The ultimate Hibernate reference

SAMPLE
CHAPTER

HIBERNATE IN ACTION

Christian Bauer
Gavin King

MANNING

Hibernate in Action

by Christian Bauer

and

Gavin King

Chapter 2

contents

2 Introducing and integrating Hibernate

This chapter covers

- Hibernate in action with “Hello World”
- The Hibernate core programming interfaces
- Integration with managed and non-managed environments
- Advanced configuration options

It's good to understand the need for object/relational mapping in Java applications, but you're probably eager to see Hibernate in action. We'll start by showing you a simple example that demonstrates some of its power.

As you're probably aware, it's traditional for a programming book to start with a "Hello World" example. In this chapter, we follow that tradition by introducing Hibernate with a relatively simple "Hello World" program. However, simply printing a message to a console window won't be enough to really demonstrate Hibernate. Instead, our program will store newly created objects in the database, update them, and perform queries to retrieve them from the database.

This chapter will form the basis for the subsequent chapters. In addition to the canonical "Hello World" example, we introduce the core Hibernate APIs and explain how to configure Hibernate in various runtime environments, such as J2EE application servers and stand-alone applications.

2.1 "Hello World" with Hibernate

Hibernate applications define *persistent classes* that are "mapped" to database tables. Our "Hello World" example consists of one class and one mapping file. Let's see what a simple persistent class looks like, how the mapping is specified, and some of the things we can do with instances of the persistent class using Hibernate.

The objective of our sample application is to store messages in a database and to retrieve them for display. The application has a simple persistent class, `Message`, which represents these printable messages. Our `Message` class is shown in listing 2.1.

Listing 2.1 `Message.java`: A simple persistent class

```
package hello;
public class Message {
    private Long id;
    private String text;
    private Message nextMessage;
    private Message() {}
    public Message(String text) {
        this.text = text;
    }
    public Long getId() {
       return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getText() {
        return text;
```

Identifier attribute

Message text

Reference to another Message

```
    }
    public void setText(String text) {
        this.text = text;
    }

     public Message getNextMessage() {
         return nextMessage;
     }
     public void setNextMessage(Message nextMessage) {
         this.nextMessage = nextMessage;
     }
}
```

Our `Message` class has three attributes: the identifier attribute, the text of the message, and a reference to another `Message`. The identifier attribute allows the application to access the database identity—the primary key value—of a persistent object. If two instances of `Message` have the same identifier value, they represent the same row in the database. We've chosen `Long` for the type of our identifier attribute, but this isn't a requirement. Hibernate allows virtually anything for the identifier type, as you'll see later.

You may have noticed that all attributes of the `Message` class have JavaBean-style property accessor methods. The class also has a constructor with no parameters. The persistent classes we use in our examples will almost always look something like this.

Instances of the `Message` class may be managed (made persistent) by Hibernate, but they don't *have* to be. Since the `Message` object doesn't implement any Hibernate-specific classes or interfaces, we can use it like any other Java class:

```
Message message = new Message("Hello World");
System.out.println( message.getText() );
```

This code fragment does exactly what we've come to expect from "Hello World" applications: It prints `"Hello World"` to the console. It might look like we're trying to be cute here; in fact, we're demonstrating an important feature that distinguishes Hibernate from some other persistence solutions, such as EJB entity beans. Our persistent class can be used in any execution context at all—no special container is needed. Of course, you came here to see Hibernate itself, so let's save a new `Message` to the database:

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
```

```
tx.commit();
session.close();
```

This code calls the Hibernate `Session` and `Transaction` interfaces. (We'll get to that `getSessionFactory()` call soon.) It results in the execution of something similar to the following SQL:

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (1, 'Hello World', null)
```

Hold on—the `MESSAGE_ID` column is being initialized to a strange value. We didn't set the `id` property of `message` anywhere, so we would expect it to be `null`, right? Actually, the `id` property is special: It's an *identifier property*—it holds a generated unique value. (We'll discuss how the value is generated later.) The value is assigned to the `Message` instance by Hibernate when `save()` is called.

For this example, we assume that the `MESSAGES` table already exists. In chapter 9, we'll show you how to use Hibernate to automatically create the tables your application needs, using just the information in the mapping files. (There's some more SQL you won't need to write by hand!) Of course, we want our "Hello World" program to print the message to the console. Now that we have a message in the database, we're ready to demonstrate this. The next example retrieves all messages from the database, in alphabetical order, and prints them:

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages =
        newSession.find("from Message as m order by m.text asc");
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
    Message message = (Message) iter.next();
    System.out.println( message.getText() );
}
newTransaction.commit();
newSession.close();
```

The literal string `"from Message as m order by m.text asc"` is a Hibernate query, expressed in Hibernate's own object-oriented Hibernate Query Language (HQL). This query is internally translated into the following SQL when `find()` is called:

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
order by m.MESSAGE_TEXT asc
```

The code fragment prints

```
1 message(s) found:
Hello World
```

If you've never used an ORM tool like Hibernate before, you were probably expecting to see the SQL statements somewhere in the code or metadata. They aren't there. All SQL is generated at runtime (actually at startup, for all reusable SQL statements).

To allow this magic to occur, Hibernate needs more information about how the `Message` class should be made persistent. This information is usually provided in an *XML mapping document*. The mapping document defines, among other things, how properties of the `Message` class map to columns of the `MESSAGES` table. Let's look at the mapping document in listing 2.2.

Listing 2.2 A simple Hibernate XML mapping

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">
        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property
            name="text"
            column="MESSAGE_TEXT"/>
        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Note that Hibernate 2.0 and Hibernate 2.1 have the same DTD!

The mapping document tells Hibernate that the `Message` class is to be persisted to the `MESSAGES` table, that the identifier property maps to a column named `MESSAGE_ID`, that the text property maps to a column named `MESSAGE_TEXT`, and that the property named `nextMessage` is an association with *many-to-one multiplicity* that maps to a column named `NEXT_MESSAGE_ID`. (Don't worry about the other details for now.)

As you can see, the XML document isn't difficult to understand. You can easily write and maintain it by hand. In chapter 3, we discuss a way of generating the

XML file from comments embedded in the source code. Whichever method you choose, Hibernate has enough information to completely generate all the SQL statements that would be needed to insert, update, delete, and retrieve instances of the `Message` class. You no longer need to write these SQL statements by hand.

NOTE Many Java developers have complained of the "metadata hell" that accompanies J2EE development. Some have suggested a movement away from XML metadata, back to plain Java code. Although we applaud this suggestion for some problems, ORM represents a case where text-based metadata really is necessary. Hibernate has sensible defaults that minimize typing and a mature document type definition that can be used for auto-completion or validation in editors. You can even automatically generate metadata with various tools.

Now, let's change our first message and, while we're at it, create a new message associated with the first, as shown in listing 2.3.

Listing 2.3 Updating a message

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

// 1 is the generated id of the first message
Message message =
        (Message) session.load( Message.class, new Long(1) );
message.setText("Greetings Earthling");
Message nextMessage = new Message("Take me to your leader (please)");
message.setNextMessage( nextMessage );
tx.commit();
session.close();
```

This code calls three SQL statements inside the same transaction:

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Notice how Hibernate detected the modification to the `text` and `nextMessage` properties of the first message and automatically updated the database. We've taken advantage of a Hibernate feature called *automatic dirty checking:* This feature

saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction. Similarly, you can see that the new message was made persistent when a reference was created from the first message. This feature is called *cascading save:* It saves us the effort of explicitly making the new object persistent by calling `save()`, as long as it's reachable by an already-persistent instance. Also notice that the ordering of the SQL statements isn't the same as the order in which we set property values. Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called *transactional write-behind*.

If we run "Hello World" again, it prints

