).

Listing 6-8. *Mapping a Property Advert with the One Table per Concrete Class Approach*

```
<hibernate-mapping>
   <class name="com.hibernatebook.chapter06.Property">
      <id name="id" type="int">
         <generator class="native"/>
      </id>
      <property name="title" type="string" length="255"/>
      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </class>
</hibernate-mapping>
```

One Table per Subclass

A slightly more complex mapping is to provide one table for each class in the hierarchy, including the abstract and interface classes. The pure *is-a* relationship of our class hierarchy is then converted into a *has-a* relationship for each entity in the schema.

Figure 6-7 shows the schema required to represent the hierarchy from Figure 6-5 using this technique.

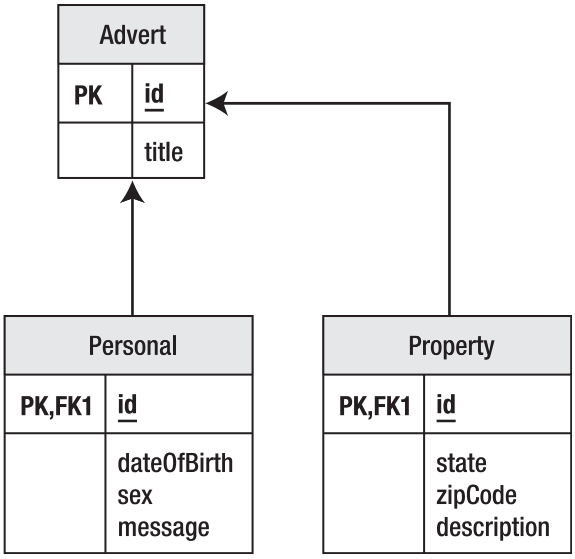

Figure 6-7. *Mapping one table per subclass*

We like this approach as it is conceptually easy to manage, does not require complex changes to the schema when a single parent class is modified, and is similar to how most JVM manage the same data behind the scenes.

The disadvantage of this approach is that while it works well from an object-oriented point of view, and is correct from a relational point of view, it can result in poor performance. As the hierarchy grows, the number of joins required to construct a leaf class also grows.

The technique works well for shallow inheritance hierarchies. Deep inheritance hierarchies are often a symptom of poorly designed code, so you may want to reconsider your application architecture before abandoning this technique. In our opinion, it should be preferred until performance issues are substantially proven to be an issue.

In Listing 6-9, we show how you can map a derived class (`Property`) as a table joined to another representing the superclass (`Advert`).

Listing 6-9. *Mapping a Property Advert with the One Table per Subclass Approach*

```
<hibernate-mapping>
   <joined-subclass
      name="com.hibernatebook.chapter06.Property"
      extends="com.hibernatebook.chapter06.Advert">

      <key column="advertid"/>

      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </joined-subclass>
</hibernate-mapping>
```

Note in the mapping that we replace `class` with `joined-subclass` to associate our mapping explicitly with the parent. You specify the entity that is being extended, and replace the `id` and `title` classes from the subclass with a single key element that maps the foreign key column to the parent class table's primary key. Otherwise the `joined-subclass` element is virtually identical to the `class` element. Note, however, that a `joined-subclass` cannot contain `subclass` elements and vice versa—the two strategies are not compatible.

One Table per Class Hierarchy

The last of the inheritance mapping strategies is to place each inheritance hierarchy in its own table. The fields from each of the child classes are added to this table, and a discriminator column contains a key to identify the base type represented by each row in the table.

Figure 6-8 shows the schema required to represent the hierarchy from Figure 6-5 using this technique.

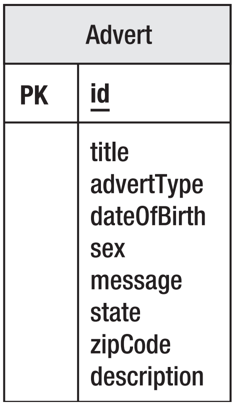

Figure 6-8. *Mapping one table per hierarchy*

This technique offers the best performance—for simple queries on simple classes even in the deepest of inheritance hierarchies, a single select may suffice to gather all of the fields to populate the entity.

Conversely, this is not a satisfying representation of the `Attribute`. Changes to members of the hierarchy will usually require a column to be altered, added, or deleted from the table. This will often be a very slow operation. As the hierarchy grows (horizontally as well as vertically), so too will the number of columns required by this table.

Each mapped subclass must specify the class that it extends and a value that can be used to discriminate this subclass from the other classes held in the same table. Thus, this is known as the discriminator value and mapped with a `discriminator-value` attribute in the `subclass` element (see Listing 6-10).

Listing 6-10. *Mapping a Property Advert with the Table per Class Hierarchy Approach*

```
<hibernate-mapping>
   <subclass
      name="com.hibernatebook.chapter06.Property"
      extends="com.hibernatebook.chapter06.Advert"
      discriminator-value="property">

      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </subclass>
</hibernate-mapping>
```

Note that this also requires the specification of a discriminator column for the root of the class hierarchy, from which the discriminator values identifying the types of the child classes can be obtained (see Listing 6-11).

Listing 6-11. *The Addition to Advert.hbm.xml Required to Support a Table per Class Hierarchy Approach*

```
<discriminator column="advertType" type="string"/>
```

A subclass mapping cannot contain `joined-subclass` elements and vice versa—the two strategies are not compatible.

More Exotic Mappings

The Hibernate mapping DTD is large. We have discussed the core set of mappings that you will use on a day-to-day basis, but before we move on we will take a very quick tour around four of the more interesting remaining mapping types.

Any

The any tag represents a polymorphic association between the attribute and several entity classes. The mapping is expressed in the schema with a column to specify the type of the related entity, and then columns for the identifier of the related entity.

Because a proper foreign key cannot be specified (being dependent upon multiple tables), this is not the generally recommended technique for making polymorphic associations. Where possible, prefer the techniques described in the previous "Mapping Inheritance Relationships" section.

Array

The array tag represents the innate array feature of the Java language. The syntax of this is virtually identical to that used for the List collection class, and we recommend the use of List except where primitive values are to be stored, or where you are constrained by an existing application architecture to work with arrays.

Join

The join mapping is a special type of one-to-one relationship, which allows you to combine properties (rather than entities) of another table into your entity. A similar effect can be achieved using views on databases that support writable views.

A new Hibernate 3 application will never have an absolute requirement for the join mapping, but we demonstrate in Chapter 13 how this can be of particular use when dealing with a legacy database schema.

Dynamic Component

While the full blown dynamic class approach, which we discussed briefly in the "Entities" section at the beginning of the chapter, is really only suitable for prototyping exercises, the dynamic component allows some of that flexibility in a package that reflects some legitimate techniques.

The dynamic-component element permits you to place any of the items that can be mapped with the normal component element into a map with a given key. For example, we could obtain and combine several items of information relating to an entity's ownership into a single Map with named elements:

```
<dynamic-component name="ownership">
   <property name="user" type="string" column="user"/>
   <many-to-one
      name="person"
      class="com.hibernatebook.chapter06.Person"
      column="person_id"/>
</dynamic-component>
```

The code to access this information in the entity is then very familiar:

```
Map map = entity.getOwnership();
System.out.println(map.get("user"));
System.out.println(map.get("person"));
```

The output would then be

```
dcminter
person: { "Dave Minter", 33, "5'10" }
```

Summary

In this chapter, we have looked at the data types supported by Hibernate 3: entities, values, and components. We have seen how all three can be expressed in a mapping file and how each relates to the underlying database schema. We have listed the attributes available to the major mapping elements, and we looked at some detailed worked examples of the elements that you will use most frequently when working with Hibernate.

In the next chapter, we will look at how subsets of the data mapped into the database can be retrieved—which is, after all, the whole point of using a database in the first place.

CHAPTER 7

Querying Objects with Criteria

In this chapter, we are going to discuss retrieving objects from the database using Hibernate's criteria query API. Hibernate provides three different ways to retrieve data: the criteria query API, the Hibernate Query Language (HQL), and native SQL queries. The criteria query API provides a set of Java objects you can use to construct your queries. Hibernate provides its own object query language, HQL, which allows you to use SQL-like syntax to retrieve objects from the database. Lastly, Hibernate provides an API for using SQL directly against the database, which allows you to use database-specific SQL that Hibernate does not support. We discuss criteria in this chapter—in Chapter 8 we will cover HQL and SQL queries.

The criteria query API lets you build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL. The criteria query API also includes Query By Example (QBE) functionality for supplying example objects that contain the properties you would like to retrieve. In Hibernate 3, the criteria query API also includes projection and aggregation methods, including count. In Hibernate 2, you had to download a patch to the criteria query API to enable row counting.

Using the Criteria Query API

The easiest way to retrieve data with Hibernate is to use the criteria query API. Instead of writing an SQL `SELECT` statement or an HQL statement, you can build up a query programmatically. You can also use QBE to retrieve objects that match an object that you have partially populated (we will discuss QBE in more detail further on in the chapter). The `org.hibernate.Criteria` interface defines the available methods for one of these criteria query objects. The Hibernate `Session` interface contains several `createCriteria()` methods. Pass the persistent object's class or its entity name to the `createCriteria()` method, and Hibernate will create a `Criteria` object that returns instances of the persistence object's class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class. We will illustrate this criteria with an example, but we wanted to first introduce the sample application we are using in this chapter, and the next chapter on HQL. The sample application has three classes: `Supplier`, `Product`, and `Software`. The `Supplier` class (see Listing 7-1) has a `name` property and a `List` collection of `Product` objects.

Listing 7-1. *The Supplier Class*

```
package com.hibernatebook.criteria;

import java.util.ArrayList;
import java.util.List;

public class Supplier
{
    private int id;
    private String name;
    private List products = new ArrayList();

    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public List getProducts()
    {
        return products;
    }
    public void setProducts(List products)
    {
        this.products = products;
    }
}
```

The `Product` class (see Listing 7-2) has `name`, `price`, and `description` properties, along with a reference to its parent supplier.

Listing 7-2. *The Product Class*

```
package com.hibernatebook.criteria;

public class Product
{
    private int id;
    private Supplier supplier;

    private String name;
    private String description;
    private double price;

    public Product()
    {
        super();
    }

    public Product(String name, String description, double price)
    {
        super();
        this.name = name;
        this.description = description;
        this.price = price;
    }

    public String getDescription()
    {
        return description;
    }
    public void setDescription(String description)
    {
        this.description = description;
    }
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }

    public Supplier getSupplier()
    {
        return supplier;
    }
    public void setSupplier(Supplier supplier)
    {
        this.supplier = supplier;
    }

    public double getPrice()
    {
        return price;
    }
    public void setPrice(double price)
    {
        this.price = price;
    }
}
```

The `Software` class (see Listing 7-3) extends the `Product` class and adds a `version` property—we added this subclass so we could demonstrate polymorphism with Hibernate's queries.

Listing 7-3. *The Software Class*

```
package com.hibernatebook.criteria;

public class Software extends Product
{
    private String version;

    public Software()
    {
        super();
    }
    public Software(String name, String description, double price, String version)
    {
        super(name, description, price);
        this.setVersion(version);
    }
    public String getVersion()
    {
        return version;
    }
    public void setVersion(String version)
    {
        this.version = version;
    }
}
```

The Hibernate mapping files for these three classes are in the source directory for this book, along with a test harness for populating the database and running these criteria examples.

The simplest way to use a criteria query is this:

```
Criteria crit = session.createCriteria(Product.class);
List results = crit.list();
```

We mentioned previously that we wanted to demonstrate polymorphism with Hibernate. When you run this example with our sample data, you will get all objects that are instances of the `Product` class—this includes any instances of the `Software` class.

Moving on from this simple example, we need to add constraints to our criteria queries so we can winnow the result set down.

Using Restrictions with Criteria

The criteria query API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over 30. You may add these restrictions to a criteria query object with the `add()` method on the `Criteria` object. The `add()` method takes an `org.hibernate.criterion.Criterion` object that represents an individual restriction. You can have more than one restriction for a criteria query.

■**Note** In Hibernate 3, the `org.hibernate.criterion.Restrictions` class is the new name of the `net.sf.hibernate.expression.Expression` class in Hibernate 2.1. In addition, the `sql()` methods on `Expression` are now `sqlRestriction()` on the `Restrictions` class.

Although you could create your own objects that either implement the `Criterion` object or extend an existing `Criterion` object, we recommend that you use Hibernate's built-in `Criterion` objects from your application's business logic, rather than implementing your business logic as `Criterion` objects. For instance, you could create your own factory class that returned instances of Hibernate's criterion objects set up for your application's restrictions.

Use the factory methods on the `org.hibernate.criterion.Restrictions` class to obtain instances of the `Criterion` objects. To retrieve objects that have a property value that equals your restriction, use the `eq()` method on `Restrictions`:

```
public static SimpleExpression eq(String propertyName, Object value)
```

We would typically nest the `eq()` method in the `add()` method on the `Criteria` object. Here is an example of how this would look if we were searching for products with the name "Mouse":

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("name","Mouse"));
List results = crit.list()
```

Next, we could search for products that do not have the name "Mouse." For this we would use the `ne()` method on the `Restrictions` class to obtain a not-equal restriction:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ne("name","Mouse"));
List results = crit.list();
```

Tip You cannot use the not-equal restriction to retrieve records with a null value in the database for that property. If you need to retrieve objects with null properties, you will have to use the `isNull()` restriction, which we discuss further on in the chapter. You can combine the two with an `OR` logical expression, which we also discuss later in the chapter.

Rather than search for just exact matches, we can also retrieve all objects that have a property that matches part of a given pattern. We would need to create an SQL `LIKE` clause, with either the `like()` method or the `ilike()` method. The `ilike()` method is case-insensitive. In either case, we have two different ways to call the method:

```
public static SimpleExpression like(String propertyName, Object value)
public static SimpleExpression like(String propertyName, String value, ➥
                                                MatchMode matchMode)
```

The first `like()` or `ilike()` method takes a pattern for matching. Use the `%` character as a wildcard to match parts of the string:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.like("name","Mou%"));
List results = crit.list();
```

The second `like()` or `ilike()` method uses an `org.hibernate.criterion.MatchMode` object to specify how to match the specified value to the stored data. The `MatchMode` object (a type-safe enumeration) has four different matches:

- `ANYWHERE`: Any place in the string
- `END`: The end of the string
- `EXACT`: An exact match
- `START`: The beginning of the string

Here is an example that uses the `ilike()` method to search for case-insensitive matches at the end of the string:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ilike("name","browser", MatchMode.END));
List results = crit.list();
```

The `isNull()` and `isNotNull()` restrictions allow you to do a search for objects that have (or do not have) null property values. This is easy to demonstrate:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.isNull("name"));
List results = crit.list();
```

Several of the restrictions are useful for doing math comparisons. The greater than comparison is `gt()`, the greater than or equal to comparison is `ge()`, the less than comparison is `lt()`, and the less than or equal to comparison is `le()`. We can do a quick retrieval of all products with prices over 25:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price",new Double(25.0)));
List results = crit.list();
```

Moving on, we can start to do more complicated queries with the criteria query API. If we want to do logical expressions, we can combine `AND` and `OR` restrictions together in logical expressions. When you add more than one constraint to a criteria query, the criteria query interprets that as an `AND`:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price",new Double(25.0)));
crit.add(Restrictions.like("name","K%"));
List results = crit.list();
```

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the `or()` method on the `Restrictions` class:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
Criterion name = Restrictions.like("name","Mou%");
LogicalExpression orExp = Restrictions.or(price,name);
crit.add(orExp);
List results = crit.list();
```

The `orExp` logical expression is treated like any other criterion after we create it. We can even add another restriction to the criteria (which will be treated as an `AND`):

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
Criterion name = Restrictions.like("name","Mou%");
LogicalExpression orExp = Restrictions.or(price,name);
crit.add(orExp);
crit.add(Restrictions.ilike("description","blocks%"));
List results = crit.list();
```

If we wanted to create an `OR` expression with more than two different criteria, we would use an `org.hibernate.criterion.Disjunction` object to represent a disjunction. You can obtain this object from the `disjunction()` factory method on the `Restrictions` class.The disjunction is more convenient than building a tree of `OR` expressions in code. To represent an `AND` expression with more than two criteria, use the `conjunction()` method, although you can easily just add those to the criteria query object. The conjunction is also more convenient than building a tree of `AND` expressions in code. Here is an example that uses the disjunction:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
Criterion name = Restrictions.like("name","Mou%");
Criterion desc = Restrictions.ilike("description","blocks%");
Disjunction disjunction = Restrictions.disjunction();
disjunction.add(price);
disjunction.add(name);
disjunction.add(desc);
crit.add(disjunction);
List results = crit.list();
```

The last type of restriction is the SQL restriction, `sqlRestriction()`. This restriction allows you to directly specify SQL in the criteria query API. This is useful if you need to use SQL clauses that Hibernate does not support through the criteria query API. Your application's code does not need to know the name of the table your class uses—use `{alias}` to signify the class's table:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.sqlRestriction("{alias}.name like 'Mou%'"));
List results = crit.list()
```

You can also use the other two `sqlRestriction()` methods to pass JDBC parameters and values into the SQL statement. Use the standard ? JDBC parameter placeholder in your SQL fragment.

Paging Through the Result Set

One common application pattern we find ourselves developing over and over again is pagination through the result set of a database query. When we say pagination, we mean an interface in which the user sees part of the result set at a time, with navigation to go forward and backward through the results. A naïve pagination implementation might load the entire result set into memory for each navigation, with terrible performance results. Both of us worked on improving performance for separate projects with this problem. The problem appeared late in testing because the sample dataset that developers were working with was trivial, and they did not notice any performance problems until the first test data load.

If you are programming directly to the database, you will typically use proprietary database SQL or database cursors to support paging. Hibernate abstracts this away for you—behind the scenes, Hibernate uses the appropriate method for your database.

There are two methods on the `Criteria` interface for paging: `setFirstResult()` and `setMaxResults()`. The `setFirstResult()` method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the `setMaxResults()` method. Using both of these together, we could construct a paging component in our web or Swing application. We have a very small dataset in our sample application, so here is an admittedly trivial example:

```
Criteria crit = session.createCriteria(Product.class);
crit.setFirstResult(1);
crit.setMaxResults(2);
List results = crit.list();
```

As you can see, this makes paging through the result set easy. You can increase the first result you return from 1, to 21, to 41, etc. to page through the result set. If you only have one result in your result set, Hibernate has a shortcut method for obtaining just that object.

Obtaining a Unique Result

Sometimes you know you are only going to return one or zero objects with a given criteria query. This could be because you are calculating an aggregate (like `COUNT`, which we discuss later), or because your restrictions lead to a unique result. You may also limit the results of any result set to just the first result, using the `setMaxResults()` method we discussed earlier. If you would like to just obtain an `Object` instead of a `List`, the `uniqueResult()` method on the `Criteria` object returns an `Object` or `null` if there are zero results. If there is more than one result, the `uniqueResult()` method throws a `HibernateException`.

A short example demonstrates having a result set that would have included more than one result, except that it was limited with the `setMaxResults()` method:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
crit.setMaxResults(1);
Product product = (Product) crit.uniqueResult();
//test for null here if needed
```

Again, we stress that you need to make sure that your query only returns one (or zero) results if you use the `uniqueResult()` method. Otherwise, Hibernate will throw a `NonUniqueResultException` exception, which may not be what you would expect. Hibernate does not just pick the first result and return it.

Sorting the Query's Results

Sorting the query's results works the same way in Hibernate as it would with SQL. The criteria query API provides the `org.hibernate.criterion.Order` class to sort your result set in either ascending or descending order according to one of your object's properties.

Create an `Order` object with either of the two static factory methods on the `Order` class: `asc()` for ascending or `desc()` for descending. Both methods take the name of the property as their only argument. After you create an `Order`, use the `addOrder()` method on the `Criteria` object to add it to the criteria query.

This example demonstrates how you would use the `Order` class:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price",new Double(25.0)));
crit.addOrder(Order.desc("price"));
List results = crit.list();
```

You may add more than one `Order` object to the `Criteria` object. Hibernate will pass them through to the underlying SQL query. Your results will be sorted by the first order, then any identical matches within the first sort will be sorted by the second order, and so on.

Hibernate passes the property value to SQL, after getting the proper database column name for the property.

Associations

To add a restriction on a class that is associated with your criteria's class, you will need to create another criteria query object. Pass the property name of the associated class to the `createCriteria()` method on the `Criteria` object, and you will have another `Criteria` object. You can get the results from either `Criteria` object, although you should pick one style and be consistent for readability's sake. We find that getting the results from the top-level `Criteria` object (the one that takes a class as a parameter) makes it clear what type of object is expected in the results.

The associations work going from one-to-many as well as from many-to-one. First, we will demonstrate how to use one-to-many associations to obtain the suppliers who sell products with a price over 25.0. Notice that we create a new `Criteria` object for the `products` property, add restrictions to the products criteria we just created, and then obtain the results from the supplier criteria query object:

```
Criteria crit = session.createCriteria(Supplier.class);
Criteria prdCrit = crit.createCriteria("products");
prdCrit.add(Restrictions.gt("price",new Double(25.0)));
List results = crit.list();
```

Going the other way, we can obtain all of the products from the supplier "MegaInc" using many-to-one associations:

```
Criteria crit = session.createCriteria(Product.class);
Criteria suppCrit = crit.createCriteria("supplier");
suppCrit.add(Restrictions.eq("name","MegaInc"));
List results = crit.list();
```

Although we used either criteria to obtain the results, when we sort the result set, it makes a difference which criteria we use for ordering the results. In the following example, we are ordering the supplier results by their name:

```
Criteria crit = session.createCriteria(Supplier.class);
Criteria prdCrit = crit.createCriteria("products");
prdCrit.add(Restrictions.gt("price",new Double(25.0)));
crit.addOrder(Order.desc("name"));
List results = prdCrit.list();
```

If we wanted to sort the suppliers by the descending price of their products, we would use the following line of code. This code would have to replace the previous `addOrder()` call on the supplier criteria object:

```
prdCrit.addOrder(Order.desc("price"));
```

Although the products are not in the result set, SQL still allows you to order by those results. If you get mixed up with which `Criteria` object you are using and pass the wrong property name for the sort by order, Hibernate will throw an exception.

Distinct Results

If you would like to work with distinct results from a criteria query, Hibernate provides a result transformer for distinct entities, `org.hibernate.transform.DistinctRootEntityResult➥Transformer`, that ensures that no duplicates will be in your criteria query's result set. Rather than using `SELECT DISTINCT` with SQL, the distinct result transformer compares each of your results using their default `hashCode()` methods, and only adds those results with unique hash codes to your result set. This may or may not be the result you would expect from an SQL `DISTINCT` query, so be careful with this. As an additional note for large databases, the comparison is done in Hibernate's Java code, not at the database, so nonunique results will still be transported across the network.

Projections and Aggregates

Instead of working with objects from the result set, you can use the results from the result set as a set of rows and columns. This is similar to how you would use data from a `SELECT` query with JDBC, and Hibernate supports properties, aggregate functions, and group by. Projections are a new feature for the criteria query API in version 3.

To use projections, start by getting the `org.hibernate.criterion.Projection` object you need from the `org.hibernate.criterion.Projections` factory class. Similar to the `Restrictions` class, `Projections` has several static factory methods for obtaining `Projection` instances. After you get a `Projection` object, add it to your `Criteria` object with the `setProjection()` method. When the `Criteria` executes, the list contains objects that you can cast to the appropriate class.

The simplest example of projections is the row counting functionality. The code looks similar to the restrictions we were working with earlier in the chapter:

```
Criteria crit = session.createCriteria(Product.class);
crit.setProjection(Projections.rowCount());
List results = crit.list();
```

The results list will contain one object, an `Integer` that contains the results of executing the `COUNT` SQL statement.

Other aggregate functions available through the `Projections` factory class include the following:

- `avg(String propertyName)`: The average of a property's value
- `count(String propertyName)`: Counts the number of times a property has a value
- `countDistinct(String propertyName)`: Counts the number of unique values the property contains
- `max(String propertyName)`: The maximum value of the property values
- `min(String propertyName)`: The minimum value of the property values
- `sum(String propertyName)`: The sum total of the property values

We can have more than one projection for a given criteria. To add multiple projections, get a projection list from the `projectionList()` method on the `Projections` class. The `org.hibernate.criterion.ProjectionList` object has an `add()` method that takes a `Projection` object. You can pass the projections list to the `setProjection()` method on the `Criteria` object, as `ProjectionList` implements the `Projection` interface. The following example demonstrates some of the aggregate functions, along with the projection list:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.max("price"));
projList.add(Projections.min("price"));
projList.add(Projections.avg("price"));
projList.add(Projections.countDistinct("description"));
crit.setProjection(projList);
List results = crit.list();
```

After you execute multiple aggregate projections, you will get a `List` with an `Object` array as the first element. The `Object` array contains all of your values, in order.

Another use of projections is to retrieve individual properties, rather than classes. For instance, we could retrieve just the name and description from our product table, instead of faulting the classes into memory. Use the `property()` method on the `Projections` class to create a `Projection` for a property. When you execute this form of query, the `list()` method will return a `List` of `Object` arrays. Each `Object` array will contain the projected properties for that row. The following example returns just the contents of the name and description columns from the `Product` data. Remember, Hibernate is polymorphic, so this also returns the name and description from the `Software` objects that inherit from `Product`.

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.property("name"));
projList.add(Projections.property("description"));
crit.setProjection(projList);
List results = crit.list();
```

Use this query style when you want to cut down on network traffic between your application servers and your database servers. For instance, if your table has a large number of columns, this can cut down your results. In other cases, you may have a large set of joins that would return a very wide result set, but you are only interested in a few columns. Lastly, if your clients have limited memory, this can save you trouble with large datasets. Be sure that you do not need to retrieve additional columns for the entire result set later, or your optimizations may actually decrease performance.

You can also group your results (SQL's `GROUP BY` clause) with the `groupProperty` projection. The following example groups the products by name and price:

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.groupProperty("name"));
projList.add(Projections.groupProperty("price"));
crit.setProjection(projList);
List results = crit.list();
```

As you can see, projections open up aggregates to the criteria query API, which means developers do not have to drop into HQL for aggregates. Projections offer a way to work with data that is closer to the JDBC result set style, which may be appropriate for some parts of your application.

Query By Example (QBE)

Another style of searching is Query By Example (QBE). Instead of programmatically building a `Criteria` object with `Criterion` objects and logical expressions, you can partially populate an instance of the object. Use the instance as an example and have Hibernate build the criteria for you behind the scenes. This keeps your code clean, and makes your project easier to test. The `org.hibernate.criterion.Example` class contains the QBE functionality. The `Example` class implements the `Criterion` interface, so you can use it like any other restriction.

For instance, if we had a user database we could construct an instance of a user object, set the property values for type and creation date, and then use the criteria query API to run a QBE query. Hibernate would return a result set containing all user objects that matched the property values that were set. Behind the scenes, Hibernate inspects the example object and constructs an SQL fragment that corresponds to the properties on the example.

To use QBE, we need to construct an example object first. Then we need to create an instance of the `Example` object, using the static `create()` method on the `Example` class. The `create()` method takes the example object as its argument. You add the `Example` object to a criteria query object just like any other `Criterion` object.

The following basic example searches for suppliers that match the name on the example `Supplier` object:

```
Criteria crit = session.createCriteria(Supplier.class);
Supplier supplier = new Supplier();
supplier.setName("MegaInc");
crit.add(Example.create(supplier));
List results = crit.list();
```

When Hibernate translates your example object into an SQL query, all of the properties on your example objects get examined. You can tell Hibernate which properties to ignore; the default is to ignore null valued properties. If we want to search our products or software with QBE, we need to either specify a price, or tell Hibernate to ignore properties with a value of zero, because we used a double primitive for storage instead of a `Double` object. The double primitive initializes to zero, while a `Double` would have been null. We can have the Hibernate `Example` object exclude zero valued properties with the `excludeZeroes()` method. We also could exclude properties by name with the `excludeProperty()` method or exclude nothing (compare both null values and zeroes) with the `excludeNone()` method:

```
Criteria crit = session.createCriteria(Product.class);
Product exampleProduct = new Product();
exampleProduct.setName("Mouse");
Example example = Example.create(exampleProduct);
example.excludeZeroes();
crit.add(example);
List results = crit.list();
```

Other options on the `Example` object include ignoring the case for strings with the `ignoreCase()` method and enabling use of SQL's `LIKE` instead of just equals for comparing strings.

We can also use associations for QBE. In the following example, we create two examples: one for the product and one for the supplier. We use the technique we explained in the "Associations" section of this chapter to retrieve objects that match both criteria:

```
Criteria prdCrit = session.createCriteria(Product.class);
Product product = new Product();
product.setName("M%");
Example prdExample = Example.create(product);
prdExample.excludeProperty("price");
prdExample.enableLike();
Criteria suppCrit = prdCrit.createCriteria("supplier");
Supplier supplier = new Supplier();
supplier.setName("SuperCorp");
suppCrit.add(Example.create(supplier));
prdCrit.add(prdExample);
List results = prdCrit.list();
```

We also ignore the price property for our product, and we use `LIKE` for object comparison, instead of equals.

The QBE API works best for searches where you are building the search from user input. The Hibernate team recommends using QBE for advanced searches with multiple fields, because its easier to set values on business objects than to manipulate restrictions with the criteria query API.

Summary

Using the criteria query API is an excellent way to get started developing with HQL. The developers of Hibernate provided a clean API for adding restrictions to queries with Java objects. Although HQL is not too difficult to learn, some developers prefer the criteria query API's compile-time syntax checking for restrictions, although column names aren't checked until runtime. The criteria query API is also more powerful in Hibernate 3 than it was in Hibernate 2.1, with the addition of projections and aggregates.

CHAPTER 8

Querying with HQL and SQL

In the last chapter, we covered Hibernate's criteria query API. In this chapter, we discuss the Hibernate Query Language (HQL) and Hibernate's native SQL functionality. Hibernate Query Language is an object-oriented query language, similar to SQL, except that instead of working with tables and columns, HQL works with persistent objects and their properties. HQL is very powerful for working with object inheritance and associations. HQL is a language, and you write HQL as strings like `from Product p`, as opposed to Hibernate's criteria queries, which are Java APIs. The criteria query API and HQL offer almost identical functionality. HQL now supports bulk update and delete queries in Hibernate 3, updated from the previous support for deleting objects with HQL in Hibernate 2. In addition, you can write SQL queries to retrieve objects from the database—Hibernate will pass the SQL through to the database, and then translate the JDBC result set into your value objects.

Hibernate Query Language

Hibernate Query Language is a powerful query language designed for use with Hibernate. Although most object-relational mapping tools and object databases offer an object query language (sometimes abbreviated as OQL), Hibernate's HQL stands out as being complete and easy to use. Although you may use SQL statements directly with Hibernate (which is covered in detail in the "Using Native SQL" section of this chapter), we strongly suggest that you use HQL (or Criteria) wherever possible to avoid database porting hassles and to take advantage of Hibernate's SQL generation and caching strategies. Another reason to use HQL over SQL is that HQL is a more compact query language. For instance, rather than specifying every relationship in the HQL query, HQL can use the relationships defined in the Hibernate mapping files.

We realize that not every developer trusts that Hibernate's generated SQL is optimized as much as possible. If you think there is a performance bottleneck with some of your queries, we recommend that you use SQL tracing on your database during performance testing of your critical components. If you do see an area that needs optimization, we suggest trying first to optimize using HQL, only later dropping into native SQL. Hibernate 3 provides statistics through a JMX MBean that you can use for analyzing Hibernate's performance—Hibernate's statistics also give you insight into how caching is performing.

■**Note** If you would like to execute HQL statements through a GUI-based tool, the Hibernate team provides a Hibernate console for Eclipse in the Hibernate Tools subproject. This console is a plug-in for recent versions of Eclipse. For each project, you will need to provide a Java object model and your Hibernate configuration and mapping information to the Hibernate console.

First Example with HQL

The simplest HQL query returns all objects for a given class in the database. Similar to SQL, we use the HQL clause `from`. With HQL, you do not have to use the leading `select` clause for this query—instead we can use the following simple shortcut query to select all objects from the Product table:

```
from Product
```

■**Note** Like all HQL syntax, you can write `from` in lowercase or uppercase (or mixed case). However, any Java classes or properties you reference in an HQL query have to be specified in the proper case. For instance, when you query for instances of a Java class named `Product`, the HQL query `from Product` is the equivalent of `FROM Product`. However, the HQL query `from product` is not the same as the HQL query `from Product`. Because Java class names are case-sensitive, Hibernate is case-sensitive about class names as well.

Embedding the following HQL statement into our application is straightforward. The `org.hibernate.Session` object contains a method named `createQuery()`:

```
public Query createQuery(String queryString) throws HibernateException
```

The `createQuery()` method takes a valid HQL statement, and returns an `org.hibernate.Query` object. The `Query` interface is analogous to the `org.hibernate.Criteria` interface we discussed in Chapter 7. The `Query` class provides methods for returning the query results as a Java `List`, as an `Iterator`, or as a unique result. Other functionality includes named parameters, results scrolling, JDBC fetch sizes, and JDBC timeouts. You can also add a comment to the SQL that Hibernate creates, which is useful for tracing which HQL statements correspond to which SQL statements.

To keep things simple with this example, we are using the same database and object model we set up for Chapter 7. The object model contains `Supplier`, `Product`, and `Software` classes. For more on this object model, see the beginning of Chapter 7.

Our first example executes our HQL statement, `from Product`, and then retrieves a `List` of `Product` objects:

```
Query query = session.createQuery("from Product");
List results = query.list();
```

Many of our other examples in this chapter use the same Java syntax as this example. We are going to provide just the HQL for these examples—you can execute them the same way we did here, substituting that HQL for the `from Product` HQL statement. This should make each example clearer as to what you should be looking at. You could also execute these HQL statements in Hibernate's Eclipse-based HQL console.

Logging the Underlying SQL

Hibernate can output the underlying SQL behind your HQL queries into your application's log file. This is especially useful if the HQL query does not give the results you expect, or the query takes longer than you wanted. You can run the SQL that Hibernate generates directly against your database in the database's query analyzer to determine what the problem is. This is not a feature you will have to use often, but it becomes very useful when you have to turn to your database administrators for help with tuning your Hibernate application.

The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the `hibernate.show_sql` property. Set this property to `true` in your `hibernate.properties` or `hibernate.cfg.xml` configuration files, and Hibernate will output the SQL into the logs. You do not need to enable any other logging settings, although setting logging for Hibernate to `debug` also outputs the generated SQL statements, along with a lot of other verbiage.

After enabling SQL output in Hibernate, you may rerun the previous example. Here is the generated SQL statement for the HQL statement `from Product`:

```
select product0_.id as id, product0_.name as name0_, product0_.description ➥
as descript3_0_, product0_.price as price0_, product0_.supplierId ➥
as supplierId0_, product0_1_.version as version1_, ➥
case when product0_1_.productId is not null then 1 ➥
when product0_.id is not null then 0 end ➥
as clazz_ from Product product0_ left outer join Software product0_1_ ➥
on product0_.id=product0_1_.productId
```

As an aside, remember that the `Software` class inherits from `Product`, which does complicate Hibernate's generated SQL for this simple query. When we select all objects from our simple `Supplier` class, the generated SQL for the HQL query `from Supplier` is much simpler:

```
select supplier0_.id as id, supplier0_.name as name2_ from Supplier supplier0_
```

When you look in your application's output for the Hibernate SQL statements, they will be prefixed with `Hibernate:`. The previous SQL statement would look like this:

```
Hibernate: select supplier0_.id as id, supplier0_.name as name2_ ➥
from Supplier supplier0_
```

If you turn your log4j logging up to `debug` for the Hibernate classes, you will see SQL statements in your log files, along with lots of information about how Hibernate parsed your HQL query and translated your HQL query into SQL.

Commenting the Generated SQL

Tracing your HQL statements to the generated SQL can be difficult, so Hibernate provides a commenting facility on the `Query` object that lets you apply a comment to an individual query. The `Query` interface has a `setComment()` method that takes a `String` object as an argument:

```
public Query setComment(String comment)
```

Use this to identify the SQL output in your application's logs, if SQL logging is enabled. For instance, if we add a comment to this example, the Java code would look like this:

```
String hql = "from Supplier";
Query query = session.createQuery(hql);
query.setComment("My HQL: " + hql);
List results = query.list();
```

The output in your application's log will have the comment in a Java-style comment before the SQL:

```
Hibernate: /*My HQL: from Supplier*/ select supplier0_.id as id, supplier0_.name ➥
as name2_ from Supplier supplier0_
```

We have found this useful for identifying SQL in our logs, especially because the generated SQL is a little difficult to follow when you are scanning large quantities of it in logs.

From Clause and Aliases

We have already discussed the basics of the `from` clause in HQL in the "First Example of HQL" section. The most important feature to note is the *alias*. Hibernate allows you to assign aliases to the classes in your query with the `as` clause. Use the aliases to refer back to the class inside the query. For instance, our previous simple example would be

```
from Product as p
```

or

```
from Product as product
```

You'll see either alias naming convention in applications. The `as` keyword is optional—you can also specify the alias directly after the class name:

```
from Product product
```

If you need to fully qualify a class name in HQL, just specify the package and class name. Hibernate will take care of most of this behind the scenes, however, so you only really need this if you have classes with duplicate names in your application. If you needed to do this in Hibernate, use syntax such as the following:

```
from com.hibernatebook.criteria.Product
```

The `from` clause is very basic and useful for working directly with objects. If you would like to work with the object's properties without loading the full objects into memory, you may use the `select` clause.

Select Clause and Projection

The select clause provides more control over the result set than the from clause. If you would like to obtain the properties of objects in the result set, use the select clause. For instance, we could run a projection query on the products in the database that only returned the names, instead of loading the full object into memory:

```
select product.name from Product product
```

The result set for this query will contain a List of Java String objects. Additionally, we could retrieve the prices and the names for each product in the database:

```
select product.name, product.price from Product product
```

This result set contains a List of Object arrays—each array represents one set of properties (in this case, a name and price pair).