```
2 message(s) found:
Greetings Earthling
Take me to your leader (please)
```

This is as far as we'll take the "Hello World" application. Now that we finally have some code under our belt, we'll take a step back and present an overview of Hibernate's main APIs.

2.2 Understanding the architecture

The programming interfaces are the first thing you have to learn about Hibernate in order to use it in the persistence layer of your application. A major objective of API design is to keep the interfaces between software components as narrow as possible. In practice, however, ORM APIs aren't especially small. Don't worry, though; you don't have to understand all the Hibernate interfaces at once. Figure 2.1 illustrates the roles of the most important Hibernate interfaces in the business and persistence layers. We show the business layer above the persistence layer, since the business layer acts as a client of the persistence layer in a traditionally layered application. Note that some simple applications might not cleanly separate business logic from persistence logic; that's okay—it merely simplifies the diagram.

The Hibernate interfaces shown in figure 2.1 may be approximately classified as follows:

- Interfaces called by applications to perform basic CRUD and querying operations. These interfaces are the main point of dependency of application business/control logic on Hibernate. They include `Session`, `Transaction`, and `Query`.

- Interfaces called by application infrastructure code to configure Hibernate, most importantly the `Configuration` class.
- *Callback* interfaces that allow the application to react to events occurring inside Hibernate, such as `Interceptor`, `Lifecycle`, and `Validatable`.
- Interfaces that allow extension of Hibernate's powerful mapping functionality, such as `UserType`, `CompositeUserType`, and `IdentifierGenerator`. These interfaces are implemented by application infrastructure code (if necessary).

Hibernate makes use of existing Java APIs, including JDBC), Java Transaction API (JTA, and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

In this section, we don't cover the detailed semantics of Hibernate API methods, just the role of each of the primary interfaces. You can find most of these interfaces in the package `net.sf.hibernate`. Let's take a brief look at each interface in turn.

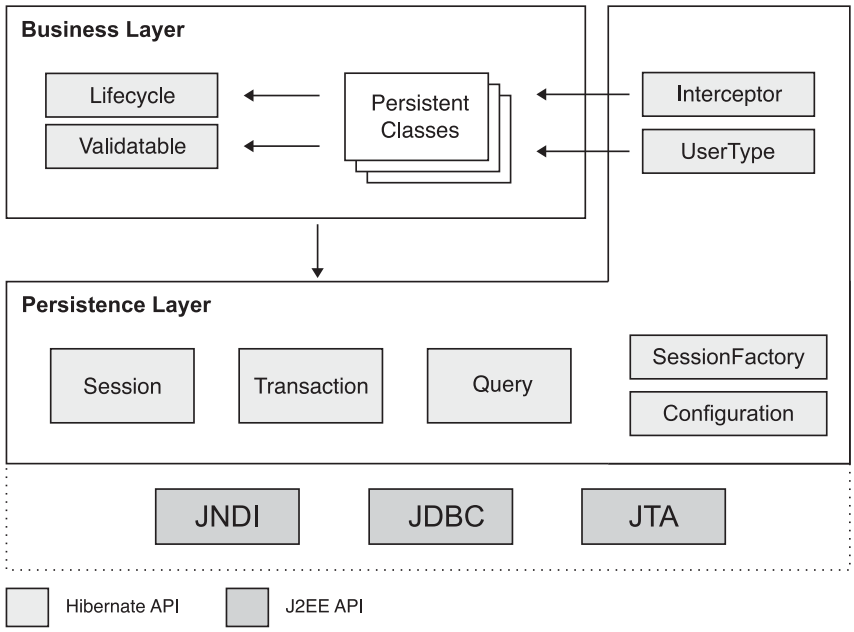

Figure 2.1 High-level overview of the HIbernate API in a layered architecture

2.2.1 The core interfaces

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

Session interface

The `Session` interface is the primary interface used by Hibernate applications. An instance of `Session` is lightweight and is inexpensive to create and destroy. This is important because your application will need to create and destroy sessions all the time, perhaps on every request. Hibernate sessions are *not* threadsafe and should by design be used by only one thread at a time.

The Hibernate notion of a *session* is something between *connection* and *transaction*. It may be easier to think of a session as a cache or collection of loaded objects relating to a single unit of work. Hibernate can detect changes to the objects in this unit of work. We sometimes call the `Session` a *persistence manager* because it's also the interface for persistence-related operations such as storing and retrieving objects. Note that a Hibernate session has nothing to do with the web-tier `HttpSession`. When we use the word *session* in this book, we mean the Hibernate session. We sometimes use *user session* to refer to the `HttpSession` object.

We describe the `Session` interface in detail in chapter 4, section 4.2, "The persistence manager."

SessionFactory interface

The application obtains `Session` instances from a `SessionFactory`. Compared to the `Session` interface, this object is much less exciting.

The `SessionFactory` is certainly not lightweight! It's intended to be shared among many application threads. There is typically a single `SessionFactory` for the whole application—created during application initialization, for example. However, if your application accesses multiple databases using Hibernate, you'll need a `SessionFactory` for each database.

The `SessionFactory` caches generated SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work (only if class and collection mappings specify that this *second-level cache* is desirable).

Configuration interface

The `Configuration` object is used to configure and bootstrap Hibernate. The application uses a `Configuration` instance to specify the location of mapping documents and Hibernate-specific properties and then create the `SessionFactory`.

Even though the `Configuration` interface plays a relatively small part in the total scope of a Hibernate application, it's the first object you'll meet when you begin using Hibernate. Section 2.3 covers the problem of configuring Hibernate in some detail.

Transaction interface

The `Transaction` interface is an optional API. Hibernate applications may choose not to use this interface, instead managing transactions in their own infrastructure code. A `Transaction` abstracts application code from the underlying transaction implementation—which might be a JDBC transaction, a JTA `UserTransaction`, or even a Common Object Request Broker Architecture (CORBA) transaction—allowing the application to control transaction boundaries via a consistent API. This helps to keep Hibernate applications portable between different kinds of execution environments and containers.

We use the Hibernate `Transaction` API throughout this book. Transactions and the `Transaction` interface are explained in chapter 5.

Query and Criteria interfaces

The `Query` interface allows you to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database. A `Query` instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

The `Criteria` interface is very similar; it allows you to create and execute object-oriented criteria queries.

To help make application code less verbose, Hibernate provides some shortcut methods on the `Session` interface that let you invoke a query in one line of code. We won't use these shortcuts in the book; instead, we'll always use the `Query` interface.

A `Query` instance is lightweight and can't be used outside the `Session` that created it. We describe the features of the `Query` interface in chapter 7.

2.2.2 Callback interfaces

Callback interfaces allow the application to receive a notification when something interesting happens to an object—for example, when an object is loaded, saved, or deleted. Hibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality, such as creating audit records.

The `Lifecycle` and `Validatable` interfaces allow a persistent object to react to events relating to its own *persistence lifecycle*. The persistence lifecycle is encompassed by an object's CRUD operations. The Hibernate team was heavily influenced by other ORM solutions that have similar callback interfaces. Later, they realized that having the persistent classes implement Hibernate-specific interfaces probably isn't a good idea, because doing so pollutes our persistent classes with nonportable code. Since these approaches are no longer favored, we don't discuss them in this book.

The `Interceptor` interface was introduced to allow the application to process callbacks without forcing the persistent classes to implement Hibernate-specific APIs. Implementations of the `Interceptor` interface are passed to the persistent instances as parameters. We'll discuss an example in chapter 8.

2.2.3 Types

A fundamental and very powerful element of the architecture is Hibernate's notion of a `Type`. A Hibernate `Type` object maps a Java type to a database column type (actually, the type may span multiple columns). All persistent properties of persistent classes, including associations, have a corresponding Hibernate type. This design makes Hibernate extremely flexible and extensible.

There is a rich range of built-in types, covering all Java primitives and many JDK classes, including types for `java.util.Currency`, `java.util.Calendar`, `byte[]`, and `java.io.Serializable`.

Even better, Hibernate supports user-defined *custom types*. The interfaces `UserType` and `CompositeUserType` are provided to allow you to add your own types. You can use this feature to allow commonly used application classes such as `Address`, `Name`, or `MonetaryAmount` to be handled conveniently and elegantly. Custom types are considered a central feature of Hibernate, and you're encouraged to put them to new and creative uses!

We explain Hibernate types and user-defined types in chapter 6, section 6.1, "Understanding the Hibernate type system."

2.2.4 Extension interfaces

Much of the functionality that Hibernate provides is configurable, allowing you to choose between certain built-in strategies. When the built-in strategies are insufficient, Hibernate will usually let you plug in your own custom implementation by implementing an interface. Extension points include:

- Primary key generation (`IdentifierGenerator` interface)
- SQL dialect support (`Dialect` abstract class)
- Caching strategies (`Cache` and `CacheProvider` interfaces)
- JDBC connection management (`ConnectionProvider` interface)
- Transaction management (`TransactionFactory`, `Transaction`, and `TransactionManagerLookup` interfaces)
- ORM strategies (`ClassPersister` interface hierarchy)
- Property access strategies (`PropertyAccessor` interface)
- Proxy creation (`ProxyFactory` interface)

Hibernate ships with at least one implementation of each of the listed interfaces, so you don't usually need to start from scratch if you wish to extend the built-in functionality. The source code is available for you to use as an example for your own implementation.

By now you can see that before we can start writing any code that uses Hibernate, we must answer this question: How do we get a `Session` to work with?

2.3 Basic configuration

We've looked at an example application and examined Hibernate's core interfaces. To use Hibernate in an application, you need to know how to configure it. Hibernate can be configured to run in almost any Java application and development environment. Generally, Hibernate is used in two- and three-tiered client/server applications, with Hibernate deployed only on the server. The client application is usually a web browser, but Swing and SWT client applications aren't uncommon. Although we concentrate on multitiered web applications in this book, our explanations apply equally to other architectures, such as command-line applications. It's important to understand the difference in configuring Hibernate for managed and non-managed environments:

- *Managed environment*—Pools resources such as database connections and allows transaction boundaries and security to be specified declaratively (that

is, in metadata). A J2EE application server such as JBoss, BEA WebLogic, or IBM WebSphere implements the standard (J2EE-specific) managed environment for Java.

- *Non-managed environment*—Provides basic concurrency management via thread pooling. A servlet container like Jetty or Tomcat provides a non-managed server environment for Java web applications. A stand-alone desktop or command-line application is also considered non-managed. Non-managed environments don't provide automatic transaction or resource management or security infrastructure. The application itself manages database connections and demarcates transaction boundaries.

Hibernate attempts to abstract the environment in which it's deployed. In the case of a non-managed environment, Hibernate handles transactions and JDBC connections (or delegates to application code that handles these concerns). In managed environments, Hibernate integrates with container-managed transactions and datasources. Hibernate can be configured for deployment in both environments.

In both managed and non-managed environments, the first thing you must do is start Hibernate. In practice, doing so is very easy: You have to create a `SessionFactory` from a `Configuration`.

2.3.1 Creating a SessionFactory

In order to create a `SessionFactory`, you first create a single instance of `Configuration` during application initialization and use it to set the location of the mapping files. Once configured, the `Configuration` instance is used to create the `SessionFactory`. After the `SessionFactory` is created, you can discard the `Configuration` class.

The following code starts Hibernate:

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

The location of the mapping file, `Message.hbm.xml`, is relative to the root of the application classpath. For example, if the classpath is the current directory, the `Message.hbm.xml` file must be in the `hello` directory. XML mapping files *must* be placed in the classpath. In this example, we also use the system properties of the virtual machine to set all other configuration options (which might have been set before by application code or as startup options).

METHOD CHAINING

Method chaining is a programming style supported by many Hibernate interfaces. This style is more popular in Smalltalk than in Java and is considered by some people to be less readable and more difficult to debug than the more accepted Java style. However, it's very convenient in most cases.

Most Java developers declare setter or adder methods to be of type `void`, meaning they return no value. In Smalltalk, which has no `void` type, setter or adder methods usually return the receiving object. This would allow us to rewrite the previous code example as follows:

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

Notice that we didn't need to declare a local variable for the `Configuration`. We use this style in some code examples; but if you don't like it, you don't need to use it yourself. If you *do* use this coding style, it's better to write each method invocation on a different line. Otherwise, it might be difficult to step through the code in your debugger.

By convention, Hibernate XML mapping files are named with the `.hbm.xml` extension. Another convention is to have one mapping file per class, rather than have all your mappings listed in one file (which is possible but considered bad style). Our "Hello World" example had only one persistent class, but let's assume we have multiple persistent classes, with an XML mapping file for each. Where should we put these mapping files?

The Hibernate documentation recommends that the mapping file for each persistent class be placed in the same directory as that class. For instance, the mapping file for the `Message` class would be placed in the `hello` directory in a file named `Message.hbm.xml`. If we had another persistent class, it would be defined in its own mapping file. We suggest that you follow this practice. The monolithic metadata files encouraged by some frameworks, such as the `struts-config.xml` found in Struts, are a major contributor to "metadata hell." You load multiple mapping files by calling `addResource()` as often as you have to. Alternatively, if you follow the convention just described, you can use the method `addClass()`, passing a persistent class as the parameter:

```
SessionFactory sessions = new Configuration()
    .addClass(org.hibernate.auction.model.Item.class)
    .addClass(org.hibernate.auction.model.Category.class)
    .addClass(org.hibernate.auction.model.Bid.class)
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

The `addClass()` method assumes that the name of the mapping file ends with the `.hbm.xml` extension and is deployed along with the mapped class file.

We've demonstrated the creation of a single `SessionFactory`, which is all that most applications need. If another `SessionFactory` is needed—if there are multiple databases, for example—you repeat the process. Each `SessionFactory` is then available for one database and ready to produce `Session`s to work with that particular database and a set of class mappings.

Of course, there is more to configuring Hibernate than just pointing to mapping documents. You also need to specify how database connections are to be obtained, along with various other settings that affect the behavior of Hibernate at runtime. The multitude of configuration properties may appear overwhelming (a complete list appears in the Hibernate documentation), but don't worry; most define reasonable default values, and only a handful are commonly required.

To specify configuration options, you may use any of the following techniques:

- Pass an instance of `java.util.Properties` to `Configuration.setProperties()`.
- Set system properties using `java -Dproperty=value`.
- Place a file called `hibernate.properties` in the classpath.
- Include `<property>` elements in `hibernate.cfg.xml` in the classpath.

The first and second options are rarely used except for quick testing and prototypes, but most applications need a fixed configuration file. Both the `hibernate.properties` and the `hibernate.cfg.xml` files provide the same function: to configure Hibernate. Which file you choose to use depends on your syntax preference. It's even possible to mix both options and have different settings for development and deployment, as you'll see later in this chapter.

A rarely used alternative option is to allow the application to provide a JDBC `Connection` when it opens a Hibernate `Session` from the `SessionFactory` (for example, by calling `sessions.openSession(myConnection)`). Using this option means that you don't have to specify any database connection properties. We don't recommend this approach for new applications that can be configured to use the environment's database connection infrastructure (for example, a JDBC connection pool or an application server datasource).

Of all the configuration options, database connection settings are the most important. They differ in managed and non-managed environments, so we deal with the two cases separately. Let's start with non-managed.

2.3.2 Configuration in non-managed environments

In a non-managed environment, such as a servlet container, the application is responsible for obtaining JDBC connections. Hibernate is part of the application, so it's responsible for getting these connections. You tell Hibernate how to get (or create new) JDBC connections. Generally, it isn't advisable to create a connection each time you want to interact with the database. Instead, Java applications should use a pool of JDBC connections. There are three reasons for using a pool:

- Acquiring a new connection is expensive.
- Maintaining many idle connections is expensive.
- Creating prepared statements is also expensive for some drivers.

Figure 2.2 shows the role of a JDBC connection pool in a web application runtime environment. Since this non-managed environment doesn't implement connection pooling, the application must implement its own pooling algorithm or rely upon a third-party library such as the open source *C3P0* connection pool. Without Hibernate, the application code usually calls the connection pool to obtain JDBC connections and execute SQL statements.

With Hibernate, the picture changes: It acts as a client of the JDBC connection pool, as shown in figure 2.3. The application code uses the Hibernate `Session` and `Query` APIs for persistence operations and only has to manage database transactions, ideally using the Hibernate `Transaction` API.

Using a connection pool

Hibernate defines a plugin architecture that allows integration with any connection pool. However, support for C3P0 is built in, so we'll use that. Hibernate will set up the configuration pool for you with the given properties. An example of a `hibernate.properties` file using C3P0 is shown in listing 2.4.

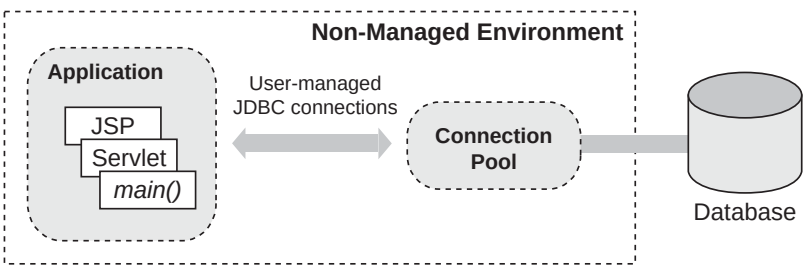

Figure 2.2 JDBC `connection` pooling in a non-managed environment

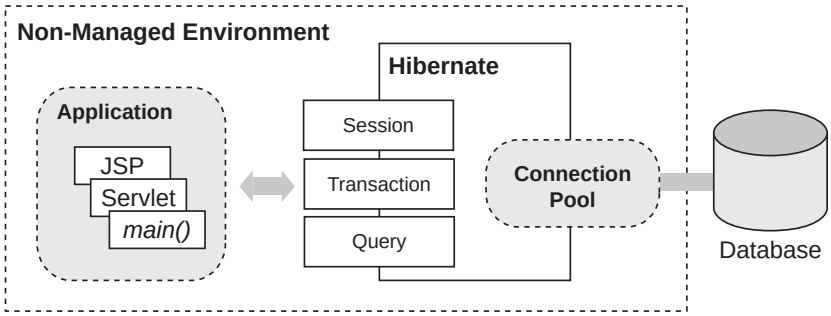

Figure 2.3 Hibernate with a connection pool in a non-managed environment

Listing 2.4 Using `hibernate.properties` for C3P0 connection pool settings

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
```

This code's lines specify the following information, beginning with the first line:

- The name of the Java class implementing the JDBC `Driver` (the driver JAR file must be placed in the application's classpath).
- A JDBC URL that specifies the host and database name for JDBC connections.
- The database user name.
- The database password for the specified user.
- A `Dialect` for the database. Despite the ANSI standardization effort, SQL is implemented differently by various databases vendors. So, you must specify a `Dialect`. Hibernate includes built-in support for all popular SQL databases, and new dialects may be defined easily.
- The minimum number of JDBC connections that C3P0 will keep ready.

- The maximum number of connections in the pool. An exception will be thrown at runtime if this number is exhausted.
- The timeout period (in this case, 5 minutes or 300 seconds) after which an idle connection will be removed from the pool.
- The maximum number of prepared statements that will be cached. Caching of prepared statements is essential for best performance with Hibernate.
- The idle time in seconds before a connection is automatically validated.

Specifying properties of the form `hibernate.c3p0.*` selects C3P0 as Hibernate's connection pool (you don't need any other switch to enable C3P0 support). C3P0 has even more features than we've shown in the previous example, so we refer you to the Hibernate API documentation. The Javadoc for the class `net.sf.hibernate.cfg.Environment` documents every Hibernate configuration property, including all C3P0-related settings and settings for other third-party connection pools directly supported by Hibernate.

The other supported connection pools are Apache DBCP and Proxool. You should try each pool in your own environment before deciding between them. The Hibernate community tends to prefer C3P0 and Proxool.

Hibernate also ships with a default connection pooling mechanism. This connection pool is only suitable for testing and experimenting with Hibernate: You should *not* use this built-in pool in production systems. It isn't designed to scale to an environment with many concurrent requests, and it lacks the fault tolerance features found in specialized connection pools.

Starting Hibernate

How do you start Hibernate with these properties? You declared the properties in a file named `hibernate.properties`, so you need only place this file in the application classpath. It will be automatically detected and read when Hibernate is first initialized when you create a `Configuration` object.

Let's summarize the configuration steps you've learned so far (this is a good time to download and install Hibernate, if you'd like to continue in a non-managed environment):

1. Download and unpack the JDBC driver for your database, which is usually available from the database vendor web site. Place the JAR files in the application classpath; do the same with `hibernate2.jar`.
2. Add Hibernate's dependencies to the classpath; they're distributed along with Hibernate in the `lib/` directory. See also the text file `lib/README.txt` for a list of required and optional libraries.

3 Choose a JDBC connection pool supported by Hibernate and configure it with a properties file. Don't forget to specify the SQL dialect.

4 Let the `Configuration` know about these properties by placing them in a `hibernate.properties` file in the classpath.

5 Create an instance of `Configuration` in your application and load the XML mapping files using either `addResource()` or `addClass()`. Build a `SessionFactory` from the `Configuration` by calling `buildSessionFactory()`.

Unfortunately, you don't have any mapping files yet. If you like, you can run the "Hello World" example or skip the rest of this chapter and start learning about persistent classes and mappings in chapter 3. Or, if you want to know more about using Hibernate in a managed environment, read on.

2.3.3 Configuration in managed environments

A managed environment handles certain cross-cutting concerns, such as application security (authorization and authentication), connection pooling, and transaction management. J2EE application servers are typical managed environments. Although application servers are generally designed to support EJBs, you can still take advantage of the other managed services provided, even if you don't use EJB entity beans.

Hibernate is often used with session or message-driven EJBs, as shown in figure 2.4. EJBs call the same Hibernate APIs as servlets, JSPs, or stand-alone applications: `Session`, `Transaction`, and `Query`. The Hibernate-related code is fully portable between non-managed and managed environments. Hibernate handles the different connection and transaction strategies transparently.

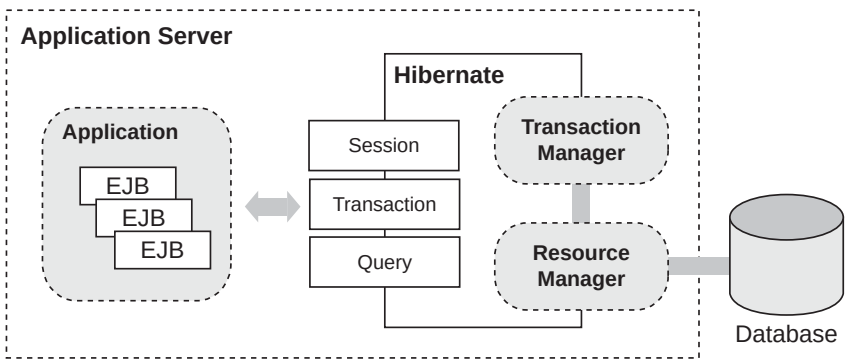

Figure 2.4 Hibernate in a managed environment with an application server

An application server exposes a connection pool as a JNDI-bound *datasource*, an instance of `javax.jdbc.Datasource`. You need to tell Hibernate where to find the datasource in JNDI, by supplying a fully qualified JNDI name. An example Hibernate configuration file for this scenario is shown in listing 2.5.

Listing 2.5 Sample `hibernate.properties` for a container-provided datasource

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

This file first gives the JNDI name of the datasource. The datasource must be configured in the J2EE enterprise application deployment descriptor; this is a vendor-specific setting. Next, you enable Hibernate integration with JTA. Now Hibernate needs to locate the application server's `TransactionManager` in order to integrate fully with the container transactions. No standard approach is defined by the J2EE specification, but Hibernate includes support for all popular application servers. Finally, of course, the Hibernate SQL dialect is required.

Now that you've configured everything correctly, using Hibernate in a managed environment isn't much different than using it in a non-managed environment: Just create a `Configuration` with mappings and build a `SessionFactory`. However, some of the transaction environment–related settings deserve some extra consideration.

Java already has a standard transaction API, JTA, which is used to control transactions in a managed environment with J2EE. This is called *container-managed transactions* (CMT). If a JTA transaction manager is present, JDBC connections are enlisted with this manager and under its full control. This isn't the case in a non-managed environment, where an application (or the pool) manages the JDBC connections and JDBC transactions directly.

Therefore, managed and non-managed environments can use different transaction methods. Since Hibernate needs to be portable across these environments, it defines an API for controlling transactions. The Hibernate `Transaction` interface abstracts the underlying JTA or JDBC transaction (or, potentially, even a CORBA transaction). This underlying transaction strategy is set with the property `hibernate.connection.factory_class`, and it can take one of the following two values:

- `net.sf.hibernate.transaction.JDBCTransactionFactory` delegates to direct JDBC transactions. This strategy should be used with a connection pool in a non-managed environment and is the default if no strategy is specified.
- `net.sf.hibernate.transaction.JTATransactionFactory` delegates to JTA. This is the correct strategy for CMT, where connections are enlisted with JTA. Note that if a JTA transaction is already in progress when `beginTransaction()` is called, subsequent work takes place in the context of that transaction (otherwise a new JTA transaction is started).

For a more detailed introduction to Hibernate's `Transaction` API and the effects on your specific application scenario, see chapter 5, section 5.1, "Transactions." Just remember the two steps that are necessary if you work with a J2EE application server: Set the factory class for the Hibernate `Transaction` API to JTA as described earlier, and declare the transaction manager lookup specific to your application server. The lookup strategy is required only if you use the second-level caching system in Hibernate, but it doesn't hurt to set it even without using the cache.

HIBERNATE WITH TOMCAT

Tomcat isn't a full application server; it's just a servlet container, albeit a servlet container with some features usually found only in application servers. One of these features may be used with Hibernate: the Tomcat connection pool. Tomcat uses the DBCP connection pool internally but exposes it as a JNDI datasource, just like a real application server. To configure the Tomcat datasource, you'll need to edit `server.xml` according to instructions in the Tomcat JNDI/JDBC documentation. You can configure Hibernate to use this datasource by setting `hibernate.connection.datasource`. Keep in mind that Tomcat doesn't ship with a transaction manager, so this situation is still more like a non-managed environment as described earlier.

You should now have a running Hibernate system, whether you use a simple servlet container or an application server. Create and compile a persistent class (the initial `Message`, for example), copy Hibernate and its required libraries to the classpath together with a `hibernate.properties` file, and build a `SessionFactory`.

The next section covers advanced Hibernate configuration options. Some of them are recommended, such as logging executed SQL statements for debugging or using the convenient XML configuration file instead of plain properties. However, you may safely skip this section and come back later once you have read more about persistent classes in chapter 3.

2.4 Advanced configuration settings

When you finally have a Hibernate application running, it's well worth getting to know all the Hibernate configuration parameters. These parameters let you optimize the runtime behavior of Hibernate, especially by tuning the JDBC interaction (for example, using JDBC batch updates).

We won't bore you with these details now; the best source of information about configuration options is the Hibernate reference documentation. In the previous section, we showed you the options you'll need to get started.

However, there is one parameter that we *must* emphasize at this point. You'll need it continually whenever you develop software with Hibernate. Setting the property `hibernate.show_sql` to the value `true` enables logging of all generated SQL to the console. You'll use it for troubleshooting, performance tuning, and just to see what's going on. It pays to be aware of what your ORM layer is doing—that's why ORM doesn't hide SQL from developers.

So far, we've assumed that you specify configuration parameters using a `hibernate.properties` file or an instance of `java.util.Properties` programmatically. There is a third option you'll probably like: using an XML configuration file.

2.4.1 Using XML-based configuration

You can use an XML configuration file (as demonstrated in listing 2.6) to fully configure a `SessionFactory`. Unlike `hibernate.properties`, which contains only configuration parameters, the `hibernate.cfg.xml` file may also specify the location of mapping documents. Many users prefer to centralize the configuration of Hibernate in this way instead of adding parameters to the `Configuration` in application code.

Listing 2.6 Sample `hibernate.cfg.xml` configuration file