If a few properties were all we were interested in, we could save network traffic to the database server and memory on the application's machine.

Using Restrictions with HQL

Similar to SQL, we use the where clause to select results that match our query's expressions. HQL provides many different possible expressions that you may use to construct a query. The list is similar to the supported restrictions for the criteria query API. From the HQL language grammar, we have the following possible expressions:

- *Logic operators*: OR, AND, NOT
- *Equality operators*: =, <>, !=, ^=
- *Comparison operators*: <, >, <=, >=, like, not like, between, not between
- *Math operators*: +, -, *, /
- *Concatentation operator*: ||
- *Cases*: case when <logical expression> then <unary expression> else <unary expression> end
- *Collection expressions*: some, exists, all, any

In addition, you may also use the following expressions in the where clause:

- *HQL named parameters*: :date, :quantity
- *JDBC query parameters*: ?
- *Date and time SQL-92 functional operators*: current_time(), current_date(), current_timestamp()
- *SQL Functions (supported by the database)*: length(), upper(), lower(), ltrim(), rtrim(), etc.

Using these restrictions, we can build a `where` clause in HQL that is as powerful as an SQL query. For many queries, HQL syntax is more compact and elegant than the criteria query API syntax. For instance, here is an example of a criteria query that uses logical expressions:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
Criterion name = Restrictions.like("name","Mou%");
LogicalExpression orExp = Restrictions.or(price,name);
crit.add(orExp);
crit.add(Restrictions.ilike("description","blocks%"));
List results = crit.list();
```

The equivalent HQL would be the following:

```
from Product where price > 25.0 and name like 'Mou%'
```

We would have to wrap that HQL in a few Java calls, but we find this particular example to be clearer in HQL. In the previous HQL example, you can see that we used the `where` clause with a greater than comparison operator, an `and` logical operator, and the `like` comparison operator. You do have to enclose literal strings in quotes in HQL, and to find names that have the literal `'Mou'` at the beginning of the string, we used `%` in the query.

Using Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing queries that accept input from the user easy, and you do not have to defend against SQL injection attacks.

■ **Note** SQL injection is an attack against applications that create SQL directly from user input with string concatenation. For instance, if we accept a name from the user through a web application form, then very bad form would be to construct an SQL (or HQL) query like this:

```
String sql = "select p from products where name = '" + name + "'";
```

A malicious user could pass a name to the application that contained a terminating quote and semicolon, followed by another SQL command (such as `delete from products`) that would let them do whatever they wanted. They would just need to end with another command that matched the SQL statement's ending quote. This is a very common attack, especially if the malicious user can guess details of your database structure.

You could escape the user's input yourself for every query, but it is much less of a security risk if you let Hibernate escape all of your input with named parameters. Hibernate's named parameters are similar to the JDBC query parameters (`?`) you may already be familiar with, but Hibernate's parameters are less confusing. It is also more straightforward to use Hibernate's named parameters if you have a query that uses the same parameter in multiple places.

When you use JDBC query parameters, any time you add, change, or delete parts of the SQL statement, you need to update your Java code that sets its parameters, because the parameters are indexed, based on the order they appear in the statement. Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters further up or back in the query.

The simplest example of named parameters uses regular SQL types for the parameters:

```
String hql = "from Product where price > :price";
Query query = session.createQuery(hql);
query.setDouble("price",25.0);
List results = query.list();
```

Although that is interesting, we can use some of HQL's object-oriented features to provide objects as values for named parameters. The `Query` interface has a `setEntity()` method that takes the name of a parameter and an object. Using this functionality, we could retrieve all of the products that have a supplier whose object we already have:

```
String supplierHQL = "from Supplier where name='MegaInc'";
Query supplierQuery = session.createQuery(supplierHQL);
Supplier supplier = (Supplier) supplierQuery.list().get(0);

String hql = "from Product as product where product.supplier=:supplier";
Query query = session.createQuery(hql);
query.setEntity("supplier",supplier);
List results = query.list();
```

If you'd like, you can also use regular JDBC query parameters in your HQL queries. We do not particularly see any reason why you would want to, but they do work.

Paging Through the Result Set

As we discussed in Chapter 7, pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user—instead we can page through the result set and retrieve the results we are going to display a chunk at a time.

There are two methods on the `Query` interface for paging: `setFirstResult()` and `setMaxResults()`, just like the `Criteria` interface. The `setFirstResult()` method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the `setMaxResults()` method. Our HQL is unchanged—we only need to modify the Java code that executes the query. Excuse our tiny dataset for this trivial example of pagination:

```
Query query = session.createQuery("from Product");
query.setFirstResult(1);
query.setMaxResults(2);
List results = query.list();
displayProductsList(results);
```

You can change the numbers around and play with the pagination. If you turn on SQL logging, you can see what SQL commands Hibernate uses for pagination. For the Open Source database HSQLDB, Hibernate uses `top` and `limit`.

If you only have one result in your HQL result set, Hibernate has a shortcut method for obtaining just that object.

Obtaining a Unique Result

Similar to the criteria query API, HQL's `Query` interface provides a `uniqueResult()` method for obtaining just one object from an HQL query. Although your query may only yield one object, you may also use the `uniqueResult()` method with other result sets if you limit the results to just the first result. You could use the `setMaxResults()` method we discussed in the previous section. The `uniqueResult()` method on the `Query` object returns an `Object` or null if there are zero results. If there is more than one result, the `uniqueResult()` method throws a `NonUniqueResultException`.

A short example demonstrates having a result set that would have included more than one result, except that it was limited with the `setMaxResults()` method:

```
String hql = "from Product where price>25.0";
Query query = session.createQuery(hql);
query.setMaxResults(1);
Product product = (Product) query.uniqueResult();
//test for null here if needed
```

Unless your query only returns one (or zero) results, the `uniqueResult()` method will throw a `NonUniqueResultException` exception. Do not expect Hibernate just to pick off the first result and return it—either set the maximum results of the HQL query to one, or obtain the first object from the result list.

Sorting Your Results with Order By

To sort your HQL query's result sets, you will need to use the `order by` clause. You can order the results by any property on the objects in the result set: either ascending (`asc`) or descending (`desc`). Use ordering on more than one property in the query if you need to. A typical HQL query for sorting results looks like this:

```
from Product p where p.price>25.0 order by p.price desc
```

If we wanted to sort by more than one property, we would just add the additional properties to the end of the `order by` clause, separated by commas. For instance, we could sort by product price and the supplier's name:

```
from Product p order by p.supplier.name asc, p.price asc
```

HQL is more straightforward for ordering than the equivalent approach using the criteria query API.

Associations

Associations allow you to use more than one class in an HQL query, just as SQL allows you to use joins between tables in the relational database. Add an association to an HQL query with the `join` clause. Hibernate supports five different types of joins: `inner`, `cross`, `left outer`, `right outer`, and `full outer join`. If you use `cross join`, just specify both classes in the `from` clause (`from Product p, Supplier s`). For the other joins, use a join clause after the `from` clause. Specify the type of join, the object property to join on, and an alias for the other class.

We can use the `inner join` to obtain the supplier for each product, and then retrieve the supplier name, product name, and product price:

```
select s.name, p.name, p.price from Product p inner join p.supplier as s
```

You can retrieve the objects using similar syntax:

```
from Product p inner join p.supplier as s
```

We used aliases in these HQL statements to refer to the entities in our query expressions. These are particularly important in queries with associations that refer to two different entities with the same class—for instance, if we are doing a join from a table back to itself. Commonly, these types of joins are used to organize tree data structures. We discussed aliases in the "From Clause and Aliases" section of this chapter.

Notice that Hibernate does not return `Object` objects in the result set; instead Hibernate returns `Object` arrays in the results. You will have to access the contents of the `Object` arrays to get the `Supplier` and the `Product` objects.

If you would like to start optimizing performance, you can ask Hibernate to fetch the associated objects and collections for an object in one query. If we were using lazy loading with Hibernate, the objects in the collection would not be initialized until we accessed them. If we use `fetch` on a join in our query, we can ask Hibernate to retrieve the objects in the collection at the time the query executes. Add the `fetch` keyword after the `join` in the query:

```
from Supplier s inner join fetch s.products as p
```

When you use `fetch` for a query like this, Hibernate will return only the `Supplier` objects, not the `Product` objects. This is because you are specifying the join so Hibernate knows which objects to fetch (instead of lazy loading). If you need to get the `Product` objects, access them through the associated `Supplier` object. You cannot use the properties of the `Product` objects in expressions in the `where` clause, for instance. Use of the `fetch` keyword overrides any settings you have in the mappings file for object initialization.

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. The `count(...)` method returns the number of times the given column name appears in the result set. You may use the `count(*)` syntax to count all of the objects in the result set, or `count(product.name)` to count the number of objects in the result set with a `name` property. Here is an example using the `count(*)` method to count all products:

```
select count(*) from Product product
```

The `distinct` keyword only counts the unique values in the row set—for instance, if there are 100 products, but 10 have the same price as another product in the results, then a `select count(distinct product.price) from Product` product query would return 90. In our database, the following query will return 2, one for each supplier:

```
select count(distinct product.supplier.name) from Product product
```

If we removed the `distinct` keyword, it would return 5, one for each product.

All of these queries return an `Integer` object in the list. You could use the `uniqueResult()` method here to obtain the result.

The aggregate functions available through HQL include the following:

- `avg(`*`property name`*`)`: Average of a property's value
- `count(`*`property name or *`*`)`: Counts the number of times a property has a value
- `max(`*`property name`*`)`: The maximum value of the property values
- `min(`*`property name`*`)`: The minimum value of the property values
- `sum(`*`property name`*`)`: Sum total of the property values

If you have more than one aggregate method, the result set `List` will contain an `Object` array with each of the aggregates you requested. Adding another aggregate to the `select` clause is straightforward:

```
select min(product.price), max(product.price) from Product product
```

You can also combine these with other projection properties in the result set.

Bulk Updates and Deletes with HQL

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The `Query` interface now contains a method called `executeUpdate()` for executing HQL update or delete statements. The `executeUpdate()` method returns an `int` that contains the number of rows affected by the update or delete:

```
public int executeUpdate() throws HibernateException
```

HQL updates look like you would expect them to, being based on SQL update statements. Do not include an alias with the update; instead, put the `set` keyword right after the class name:

```
String hql = "update Supplier set name = :newName ➥
where name = :name";

Query query = session.createQuery(hql);
query.setString("name","SuperCorp");
query.setString("newName","MegaCorp");
int rowCount = query.executeUpdate();
System.out.println("Rows affected: " + rowCount);
```

```
//See the results of the update
query = session.createQuery("from Supplier");
List results = query.list();
```

You may use a `where` clause with updates to control which rows get updated, or you may leave it off to update all rows. Notice that we printed out the number of rows affected by the query. We also used named parameters in our HQL for this bulk update.

Bulk deletes work in a similar way. Use the `delete from` clause with the class name you would like to delete from. Then use the `where` clause to narrow down which entries in the table you would like to delete. Use the `executeUpdate()` method to execute deletes against the database as well. Our code surrounding the delete HQL statement is basically the same—we used named parameters and we print out the number of rows affected by the delete:

```
String hql = "delete from Product where name = :name";
Query query = session.createQuery(hql);
query.setString("name","Mouse");
int rowCount = query.executeUpdate();
System.out.println("Rows affected: " + rowCount);

//See the results of the update
query = session.createQuery("from Product");
List results = query.list();
```

■ **Caution** Using bulk updates and deletes in HQL works almost the same as you would expect in SQL, so keep in mind that these are powerful and can erase the data in your tables if you make a mistake with the `where` clause.

Named Queries for HQL and SQL

One of Hibernate's best features is named queries, where your application can store its HQL queries outside the application in the mapping file. This has many benefits for application maintenance. The first benefit is that many objects can share queries—you could set up static final strings on classes with the HQL queries, but Hibernate already provides a nice facility for the same thing. The next benefit is that named queries could also contain native SQL queries—the application calling the named query does not need to know if the named query is SQL or HQL. This has enormous benefits for migrating SQL-based applications to Hibernate. The last benefit is that you can provide your HQL and SQL queries in a configuration file to your database administrators, who would probably find it easier to work with an XML-mapping file than HQL statements embedded in Java code.

Add named queries in the appropriate Hibernate mapping file. HQL queries use the `<query>` XML element, and SQL queries use the `<sql-query>` XML element. Both of these XML elements require a name attribute that uniquely identifies the query in the application. With one simple HQL named query, and one simple SQL query that does the same exact thing, we have this Hibernate mapping file, `Product.hbm.xml`, as seen in Listing 8-1.

Listing 8-1. *Product.hbm.xml*

```
<?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 package="com.hibernatebook.criteria">
   <class name="Product">
      <id name="id" type="int">
         <generator class="native"/>
      </id>

      <property name="name" type="string"/>
      <property name="description" type="string"/>
      <property name="price" type="double"/>
      <many-to-one name="supplier" class="Supplier" column="supplierId"/>

   </class>

   <query name="com.hibernatebook.criteria.Product.HQLpricing"><![CDATA[
      select product.price from Product product where product.price > 25.0]]>
   </query>
   <sql-query name="com.hibernatebook.criteria.Product.SQLpricing">
     <return-scalar column="price" type="double"/>
     <![CDATA[
      select product.price from Product as product where product.price > 25.0]]>
   </sql-query>
</hibernate-mapping>
```

Notice that we embedded the SQL and HQL queries in `CDATA` regions. This protects our SQL queries from interfering with the XML parser—we don't have to worry about special characters breaking the XML. For the native SQL query, we also had to specify a return type, so Hibernate knows what type of result data to expect from the database. When you use HQL, Hibernate handles that mapping behind the scenes, because it knows which objects went in. With SQL, you have to specify the return types yourself. In this case, we used the `<return-scalar>` XML element to define our return type as a column named `price`, with a type of `double`. Hibernate converts the JDBC result set into an array of objects, just like the previous HQL query. Functionally, they are identical. We discuss native SQL in more detail in the next section of this chapter.

You may also specify the flush mode, whether or not the query is cacheable, the cache region, the fetch size, and the timeout for the HQL and SQL queries. For the SQL query, you may additionally specify whether the SQL query is callable.

Using Native SQL

Although you should probably use HQL wherever possible, Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that is not supported in HQL. Another reason is that you would like to call stored procedures from your Hibernate application. We discuss stored procedures and other database-specific integration solutions in Chapter 13. Rather than just providing an interface to the underlying JDBC connection, like other Java ORM tools, Hibernate provides a way to define the entity (or join) the query uses. This makes integration with the rest of your ORM-oriented application easy.

You can modify your SQL statements to make them work with Hibernate's object-relational mapping layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with `{objectname.*}`, or you can specify the aliases directly with `{objectname.property}`. Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so you select the right properties off of the table.

Underlying Hibernate's native SQL support is the `org.hibernate.SQLQuery` interface, which extends the `org.hibernate.Query` interface we already discussed. Your application will create a native SQL query from the session with the `createSQLQuery()` method on the `Session` interface.

```
public SQLQuery createSQLQuery(String queryString) throws HibernateException
```

After you pass a string containing the SQL query to the `createSQLQuery()` method, you should associate the SQL result with an existing Hibernate entity, a join, or a scalar result. The `SQLQuery` interface has `addEntity()`, `addJoin()`, and `addScalar()` methods. For the entities and joins, you can specify a lock mode, which we discuss in Chapter 9. The `addEntity()` methods take an alias argument and either a class name or an entity name. The `addJoin()` methods take an alias argument and a path to join.

Using native SQL with scalar results is the simplest way to get started with native SQL. Our Java code looks like this:

```
  String sql = "select avg(product.price) as avgPrice from Product product";
  SQLQuery query = session.createSQLQuery(sql);
  query.addScalar("avgPrice",Hibernate.DOUBLE);
  List results = query.list();
```

Hibernate does execute the same SQL we passed through, because we did not need to specify any entity aliases:

```
select avg(product.price) as avgPrice from Product product
```

The SQL is regular SQL (we did not have to do any aliasing here). We created an `SQLQuery` object, and then added a scalar mapping with the built-in `double` type (from the `org.hibernate.Hibernate` class). We needed to map the `avgPrice` SQL alias to the object type. The results are a `List` with one object, a `Double`.

One step more complicated than the previous example is native SQL that returns a result set of objects. In this case, we will need to map an entity to the SQL query. The entity consists of the alias we used for the object in the SQL query and its class. For this example, we used our `Supplier` class:

```
String sql = "select {supplier.*} from Supplier supplier";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity("supplier", Supplier.class);
List results = query.list();
```

Hibernate modifies the SQL, and executes the following command against the database:

```
select Supplier.id as id0_, Supplier.name as name2_0_ from Supplier supplier
```

The special aliases allow Hibernate to map the database columns back to the object properties.

In Chapter 13, we discuss more advanced ways to use native SQL to integrate your Hibernate application with existing databases. We cover the use of stored procedures, and we also discuss how to use persistent object custom loading with native SQL.

Summary

The Hibernate Query Language (HQL) is a powerful object-oriented query language that provides the power of SQL, while taking advantage of Hibernate's object-relational mapping and caching. If you are porting an existing application to Hibernate, you can use Hibernate's native SQL facilities to execute SQL against the database. The SQL functionality is also useful for executing SQL statements specific to a given database that have no equivalents in HQL.

You may turn on SQL logging for Hibernate, and Hibernate will log the generated SQL Hibernate executes against the database. If you add a comment to your HQL query object, Hibernate will display a comment in the log next to the SQL statement—this helps with tracing SQL statements back to HQL in your application.

CHAPTER 9

Using the Session

In earlier chapters, we encountered the `Session` object often enough that it has become apparent that it is the central point of access to Hibernate functionality. We will now look at what it embodies and what that implies about how you should use it.

Sessions

From our examples in the earlier chapters, you will have noticed that a small number of classes dominates our interactions with Hibernate. Of these, the `Session` is the lynchpin.

The `Session` is used to create a new database object, read in objects from the database, update the objects in the database, and delete objects from the database. It allows us to manage the transaction boundaries of database access, and (in a pinch) it allows us to obtain a traditional JDBC connection object so that we can do anything to the database that the Hibernate developers have not already considered (precious little) in their existing design.

If you are familiar with the connected approach, it helps to think of a `Session` as being somewhat like a JDBC connection, and the `SessionFactory`, which provides `Session` objects, as being somewhat like a `ConnectionPool`, which provides `Connection` objects. The `SessionFactory` is an expensive object—needlessly duplicating it will cause problems quickly, and creating a `SessionFactory` is a relatively time-consuming process. Ideally, you should have a single `SessionFactory` for each database your application will access. The `SessionFactory` is threadsafe, so it is not necessary to obtain one for each thread. However, you will create numerous `Session` objects—at least one for each thread using Hibernate as the `Session` is *not* threadsafe—and, often, you will want to create multiple `Session` instances even during the lifetime of a specific thread (see the "Threads" section for concurrency issues).

Caution The analogy between a Hibernate `Session` and a JDBC connection only goes so far. One important difference is that if a Hibernate `Session` throws an exception of any sort, you should discard it and obtain a new one. This prevents data in the session's cache from becoming inconsistent with the database.

We will not try to discuss each of the methods available to you through the `Session` interface, but we will look at some of the method categories that `Session` provides (see Table 9-1). For an exhaustive look at what is available, you should read the API documentation on the Hibernate website, or in the Hibernate 3 download.

Table 9-1. *Hibernate Method Summary*

Methods	Description
Create, Read, Update, and Delete	
`save`	Saves an object to the database.
`saveOrUpdate`	Saves an object to the database, or updates the database if it already exists.
`persist`	Reassociates an object with the session so that updates to it will be persisted.
`get`	Retrieves a specific object from the database by the object's identifier.
`getIdentifier`	Determines the identifier for a specific object associated with the session.
`load`	Loads an object from the database by the object's identifier (you should use the `get` method if you are not certain it is in the database).
`refresh`	Refreshes the state of an object from the database.
`update`	Updates the database with changes to an object.
`delete`	Deletes an object from the database.
`createFilter`	Creates a filter to narrow operations on the database.
`createQuery`	Creates a Hibernate query to be applied to the database.
Transactions and Locking	
`beginTransaction`	Begins a transaction.
`lock`	Gets a database lock for an object (or can be used like persist if `LockMode.NONE` is given).
Managing Resources	
`openSession`	Opens the session object, acquiring necessary resources.
`contains`	Determines if a specific object is associated with the database.
`clear`	Clears the session of all loaded instances, cancels any saves, updates, or deletions that have not been completed. Retains any iterators that are in use.
`evict`	Disassociates an object from the session so that subsequent changes to it will not be persisted.
`flush`	Flushes all pending changes into the database—all saves, updates, and deletions will be carried out; essentially, this synchronizes the session with the database.
`isOpen`	Determines if the session has been closed.
`isDirty`	Determines if the database is synchronized with the database.
The JDBC Connection	
`close`	Closes the session, and, hence, the underlying database connection; releases other resources (such as the cache). This is not essential if you have called `disconnect`, but it is usually a good idea. You must not perform operations on the session after calling `close` as they will generate exceptions.
`disconnect`	Disconnects the underlying database connection, but maintains the cached `Session` objects.
`reconnect`	Reconnects the underlying database connection.
`isConnected`	Determines if the underlying database connection is connected.

Transactions and Locking

Transactions and locking are intimately related—the locking techniques chosen to enforce a transaction can determine both the performance and likelihood of the success of a transaction. The type of transaction selected dictates, to some extent, the type of locking that it must use.

You are not obliged to use transactions if they do not suit your needs. Though, you will need to invoke the `flush()` method on the session to ensure that your changes are persisted to the database if you choose not to use them.

Transactions

A transaction is a unit of work guaranteed to behave as if you had exclusive use of the database. If you wrap your work in a transaction, the behavior of other system users will not affect your data. A transaction can be started, committed to write data to the database, or rolled back to remove all changes from the beginning onward (usually as the result of an error). To achieve this, you obtain a `Transaction` object from the database (beginning the transaction) and manipulate this:

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();

  // Normal session usage here...

  tx.commit();
  tx = null;
} catch (HibernateException e) {
  if (tx != null) tx.rollback();
} finally {
  session.close();
}
```

In the real world, transactions are not truly ACID because you can adjust how compliant your transaction will be; however, in a perfect world, all transactions would be carried out strictly observing these rules with no ill effects. Most databases supported by Hibernate support the ACID requirements of relational theory as closely as possible.

Different database suppliers support and permit you to break the ACID rules to a lesser or greater extent, but the degree of control over the isolation rule is actually mandated by the SQL-92 standard. There are important reasons why you might want to break this rule, so both JDBC and Hibernate also make explicit allowances for it.

THE ACID TESTS

Atomicity: A transaction should be all or nothing. If it fails to complete, the database will be left as if none of the operations had ever been performed—known as a *rollback*.

Consistency: A transaction should be incapable of breaking any rules defined for the database. For example, foreign keys must be obeyed. If for some reason this is impossible, the transaction will be rolled back.

Isolation: The effects of the transaction will be completely invisible to all other transactions until it has completed successfully. This guarantees that the transaction will always see the data in a sensible state. For example, an update to a user's address should only contain a correct address (i.e., it will never have the house name for one location but the zip code for another); without this rule, a transaction could easily see when another transaction had updated the first part but not yet completed.

Durability: The data should be retained intact. If the system fails for any reason, it should always be possible to retrieve the database up to the moment of the failure.

The isolation levels permitted by JDBC and Hibernate are listed in Table 9-2.

Table 9-2. *JDBC Isolation levels*

Level	Name	Permitted
0	None	Anything. The database or driver does not support transactions.
1	Read uncommitted	Dirty, nonrepeatable, and phantom reads.
2	Read committed	Nonrepeatable reads and phantom reads.
4	Repeatable read	Phantom reads.
8	Serializable	The rule must be obeyed absolutely.

A *dirty read* may see the in-progress changes of an uncommitted transaction. As in our example, it could see the wrong zip code for an address.

A *nonrepeatable read* sees different data for the same query. For example, it might determine a specific user's zip code at the beginning of the transaction and again at the end, and get a different answer both times without making any updates.

A *phantom read* sees different numbers of rows for the same query. For example, it might see 100 users in the database at the beginning of the query and 105 at the end without making any updates.

Hibernate treats the isolation as a global setting—you apply the configuration option `hibernate.connection.isolation` in the usual manner, setting it to one of the values permitted in Table 9-2. This is not always ideal. You will sometimes want to treat one particular transaction at a high (usually serializable) level of isolation, while permitting lower degrees of isolation for others. To do so, you will need to obtain the JDBC connection directly, alter the isolation level, begin the transaction, roll back or clean up the transaction as appropriate, and reset the isolation level back to its original value before releasing the connection for general usage. The `createUser()` method (discussed in the "Deadlock" section later in this chapter) demonstrates the additional complexity that the connection-specific transaction isolation involves.

Fortunately, the normal case for a transaction using the global isolation level is much simpler. We provide a more standard implementation of the `createUser()` method for comparison in Listing 9-1.

Listing 9-1. *Using the Global (Default) Isolation Level*

```
public static void createUser(String username)
      throws HibernateException {
   Session session = sessions.openSession();
   Transaction tx = null;
   try {
      // Begin the transaction
      tx = session.beginTransaction();

      // Normal usage of the Session here...
      Publisher p = new Publisher(username);
      Subscriber s = new Subscriber(username);
      session.create(p);
      session.create(s);

      // Commit the transaction
      tx.commit();
      tx = null;
   } catch (HibernateException e) {
      // Handle a failed transaction
      if (tx != null)
         tx.rollback();
   } finally {
      // Close the session
      session.close();
   }
}
```

Locking

A database can conform to these various levels of isolation in a number of ways, and you will need a working knowledge of locking to elicit the desired behavior and performance from your application in all circumstances.

To prevent simultaneous access to data, the database itself will acquire a lock on that data. This may be acquired only for the momentary operation on the data, or it may be retained until the end of the transaction. The former is *optimistic locking*, the latter is *pessimistic locking*.

The Read Uncommitted isolation level always acquires optimistic locks, whereas the serializable isolation level will only acquire pessimistic locks. Some databases offer a feature that allows you to append the `FOR UPDATE` query to a select operation, which requires the database to acquire a pessimistic lock even in the lower isolation levels.

Hibernate provides some support for this feature when it is available, and takes it somewhat further by adding facilities that describe additional degrees of isolation obtainable from Hibernate's own cache.

The `LockMode` object controls this fine-grained isolation (see Table 9-3). It is only applicable to the `Session.get` methods, so it is limited; however, where possible, it is preferable to the direct control of isolation mentioned previously.

Table 9-3. *Lock Modes*

Mode	Description
`NONE`	Only reads from the database if the object is not available from the caches.
`READ`	Reads from the database regardless of the contents of the caches.
`UPGRADE`	If you are using versioning, the version is checked. If the version is not stale, a dialect-specific upgrade lock will be obtained for the data to be accessed (if this is available from your database).
`UPGRADE_NOWAIT`	Behaves like `UPGRADE`, but where support is available from the database and dialect, the method will fail with a locking exception immediately. Without this option, or on databases where it is not supported, the query must wait for a lock to be granted (or for a timeout to occur).

An additional lock mode, `WRITE`, is acquired by Hibernate automatically when it has written to a row within the current transaction. This mode cannot be set explicitly.

Having discussed locking in general, we need to touch on some of the problems that locks can cause.

Deadlocks

Even if you have not encountered deadlock (sometimes given the rather louche name of "deadly embrace") in databases, you probably have encountered the problem in multithreaded Java code. The problem arises from similar origins.

Two threads of execution can get into a situation where each is waiting for the other to release a resource that it needs. The most common way to create this situation in a database is shown in Figure 9-1.

Each thread obtains a lock on its table when the update begins. Each thread proceeds until the table held by the other user is required. Neither thread can release the lock on its own table until the transaction completes—so something has to give.

Fortunately, a Database Management System (DBMS) can detect this situation automatically, at which point the transaction of one or more of the offending processes will be aborted by the database. The resulting deadlock error will be received and handled by Hibernate as a normal `HibernateException`. Now you must roll back your transaction, close the session, and then (optionally) try again.

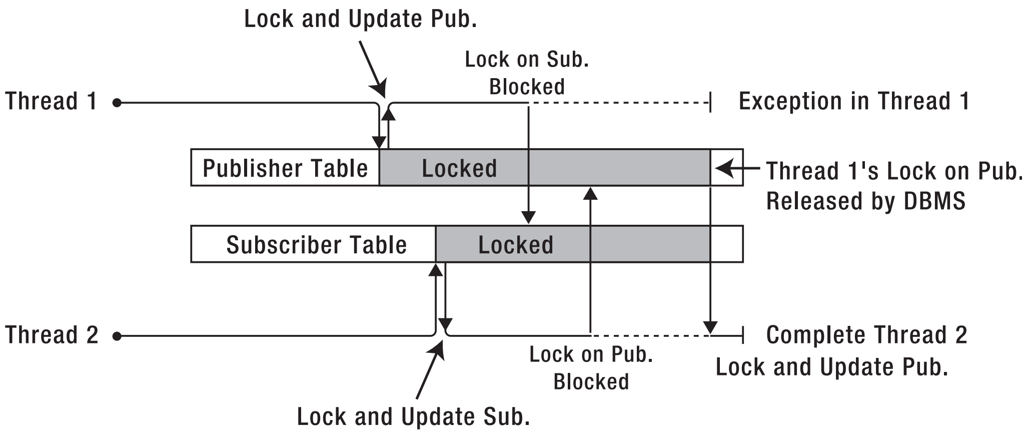

Figure 9-1. *The anatomy of a deadlock*

In Listing 9-2, we demonstrate how four updates on two threads can cause a deadlock. If you look at the output from the threads, you will see that one of them completes while the other fails with a deadlock error. Looking at the database after completion, you will see that the "test" user has been replaced with either "jeff" or "dave" in both tables (you will never see "dave" from one thread and "jeff" from the other). Though it is not necessary here, because we close the session regardless, in a more extensive application it is important to ensure that the session associated with a deadlock or any other Hibernate exception is closed and never used again.

It is worth building and running Listing 9-2 to ensure that you are familiar with the symptoms of a deadlock when they occur.

Listing 9-2. *Code to Generate a Deadlock*

```
package com.hibernatebook.chapter09.deadlock;

import java.sql.Connection;
import java.sql.SQLException;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
```

```
public class GenerateDeadlock {

   public abstract static class Task implements Runnable {

      // Specify the name of the task to carry out
      // and the name of the user to look up and
      // modify
      public Task(String taskName, String username) {
         this.username = username;
         this.taskName = taskName;
      }

      abstract public void step1(Session session);

      abstract public void step2(Session session);

      // Carry out two database steps with a pause between
      // them.
      public void run() {
         Session session = sessions.openSession();
         Transaction tx = null;
         try {
            System.out.println(taskName + " begins a transaction.");
            tx = session.beginTransaction();

            // Step 1
            System.out.println(taskName + " step 1");
            step1(session);

            // Pause to ensure proper ordering
            // of the steps. You would never
            // normally do this!
            pause();

            // Step 2
            System.out.println(taskName + " step 2");
            step2(session);

            tx.commit();
            System.out
                  .println(taskName + " committed its transaction.");
            tx = null;
         } catch (HibernateException e) {
            if (tx != null)
               tx.rollback();
            System.out.println(taskName
                  + " rolled back its transaction: " + e);
         } finally {
            System.out.println("Session for " + taskName + " closed.");
            session.close();
         }
      }

      private void pause() {
         setPaused(true);
         while (isPaused()) {
            synchronized (this) {
               try {
                  System.out.println("Pausing " + taskName);
                  wait();
               } catch (InterruptedException e) {
                  // No need to process this.
               }
            }
         }
         System.out.println(taskName + " awoken, continuing...");
      }

      public boolean isPaused() {
         return isPaused;
      }

      public void setPaused(boolean isPaused) {
         this.isPaused = isPaused;
         if (!isPaused) {
            synchronized (this) {
               notifyAll();
            }
         }
      }

      protected String taskName;

      protected String username;

      private boolean isPaused = false;
   }

   private static SessionFactory sessions = new Configuration()
         .configure().buildSessionFactory();

   // Create a user in the database - illustrates
   // explicit setting of the isolation transaction
   // for a single transaction.
```

```
public static void createUser(String username)
      throws HibernateException {
   Session session = sessions.openSession();
   Connection conn = null;

   int isolation = -1;
   try {
      // Obtain the JDBC connection
      conn = session.connection();

      // Determine the initial isolation level of the transaction
      isolation = conn.getTransactionIsolation();

      // Explicitly set the transaction's isolation level
      // before beginning the transaction.
      conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
      conn.setAutoCommit(false);

      // Normal usage of the Session here...
      Publisher p = new Publisher(username);
      Subscriber s = new Subscriber(username);
      session.create(p);
      session.create(s);

      // Force data to be written to the database
      // BEFORE the transaction is committed.
      session.flush();
      // Commit the transaction
      conn.commit();
   } catch (Exception e1) {
      // Handle a failed transaction
      try {
         if (conn != null) {
            conn.rollback();
         }
      } catch (SQLException e2) {
         System.out.println("Could not rollback the connection: "
               + e2);
      }
   } finally {
      // Restore the original isolation of the committed
      // or rolled back transaction
      if (conn != null) {
         if (isolation != -1) {
            try {
               conn.setTransactionIsolation(isolation);
            } catch (SQLException e2) {
               System.out.println(
                  "Could not restore the connection isolation level: "
                   + e2);
            }
         }
      }

      // Close the session
      session.close();
   }
}

public static void main(String[] argv) {
   String username = "test";

   // First, we'll create the Publisher and
   // Subscriber.
   createUser(username);

   // We create a task (ready to be run) that will
   // update the Publisher, pause, then update the
   // Subscriber.
   Task taskA = new Task("Task A", username) {
      public void step1(Session session) {
         Query query = session
               .createQuery("from Publisher where :foo = username");
         query.setString("foo", username);
         Publisher p = (Publisher) query.uniqueResult();
         p.setUsername("jeff");
      }

      public void step2(Session session) {
         Query query = session
               .createQuery("from Subscriber where :foo = username");
         query.setString("foo", username);
         Subscriber s = (Subscriber) query.uniqueResult();
         s.setUsername("jeff");
      }
   };

   // We create a task (ready to be run) that will
   // update the Subscriber, pause, then update the
   // Publisher (the opposite order to TaskA).
   Task taskB = new Task("Task B", username) {
      public void step1(Session session) {
         Query query = session
               .createQuery("from Subscriber where username = :who");
         query.setString("who", username);
         Subscriber s = (Subscriber) query.uniqueResult();
         s.setUsername("dave");
      }

      public void step2(Session session) {
         Query query = session
               .createQuery("from Publisher where username = :who");
         query.setString("who", username);
         Publisher p = (Publisher) query.uniqueResult();
         p.setUsername("dave");
      }
   };

   // Run the first step of task A to completion
   new Thread(taskA).start();
   while (!taskA.isPaused());
   System.out.println("Task A is complete (paused)");

   // Run the first step of task B to completion
   new Thread(taskB).start();
   while (!taskB.isPaused());
   System.out.println("Task B is complete (paused)");

   // Both tasks have completed their first steps - we'll now
   // release the brakes from both of them, and see what
   // happens.

   taskA.setPaused(false);
   taskB.setPaused(false);
}
}
```

Caching

Accessing a database is an expensive operation, even for a simple query. The request has to be sent (usually over the network) to the server. The database server may have to compile the SQL into a query plan. The query plan has to be run and is limited largely by disk performance. The resulting data has to be shuttled back (again, usually across the network) to the client, and only then can the application program begin to process the results.

Most good databases will cache the results of a query if we run it multiple times, eliminating the disk IO and query compilation times; but this will be of limited value if there are large numbers of clients making substantially different requests. Even if the cache generally holds the results, the time taken to transmit the information across the network is often the larger part of the delay.

Some applications will be able to take advantage of in-process databases, but this is the exception rather than the rule—and such databases have their own limitations.

The natural and obvious answer is to have a cache at the client end of the database connection. This is not a feature provided or supported by JDBC directly, but Hibernate provides one cache (the first-level cache) through which all requests must pass. A second-level cache is optional and configurable.

The level 1 (L1) cache ensures that within a session requests for a given object from a database will always return the same object instance, thus preventing data from conflicting and preventing Hibernate from trying to load an object multiple times.

Items in the L1 cache can be individually discarded by invoking the `evict()` method on the session for the object that you wish to discard. To discard all items in the L1 cache, invoke the `clear()` method.

In this way, Hibernate has a major advantage over the traditional connected approach. With no additional effort from the developer, a Hibernate application gains the benefits of a client-side database cache.

Figure 9-2 shows the two caches available to the session. The compulsory L1 cache, through which all requests must pass, and the optional level 2 (L2) cache. The L1 cache will always be consulted before any attempt is made to locate an object in the L2 cache. You will notice that the L2 cache is external to Hibernate and, though accessed via the session in a way that is transparent to the user of Hibernate, is a pluggable interface to any one of a variety of caches that are maintained on the same JVM as your Hibernate application, or on an external JVM. This allows a cache to be shared between applications on the same machine or even between multiple applications on multiple machines.

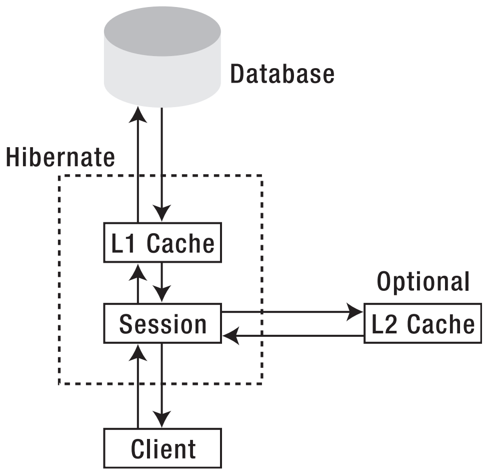

Figure 9-2. *The session's relationship to the caches*

In principle, any third-party cache can be used with Hibernate. An `org.hibernate.cache.CacheProvider` interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation. The cache provider is then specified by giving the implementation class name as the value of the `hibernate.cache.provider_class` property.