```
?xml version='1.0'encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory name="java:/hibernate/HibernateFactory">
    <property name="show_sql">true</property>
    <property name="connection.datasource">
        java:/comp/env/jdbc/AuctionDB
    </property>
    <property name="dialect">
        net.sf.hibernate.dialect.PostgreSQLDialect
    </property>
```

Document type declaration ❶

❷ Name attribute

❸ Property specifications

```
    <property name="transaction.manager_lookup_class">
        net.sf.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <mapping resource="auction/Item.hbm.xml"/>
    <mapping resource="auction/Category.hbm.xml"/>
    <mapping resource="auction/Bid.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

❹ **Mapping document specifications**

❶ The *document type* declaration is used by the XML parser to validate this document against the Hibernate configuration DTD.

❷ The optional `name` attribute is equivalent to the property `hibernate.session_factory_name` and used for JNDI binding of the `SessionFactory`, discussed in the next section.

❸ Hibernate properties may be specified without the `hibernate` prefix. Property names and values are otherwise identical to programmatic configuration properties.

❹ Mapping documents may be specified as application resources or even as hard-coded filenames. The files used here are from our online auction application, which we'll introduce in chapter 3.

Now you can initialize Hibernate using

```
SessionFactory sessions = new Configuration()
        .configure().buildSessionFactory();
```

Wait—how did Hibernate know where the configuration file was located?

When `configure()` was called, Hibernate searched for a file named `hibernate.cfg.xml` in the classpath. If you wish to use a different filename or have Hibernate look in a subdirectory, you must pass a path to the `configure()` method:

```
SessionFactory sessions = new Configuration()
        .configure("/hibernate-config/auction.cfg.xml")
          .buildSessionFactory();
```

Using an XML configuration file is certainly more comfortable than a properties file or even programmatic property configuration. The fact that you can have the class mapping files externalized from the application's source (even if it would be only in a startup helper class) is a major benefit of this approach. You can, for example, use different sets of mapping files (and different configuration options), depending on your database and environment (development or production), and switch them programatically.

If you have both `hibernate.properties` and `hibernate.cfg.xml` in the classpath, the settings of the XML configuration file will override the settings used in the properties. This is useful if you keep some base settings in properties and override them for each deployment with an XML configuration file.

You may have noticed that the `SessionFactory` was also given a `name` in the XML configuration file. Hibernate uses this name to automatically bind the `SessionFactory` to JNDI after creation.

2.4.2 JNDI-bound SessionFactory

In most Hibernate applications, the `SessionFactory` should be instantiated once during application initialization. The single instance should then be used by all code in a particular process, and any `Session`s should be created using this single `SessionFactory`. A frequently asked question is where this factory must be placed and how it can be accessed without much hassle.

In a J2EE environment, a `SessionFactory` bound to JNDI is easily shared between different threads and between various Hibernate-aware components. Or course, JNDI isn't the only way that application components might obtain a `SessionFactory`. There are many possible implementations of this Registry pattern, including use of the `ServletContext` or a `static final` variable in a singleton. A particularly elegant approach is to use an application scope IoC (Inversion of Control) framework component. However, JNDI is a popular approach (and is exposed as a JMX service, as you'll see later). We discuss some of the alternatives in chapter 8, section 8.1, "Designing layered applications."

NOTE The Java Naming and Directory Interface (JNDI) API allows objects to be stored to and retrieved from a hierarchical structure (directory tree). JNDI implements the Registry pattern. Infrastructural objects (transaction contexts, datasources), configuration settings (environment settings, user registries), and even application objects (EJB references, object factories) may all be bound to JNDI.

The `SessionFactory` will automatically bind itself to JNDI if the property `hibernate.session_factory_name` is set to the name of the directory node. If your runtime environment doesn't provide a default JNDI context (or if the default JNDI implementation doesn't support instances of `Referenceable`), you need to specify a JNDI initial context using the properties `hibernate.jndi.url` and `hibernate.jndi.class`.

Here is an example Hibernate configuration that binds the `SessionFactory` to the name `hibernate/HibernateFactory` using Sun's (free) file system–based JNDI implementation, `fscontext.jar`:

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

Of course, you can also use the XML-based configuration for this task. This example also isn't realistic, since most application servers that provide a connection pool through JNDI also have a JNDI implementation with a writable default context. JBoss certainly has, so you can skip the last two properties and just specify a name for the `SessionFactory`. All you have to do now is call `Configuration.configure().buildSessionFactory()` once to initialize the binding.

NOTE Tomcat comes bundled with a read-only JNDI context, which isn't writable from application-level code after the startup of the servlet container. Hibernate can't bind to this context; you have to either use a full context implementation (like the Sun FS context) or disable JNDI binding of the `SessionFactory` by omitting the `session_factory_name` property in the configuration.

Let's look at some other very important configuration settings that log Hibernate operations.

2.4.3 Logging

Hibernate (and many other ORM implementations) executes SQL statements *asynchronously*. An `INSERT` statement isn't usually executed when the application calls `Session.save()`; an `UPDATE` isn't immediately issued when the application calls `Item.addBid()`. Instead, the SQL statements are usually issued at the end of a transaction. This behavior is called *write-behind*, as we mentioned earlier.

This fact is evidence that tracing and debugging ORM code is sometimes nontrivial. In theory, it's possible for the application to treat Hibernate as a black box and ignore this behavior. Certainly the Hibernate application can't detect this asynchronicity (at least, not without resorting to direct JDBC calls). However, when you find yourself troubleshooting a difficult problem, you need to be able to see *exactly* what's going on inside Hibernate. Since Hibernate is open source, you can

easily step into the Hibernate code. Occasionally, doing so helps a great deal! But, especially in the face of asynchronous behavior, debugging Hibernate can quickly get you lost. You can use logging to get a view of Hibernate's internals.

We've already mentioned the `hibernate.show_sql` configuration parameter, which is usually the first port of call when troubleshooting. Sometimes the SQL alone is insufficient; in that case, you must dig a little deeper.

Hibernate logs all interesting events using Apache `commons-logging`, a thin abstraction layer that directs output to either Apache log4j (if you put `log4j.jar` in your classpath) or JDK1.4 logging (if you're running under JDK1.4 or above and log4j isn't present). We recommend log4j, since it's more mature, more popular, and under more active development.

To see any output from log4j, you'll need a file named `log4j.properties` in your classpath (right next to `hibernate.properties` or `hibernate.cfg.xml`). This example directs all log messages to the console:

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
⇒ %5p %c{1}:%L - %m%n
### root logger option ###
log4j.rootLogger=warn, stdout
### Hibernate logging options ###
log4j.logger.net.sf.hibernate=info
### log JDBC bind parameters ###
log4j.logger.net.sf.hibernate.type=info
### log PreparedStatement cache activity ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

With this configuration, you won't see many log messages at runtime. Replacing info with `debug` for the `log4j.logger.net.sf.hibernate` category will reveal the inner workings of Hibernate. Make sure you don't do this in a production environment—writing the log will be much slower than the actual database access.

Finally, you have the `hibernate.properties`, `hibernate.cfg.xml`, and `log4j.properties` configuration files.

There is another way to configure Hibernate, if your application server supports the Java Management Extensions.

2.4.4 Java Management Extensions (JMX)

The Java world is full of specifications, standards, and, of course, implementations of these. A relatively new but important standard is in its first version: the *Java*

Management Extensions (JMX). JMX is about the management of systems components or, better, of system services.

Where does Hibernate fit into this new picture? Hibernate, when deployed in an application server, makes use of other services like managed transactions and pooled database transactions. But why not make Hibernate a managed service itself, which others can depend on and use? This is possible with the Hibernate JMX integration, making Hibernate a managed JMX component.

The JMX specification defines the following components:

- *The JMX MBean*—A reusable component (usually infrastructural) that exposes an interface for *management* (administration)
- *The JMX container*—Mediates generic access (local or remote) to the MBean
- *The (usually generic) JMX client*—May be used to administer any MBean via the JMX container

An application server with support for JMX (such as JBoss) acts as a JMX container and allows an MBean to be configured and initialized as part of the application server startup process. It's possible to monitor and administer the MBean using the application server's administration console (which acts as the JMX client).

An MBean may be packaged as a JMX service, which is not only portable between application servers with JMX support but also deployable to a running system (a *hot deploy*).

Hibernate may be packaged and administered as a JMX MBean. The Hibernate JMX service allows Hibernate to be initialized at application server startup and controlled (configured) via a JMX client. However, JMX components aren't automatically integrated with container-managed transactions. So, the configuration options in listing 2.7 (a JBoss service deployment descriptor) look similar to the usual Hibernate settings in a managed environment.

Listing 2.7 Hibernate `jboss-service.xml` JMX deployment descriptor

```
<server>
<mbean
     code="net.sf.hibernate.jmx.HibernateService"
     name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
  <depends>jboss.jca:service=RARDeployer</depends>
  <depends>jboss.jca:service=LocalTxCM,name=DataSource</depends>
  <attribute name="MapResources">
    auction/Item.hbm.xml, auction/Bid.hbm.xml
  </attribute>
```

```
  <attribute name="JndiName">
     java:/hibernate/HibernateFactory
  </attribute>
  <attribute name="Datasource">
     java:/comp/env/jdbc/AuctionDB
  </attribute>
  <attribute name="Dialect">
     net.sf.hibernate.dialect.PostgreSQLDialect
  </attribute>
  <attribute name="TransactionStrategy">
     net.sf.hibernate.transaction.JTATransactionFactory
  </attribute>
  <attribute name="TransactionManagerLookupStrategy">
     net.sf.hibernate.transaction.JBossTransactionManagerLookup
  </attribute>
  <attribute name="UserTransactionName">
     java:/UserTransaction
  </attribute>
</mbean>
</server>
```

The `HibernateService` depends on two other JMX services: `service=RARDeployer` and `service=LocalTxCM,name=DataSource`, both in the `jboss.jca` service domain name.

The Hibernate MBean may be found in the package `net.sf.hibernate.jmx`. Unfortunately, lifecycle management methods like starting and stopping the JMX service aren't part of the JMX 1.0 specification. The methods `start()` and `stop()` of the `HibernateService` are therefore specific to the JBoss application server.

NOTE If you're interested in the advanced usage of JMX, JBoss is a good open source starting point: All services (even the EJB container) in JBoss are implemented as MBeans and can be managed via a supplied console interface.

We recommend that you try to configure Hibernate programmatically (using the `Configuration` object) before you try to run Hibernate as a JMX service. However, some features (like hot-redeployment of Hibernate applications) may be possible only with JMX, once they become available in Hibernate. Right now, the biggest advantage of Hibernate with JMX is the automatic startup; it means you no longer have to create a `Configuration` and build a `SessionFactory` in your application code, but can simply access the `SessionFactory` through JNDI once the `HibernateService` has been deployed and started.

2.5 Summary

In this chapter, we took a high-level look at Hibernate and its architecture after running a simple "Hello World" example. You also saw how to configure Hibernate in various environments and with various techniques, even including JMX.

The `Configuration` and `SessionFactory` interfaces are the entry points to Hibernate for applications running in both managed and non-managed environments. Hibernate provides additional APIs, such as the `Transaction` interface, to bridge the differences between environments and allow you to keep your persistence code portable.

Hibernate can be integrated into almost every Java environment, be it a servlet, an applet, or a fully managed three-tiered client/server application. The most important elements of a Hibernate configuration are the database resources (connection configuration), the transaction strategies, and, of course, the XML-based mapping metadata.

Hibernate's configuration interfaces have been designed to cover as many usage scenarios as possible while still being easy to understand. Usually, a single file named `hibernate.cfg.xml` and one line of code are enough to get Hibernate up and running.

None of this is much use without some persistent classes and their XML mapping documents. The next chapter is dedicated to writing and mapping persistent classes. You'll soon be able to store and retrieve persistent objects in a real application with a nontrivial object/relational mapping.

JAVA

HIBERNATE IN ACTION

Christian Bauer and Gavin King

"The Bible of Hibernate"

—Ara Abrahamian, XDoclet Lead Developer

Hibernate practically exploded on the Java scene. Why is this open-source tool so popular? Because it automates a tedious task: persisting your Java objects to a relational database. The inevitable mismatch between your object-oriented code and the relational database requires you to write code that maps one to the other. This code is often complex, tedious and costly to develop. Hibernate does the mapping for you.

Not only that, Hibernate makes it easy. Positioned as a layer between your application and your database, Hibernate takes care of loading and saving of objects. Hibernate applications are cheaper, more portable, and more resilient to change. And they perform better than anything you are likely to develop yourself.

Hibernate in Action carefully explains the concepts you need, then gets you going. It builds on a single example to show you how to use Hibernate in practice, how to deal with concurrency and transactions, how to efficiently retrieve objects and use caching.

The authors created Hibernate and they field questions from the Hibernate community every day—they know how to make Hibernate sing. Knowledge and insight seep out of every pore of this book.

What's Inside

- ORM concepts
- Getting started
- Many real-world tasks
- The Hibernate application development process

Ask the Authors

Ebook edition

www.manning.com/bauer

A member of the core Hibernate developer team, **Christian Bauer** maintains the Hibernate documentation and website. He is a senior software engineer in Frankfurt, Germany. **Gavin King** is the Hibernate founder and principal developer. He is a J2EE consultant based in Melbourne, Australia.

$44.95 US/$62.95 Canada

ISBN 1-932394-15-X

The ultimate Hibernate reference

SAMPLE
CHAPTER

HIBERNATE IN ACTION

Christian Bauer
Gavin King

MANNING

Hibernate in Action

by Christian Bauer

and

Gavin King

Chapter 6

contents

6 Advanced mapping concepts

This chapter covers

- The Hibernate type system
- Custom mapping types
- Collection mappings
- One-to-one and many-to-many associations

In chapter 3, we introduced the most important ORM features provided by Hibernate. You've met basic class and property mappings, inheritance mappings, component mappings, and one-to-many association mappings. We now continue exploring these topics by turning to the more exotic collection and association mappings. At various places, we'll warn you against using a feature without careful consideration. For example, it's usually possible to implement any domain model using only component mappings and one-to-many (occasionally one-to-one) associations. The exotic mapping features should be used with care, perhaps even *avoided* most of the time.

Before we start to talk about the exotic features, you need a more rigorous understanding of Hibernate's type system—particularly of the distinction between entity and value types.

6.1 Understanding the Hibernate type system

In chapter 3, section 3.5.1, "Entity and value types," we first distinguished between entity and value types, a central concept of ORM in Java. We must elaborate that distinction in order for you to fully understand the Hibernate type system of entities, value types, and mapping types.

Entities are the coarse-grained classes in a system. You usually define the features of a system in terms of the entities involved: "the user places a bid for an item" is a typical feature definition that mentions three entities. Classes of value type often don't appear in the business requirements—they're usually the fine-grained classes representing strings, numbers, and monetary amounts. Occasionally, value types *do* appear in feature definitions: "the user changes billing address" is one example, assuming that `Address` is a value type, but this is atypical.

More formally, an *entity* is any class whose instances have their own persistent identity. A *value type* is a class that doesn't define some kind of persistent identity. In practice, this means entity types are classes with identifier properties, and value-type classes depend on an entity.

At runtime, you have a graph of entity instances interleaved with value type instances. The entity instances may be in any of the three persistent lifecycle states: transient, detached, or persistent. We don't consider these lifecycle states to apply to the value type instances.

Therefore, entities have their own lifecycle. The `save()` and `delete()` methods of the Hibernate `Session` interface apply to instances of entity classes, never to value type instances. The persistence lifecycle of a value type instance is completely tied to the lifecycle of the owning entity instance. For example, the username

becomes persistent when the user is saved; it never becomes persistent independently of the user.