In practice, the four production-ready caches, which are already supported, will be adequate for most users (see Table 9-4).

Table 9-4. *Level 2 Cache Implementations Supported "Out of the Box" by Hibernate*

Cache Name	Description
`EHCache`	An in-process cache
`OSCache`	An in-process cache
`SwarmCache`	A multicast distributed cache
`TreeCache`	A multicast distributed transactional cache

The type of access to the L2 cache can be configured on a per-session basis by selecting a `CacheMode` (see Table 9-5) and applying it with the `setCacheMode()` method.

Table 9-5. *CacheMode Options*

Mode	Description
`NORMAL`	Data is read from, and written to, the cache as necessary.
`GET`	Data is never added to the cache (although cache entries are invalidated when updated by this session).
`PUT`	Data is never read from the cache, but cache entries will be updated as they are read from the database by this session.
`REFRESH`	As for `PUT`, but the `use_minimal_puts` Hibernate configuration option will be ignored if it has been set.
`IGNORE`	Data is never read from, or written to, the cache (except that cache entries will still be invalidated when they are updated by this session).

The `CacheMode` setting does not affect the way in which the L1 cache is accessed.

The decision to use an L2 cache is not clear-cut. Although it has the potential to greatly reduce access to the database, the benefits depend on the type of cache and the way in which it will be accessed.

A distributed cache will cause additional network traffic. Some types of database access may result in the contents of the cache being flushed before they are used—in which case it will be adding unnecessary overhead to the transactions.

The L2 cache cannot account for the changes in the underlying data, which are the result of actions by an external program that is not cache-aware. This could potentially lead to problems with stale data, which is not an issue with the L1 cache.

In practice, as with most optimization problems, it is best to carry out performance testing under realistic load conditions. This will let you determine if a cache is necessary and help you select which one will offer the greatest improvement.

Threads

Having considered the caches available to a Hibernate application, you may now be concerned about the risk of a conventional Java deadlock if two threads of execution were to contend for the same object in the Hibernate session cache.

In principle, this is possible—and unlike a database deadlock, Java thread deadlocks do not time out with an error message. Fortunately, there is a very simple solution:

"Doctor, it hurts when I do this"
"Don't do that then"

Do not share the `Session` object between threads. This will avoid all risks of deadlocking on objects contained within the `Session` cache.

The easiest way to ensure that you do not use the same `Session` object outside the current thread is to use an instance local to the current method. If you absolutely must maintain an instance for a longer duration, maintain the instance within a `ThreadLocal` object. For most purposes, however, the lightweight nature of the `Session` object makes it practical to construct, use, and destroy an instance, rather than to store a session.

Summary

In this chapter, we have discussed the nature of the session object and how it can be used to obtain and manage transactions. We have looked at the two levels of caching that are available to applications, and how concurrent threads should manage sessions.

In the next chapter, we present a case study that draws the themes of the earlier chapters together, showing how a Hibernate persistence layer should be constructed.

CHAPTER 10

Design Considerations with Hibernate 3

In this chapter, we will look at a simple case study to show the steps required in designing the persistence layer of a Hibernate-based application. Our application will be an order server, taking orders from a web application, processing them against the product inventory, and tracking the order status.

Most of the code and configuration information is omitted from this chapter for reasons of space and because the specifics are covered in other chapters. You can download the source with the code for the other chapters from the Downloads section of the Apress website (`http://www.apress.com`).

We do not attempt to follow a standard methodology in depth in this example. Instead, we try to illustrate the steps required when designing a Hibernate-based application. In particular, we do not delve too deeply into the intricacies of drawing up the initial requirements.

Application Requirements

As our first step in the design process, we need to specify the requirements incumbent upon our order server. Our design is focused upon the persistence layer of the order server, rather than the entire application, so we will express the requirements in terms of the operations that are offered to client code.

Client code should be able to do the following:

- Add a product to the catalog
- Remove a product from the catalog
- Amend a product in the catalog
- List the products in the catalog
- Set the price of a product in the catalog
- Amend the price of a product in the catalog
- Specify the stock of a product
- Determine the stock of a product

- Place an order for a product
- Expedite an order for a product

Designing the Object Model

For a Hibernate persistence layer, our object model will consist primarily of the POJOs to be persisted into and retrieved from the database, and the DAOs through which this will be achieved.

Designing the POJOs

Taking the requirements listed previously, we scan for nouns to identify likely candidates for our POJO objects. These are

- Client
- Product
- Catalog
- Price
- Stock
- Order

We will discard "Client" since we are assuming that client logic already exists. For our purposes, this will essentially correspond to the test harness logic. We examine our classes in turn and outline their attributes.

The `Product` consists of a Stock Keeping Unit (SKU) and a description.

The `Price` consists of the price and the `Product` with which it is associated.

The `Catalog` consists of the catalog name and a list of `Products` at specific `Prices`. We will separate out the pairs of priced products as `CatalogEntry` classes. A `Catalog`, therefore, consists of a set of `CatalogEntry` objects.

The `CatalogEntry` consists of a `Product` and a `Price` attribute.

The `Stock` consists of the number of units in stock and the `Product` with which it is associated.

The `Order` needs to represent the set of items that a customer has purchased, the price that they were to be charged, and the number of units of each product that they were purchasing. The combination of price, product, and quantity is usually referred to as a *line item* (simply because it represents a single line on an invoice) and so we will refactor this out as a `LineItem` class.

To recap, `LineItems`, therefore, consist of a Price, Product, and quantity attribute.

Our `Order` class now consists of a set of `LineItem` entries—but we would like to distinguish between the orders that are pending and the orders that have been expedited. There are a number of ways to represent this distinction, but we will select the simplest for our example, and add an attribute to the `Order` class.

To all of our classes, we will add an identifier attribute. This will represent the surrogate key to be used in the tables. This is a convenience, not an absolute requirement, but on a new build when all else is equal, it is preferable to include it.

Our refined class list is now

- `Client`
- `Product`
- `Catalog`
- `CatalogEntry`
- `Price`, `Stock`
- `Order`
- `LineItem`

The class relationships in the Order Server model are shown in Figure 10-1.

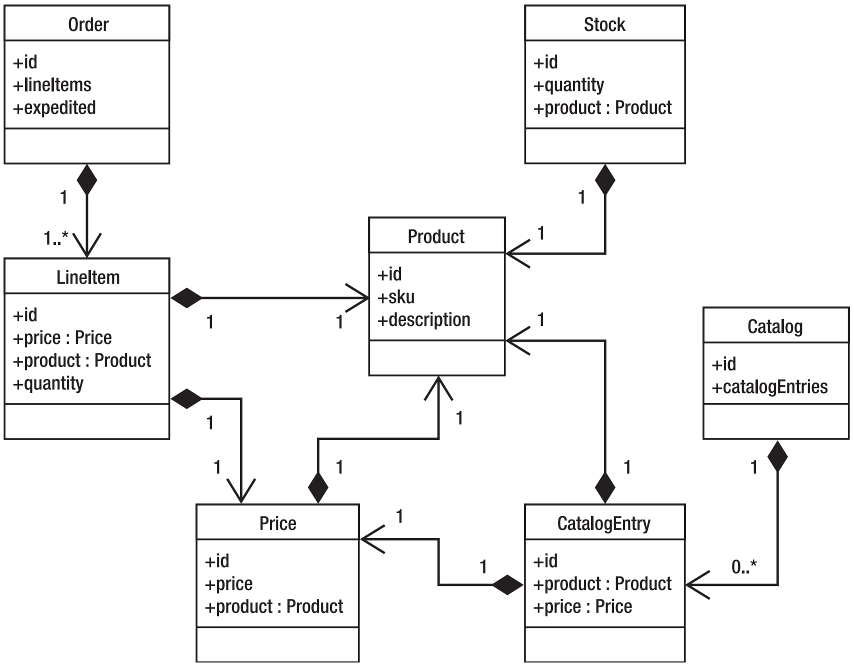

Figure 10-1. *The Order Server model*

Having determined the classes and their attributes, you should create suitable POJOs to represent them. By default, for each of these you should do the following:

- Provide a package, protected, or public default constructor.
- Keep the fields private—provide getter and setter methods for the attributes.

You may want to provide suitable constructors for creating the POJOs from within the client code, and you will usually want to provide additional business logic methods. To determine these business methods, you should walk through the processes described in the requirements specification and determine how each one would be achieved using the classes that you have selected.

Listing 10-1 is a typical example of the sort of POJO class definition that you should end up with. As with most of the examples in this book, we have removed the bulk of the commenting for reasons of brevity, but, in this case, we have retained the XDoclet markup tags (see the sidebar later in this chapter) used to generate the corresponding mapping file.

Listing 10-1. *The POJO Class Representing Our Catalog*

```
package com.hibernatebook.casestudy.pojo;

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

/**
 * @hibernate.class
 * table="catalog"
 */
public class Catalog {
   // The default constructor
   // required by Hibernate
   protected Catalog() {
   }

   // A convenience constructor for
   // client code
   public Catalog(String description) {
      setDescription(description);
   }

   /**
    * @hibernate.id column="id"
    * generator-class="native"
    * type="int"
    */
   public void setId(int id) {
      this.id = id;
   }

   /**
    * @hibernate.set role="catalogEntries"
    * table="catalog_catalogentry"
    * cascade="save-update"
    * @hibernate.collection-key
```

```
     * column="catalog_id"
     * @hibernate.collection-many-to-many
     * class="com.hibernatebook.casestudy.pojo.CatalogEntry"
     * column="catalogentry_id"
     */
    public void setCatalogEntries(Set catalogEntries) {
       this.catalogEntries = catalogEntries;
    }

    /**
     * @hibernate.property column="description"
     * type="string"
     * not-null="true"
     */
    public void setDescription(String description) {
       this.description = description;
    }

    public int getId() {
       return id;
    }

    public Set getCatalogEntries() {
       return catalogEntries;
    }

    public String getDescription() {
       return description;
    }

    private int id;
    private Set catalogEntries = new HashSet();
    private String description;
}
```

Our first requirement on the class list was that we should be able to add a product (one that already exists in the database, or a newly created one) to the catalog. The steps necessary to achieve this will be as follows:

1. Obtain a specific `Catalog` object.
2. Obtain a specific `Product` object.
3. Create a new `CatalogEntry` object.
4. Add the `CatalogEntry` object to the `Catalog`'s set of entries.

When they are not created directly, objects will be obtained from the database, so methods to retrieve objects (or delete them) are immediate candidates for DAO methods. Creation of an object is usually achieved through one of the constructors (though it can be a DAO method).

The catalog entry could be created and populated directly, or we could add a method directly to the `Catalog` class to achieve this. Taking these factors into account, Listing 10-2 is a plausible snapshot of the way we might want to achieve the requirement.

Listing 10-2. *Plausible Code for Adding a Priced Product to a New Catalog*

```
CatalogDAO dao = new CatalogDAO();
Catalog catalog = new Catalog("Spring Catalog");
Product product = new Product("Sunglasses","00000001");
Price price = new Price(product,4,45); // $4.45
dao.addProduct(catalog,product,price);
```

This exercise has forced us to consider the business methods to be added to the POJOs, but it has also forced us to look at candidates for DAO methods, some primitive exception handling, and the specifics of our currency representation (here we simply use dollars and cents, but an internationalized application would need to represent this as a more abstract `Currency` class).

Designing our DAOs

On completion of the exercises required to determine the business methods for the POJOs, you should also find that you have a number of methods that are candidates for inclusion in your DAOs. However, it may not be clear what DAOs are required.

You should attempt to align your DAOs with the operations that you want to perform through them, rather than with the classes and database tables that they operate on. This will make changing these implementation details a less fragile process. Note that there will often be some correspondence between the DAOs selected and the POJOs but to a subset rather than the complete list.

In our study, we would contend that the operations described in the requirements fall into three broad categories:

- *Catalog maintenance*
  - Add a product to the catalog.
  - Remove a product from the catalog.
  - Amend a product in the catalog.
  - List the products in the catalog.
  - Set the price of a product in the catalog.
  - Amend the price of a product in the catalog.
- *Stock management*
  - Specify the stock of a product.
  - Determine the stock of a product.

- *Order management*
    - Place an order for a product.
    - Expedite an order for a product.

Thus, we will create three DAOs:

- `CatalogDAO`
- `StockDAO`
- `OrderDAO`

Of our seven classes selected as POJOs, the DAOs are closely (but not directly) related to three of them. The candidate methods selected should be distributed between them such that the method and its DAO are clearly related; if the relationship seems forced, then you should consider adding a new DAO to accommodate it, or rethink the selection of DAOs. For example, it would probably be inappropriate to add an `assignNewClient` method to the `ProductDAO` in response to a new requirement. This sort of conflict suggests a need for new classes to be added to the design, or for some refactoring of the existing classes.

In Listing 10-3, we show the initial parts of a typical correct DAO implementation that implement the methods for managing operations closely pertaining to the catalog.

Listing 10-3. *The Beginning of Our CatalogDAO Implementation*

```
public class CatalogDAO extends DAO {
   public CatalogDAO()
      throws DAOException
   {
      super();
   }

   public void addProduct(Catalog catalog,
                          Product product,
                          Price price)
      throws DAOException
   {
      Session session = null;
      Transaction tx = null;
      try {
         session = getSession();
         tx = session.beginTransaction();
         CatalogEntry entry =
            new CatalogEntry(product,price);
         catalog.getCatalogEntries().add(entry);
         session.saveOrUpdate(catalog);
         tx.commit();
      } catch( HibernateException e ) {
         rollback(tx);
         throw new DAOException(
            "Could not add product: " + product,e);
      } finally {
         close(session);
      }
   }

   // etc.
```

Mapping with Hibernate

Having created the POJOs to your satisfaction, the next step will be to represent the mappings (see Listing 10-4). We looked at the basics of mapping file creation in Chapter 3 and the format of the mapping file in detail in Chapter 6, so you should refer to those chapters for the details of this step.

Listing 10-4. *A Typical Mapping File Corresponding to Our Catalog Class*

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

<hibernate-mapping
>
  <class
    name="com.hibernatebook.casestudy.pojo.Catalog"
    table="catalog"
  >

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

    <set
      name="catalogEntries"
      table="catalog_catalogentry"
      lazy="false"
      cascade="save-update"
      sort="unsorted">

      <key
        column="catalog_id"/>
      <many-to-many
        class="com.hibernatebook.casestudy.pojo.CatalogEntry"
        column="catalogentry_id"
        outer-join="auto"/>

    </set>

    <property
      name="description"
      type="string"
      update="true"
      insert="true"
      column="description"
      not-null="true"/>

  </class>

</hibernate-mapping>
```

In general, the process is simply one of expressing the relationships between the POJOs and their underlying tables. There are some basic decisions to make, of course—you need to decide how the primary key will be handled and what constraints should apply. There are also some performance design decisions to make—specifically, the selection of indexes for your tables.

XDOCLET

Hibernate 2 had a `CodeGenerator` tool that allowed you to create a skeleton Hibernate mapping file directly from your POJOs. Hibernate 3 no longer provides this tool. Instead, you can use the new Annotations facility (see Chapter 4), but currently this is still at a beta release level, and even with a production release it will oblige you to use Java 5, which may not be an option for many users.

We suggest that as an interim measure you should use XDoclet and its associated markup tags. The XDoclet XML generation tool allows you to use JavaDoc style @ tags in the comments preceding classes and methods to indicate how you would like these to be represented in the mapping files. The tool integrates into your Ant build tool and we think it works well in reducing the burden of this task on the developer. It is worth getting to know XDoclet even if you decide to use other mechanisms for mapping file generation, as the tool can be used for generation of XML files in a number of other circumstances. See the XDoclet home page at `http://xdoclet.sourceforge.net/xdoclet/index.html` for more details.

See the sample code for this chapter for examples of XDoclet style markup.

An alternative to annotations and XDoclet is the `MiddleGen` tool (`http://boss.bekk.no/boss/middlegen/`), which takes the reverse view and generates code from your mapping files. You could also consider the Hibernate tools project (`http://tools.hibernate.org/`) that includes Eclipse IDE plug-ins to generate mappings—but this is currently only at an alpha release level of maturity.

Creating the Database Schema

As we have recommended in earlier chapters, you should generate your schema twice, once by hand, and once using the `SchemaExport` tool (which we discussed in Chapter 3) from your mappings. You should then spend some time reconciling the two designs.

For example, in this chapter we essentially assign a table for each class. In a traditional database schema, we could potentially have placed the stock figure and the price of the items as columns in the product table. However, this would cause substantial complications if the format of the price were to change; for example if we suddenly decided that we needed to accommodate multiple currencies, we would need to arrange for a single product to have a number of different prices. We might also want to assign a single price instance to a number of products that differed only in color. Our selected approach permits all of these changes or differing usages to be applied with minimal disruption to unrelated parts of the schema.

The aims of the hand-designed process are to ensure that you are familiar with your schema and to ensure that you have considered issues such as indexing, uniqueness constraints, not-null constraints, as well as the sensible naming of tables, keys, views, and indexes.

■ **Tip** We recommend that a standard for pluralization of table names be selected. We prefer the singular form—thus the Order and Catalog tables, rather than Orders and Catalogs—but it does not matter much. The point is that all tables should have the same form, which in turn ensures that your programmers will not have to keep cross-referencing the schema design when constructing SQL queries and modifying mapping files.

The aim of the mapping file–generated schema is simply to provide you with a point of comparison against the manually generated file. As you compare the two schemas, you will almost always find disparities between them. When the disparities are in the names of the objects, you should prefer your manually generated names. However, when they are in the structures of the objects (a missing foreign key, a missing table, etc.), this can indicate several scenarios:

- You may have made a mistake in your manual schema design.
- You may have made a mistake in your mapping file specification.
- You may have made a mistake in your class design.

The first scenario is fairly benign (but it helps indicate which parts of the schema may need more documentation for unwary developers). The latter pair, however, must be remedied immediately. In Listing 10-5, we show a generated schema that has been through these iterations and is now correct.

Listing 10-5. *A Typical (Generated) Schema to Represent the Catalog Entity*