In Hibernate, a value type may define associations; it's possible to navigate from a value type instance to some other entity. However, it's *never* possible to navigate from the other entity back to the value type instance. Associations *always* point to entities. This means that a value type instance is owned by exactly one entity when it's retrieved from the database—it's never shared.

At the level of the database, any table is considered an entity. However, Hibernate provides certain constructs to hide the existence of a database-level entity from the Java code. For example, a many-to-many association mapping hides the intermediate association table from the application. A collection of strings (more accurately, a collection of value-typed instances) behaves like a value type from the point of view of the application; however, it's mapped to its own table. Although these features seem nice at first (they simplify the Java code), we have over time become suspicious of them. Inevitably, these hidden entities end up needing to be exposed to the application as business requirements evolve. The many-to-many association table, for example, often has additional columns that are added when the application is maturing. We're almost prepared to recommend that every database-level entity be exposed to the application as an entity class. For example, we'd be inclined to model the many-to-many association as two one-to-many associations to an intervening entity class. We'll leave the final decision to you, however, and return to the topic of many-to-many entity associations later in this chapter.

So, entity classes are always mapped to the database using `<class>`, `<subclass>`, and `<joined-subclass>` mapping elements. How are value types mapped?

Consider this mapping of the CaveatEmptor `User` and email address:

```
<property
        name="email"
        column="EMAIL"
        type="string"/>
```

Let's focus on the `type="string"` attribute. In ORM, you have to deal with Java types and SQL data types. The two different type systems must be bridged. This is the job of the Hibernate *mapping types*, and `string` is the name of a built-in Hibernate mapping type.

The `string` mapping type isn't the only one built into Hibernate; Hibernate comes with various mapping types that define default persistence strategies for primitive Java types and certain JDK classes.

6.1.1 Built-in mapping types

Hibernate's built-in mapping types usually share the name of the Java type they map; however, there may be more than one Hibernate mapping type for a particular Java type. Furthermore, the built-in types may *not* be used to perform arbitrary conversions, such as mapping a `VARCHAR` field value to a Java `Integer` property value. You may define your own *custom value types* to do this kind of thing, as discussed later in this chapter.

We'll now discuss the basic, date and time, large object, and various other built-in mapping types and show you what Java and SQL data types they handle.

Java primitive mapping types

The basic mapping types in table 6.1 map Java primitive types (or their wrapper types) to appropriate built-in SQL standard types.

Table 6.1 Primitive types

Mapping type	Java type	Standard SQL built-in type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes_no	boolean or java.lang.Boolean	`CHAR(1)` (`'Y'` or `'N'`)
true_false	boolean or java.lang.Boolean	`CHAR(1)` (`'T'` or `'F'`)

You've probably noticed that your database doesn't support some of the SQL types listed in table 6.1. The listed names are ANSI-standard data types. Most database vendors ignore this part of the SQL standard (because their type systems sometimes predate the standard). However, the JDBC driver provides a partial abstraction of vendor-specific SQL data types, allowing Hibernate to work with ANSI-standard

types when executing data manipulation language (DML). For database-specific DDL generation, Hibernate translates from the ANSI-standard type to an appropriate vendor-specific type, using the built-in support for specific SQL dialects. (You usually don't have to worry about SQL data types if you're using Hibernate for data access and data schema definition.)

Date and time mapping types

Table 6.2 lists Hibernate types associated with dates, times, and timestamps. In your domain model, you may choose to represent date and time data using either `java.util.Date`, `java.util.Calendar`, or the subclasses of `java.util.Date` defined in the `java.sql` package. This is a matter of taste, and we leave the decision to you—make sure you're consistent, however!

Table 6.2 Date and time types

Mapping type	Java type	Standard SQL built-in type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Large object mapping types

Table 6.3 lists Hibernate types for handling binary data and large objects. Note that none of these types may be used as the type of an identifier property.

Table 6.3 Binary and large object types

Mapping type	Java type	Standard SQL built-in type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

`java.sql.Blob` and `java.sql.Clob` are the most efficient way to handle large objects in Java. Unfortunately, an instance of `Blob` or `Clob` is only useable until the JDBC transaction completes. So if your persistent class defines a property of `java.sql.Clob` or `java.sql.Blob` (not a good idea anyway), you'll be restricted in how instances of the class may be used. In particular, you won't be able to use instances of that class as detached objects. Furthermore, many JDBC drivers don't feature working support for `java.sql.Blob` and `java.sql.Clob`. Therefore, it makes more sense to map large objects using the `binary` or `text` mapping type, assuming retrieval of the entire large object into memory isn't a performance killer.

Note you can find up-to-date design patterns and tips for large object usage on the Hibernate website, with tricks for particular platforms.

Various JDK mapping types

Table 6.4 lists Hibernate types for various other Java types of the JDK that may be represented as `VARCHAR`s in the database.

Table 6.4 Other JDK-related types

Mapping type	Java type	Standard SQL built-in type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Certainly, `<property>` isn't the only Hibernate mapping element that has a `type` attribute.

6.1.2 Using mapping types

All of the basic mapping types may appear almost anywhere in the Hibernate mapping document, on normal property, identifier property, and other mapping elements.

The `<id>`, `<property>`, `<version>`, `<discriminator>`, `<index>`, and `<element>` elements all define an attribute named `type`. (There are certain limitations on which mapping basic types may function as an identifier or discriminator type, however.)

You can see how useful the built-in mapping types are in this mapping for the `BillingDetails` class:

```
<class name="BillingDetails"
       table="BILLING_DETAILS"
       discriminator-value="null">

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

    <discriminator type="character" column="TYPE"/>

    <property name="number" type="string"/>
    ...
</class>
```

The `BillingDetails` class is mapped as an entity. Its `discriminator`, identifier, and `number` properties are value typed, and we use the built-in Hibernate mapping types to specify the conversion strategy.

It's often not necessary to explicitly specify a built-in mapping type in the XML mapping document. For instance, if you have a property of Java type `java.lang.String`, Hibernate will discover this using reflection and select `string` by default. We can easily simplify the previous mapping example:

```
<class name="BillingDetails"
       table="BILLING_DETAILS"
       discriminator-value="null">

    <id name="id" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>

    <discriminator type="character" column="TYPE"/>

    <property name="number"/>
    ....
</class>
```

The most important case where this approach doesn't work well is a `java.util.Date` property. By default, Hibernate interprets a `Date` as a `timestamp` mapping. You'd need to explicitly specify `type="time"` or `type="date"` if you didn't wish to persist both date and time information.

For each of the built-in mapping types, a constant is defined by the class `net.sf.hibernate.Hibernate`. For example, `Hibernate.STRING` represents the `string` mapping type. These constants are useful for query parameter binding, as discussed in more detail in chapter 7:

```
session.createQuery("from Item i where i.description like :desc")
    .setParameter("desc", desc, Hibernate.STRING)
    .list();
```

These constants are also useful for programmatic manipulation of the Hibernate mapping metamodel, as discussed in chapter 3.

Of course, Hibernate isn't limited to the built-in mapping types. We consider the extensible mapping type system one of the core features and an important aspect that makes Hibernate so flexible.

Creating custom mapping types

Object-oriented languages like Java make it easy to define new types by writing new classes. Indeed, this is a fundamental part of the definition of object orientation. If you were limited to the predefined built-in Hibernate mapping types when declaring properties of persistent classes, you'd lose much of Java's expressiveness. Furthermore, your domain model implementation would be tightly coupled to the physical data model, since new type conversions would be impossible.

Most ORM solutions that we've seen provide some kind of support for user-defined strategies for performing type conversions. These are often called *converters*. For example, the user would be able to create a new strategy for persisting a property of JDK type `Integer` to a `VARCHAR` column. Hibernate provides a similar, much more powerful, feature called *custom mapping types*.

Hibernate provides two user-friendly interfaces that applications may use when defining new mapping types. These interfaces reduce the work involved in defining custom mapping types and insulate the custom type from changes to the Hibernate core. This allows you to easily upgrade Hibernate and keep your existing custom mapping types. You can find many examples of useful Hibernate mapping types on the Hibernate community website.

The first of the programming interfaces is `net.sf.hibernate.UserType`. `UserType` is suitable for most simple cases and even for some more complex problems. Let's use it in a simple scenario.

Our `Bid` class defines an `amount` property; our `Item` class defines an `initialPrice` property, both monetary values. So far, we've only used a simple `BigDecimal` to represent the value, mapped with `big_decimal` to a single `NUMERIC` column.

Suppose we wanted to support multiple currencies in our auction application and that we had to refactor the existing domain model for this (customer-driven) change. One way to implement this change would be to add new properties to `Bid` and `Item: amountCurrency` and `initialPriceCurrency`. We would then map these new properties to additional `VARCHAR` columns with the built-in `currency` mapping type. We hope you *never* use this approach!

Creating a UserType

Instead, we should create a `MonetaryAmount` class that encapsulates both currency and amount. Note that this is a class of the domain model; it doesn't have any dependency on Hibernate interfaces:

```
public class MonetaryAmount implements Serializable {

    private final BigDecimal value;
    private final Currency currency;

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() { return value; }

    public Currency getCurrency() { return currency; }

    public boolean equals(Object o) { ... }
    public int hashCode() { ...}
}
```

We've made `MonetaryAmount` an immutable class. This is a good practice in Java. Note that we have to implement `equals()` and `hashCode()` to finish the class (there is nothing special to consider here). We use this new `MonetaryAmount` to replace the `BigDecimal` of the `initialPrice` property in `Item`. Of course, we can, and should use it for all other `BigDecimal` prices in our persistent classes (such as the `Bid.amount`) and even in business logic (for example, in the billing system).

Let's map this refactored property of `Item` to the database. Suppose we're working with a legacy database that contains all monetary amounts in USD. Our application is no longer restricted to a single currency (the point of the refactoring), but it takes time to get the changes done by the database team. We need to convert the amount to USD when we persist the `MonetaryAmount` and convert it back to USD when we are loading objects.

For this, we create a `MonetaryAmountUserType` class that implements the Hibernate interface `UserType`. Our custom mapping type, is shown in listing 6.1.

Listing 6.1 Custom mapping type for monetary amounts in USD

```
package auction.customtypes;

import ...;

public class MonetaryAmountUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.NUMERIC};
```

```
public int[] sqlTypes() { return SQL_TYPES; }  ❶

public Class returnedClass() { return MonetaryAmount.class; }  ❷

public boolean equals(Object x, Object y) {  ❸
    if (x == y) return true;
    if (x == null || y == null) return false;
    return x.equals(y);
        }

public Object deepCopy(Object value) { return value; }  ❹

public boolean isMutable() { return false; }  ❺

public Object nullSafeGet(ResultSet resultSet,  ❻
                          String[] names,
                          Object owner)
        throws HibernateException, SQLException {

   if (resultSet.wasNull()) return null;
   BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);

   return new MonetaryAmount(valueInUSD, Currency.getInstance)"USD"));
}

public void nullSafeSet(PreparedStatement statement,  ❼
                        Object value,
                        int index)
        throws HibernateException, SQLException {
    if (value == null) {
        statement.setNull(index, Types.NUMERIC);
    } else {
        MonetaryAmount anyCurrency = (MonetaryAmount)value;
        MonetaryAmount amountInUSD =
          MonetaryAmount.convert( anyCurrency,
                                  Currency.getInstance("USD") );
        // The convert() method isn't shown in our examples
        statement.setBigDecimal(index, amountInUSD.getValue());
    }
  }
}
```

❶ The `sqlTypes()` method tells Hibernate what SQL column types to use for DDL schema generation. The type codes are defined by `java.sql.Types`. Notice that this method returns an array of type codes. A `UserType` may map a single property to *multiple* columns, but our legacy data model only has a single `NUMERIC`.

❷ `returnedClass()` tells Hibernate what Java type is mapped by this `UserType`.

❸ The `UserType` is responsible for dirty-checking property values. The `equals()` method compares the current property value to a previous snapshot and determines whether the property is dirty and must by saved to the database.

❹ The `UserType` is also partially responsible for creating the snapshot in the first place. Since `MonetaryAmount` is an immutable class, the `deepCopy()` method returns its argument. In the case of a mutable type, it would need to return a copy of the argument to be used as the snapshot value. This method is also called when an instance of the type is written to or read from the second-level cache.

❺ Hibernate can make some minor performance optimizations for immutable types like this one. The `isMutable()` method tells Hibernate that this type is immutable.

❻ The `nullSafeGet()` method retrieves the property value from the JDBC `ResultSet`. You can also access the `owner` of the component if you need it for the conversion. All database values are in USD, so you have to convert the `MonetaryAmount` returned by this method before you show it to the user.

❼ The `nullSafeSet()` method writes the property value to the JDBC `PreparedStatement`. This method takes whatever currency is set and converts it to a simple `BigDecimal` USD value before saving.

We now map the `initialPrice` property of `Item` as follows:

```
<property name="initialPrice"
          column="INITIAL_PRICE"
          type="auction.customtypes.MonetaryAmountUserType"/>
```

This is the simplest kind of transformation that a `UserType` could perform. *Much* more sophisticated things are possible. A custom mapping type could perform validation; it could read and write data to and from an LDAP directory; it could even retrieve persistent objects from a different Hibernate `Session` for a different database. You're limited mainly by your imagination!

We'd prefer to represent both the amount and currency of our monetary amounts in the database, especially if the schema isn't legacy but can be defined (or updated quickly). We could still use a `UserType`, but then we wouldn't be able to use the amount (or currency) in object queries. The Hibernate query engine (discussed in more detail in the next chapter) wouldn't know anything about the individual properties of `MonetaryAmount`. You can access the properties in your Java code (`MonetaryAmount` is just a regular class of the domain model, after all), but not in Hibernate queries.

Instead, we should use a `CompositeUserType` if we need the full power of Hibernate queries. This (slightly more complex) interface exposes the properties of our `MonetaryAmount` to Hibernate.

Creating a CompositeUserType

To demonstrate the flexibility of custom mapping types, we don't change our `MonetaryAmount` class (and other persistent classes) at all—we change only the custom mapping type, as shown in listing 6.2.

Listing 6.2 Custom mapping type for monetary amounts in new database schemas

```
package auction.customtypes;

import ...;

public class MonetaryAmountCompositeUserType
        implements CompositeUserType {

    public Class returnedClass() { return MonetaryAmount.class; }

    public boolean equals(Object x, Object y) {
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public Object deepCopy(Object value) {
        return value; // MonetaryAmount is immutable
    }

    public boolean isMutable() { return false; }

    public Object nullSafeGet(ResultSet resultSet,
                              String[] names,
                              SessionImplementor session,
                              Object owner)
            throws HibernateException, SQLException {

        if (resultSet.wasNull()) return null;
        BigDecimal value = resultSet.getBigDecimal( names[0] );
        Currency currency =
            Currency.getInstance(resultSet.getString( names[1] ));
        return new MonetaryAmount(value, currency);
    }

    public void nullSafeSet(PreparedStatement statement,
                            Object value,
                            int index,
                            SessionImplementor session)
            throws HibernateException, SQLException {
```

```
        if (value==null) {
            statement.setNull(index, Types.NUMERIC);
            statement.setNull(index+1, Types.VARCHAR);
        } else {
            MonetaryAmount amount = (MonetaryAmount) value;
            String currencyCode =
                        amount.getCurrency().getCurrencyCode();
            statement.setBigDecimal( index, amount.getValue() );
            statement.setString( index+1, currencyCode );
        }
    }

    public String[] getPropertyNames() {  (1)
        return new String[] { "value", "currency" };

    }

    public Type[] getPropertyTypes() {  (2)
        return new Type[] { Hibernate.BIG_DECIMAL, Hibernate.CURRENCY };
    }

    public Object getPropertyValue(Object component,  (3)
                                   int property)
            throws HibernateException {
        MonetaryAmount MonetaryAmount = (MonetaryAmount) component;
        if (property == 0)
            return MonetaryAmount.getValue()();
        else
            return MonetaryAmount.getCurrency();
    }

    public void setPropertyValue(Object component,  (4)
                                 int property,
                                 Object value) throws HibernateException {
       throw new UnsupportedOperationException("Immutable!");
    }

    public Object assemble(Serializable cached,  (5)
                           SessionImplementor session,
                           Object owner)
       throws HibernateException {
        return cached;
    }

    public Serializable disassemble(Object value,  (6)
                                    SessionImplementor session)
        throws HibernateException {
        return (Serializable) value;
    }

}
```

❶ A `CompositeUserType` has its own properties, defined by `getPropertyNames()`.

❷ The properties each have their own type, as defined by `getPropertyTypes()`.

❸ The `getPropertyValue()` method returns the value of an individual property of the `MonetaryAmount`.

❹ Since `MonetaryAmount` is immutable, we can't set property values individually (no problem; this method is optional).

❺ The `assemble()` method is called when an instance of the type is read from the second-level cache.

❻ The `disassemble()` method is called when an instance of the type is written to the second-level cache.

The order of properties must be the same in the `getPropertyNames()`, `getPropertyTypes()`, and `getPropertyValues()` methods. The `initialPrice` property now maps to two columns, so we declare both in the mapping file. The first column stores the value; the second stores the currency of the `MonetaryAmount` (the order of columns must match the order of properties in your type implementation):

```
<property name="initialPrice"
          type="auction.customtypes.MonetaryAmountCompositeUserType">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>
```

In a query, we can now refer to the `amount` and `currency` properties of the custom type, even though they don't appear anywhere in the mapping document as individual properties:

```
from Item i
where i.initialPrice.value > 100.0
  and i.initialPrice.currency = 'AUD'
```

We've expanded the buffer between the Java object model and the SQL database schema with our custom composite type. Both representations can now handle changes more robustly.

If implementing custom types seems complex, relax; you rarely need to use a custom mapping type. An alternative way to represent the `MonetaryAmount` class is to use a component mapping, as in section 3.5.2, "Using components." The decision to use a custom mapping type is often a matter of taste.

Let's look at an extremely important, application of custom mapping types. The *type-safe enumeration* design pattern is found in almost all enterprise applications.

Using enumerated types

An *enumerated type* is a common Java idiom where a class has a constant (small) number of immutable instances.

For example, the `Comment` class (users giving comments about other users in CaveatEmptor) defines a `rating`. In our current model, we have a simple `int` property. A typesafe (and much better) way to implement different ratings (after all, we probably don't want arbitrary integer values) is to create a `Rating` class as follows:

```
package auction;

public class Rating implements Serializable {

    private String name;

    public static final Rating EXCELLENT = new Rating("Excellent");
    public static final Rating OK = new Rating("OK");
    public static final Rating LOW = new Rating("Low");
    private static final Map INSTANCES = new HashMap();

    static {
        INSTANCES.put(EXCELLENT.toString(), EXCELLENT);
        INSTANCES.put(OK.toString(), OK);
        INSTANCES.put(LOW.toString(), LOW);
    }
    private Rating(String name) {
        this.name=name;
    }

    public String toString() {
        return name;
    }

    Object readResolve() {
        return getInstance(name);
    }

    public static Rating getInstance(String name) {
        return (Rating) INSTANCES.get(name);
    }
}
```

We then change the `rating` property of our `Comment` class to use this new type. In the database, ratings would be represented as `VARCHAR values`. Creating a `UserType` for `Rating`-valued properties is straightforward:

```
package auction.customtypes;

import ...;
public class RatingUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.VARCHAR};
```

```
    public int[] sqlTypes() { return SQL_TYPES; }
    public Class returnedClass() { return Rating.class; }
    public boolean equals(Object x, Object y) { return x == y; }
    public Object deepCopy(Object value) { return value; }
    public boolean isMutable() { return false; }

    public Object nullSafeGet(ResultSet resultSet,
                              String[] names,
                              Object owner)
            throws HibernateException, SQLException {

      String name = resultSet.getString(names[0]);
      return resultSet.wasNull() ? null : Rating.getInstance(name);
    }

    public void nullSafeSet(PreparedStatement statement,
                            Object value,
                            int index)
            throws HibernateException, SQLException {

        if (value == null) {
            statement.setNull(index, Types.VARCHAR);
        } else {
            statement.setString(index, value.toString());
        }
    }
}
```

This code is basically the same as the `UserType` implemented earlier. The implementation of `nullSafeGet()` and `nullSafeSet()` is again the most interesting part, containing the logic for the conversion.