```
create table catalog (
   id int4 not null,
   description varchar(255) not null,
   primary key (id)
)

create table catalog_catalogentry (
   catalog_id int4 not null,
   catalogentry_id int4 not null,
   primary key (catalog_id, catalogentry_id)
)

create table catalogentry (
   id int4 not null,
   price int4 not null,
   product int4 not null,
   primary key (id)
)

alter table
   catalog_catalogentry
add
   constraint FK620D8A5F505ECF25
foreign key
   (catalog_id) references catalog

alter table
   catalog_catalogentry
add
   constraint FK620D8A5F5240A1CF
foreign key
   (catalogentry_id) references catalogentry

alter table
   catalogentry
add
   constraint FKBBE8CFF96CD3E95D
foreign key
   (price) references price

alter table
   catalogentry
add
   constraint FKBBE8CFF9882999E9
foreign key
   (product) references product
```

Once your manual and generated schemas agree, you have consonance between your mental model, class model, and mapping of the relationships between database entities and objects, and you are ready to implement the rest of your application.

The Java Application

Your client Java code is now reliant upon your DAOs for its access to the database. There are several ways in which they can be managed, which to some extent depend, in turn, on the context within which you will run them.

You can make them "stand alone." This has the advantage that the client code is entirely independent of the underlying data mechanism. Unfortunately, a problem then arises when you come to define the transaction boundaries of your calls into the DAOs.

If you are running within a context that supports Java Transaction Service (JTS; basically in a J2EE server) and your DAOs acquire their Hibernate transactions through the Java Transaction API via a `JTATransactionFactory` (configurable with the `hibernate.transaction.factory` property), then the DAO transactions will join any extant JTS transaction. So, you can manage the scope of your transaction across the boundaries of the DAOs. You will also need to set the `auto_close_session` and `flush_before_completion` configuration flags to ensure that your data is committed properly.

If your client code will not have the luxury of JTS, then you will have to compromise the design in one of two ways.

The first (and probably the preferred) approach is to maintain the transaction boundaries within the lifespan of the DAO. In our case study, changes to the Catalog, Stock, and Orders are mostly orthogonal to each other; you will not generally have a need to manage a transaction across these boundaries anyway, so you can add explicit transaction management to the three DAOs and allow them to operate independently. Where a problem arises (such as trying to update the stock of a product while an order for it is being expedited), your client will be informed of this via the normal exception mechanism.

Your second option is to make your client code responsible for acquiring the session, or at least managing the transaction associated with the session, when accessing the DAOs. While this does not present any particular technical obstacles, it means that your DAOs will not be as generic as they otherwise could be—your client code will have a dependency upon Hibernate's way of doing things. Often this approach will indicate that there is a design error in your choice of DAOs, but as long as your application does not *need* to be portable to another ORM mechanism (and the likelihood of such a move is sometimes overrated), it can at least be considered.

■ **Caution** Do not forget the preferred scope of the `SessionFactory` and the `Session` objects. `SessionFactory` is once per application per database. Use a suitable singleton class to enforce this. `Session`s are once per thread. Use `ThreadLocal` variables to enforce this, or use the built-in `SessionFactory.getCurrentSession()` method available in Hibernate 3.0.1 and higher.

With your DAOs in place and a policy to allow your client code access to appropriate transactionality, the world is your oyster. Only support and enhancements cloud the horizon; if you have followed our design recommendations and documented your decisions, then your Hibernate-based application will doubtless be a credit to you.

Summary

In this chapter we have shown you the basic steps required to design a persistence layer based on Hibernate. We have looked at the steps involved in selecting the POJO and DAO classes, and we have looked at some of the steps you can take to help verify your database design.

There is no upper limit on how much one can write on the various design methodologies currently in widespread use, but we hope this chapter gives you a good feel for how the use of Hibernate impacts the design process.

In the next chapter, we look at two closely related features: Events that are new with Hibernate 3, and Interceptors that are broadly unchanged since version 2.

CHAPTER 11

Events and Interceptors

Applications often require custom behavior from the persistence layer that is not directly a part of the application. Hibernate 2 provided the notion of Interceptors—essentially callbacks—allowing your custom code to be invoked as Hibernate carried out its persistence logic. Your code could, thus, eavesdrop on (but not interfere with) Hibernate's behavior.

In Hibernate 3, the capabilities have been extended with Events and Listeners so that your custom logic can be invoked by Hibernate instead of its own persistence logic. The default behavior is available to you, so by calling it you can imitate the behavior of interceptors, but you can also change the behavior entirely.

While there is obviously some overlap in the capabilities of these two techniques, they answer broadly different types of problems, so the Events of Hibernate 3 supplement, rather than supplant, the Interceptors of Hibernate 2.

Interceptors

Interceptors are privy to a blow-by-blow account of what is going on as Hibernate carries out its duties. While you can listen in, you can only make limited changes to the way in which Hibernate actually behaves. This is the common requirement; unless you are making substantial changes to the persistence behavior, you will usually want only to track what is going on.

Financial packages often require considerable auditing information to be maintained to prevent fraud and aid accountability. Auditing is a natural candidate for implementation as an interceptor, as it would normally require that no changes be made to the persistence process at all.

The question that usually arises when discussing interceptors is "why not use triggers?" Triggers should never embody application logic, only business logic. If any application is going to have audit-free access to the database, we cannot implement the auditing in triggers. Worse, the triggers may not have access to the user information that's needed. In most multitier situations, the need to pool the database connections precludes establishing individual user logins to the database, so the trigger only knows that the user login "MonolithicApplication" carried out an update of last year's sales figures, not "Jim from accounts," which is who the auditors are likely to be interested in!

Hibernate 3 supports most of the original API with little change, but also extends it by adding additional methods to the Interceptor interface, making much finer distinctions between application states possible. Table 11-1 summarizes the points in the application lifecycle at which the various methods will be invoked, and indicates which are available in each version.

Table 11-1. *The Interceptor Methods Hibernate 2 and 3*

Name	v2	v3	Invoked when . . .	Comments
`afterTransactionBegin`	yes	yes	Immediately after a call to `begin()` on a `Transaction` object retrieved from the `Session`.	This method can change the state of the transaction—for example, it can call `rollback()`.
`instantiate`	yes	yes	When the `Session` needs to create an entity instance.	Because the "empty" object can be created here, this allows Hibernate to use entities, which do not have a default constructor (in legacy applications, for example).
`getEntity`	no	yes	When an entity not in the `Session`'s own cache is requested by its identifier.	
`getEntityName`	no	yes	When the `Session` needs to determine the name of a given `entity`.	
`onLoad`	yes	yes	Immediately before an `entity` object is populated from the database.	The loading can be overridden (by returning `false`), and the instantiated, but uninitialized object, is available if supplementary initialization from the listener is needed.
`findDirty`	yes	yes	During calls to `flush()`.	Allows the saving of changes to attributes to be prevented or forced.
`isTransient`	no	yes	When the `Session` needs to determine if an `entity` it has been asked to persist is transient—for example, during calls to `saveOrUpdate()`.	This replaces the `isUnsaved()` method in version 2.
`isUnsaved`	yes	no	Replaced by `isTransient()`.	
`onSave`	yes	yes	Before an object is saved.	Permits the state of the object to be changed immediately before it is saved.

Name	v2	v3	Invoked when . . .	Comments
`onDelete`	yes	yes	Before an object is saved.	The object's state should not be tampered with at this point.
`preFlush`	yes	yes	Immediately before the `Session` is flushed.	
`onFlushDirty`	yes	yes	During a call to `flush()` after entities have been determined to be dirty.	
`postFlush`	yes	yes	After the `Session` is flushed, if and only if the `Session` had to carry out SQL operations to synchronize state with the database.	
`beforeTransactionCompletion`	no	yes	Immediately prior to the completion of a transaction.	This method can change the state of the transaction—for example, it can call `rollback()`.
`afterTransactionCompletion`	no	yes	Immediately after the completion of a transaction.	

An Example Interceptor

To illustrate how all this works in practice, we will create a simple interceptor from scratch. While the auditing example is a good one, it is rather too involved for our demonstration. Instead, we will consider a concert hall seat-booking system (see Listing 11-1) where the details of bookings will be sent out to customers as they are pushed into the database.

Listing 11-1. *The Booking POJO*

```
package com.hibernatebook.chapter11.events;

public class Booking {
   public Booking(String name, String seat) {
      this.name = name;
      this.seat = seat;
   }

   Booking() {
   }
   protected String getName() {
      return name;
   }

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

   protected String getSeat() {
      return seat;
   }

   protected void setSeat(String seat) {
      this.seat = seat;
   }

   private String seat;
   private String name;
}
```

Interceptors have to override the `org.hibernate.Interceptor` interface. You can set a global interceptor for the configuration (see Listing 11-2), or you can apply interceptors on a per-session basis. You have to install the interceptor programmatically—there is no syntax for specifying this in the Hibernate configuration file.

Listing 11-2. *Installing a Global Interceptor*

```
package com.hibernatebook.chapter11.events;

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

public class MailingExample {
   public static void main(String[] argv) {
      Configuration config = new Configuration();

      // Apply this interceptor at a global level...
      config.setInterceptor(new BookingInterceptor());

      SessionFactory factory = config.configure().buildSessionFactory();
      Session session = factory.openSession();

      // A local interceptor could alternatively
      // be applied here:
      // session.setInterceptor(new BookingInterceptor());
      Transaction tx = session.beginTransaction();

      // Make our bookings...
      session.save(new Booking("dave","F1"));
      session.save(new Booking("jeff","C3"));

      // The confirmation letters should not be sent
      // out until AFTER the commit completes.
      tx.commit();
   }
}
```

The interceptor that we are applying is going to capture the information from the `Booking` objects that we are storing in the database. Listing 11-3 demonstrates the basic mechanism, but it is only a toy example. We will discuss some of its deficiencies in a moment.

Listing 11-3. *An Interceptor Implementation*

```
package com.hibernatebook.chapter11.events;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

import org.hibernate.CallbackException;
import org.hibernate.EntityMode;
import org.hibernate.Interceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class BookingInterceptor implements Interceptor {

   public BookingInterceptor() {
   }

   private ThreadLocal stored = new ThreadLocal();

   public void afterTransactionBegin(Transaction tx) {
      stored.set(new HashSet());
   }

   public void afterTransactionCompletion(Transaction tx) {
      if (tx.wasCommitted()) {
         Iterator i = ((Collection) stored.get()).iterator();
         while (i.hasNext()) {
            Booking b = (Booking) i.next();
            sendMail(b);
         }
      }
      stored.set(null);
   }

   public boolean onSave(Object entity, Serializable id,
         Object[] state, String[] propertyNames, Type[] types)
         throws CallbackException {
      ((Collection) stored.get()).add(entity);
      return false;
   }

   private void sendMail(Booking b) {
      // Here we would actually send out the email
      System.out.print("Name: " + b.getName());
      System.out.println(", Seat: " + b.getSeat());
   }

   public void beforeTransactionCompletion(Transaction tx) {
   }

   public int[] findDirty(Object entity, Serializable id,
         Object[] currentState, Object[] previousState,
         String[] propertyNames, Type[] types) {
      return null;
   }

   public Object getEntity(String entityName, Serializable id)
         throws CallbackException {
      return null;
   }

   public String getEntityName(Object object) throws CallbackException {
      return null;
   }

   public Object instantiate(String entityName, EntityMode entityMode,
         Serializable id) throws CallbackException {
      return null;
   }

   public Boolean isTransient(Object object) {
      return null;
   }
   public void onDelete(Object entity, Serializable id, Object[] state,
         String[] propertyNames, Type[] types) throws CallbackException {
   }

   public boolean onFlushDirty(Object entity, Serializable id,
         Object[] currentState, Object[] previousState,
         String[] propertyNames, Type[] types) throws CallbackException {
      return false;
   }

   public boolean onLoad(Object entity, Serializable id,
         Object[] state, String[] propertyNames, Type[] types)
         throws CallbackException {
      return false;
   }

   public void postFlush(Iterator entities) throws CallbackException {
   }

   public void preFlush(Iterator entities) throws CallbackException {
   }
}
```

Our interceptor makes use of the `afterTransactionBegin()` method to prepare to collect booking details, the `onSave()` method to do so, and the `afterTransactionCompletion()` to report the successful bookings. This sequence guarantees that bookings will not be reported to the users until after we are confident that they have been retained in the database.

A minor deficiency of this implementation is that the email is sent outside the transaction—a system failure immediately after the commit completes could cause the email not to be sent. In our scenario, this is unimportant because email is already an unreliable transport mechanism, but there are other situations, such as the auditing example discussed earlier, where this may be unacceptable. In these cases, interception may not be appropriate and an integrated solution tied into a two-phase–commit transaction may be required.

More importantly, our example assumes that the `Booking` object will not be altered between its addition to the set of emails to be sent and their transmission. This is an extremely dangerous assumption! A safer approach would be to create copies of the `Booking` objects or, better yet, to copy their data into a more appropriate object, as seen in Listing 11-4.

Listing 11-4. *A Better Approach to Preparing the Mailshot*

```
   public boolean onSave(Object entity, Serializable id,
         Object[] state, String[] propertyNames, Type[] types)
         throws CallbackException
   {
      if( entity instanceof Booking ) {
         Booking booking = (Booking)entity.
         Mailshot mailshot = new Mailshot(booking.getName(), booking.getSeat() );
         ((Collection) stored.get()).add(mailshot);
      }
      return false;
   }
```

Finally, we don't necessarily have enough information to prepare our mailshot—the email address may be missing. If the `name` field actually represents the email address, then we are fine, but if it represents a key into other objects, and hence tables in the database, then we have to be careful. It is possible to write database logic from within an interceptor, but the risk of accidentally recursing back into your interceptor logic is high, so we don't recommend it. It's slightly less tricky if you are only using a session-scoped interceptor, but there are probably safer ways to achieve the same end result.

In this example, the methods that we are not using have been given a default implementation. You will note that some of these methods return `null`, or `false`. These methods are permitted to change the data that is preserved or returned by the session. Returning to the `onSave` method, we will consider another possible implementation, as shown in Listing 11-5.

Listing 11-5. *Changing the Data from Within an Interceptor*

```
   public boolean onSave(Object entity, Serializable id,
         Object[] state, String[] propertyNames, Type[] types)
         throws CallbackException
   {
      if( entity instanceof Booking ) {
         state[1] = "unknown";
      }
      return true;
   }
```

Here we are altering the `state` array. This contains the values of each of the objects' fields that are to be stored (in the order defined in the mapping file). In our `Booking` class, field `0` is the `id`, field `1` is the `name`, and field `2` is the `seat`, so here we have changed the name value to be preserved in the database. Returning `true` causes Hibernate to reflect this change when it saves the data. If we left the `return` flag as `false`, nothing would happen when the method was called.

The temptation is to assume that returning `false` guarantees the safety of the data to be preserved, but, in fact, this is not the case. The `state` array represents copies of the data to be preserved, but we have also been given access to the actual object (entity) that contains the original values. If you amend the fields of the entity before returning, the flag will not prevent your changes from being made. Listing 11-6 illustrates how this might occur.

Listing 11-6. *Changing the Data in an Unorthodox Way*

```
public boolean onSave(Object entity, Serializable id,
         Object[] state, String[] propertyNames, Type[] types)
         throws CallbackException
   {
      if( entity instanceof Booking ) {
         Booking booking = (Booking)entity;
         booking.setName("unknown");
      }
      // The flag can't save us from ourselves here!
      return false;
   }
```

Again, this is probably not the best way to make the changes, but it can be useful where you already have a considerable body of logic prepared to process the entity type.

Events

Hibernate 3 actually implements most of its functionality as event listeners. When you register a listener with Hibernate, the listener entirely supplants the default functionality. If you then choose not to do anything with the associated event, the event is lost completely.

If you look at the implementation of the `Session` (see Listing 11-7), you'll see why this is the case. Most of the methods have a form very similar to the following:

Listing 11-7. *The Implementation of a Typical Method in Session*

```
SaveOrUpdateEvent event = new SaveOrUpdateEvent(entityName, obj, this);
listeners.getSaveOrUpdateEventListener().onSaveOrUpdate(event);
```

The `listeners` field is an instance of `SessionEventListenerConfig` which provides the requested `EventListener`, or the default if none is specified. So, if your `EventListener` is provided and doesn't call the default one, nothing else can.

`EventListeners` are always registered globally for the event that they handle. You can register them in the configuration file or programmatically. Either way, you will need to map your implementation of one of the interfaces to the associated types, which you can look up in Table 11-2 (the names are almost-but-not-quite standardized!).

Table 11-2. *The Listener Names and Their Corresponding Interfaces*

Type Name	Listener
`auto-flush`	`AutoFlushEventListener`
`merge`	`MergeEventListener`
`create`	`CreateEventListener`
`delete`	`DeleteEventListener`
`dirty-check`	`DirtyCheckEventListener`
`evict`	`EvictEventListener`
`flush`	`FlushEventListener`
`flush-entity`	`FlushEntityEventListener`
`load`	`LoadEventListener`
`load-collection`	`InitializeCollectionEventListener`

Continued

Table 11-2. *Continued*

Type Name	Listener
lock	LockEventListener
refresh	RefreshEventListener
replicate	ReplicateEventListener
save-update	SaveOrUpdateEventListener
pre-load	PreLoadEventListener
pre-update	PreUpdateEventListener
pre-delete	PreDeleteEventListener
pre-insert	PreInsertEventListener
post-load	PostLoadEventListener
post-update	PostUpdateEventListener
post-delete	PostDeleteEventListener
post-insert	PostInsertEventListener

So, for example, your listener for the `SaveOrUpdateEvent` is mapped to the type name `save-update`, must implement the `SaveOrUpdateEventListener` interface, and would normally have been implemented by the `DefaultSaveOrUpdateEventListener` class. It is wise to follow a similar convention with your own naming, so your mapping file `listener` entry might read like this:

```
<listener type="save-or-update"
         class="com.hibernatebook.chapter11.BookingSaveOrUpdateEventListener"/>
```

Alternatively, a programmatic registration of the same event would be given thus:

```
Configuration config = new Configuration();
config.setListener("save-update", new BookingSaveOrUpdateEventListener());
```

Because they override the default behavior, events are suitable for situations in which you want to fundamentally change the `Session`'s behavior—particularly if you want to prevent a certain event from being processed. Probably the best example of this requirement is in authorizing access to the database, and, in fact, Hibernate provides a set of event listeners for just this purpose. The four events listeners in question override the `PreDelete`, `PreUpdate`, `PreInsert`, and `PreLoad` listeners. The logic in each case in pseudocode runs something like this:

```
if( user does not have permission ) throw RuntimeException
Invoke default listener…
```

Because events are invoked in the same thread as the user's call to the session, the result of an exception in the first step will be an exception (actually a security exception) as the unprivileged user carries out the relevant operation.

To enable policy configuration of security, you would add the following:

```
<listener type="pre-delete"
          class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update"
          class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert"
          class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load"
          class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

An Example Event Listener

Before we get stuck in a simple example, a word of caution: events are very much an exposed part of the inner workings of the Session. While this is ideal for something requiring the level of interference of a security tool, you will not need this for most purposes. Listing 11-8 is more in the nature of an illustrative "hack" than a real solution. In a real application, you would probably solve this particular problem either within the body of the application, by using interceptors, or by using triggers.

Our example assumes a scenario similar to the one we used for our example Interceptor, but now we would like to prevent the booking of certain seats from being persisted to the database from the application.

Listing 11-8. *Programmatically Installing an Event Listener*

```
public class EventExample {
   public static void main(String[] args) {
      Configuration config = new Configuration();

      // Apply this event listener (programmatically)
      config.setListener("save-update", new BookingSaveOrUpdateEventListener());

      SessionFactory factory = config.configure().buildSessionFactory();
      Session session = factory.openSession();

      Transaction tx = session.beginTransaction();

      // Make our bookings... seat R1 is NOT to be saved.
      session.saveOrUpdate(new Booking("charles","R1"));
      session.saveOrUpdate(new Booking("camilla","R2"));

      // The confirmation letters should not be sent
      // out until AFTER the commit completes.
      tx.commit();
   }
}
```

Our example is only going to implement the SaveOrUpdateEventListener interface. You will notice that in Listing 11-8, the original calls to Session.save() have been replaced with calls to Session.saveOrUpdate(). There is a close correspondence between the methods on the Session interface and the event listeners with similar names. A call to Session.save() will not invoke Session.saveOrUpdate() or vice versa. Try using the save() method in the EventExample and you will see that the BookingSaveOrUpdateListener is not invoked. In Listing 11-9, we present the logic of the listener registered in Listing 11-8.

Listing 11-9. *The Implementation of an Event Listener*

```
package com.hibernatebook.chapter11.events;

import java.io.Serializable;

import org.hibernate.HibernateException;
import org.hibernate.event.SaveOrUpdateEvent;
import org.hibernate.event.def.DefaultSaveOrUpdateEventListener;


public class BookingSaveOrUpdateEventListener
   extends DefaultSaveOrUpdateEventListener
{
   public Serializable onSaveOrUpdate(SaveOrUpdateEvent event)
         throws HibernateException {
      if( event.getObject() instanceof Booking ) {
         Booking booking = (Booking)event.getObject();
         System.out.println("Preparing to book seat " + booking.getSeat());

         if( booking.getSeat().equalsIgnoreCase("R1")) {
            System.out.println("Royal box booked");
            System.out.println("Conventional booking not recorded.");

            // By returning null instead of invoking the
            // default behavior, we prevent the invocation
            // of saveOrUpdate on the Session from having
            // any effect on the database!
            return null;
         }
      }

      // The default behavior:
      return super.onSaveOrUpdate(event);
   }
}
```

Summary

In this chapter, we have looked at the powerful ways in which interceptors and events can be used to enhance the Hibernate lifecycle. Hopefully you've gained an appreciation of the advantages and risks that these techniques can confer.

Much as Interceptors and Events are analogous to database triggers, in the next chapter we look at how we can create Filters, which offer a way of constraining results that is similar to views.

CHAPTER 12

Hibernate Filters

Many applications do not need to work with all of the data in a table at once. In these cases, we can create a Hibernate *filter* to work only with a subset of data. Filters provide a way for your application to limit the results of a query to data that passes through the filter. Filters are not a new concept—you could do the same thing with database views, or maintain the logic in your application. Hibernate offers a centralized management system for these filters that will help clean up your application's data access layers over an application-specific solution.

Creating custom views for your tables in the database is another solution, but these tend to be inflexible. Hibernate filters can be enabled or disabled during a Hibernate session. In addition, Hibernate filters can take parameters that become very useful when you build an application on top of Hibernate that uses security or personalization.

Note Hibernate 3 introduces new filter functionality. The `filter()` methods on the `Session` interface from Hibernate 2 are deprecated—the Hibernate 3 functionality has a slightly different purpose.

Where to Use Filters

As an example, let us consider a web application that manages user profiles. Currently, your application presents all users through the same web interface, but you received a requirement from your end user that specifies that active users and expired users should now be managed separately. For this example, the status is stored as a column on the user table. One way to solve this problem is to rewrite every `SELECT` HQL query in your application to add a `WHERE` clause that specifies the user's status. Depending on how you built your application, this could be an easy undertaking or it could be complex, but you still end up modifying code you already tested thoroughly and changing it in many different places. With Hibernate 3, another way to solve the problem would be to create a filter for user status for the application. After your end user selects which set of users to work with (active or expired), your application activates the user status filter (with the proper status) for the end user's Hibernate session. Now any `SELECT` queries will return the correct subset of results, and the relevant code for the user status is limited to two locations: the Hibernate session and the user status filter.

Conceptually, you could use a `WHERE` clause in your queries to accomplish the same thing as Hibernate filters. You could also create a view in your database (if your database supports views). In all three cases, you are limiting the results of your query by imposing one or more conditions on the results. The advantages of Hibernate filters are that you can programmatically turn filters on or off in your application code, and your filters are defined in your Hibernate mapping documents for easy maintainability. The major disadvantage of filters is that you cannot create new filters at runtime. Instead, any filters your application requires need to be specified in the proper Hibernate mapping document. Although this does sound very limiting, filters can be parameterized. For our user status filter example, only one filter would need to be defined in the mapping document (although in two parts). That filter would specify that the status column would need to match a named parameter. You would not need to define the possible values for status in the Hibernate mapping document—the application can specify those parameters at runtime, so the system is fairly flexible.

Although it is certainly possible to write applications with Hibernate that do not use filters, we have found them to be a very good solution to certain problems (notably, security and personalization). Along with interceptors, filters provide a way to implement application-wide logic in one well-defined place.

Defining Filters

The first step is to define filters in your application's Hibernate mapping documents, using the `<filter-def>` XML element. These filter definitions contain the name of the filter and the names and types of any filter parameters. Specify filter parameters with the `<filter-param>` XML element. Filter parameters are similar to named parameters for HQL queries. Both use a `:` in front of the parameter name. Here is an example of a Hibernate mapping document with a filter definition:

```
<?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 ...

  </class>
  <filter-def name="latePaymentFilter">
    <filter-param name="dueDate" type="date"/>
  </filter-def>
</hibernate-mapping>
```

After you create the filter definitions, you need to attach the filters to class or collection mapping elements. You can attach one filter to more than one class or collection. In either case, you add a `<filter>` XML element. The `<filter>` XML element has two attributes: `name` and `condition`. The name refers to a filter that was defined earlier (for instance, `latePaymentFilter`). The condition corresponds to a `WHERE` clause in HQL. Here is an example:

```
  <class ...
    <filter name="latePaymentFilter" condition=":dueDate = paymentDate"/>
  </class>
```

Each `<filter>` XML element needs to correspond to a filter definition that you have already defined. You may have more than one filter for each filter definition, and each class can certainly have more than one filter. This is a little confusing—the extra level of abstraction allows you to define all of the filter parameters in one place and then refer to them in the individual filter conditions.

Using Filters in Your Application

Your application programmatically determines which filters to activate or deactivate for a given Hibernate session. Each session can have a different set of filters with different values for the parameters. By default, all sessions have no active filters—you will need to enable filters programmatically for each session. The `Session` interface contains several methods for working with filters:

- `public Filter enableFilter(String filterName)`
- `public Filter getEnabledFilter(String filterName)`
- `public void disableFilter(String filterName)`

Each of these methods is new to Hibernate 3, as is the `org.hibernate.Filter` interface. The `enableFilter(String filterName)` method activates the specified filter, the `disableFilter(String filterName)` method deactivates the method, and if you have previously activated the named filter, `getEnabledFilter(String filterName)` retrieves the filter.

The `Filter` interface has five methods. You are unlikely to use `validate()`; Hibernate uses that method when it processes the filters. The other four methods are

- `public Filter setParameter(String name, Object value)`
- `public Filter setParameterList(String name, Collection values)`
- `public Filter setParameterList(String name, Object[] values)`
- `public String getName()`

The `setParameter()` method is the most useful. You can substitute any Java object for the parameter, although its type should match the type you specified for the parameter when you defined the filter. The two `setParameterList()` methods are useful for doing `IN` clauses in your filters. If you want to do `BETWEEN` clauses, use two different filter parameters with different names.

After you enable a filter on the session, you do not have to do anything else to your application to take advantage of filters. Your application can run unchanged, except for the filters, as we demonstrate in the following example.

Basic Filtering Example

Because filters are a relatively straightforward concept, a basic example allows us to demonstrate most of the filter functionality, including both activating filters and defining filters in mapping documents.

In the following Hibernate XML mapping document (User.hbm.xml), we created a filter definition called activatedFilter. When you define a filter, you need to use the <filter-def> XML element, and provide a name for the filter. Any parameters for the filter should be specified with <filter-param> XML elements, as shown in Listing 12-1 with the activatedParam XML element. You need to specify a type for the filter parameter so Hibernate knows how to map values to parameters. After you define a filter, you need to attach the filter definition to a class. At the end of our User class definition, we specified a filter named activatedFilter that corresponds to the filter we defined. We also need to set a condition for the attached filter. In our case, we used :activatedParam = activated, where :activatedParam is the name of the parameter we specified on the filter definition, and activated is the column name for the user table. One important piece of information to note is that the parameter should go on the left-hand side so Hibernate's generated SQL doesn't interfere with any joins.

Listing 12-1. *Hibernate XML Mapping for User*

```
<?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.filters.User">
    <id name="id" type="int">
      <generator class="native"/>
    </id>

    <property name="username" type="string" length="32"/>
    <property name="activated" type="boolean"/>
    <filter name="activatedFilter" condition=":activatedParam = activated"/>
  </class>
  <filter-def name="activatedFilter">
    <filter-param name="activatedParam" type="boolean"/>
  </filter-def>
</hibernate-mapping>
```

After we created the filter definition and attached the filter to a class with a condition, we need to activate the filter from our application on a session. The next class, SimpleFilterExample, inserts several user records into the database and then immediately displays them to screen. The class uses a very simple HQL query ("from User") to obtain the result set from Hibernate. The displayUsers() method writes the usernames and activation status out to the console. Before we enable any filters on the database, this method returns all of the users. After we enable the first filter (activatedFilter) to only show activated users, we call the same displayUsers() method, but now the results of the query are the same as if we added a WHERE clause that contained an "activated=true" clause. We can just as easily change the filter's parameter to show nonactivated users, as seen in Listing 12-2.

Listing 12-2. *SimpleFilterExample.java Source Code Listing*

```
package com.hibernatebook.filters;

import java.util.Iterator;

import org.hibernate.Filter;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;


public class SimpleFilterExample
{
    public static void main (String args[])
    {
        Session session = HibernateHelper.getSession();

        //insert the users
        insertUser("ray",true,session);
        insertUser("jason",true,session);
        insertUser("beth",false,session);
        insertUser("judy",false,session);
        insertUser("rob",false,session);

        //Show all users
        System.out.println("===ALL USERS===");
        displayUsers(session);

        //Show activated users
        Filter filter = session.enableFilter("activatedFilter");
        filter.setParameter("activatedParam",new Boolean(true));
        System.out.println("===ACTIVATED USERS===");
        displayUsers(session);

        //Show nonactivated users
        filter.setParameter("activatedParam",new Boolean(false));
        System.out.println("===NON-ACTIVATED USERS===");
        displayUsers(session);

        session.close();
    }

    public static void displayUsers(Session session)
    {
        Transaction trans = session.beginTransaction();
        Query query = session.createQuery("from User");
        Iterator results = query.iterate();
        while (results.hasNext())
        {
            User user = (User) results.next();
            System.out.print(user.getUsername() + " is ");
            if (user.isActivated())
            {
                System.out.println("activated.");
            }
            else
            {
                System.out.println("not activated.");
            }
        }

        trans.commit();

    }

    public static void insertUser(String name, boolean activated, Session session)
    {
        Transaction trans = session.beginTransaction();

        User user = new User();
        user.setUsername(name);
        user.setActivated(activated);
        session.save(user);

        trans.commit();
    }
}
```

The results from running `SimpleFilterExample` are

```
===ALL USERS===
ray is activated.
jason is activated.
beth is not activated.
judy is not activated.
rob is not activated.
===ACTIVATED USERS===
ray is activated.
jason is activated.
===NON-ACTIVATED USERS===
beth is not activated.
judy is not activated.
rob is not activated.
```

Listing 12-3 is the `User` class, which is a POJO. It only contains the `id` for the primary key, the `username`, and a `boolean` variable called `activated`.

Listing 12-3. *User.java Source Code Listing*

```
package com.hibernatebook.filters;

public class User
{
    private int id;
    private String username;
    private boolean activated;


    public boolean isActivated()
    {
        return activated;
    }
    public void setActivated(boolean activated)
    {
        this.activated = activated;
    }
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
    public String getUsername()
    {
        return username;
    }
    public void setUsername(String username)
    {
        this.username = username;
    }
}
```

We are using the same `HibernateHelper` class we used in earlier chapters (see Listing 12-4).

Listing 12-4. *HibernateHelper.java Source Code Listing*

```
package com.hibernatebook.filters;

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

public class HibernateHelper {
   private HibernateHelper() {
   }

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

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

We used the HSQL database for the example. If you are using another database, this example will work as expected, as it does not use any database-specific functionality beyond the Hibernate configuration. The Hibernate configuration file defines the database configuration and connection information, along with the XML mapping document for the `User` class (see Listing 12-5).

Listing 12-5. *Hibernate XML Configuration File for the Example*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class"> ➥
org.hsqldb.jdbcDriver</property>
        <property name="hibernate.connection.url"> ➥
jdbc:hsqldb:file:filterdb</property>
        <property name="hibernate.username">user</property>
        <property name="hibernate.password">user</property>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Mapping files -->
        <mapping resource="com/hibernatebook/filters/User.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

The source code for this chapter includes the schema we used for the HSQL database to create the table for the `filterdb` database.

Summary

Filters are a very useful way to separate some database concerns from the rest of your code. A set of filters can cut back on the complexity of the HQL queries in the rest of your application, at the expense of some runtime flexibility. Instead of using views, which need to be set up at the database level, your applications can take advantage of filters that may be activated or deactivated for individual user sessions.

CHAPTER 13

Fitting Hibernate into the Existing Environment

As we have discussed in earlier chapters, Hibernate represents a fundamentally different approach to database integration from the traditional connected API. The vast body of existing code, therefore, is not written with Hibernate's techniques in mind, and the typical migration to Hibernate will need to support at least a subset of the legacy design.

In this chapter, we will discuss stored procedures, which are not supported by the HSQL database that we used in earlier chapters. Therefore, we will make use of the Firebird database (available from `http://firebird.sourceforge.net/` and the JayBird JDBC driver can be obtained from `http://jaybirdwiki.firebirdsql.org/`) to conduct the remaining examples. As with HQL, Firebird was chosen because it is a freely available Open Source project. Hibernate has support for stored procedures for a number of other commercial and free databases, including Oracle, Microsoft SQL Server, Sybase Adaptive Server, and MySQL.

Limitations of Hibernate

First and foremost, Hibernate wants every entity to be identifiable with a primary key. Ideally, it would like this to be a *surrogate key* (a single column distinct from the fields of the table). Hibernate will accept a primary key that is not a surrogate key. For example, the username column might be used to uniquely identify an entry in the user table. Hibernate will also accept a composite key as its primary key, so that the username and hostname might be used to form the primary key if the username alone did not serve to identify the row.

In the real world, things do not really work like that. Any database that has been around the block a few times is likely to have at least one table for which the primary key has been omitted. For instance, the contents of the table may not have needed to be involved in any relations with other tables. While this is still bad database design, the error is only exposed when Hibernate tries to map objects to data. It may be that adding a suitable surrogate key column is an option—where this is the case, we urge you to do so. In practice, however, the fundamental schema may not be under the developer's control, or other applications may break if the schema is radically changed.

In most scenarios, a developer will be able to arrange the creation of views, or stored procedures. It may be possible to create the appearance of a suitable primary key using these if no other options present themselves, but you should consult with your database administrators, since a table for which no true primary key can be obtained is likely to cause long-term corruption of your data!

Finally, if you cannot change a broken schema, and you cannot add views or stored procedures to ameliorate its effects, you have the option of obtaining a pure JDBC connection (see Listing 13-1) from the session to the database and carrying out traditional connected database access. This is the option of last resort and is only truly of value when you anticipate being able to correct the faulty schema at some future time.

Listing 13-1. *Obtaining a JDBC Connection from Hibernate*

```
SessionFactory sessions =
      new Configuration().configure().buildSessionFactory();
Session session = sessions.openSession();
Connection connection = session.getConnection();
```

If you dislike the mixture of Hibernate and connected code, you could also look at some of the competing connectivity tools. For example, iBatis (downloadable from `http://www.ibatis.com/`) offers an intermediate solution somewhere between the relaxed API of JDBC and the strictures of Hibernate. You could also investigate the Spring framework's support for JDBC, which offers excellent support for stored procedures, aggregate functions, and many other features through a clear, yet terse, syntax. You can obtain Spring from `http://www.springframework.org/`.

Hand-Rolled SQL

Although Hibernate cannot operate upon entities that lack primary keys, it is also awkward to use when there is a poor correspondence between the tables and the classes of your object model.

Using a Direct Mapping

Figure 13-1 presents a fairly typical example of a valid database model that may be painful to represent in our mapping files.

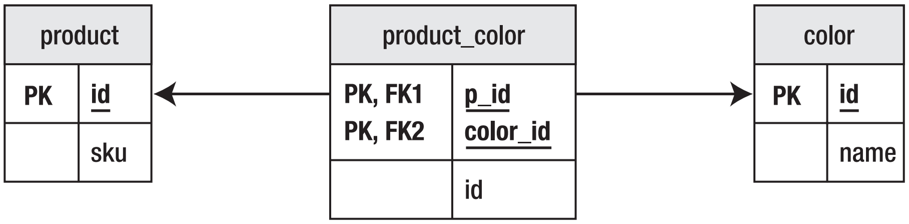

Figure 13-1. *A problematic, but legal, schema*

Here the `product` table represents a product (for example, a flashlight). The `color` table represents the colors in which it is sold. The link table named `product_color` then allows us to identify a product, by Stock Keeping Unit (SKU), and identify the colors in which it is available.

If we do not mind the `Product` object retaining a set of colors (representing the colors in which it can be sold), then we have no problem; but if we want to distinguish between a red flashlight and a green one, things become more difficult (see Listing 13-2).

Listing 13-2. *A Fairly Direct Representation of the Product*

```
<class name="com.hibernatebook.legacy.Product" table="product_color">

   <composite-id
      class="com.hibernatebook.legacy.ProductKey"
      name="key">

      <key-property type="int" name="id" column="product_id"/>
      <key-property type="int" name="colorId" column="color_id"/>

   </composite-id>


   <many-to-one
      name="color"
      class="com.hibernatebook.legacy.Color"
      column="color_id"
      insert="false"
      update="false"/>

   <many-to-one
      name="data"
      class="com.hibernatebook.legacy.ProductData"
      column="product_id"
      insert="false"
      update="false"/>

</class>
```

There are several dissatisfying aspects to the mapping in Listing 13-2. First, rather than map our product table, we have mapped the link table. This makes sense when you consider that the primary key formed from the two columns of this table uniquely identifies a “colored product” that the product table alone cannot do.

Second, we are obliged to create a number of distinct objects to represent the class—the `Product` class itself, a class to represent the primary key (inevitable where a composite id occurs), a class to represent the other attributes of the product, and the `Color` class.

Last, the use of the columns more than once within the mapping requires us to flag them so that they cannot be written—this is a *read-only mapping*.

Using a View

Fortunately, most databases provide a simple mechanism for manipulating a schema so that it better matches the business requirements. A database view will allow you to put together a join that appears to be a table. By a suitable choice of columns from the existing tables, you can construct a view that is much easier to map. Our example in Listing 13-3 is for Firebird; your database may have a slightly different syntax.

Listing 13-3. *A View on Our Product Tables*

```
create view vwProduct (ProductKey,ColorKey,Id,SKU,ColorId)
AS
   select
      p.id as ProductKey,
      c.id as ColorKey,
      p.id as Id,
      p.sku as SKU,
      c.id as ColorId
   from
      product p,
      product_color pc,
      color c
   where
      p.id = pc.product_id
   and
      pc.color_id = c.id;
```

This view effectively reformats our table so that it has a correct (composite) primary key formed from the link table's two columns. It makes the SKU data available directly, and it retains the foreign key into the color table.

Listing 13-4 is a much more natural mapping.

Listing 13-4. *The Revised Mapping*

```
<class name="com.hibernatebook.legacy.Product" table="vwProduct">
  <composite-id
      class="com.hibernatebook.legacy.ProductKey"
      name="key">
      <key-property
         type="int"
         name="id"
         column="ProductKey"/>
      <key-property
         type="int"
         name="colorId"
         column="ColorKey" />
   </composite-id>
   <property
      name="id"
      type="int"
      column="id"
      insert="false"
      update="false"
      unique="true"/>

   <property
      name="SKU"
      type="int"
      column="sku"
      insert="false"/>

   <many-to-one
      name="color"
      class="com.hibernatebook.legacy.Color"
      column="ColorId"/>
</class>
```

The behavior of the composite primary key is unchanged, but the SKU now becomes a simple property. The color entity is mapped as before.

The caveat for this approach is the problem of writing data to the mapping. Some databases (for example versions 4 and lower of MySQL) do not support writable views, and others may have only limited support for them. To avoid views in these circumstances, we must abandon complete portability in favor of database-specific SQL inserted directly into the mapping file.

Putting SQL into a Mapping

Hibernate 3 provides three tags that can be used to override the default behavior when writing to the database. Instead of accepting the SQL generated by Hibernate from the information in the mapping file, you can dictate exactly how changes to an entity should be enforced. The disadvantage is that you will lose Hibernate's guarantee of cross-database platform portability. The advantage is that you can carry out operations that are not explicitly described in the mapping, such as calculating and inserting values in the process of carrying out an insert.

The tags are `<sql-insert>`, `<sql-update>`, and `<sql-delete>`. All three work in the same way.

If you take a look at the Database Definition Language (DDL) script for this chapter, you will see that our client table includes seven fields, last amongst which is the `country` field as shown in Listing 13-5.

Listing 13-5. *The DDL Script to Create Our Client Table*

```
create table client (
   id int not null primary key,
   name varchar(32) not null,
   number varchar(10),
   streetname varchar(128),
   town varchar(32),
   city varchar(32),
   country varchar(32)
);
```

We will, however, ignore the `country` field in our mapping file. We would like this to be automatically set to "UK" whenever a client entity is persisted.

Depending on the database, this could be implemented as part of the view's support for writing operations or as a trigger invoked when the other fields are written, but we use the `<sql-insert>` tag to specify the operation to perform.

The necessary ordering of the parameters can be determined by running Hibernate with logging enabled for the `org.hibernate.persister.entity` level. You must do this before you add the mapping. Listing 13-6 shows a suitably formatted `<sql-insert>` element with the parameters suitably ordered. Note that the identifier field `id` is in the last position, rather than the first as you might have expected.

Listing 13-6. *The Mapping File Using Explicit SQL to Update the Tables*

```
<class name="com.hibernatebook.legacy.Client" table="Client">
   <id type="int" name="id" column="id">
      <generator class="native"/>
   </id>
   <property type="text" name="name" column="name"/>
   <property type="text" name="number" column="number"/>
   <property type="text" name="streetname" column="streetname"/>
   <property type="text" name="town" column="town"/>
   <property type="text" name="city" column="city"/>

   <sql-insert>
   insert into client(name,number,streetname,town,city,id,country)
   values (?,?,?,?,?,?,'UK');
   </sql-insert>
</class>
```

In addition to the three SQL terms for writing to the database, you can specify hand-rolled SQL for reading. This is appended as `<sql-query>` tags outside the `class` tag (see Listing 13-7). They are not intrinsically a part of the mapping. However, you *can* specify that one of them should be used as the default loader for your class.

Listing 13-7. *An Alternative Mapping File Defining a Default Loader*

```
...
<hibernate-mapping>
   <class name="com.hibernatebook.legacy.Client"
          table="Client">
      <id type="int" name="id" column="id">
         <generator class="native"/>
      </id>
      <property name="name"/>
      <property name="number"/>
      <property name="streetname"/>
      <property name="town"/>
      <property name="city"/>

      <loader query-ref="DefaultQuery"/>
   </class>

<sql-query name="DefaultQuery">
   <return alias="c"
           class="com.hibernatebook.legacy.Client"/>
   SELECT
      id as {c.id},
      'NOT SPECIFIED' as {c.name},
      number as {c.number},
      streetname as {c.streetname},
      town as {c.town},
      city as {c.city}
   FROM
      Client
   WHERE
      id = ?
</sql-query>

</hibernate-mapping>
```

Tip Unfortunately, this technique is not quite as sophisticated as you might hope—the custom SQL will not be invoked in general terms. Only if the `id` is explicitly supplied as with a call to `Session.get()` will the default handling be overridden in favor of the loader query.

Invoking Stored Procedures

Data outlives application logic. This is a general rule of thumb and, as the authors can attest, it holds true in practice. The natural lifespan of a database will tend to see multiple applications. The lifespan of some of these applications will, in turn, tend to overlap, so that at any one time we expect substantially different code bases to be accessing the same data.

■ **Tip** Different code bases accessing the same data can cause a fundamental problem: one erroneous application can corrupt the data being used by multiple applications. This is a distinct problem from the issues that the ACID rules described in Chapter 9 can resolve. For example, an application that writes usernames in mixed case may break an application that expects to see only uppercase!

To resolve such issues, databases usually provide their own programming language to allow complex business rules to be expressed and enforced within the boundary of the database itself. These languages are expressed in stored procedures—essentially an API to the database. Often free-form SQL access to such a database is denied, and only stored procedure access is permitted, as barring errors in the stored procedures themselves this removes the risk of corruption.

One final advantage of using stored procedures is that where a substantial calculation is required, the use of a stored procedure can reduce the network traffic involved. For example, invoking a stored procedure to calculate the grand total of a table of accounts, only the request and the result figure would need to traverse the network. The equivalent client-side implementation would need to acquire the value to be totaled from every row!

■ **Tip** Unfortunately, there is no universal standard either for the language of stored procedures themselves or the way that they interface to the outside world. Hibernate supports only the common subset of stored procedures that permit both input and output parameters in the initial call, and that returns one (and only one) result set when invoked. Hibernate also makes some assumptions about how a stored procedure will behave when invoked. For example, a `PostgreSQL` function (its equivalent of stored procedures) will always return a value `void` when called, which prevents these from being invoked for insertions where no return value is expected.

Because stored procedures are such a fundamental part of the way real businesses manage their data, we are optimistic that the Hibernate team will move to support additional types of invocation in the near future—and, in fact, the JDBC specification makes no particular assumptions about the behavior of stored procedures, so it should be possible for them to be supported by the dialect classes, or a similar mechanism, for all databases that provide a stored procedure mechanism.

Taking the Client example from the “Putting SQL into a Mapping” section, we could replace the SQL logic in the `<sql-insert>` tag with a call to a suitable stored procedure. The callable attribute is set to `true` to indicate that Hibernate needs to issue a call to a stored procedure instead of a standard query (see Listing 13-8).

Listing 13-8. *Mapping a Call to the Stored Procedure*

```
<sql-insert callable="true">
   {call insertClient(?,?,?,?,?,?)}
</sql-insert>
```

In the stored procedure definition (see Listing 13-9), you will note that the order of the parameters to be passed in has been tailored to match the order in which they will be provided by Hibernate.

Listing 13-9. *The Logic of the Stored Procedure*

```
SET TERM ^ ;
CREATE PROCEDURE
   insertClient( p_name varchar(32),
                 p_number varchar(10),
                 p_streetname varchar(128),
                 p_town varchar(32),
                 p_city varchar(32),
                 p_id int)
AS
BEGIN
   INSERT INTO client
      (id,name,number,streetname,town,city,country)
   VALUES
      (:p_id,:p_name,:p_number,:p_streetname,:p_town,:p_city,'UK');
END^
SET TERM ; ^
```

By obtaining a JDBC connection from the `Session`, it is of course possible to invoke stored procedures directly; however, you must be aware that such updates cannot be tracked by the Hibernate session cache.

Replacing JDBC Calls in Existing Code

When Hibernate is to be used to replace existing JDBC logic in a legacy application, we recommend that you convert your application to use Data Access Objects (DAOs) as described in brief in Chapter 3.

First, establish your goals—why are you converting a working application to Hibernate?

Second, your next step could be to implement new functionality in Hibernate, being careful with caching to always be pessimistic about what is going on in the database. As an interim step, make sure all of your existing JDBC code uses transactions properly.

After that, follow our directions in this chapter for ensuring that your database has all of its primary keys, so we can start moving JDBC statements to Hibernate Native SQL. Native SQL does perform object mapping. Also, start writing Hibernate mapping documents for your classes and tables—refactoring your Java business objects as necessary. Use views as a transitional tool. Try to avoid the urge to completely rewrite the working parts of the application at this stage, unless you have got the time to do it right!

As you convert the application to native SQL in Hibernate, replace your JDBC statements with calls to Hibernate named queries.

While every database conversion is different, we believe that you now have the tools you need to understand any hurdles that you may encounter.

Summary

In this chapter, we have looked at the way in which Hibernate can integrate with your existing application environment. We have shown some techniques that allow an extant database to be accommodated and we have shown how a legacy application can be integrated with the Hibernate API. In addition, we have considered some of the limitations and constraints that Hibernate's "do it right" philosophy imposes.

This is the penultimate chapter of the book and the last in which we introduce the new features of Hibernate 3. In the final chapter, we will discuss the differences between Hibernate 2 and 3. We will also discuss some of the features that Hibernate 3 offers to support Hibernate 2 users.

CHAPTER 14

Upgrading from Hibernate 2

Hibernate 3 is a major change from the ways of doing things in Hibernate 2. On the whole it is a better product, and we applaud the Hibernate developers for their efforts. One particular group of users will be made nervous by all the changes, and that group is the existing users of Hibernate 2.

Well, there is good news, and there is . . . no bad news! Hibernate 3 has gone the extra mile to allow earlier users to get along. In this chapter, we will take the view from a height of the differences between the two versions, and explain how a Hibernate 2 user can take advantage of them without conducting a major code rewrite.

Hibernate 3 does make changes: The package names have changed; the DTDs have changed; the required libraries are different; and method names and signatures have been altered. Even so, we think you will find that these differences will not cause you much grief when upgrading to the new version.

Once you have read this chapter, we also recommend that you consult the Hibernate 3 Migration Guide in the documentation section of the Hibernate website. The Hibernate team maintains and updates this section to reflect the users' experiences, so you can find hints and tips gathered from users at the cutting edge of just this sort of upgrade.

Package and DTD Changes

The package names for Hibernate 2 and Hibernate 3 have been changed. Hibernate 2 used a base package of `net.sf.hibernate` and Hibernate 3 uses a base package of `org.hibernate`.

This is, of itself, a completely trivial difference—you might imagine that it is purely the result of a migration from Hibernate's hosting moving from SourceForge (`http://sf.net/` or `http://sourceforge.net/`) to their own website (`http://hibernate.org/`), but, in fact, there is another reason for the change.

Because of the package name change, it is possible for an application to use Hibernate 2 and Hibernate 3 simultaneously. If the same package name had been used, then it would be nearly impossible to achieve this.

This is not a coincidence; in addition to the package name, there are now two versions of the DTDs for the XML configuration files. Unchanged Hibernate 2 code should use the usual mapping DTD reference of

```
http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd
```

And for your new Hibernate 3 code, you will use

```
http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd
```

Similarly, for the Hibernate configuration file, your version 2 code will use

```
http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd
```

And version 3 code will use

```
http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd
```

■ **Caution** If you do not update your mapping configuration from the Hibernate 2 form to the Hibernate 3 form, the time it takes to create a configuration and session factory will rise from around 20 seconds to a matter of minutes.

Obviously, it will not be possible for you to have two configuration files with the same (default) name of `hibernate.cfg.xml`, but either version of Hibernate permits you to construct a `Configuration` object and then specify an explicit location for the configuration file using the `configure()` methods (as seen in Listing 14-1).

Listing 14-1. *Using an Explicitly Named Configuration File in Hibernate 3*

```
File configFile = new File("hibernate3.cfg.xml");
Configuration v3Config = new Configuration();
v3Config.configure(configFile);
SessionFactory sessionFactory =
   v3Config.buildSessionFactory();

Session session = sessionFactory.openSession();
// ...
```

In your Hibernate 3 logic, you should be aware that some of the defaults for entries in the mapping file have changed. If you have logic that relies upon implicit settings, you should review your converted mapping files against the version 3 DTD to check that they will behave as expected. The most significant change is that all mappings now default to lazy loading.

Related to this, in Hibernate 2 your POJO classes were not required to provide a public- or protected-access default constructor. Because Hibernate 3 provides lazy loading as the default and because it instruments your POJOs to be their own proxies, you must provide a default (zero argument) constructor with protected or public access. This issue does not manifest itself at compile time, so you should apply this change methodically before trying to run the converted application.

New Features and Support for Old Ones

If you are a Hibernate 2 developer and you have browsed through the earlier chapters, you will have realized that Hibernate 3 offers a lot of new features. You will also have realized that the Hibernate 2 features that you rely on may no longer be supported in version 3. For the most part, though, this is not the case.

Changes and Deprecated Features

If you plan to take advantage of the Hibernate 3 features in any of your existing code, you can, as discussed, simply run the two versions side by side without concern. If you are prepared to make some changes to your existing code, then it is better to take the opportunity to update your existing code to use Hibernate 3 directly. In order to make this second choice a little easier, Hibernate 3 provides a number of "deprecated" APIs that permit fewer changes than a full-blown conversion.

This reduces the immediate impact of the change, and allows you to conduct the rest of the conversion at your leisure, while still allowing you to remove the legacy Hibernate 2 libraries from your application.

■ **Tip** Hibernate exceptions are now thrown as unchecked exceptions. This will not impact existing code, but you may want to revisit APIs that explicitly declare the exception. This change is intended to increase the clarity of API signatures by removing the need for the explicit `throws` clause in code, which uses Hibernate but does not catch its exceptions. There are ongoing debates over the relative merits of the two approaches, but certainly the change *from* checked *to* unchecked will not introduce any incompatibilities (where the reverse would not be true).

Some changes to HQL have occurred between versions 2 and 3. If you have a substantial body of existing HQL syntax, you can elect to retain the old syntax. The selection is made with the `hibernate.query.factoryclass` configuration attribute, which selects the class to load for translating HQL into database queries. The options are listed in Table 14-1.

Table 14-1. *The HQL Processing Classes*

Query Factory Class	HQL Version
`org.hibernate.hql.ast.ASTQueryTranslatorFactory`	3 (default)
`org.hibernate.hql.classic.ClassicQueryTranslatorFactory`	2

It is not possible to switch between the two query translators within a `SessionFactory` instance. Because HQL queries are not parsed until runtime,[1] you will need to run extensive tests to ensure that your modified queries are correct if you decide to convert to the Hibernate 3 syntax.

The object retrieved from the `SessionFactory` in Hibernate 3 implements both the pure Hibernate 3 `org.hibernate.Session` interface and a Hibernate 2 friendly `org.hibernate.classic.Session` interface. By using a classic `Session` reference instead of the standard, you will have access to the methods that have now been deprecated. A fully converted Hibernate 3 application will not need to invoke any of these methods, so you should use a reference to the standard interface unless absolutely compelled by existing logic.

Other deprecated features also reside in the classic package and its subpackages. Notable examples are listed in Table 14-2.

1. Named queries were introduced in Hibernate 3. These are stored in the mapping file and are parsed on application initialization—so while they are still parsed at runtime, you will not need to run extensive tests to spot problems with them.

Table 14-2. *Feature Replacements in Hibernate 3*

Feature	Location in Hibernate 3	Use in Preference
Lifecycle	`org.hibernate.classic`	Interceptor or Event
Validatable	`org.hibernate.classic`	Interceptor or Event

Some of the changes to Hibernate 3 have not justified the creation of a replacement class. Here a few methods will have been removed, replaced, or renamed. In these few cases, if you do not want to run Hibernate 2 and 3 side by side, you will be forced to update your code. When compilation produces errors, consult the JavaDoc API at `http://hibernate.org` to see if the API has changed and to determine the preferred technique. In practice, there are few changes in the core API between versions 2 and 3, and such changes as do exist have well signposted successors in the new API.

Additions

The Event class is new to Hibernate 3. If you are familiar with the Interceptor class (which is retained), you will have some idea of what to expect. The topic is discussed in Chapter 11.

The criteria and filter APIs have been extended considerably. These are discussed in detail in Chapters 7 and 12, respectively.

The flexibility of the mappings has been improved. For example, the `join` element permits a single class to be represented by multiple tables. Support for stored procedures allows better integration with legacy databases. The mapping file format is discussed in Chapter 6 and support for stored procedures and other legacy features are discussed in Chapter 13.

Changes to Tools and Libraries

As you may expect, the libraries upon which Hibernate is based have changed in version 3. Some have been added, others have been brought up to date. Rather than enumerate these here, we refer you to the `lib/README.txt` file in your Hibernate 3 distribution, which explains in detail whether or not individual libraries are required, and what purpose each serves.

Hibernate 2 provided a number of aids to the generation of POJOs, mapping files, and database schema. Hibernate 3 has started a process of migrating to external support for these processes. Where these tools are retained, they can be found in the `org.hibernate.tool` package and its subpackages. For example, the fully qualified name of the `SchemaExport` class is now `org.hibernate.tool.hbm2ddl.SchemaExport`, but the generation of mapping files from POJOs (previously a facility provided by the `CodeGenerator` class, which is not provided with Hibernate 3) would usually be conducted with an Eclipse plug-in, XDoclet, or be replaced by Java 5 annotations.

The Eclipse IDE has grown phenomenally in popularity during the development of Hibernate 3. As a result of this, Hibernate 3 now provides the Hibernate tools project to supply Eclipse plug-ins. This project (`http://tools.hibernate.org/`) is in alpha release at the time of this writing, but we have encountered few problems using it during the writing of this book. It provides aid in generating and reverse-engineering the POJO, mapping, and schema components. It also provides an interactive HQL console, which is an invaluable debugging tool. If you propose to use Eclipse while developing Hibernate, we would recommend that you install these plug-ins.

Changes with Java 5

The latest release of Java introduced some pretty substantial changes to the language. It also introduced some incompatibilities with the previous class file format.

The core `Hibernate` library and its dependencies are still being compiled under the 1.2 series of Java so this should present you with no problems. The only significant addition to Hibernate 3 that relies upon a Java 5 specific feature is the `Annotations` library, which at the time of this writing is still in beta. If you are using Java 5, check the Annotations page (`http://annotations.hibernate.org/`) for the latest status of this library.

The use of annotations in Hibernate 3 is discussed in more depth in Chapter 4.

Summary

In this chapter, we examined some of the changes that have been introduced with Hibernate 3 and seen how code written for Hibernate 2 can be run parallel to Hibernate 3, or be readily adapted to run directly under Hibernate 3.

Index