One problem you might run into is using enumerated types in Hibernate queries. Consider the following query in HQL that retrieves all comments rated "Low":

```
Query q =
    session.createQuery("from Comment c where c.rating = Rating.LOW");
```

This query doesn't work, because Hibernate doesn't know what to do with `Rating.LOW` and will try to use it as a literal. We have to use a `bind` parameter and set the rating value for the comparison dynamically (which is what we need for other reasons most of the time):

```
Query q =
    session.createQuery("from Comment c where c.rating = :rating");
q.setParameter("rating",
               Rating.LOW,
               Hibernate.custom(RatingUserType.class));
```

The last line in this example uses the static helper method `Hibernate.custom()` to convert the custom mapping type to a Hibernate `Type`, a simple way to tell Hibernate about our enumeration mapping and how to deal with the `Rating.LOW` value.

If you use enumerated types in many places in your application, you may want to take this example `UserType` and make it more generic. JDK 1.5 introduces a new language feature for defining enumerated types, and we recommend using a custom mapping type until Hibernate gets native support for JDK 1.5 features. (Note that the Hibernate2 `PersistentEnum` is considered deprecated and shouldn't be used.)

We've now discussed all kinds of Hibernate mapping types: built-in mapping types, user-defined custom types, and even components (chapter 3). They're all considered value types, because they map objects of value type (not entities) to the database. We're now ready to explore *collections* of value typed instances.

6.2 Mapping collections of value types

You've already seen collections in the context of entity relationships in chapter 3. In this section, we discuss collections that contain instances of a value type, including collections of components. Along the way, you'll meet some of the more advanced features of Hibernate collection mappings, which can also be used for collections that represent entity associations, as discussed later in this chapter.

6.2.1 Sets, bags, lists, and maps

Suppose that our sellers can attach images to `Item`s. An image is accessible only via the containing item; it doesn't need to support associations to any other entity in our system. In this case, it isn't unreasonable to model the image as a value type. `Item` would have a collection of images that Hibernate would consider to be part of the `Item`, without its own lifecycle.

We'll run through several ways to implement this behavior using Hibernate. For now, let's assume that the image is stored somewhere on the filesystem and that we keep just the filename in the database. How images are stored and loaded with this approach isn't discussed.

Using a set

The simplest implementation is a `Set` of `String` filenames. We add a collection property to the `Item` class:

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}

public void setImages(Set images) {
    this.images = images;
}
```

We use the following mapping in the `Item`:

```
<set name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

The image filenames are stored in a table named `ITEM_IMAGE`. From the database's point of view, this table is separate from the `ITEM` table; but Hibernate hides this fact from us, creating the illusion that there is a single entity. The `<key>` element declares the foreign key, `ITEM_ID` of the parent entity. The `<element>` tag declares this collection as a collection of value type instances: in this case, of strings.

A set can't contain duplicate elements, so the primary key of the `ITEM_IMAGE` table consists of both columns in the `<set>` declaration: `ITEM_ID` and `FILENAME`. See figure 6.1 for a table schema example.

It doesn't seem likely that we would allow the user to attach the same image more than once, but suppose we did. What kind of mapping would be appropriate?

Using a bag

An unordered collection that permits duplicate elements is called a *bag*. Curiously, the Java `Collections` framework doesn't define a `Bag` interface. Hibernate lets you use a `List` in Java to simulate bag behavior; this is consistent with common usage in the Java community. Note, however, that the `List` contract specifies that a list is an ordered collection; Hibernate won't preserve the ordering when persisting a `List` with bag semantics. To use a bag, change the type of `images` in `Item` from `Set` to `List`, probably using `ArrayList` as an implementation. (You could also use a `Collection` as the type of the property.)

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Figure 6.1
Table structure and example data for a collection of strings

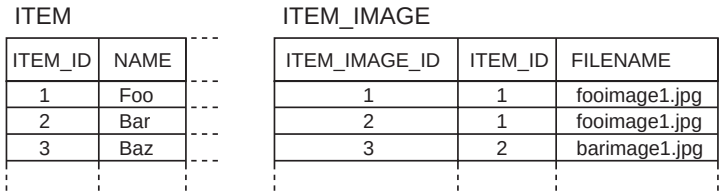

Figure 6.2 Table structure using a bag with a surrogate primary key

Changing the table definition from the previous section to permit duplicate `FILENAME`s requires another primary key. An `<idbag>` mapping lets us attach a surrogate key column to the collection table, much like the synthetic identifiers we use for entity classes:

```
<idbag name="images" lazy="true" table="ITEM_IMAGE">
    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

In this case, the primary key is the generated `ITEM_IMAGE_ID`. You can see a graphical view of the database tables in figure 6.2.

You might be wondering why the Hibernate mapping was `<idbag>` and if there is also a `<bag>` mapping. You'll soon learn more about bags, but a more likely scenario involves preserving the order in which images were attached to the `Item`. There are a number of good ways to do this; one way is to use a real list instead of a bag.

Using a list

A `<list>` mapping requires the addition of an *index column* to the database table. The index column defines the position of the element in the collection. Thus, Hibernate can preserve the ordering of the collection elements when retrieving the collection from the database if we map the collection as a `<list>`:

```
<list name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

The primary key consists of the `ITEM_ID` and `POSITION` columns. Notice that duplicate elements (`FILENAME`) are allowed, which is consistent with the semantics of a

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage1.jpg
1	2	fooimage2.jpg

Figure 6.3 Tables for a list with positional elements

list. (We don't have to change the `Item` class; the types we used earlier for the bag are the same.)

If the collection is `[fooimage1.jpg, fooimage1.jpg, fooimage2.jpg]`, the `POSITION` column contains the values `0`, `1`, and `2`, as shown in figure 6.3.

Alternatively, we could use a Java array instead of a list. Hibernate supports this usage; indeed, the details of an array mapping are virtually identical to those of a list. However, we very strongly recommend against the use of arrays, since arrays can't be lazily initialized (there is no way to proxy an array at the virtual machine level).

Now, suppose that our images have user-entered names in addition to the filenames. One way to model this in Java would be to use a `Map`, with names as keys and filenames as values.

Using a map

Mapping a `<map>` (pardon us) is similar to mapping a list:

```
<map name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The primary key consists of the `ITEM_ID` and `IMAGE_NAME` columns. The `IMAGE_NAME` column stores the keys of the map. Again, duplicate elements are allowed; see figure 6.4 for a graphical view of the tables.

This `Map` is unordered. What if we want to always sort our map by the name of the image?

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGE_NAME	FILENAME
1	Foo Image 1	fooimage1.jpg
1	Foo Image One	fooimage1.jpg
1	Foo Image 2	fooimage2.jpg

Figure 6.4 Tables for a map, using strings as indexes and elements

Sorted and ordered collections

In a startling abuse of the English language, the words *sorted* and *ordered* mean different things when it comes to Hibernate persistent collections. A *sorted collection* is sorted in memory using a Java comparator. An *ordered collection* is ordered at the database level using an SQL query with an `order by` clause.

Let's make our map of images a sorted map. This is a simple change to the mapping document:

```
<map name="images"
        lazy="true"
        table="ITEM_IMAGE"
        sort="natural">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

By specifying `sort="natural"`, we tell Hibernate to use a `SortedMap`, sorting the image names according to the `compareTo()` method of `java.lang.String`. If you want some other sorted order—for example, reverse alphabetical order—you can specify the name of a class that implements `java.util.Comparator` in the `sort` attribute. For example:

```
<map name="images"
        lazy="true"
        table="ITEM_IMAGE"
        sort="auction.util.comparator.ReverseStringComparator">

    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The behavior of a Hibernate sorted map is identical to `java.util.TreeMap`. A sorted set (which behaves like `java.util.TreeSet`) is mapped in a similar way:

```
<set name="images"
        lazy="true"
        table="ITEM_IMAGE"
        sort="natural">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Bags can't be sorted (there is no `TreeBag`, unfortunately), nor may lists; the order of list elements is defined by the list index.

Alternatively, you might choose to use an ordered map, using the sorting capabilities of the database instead of (probably less efficient) in-memory sorting:

```
<map name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The expression in the `order-by` attribute is a fragment of an SQL `order by` clause. In this case, we order by the `IMAGE_NAME` column, in ascending order. You can even write SQL function calls in the `order-by` attribute:

```
<map name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="lower(FILENAME) asc">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Notice that you can order by any column of the collection table. Both sets and bags accept the `order-by` attribute; but again, lists don't. This example uses a bag:

```
<idbag name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="ITEM_IMAGE_ID desc">
    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

Under the covers, Hibernate uses a `LinkedHashSet` and a `LinkedHashMap` to implement ordered sets and maps, so this functionality is only available in JDK 1.4 or later. Ordered bags are possible in all JDK versions.

In a real system, it's likely that we'd need to keep more than just the image name and filename; we'd probably need to create an `Image` class for this extra information. We could map `Image` as an entity class; but since we've already concluded that this isn't absolutely necessary, let's see how much further we can get without an `Image` entity (which would require an association mapping and more complex lifecycle handling).

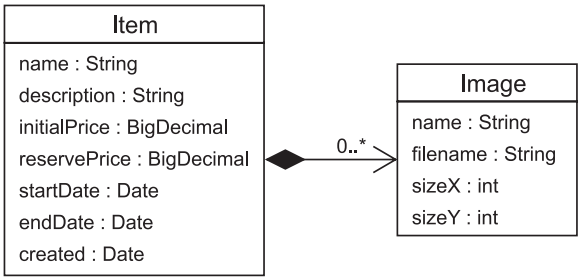

Figure 6.5
Collection of `Image` components in `Item`

In chapter 3, you saw that Hibernate lets you map user-defined classes as components, which are considered to be value types. This is still true even when component instances are collection elements.

Collections of components

Our `Image` class defines the properties `name`, `filename`, `sizeX`, and `sizeY`. It has a single association, with its parent `Item` class, as shown in figure 6.5.

As you can see from the aggregation association style (the black diamond), `Image` is a component of `Item`, and `Item` is the entity that is responsible for the lifecycle of `Image`. References to images aren't shared, so our first choice is a Hibernate component mapping. The multiplicity of the association further declares this association as many-valued—that is, many (or zero) `Images` for the same `Item`.

Writing the component class

First, we implement the `Image` class. This is just a POJO, with nothing special to consider. As you know from chapter 3, component classes don't have an identifier property. However, we must implement `equals()` (and `hashCode()`) to compare the `name`, `filename`, `sizeX`, and `sizeY` properties, to allow Hibernate's dirty checking to function correctly. Strictly speaking, implementing `equals()` and `hashCode()` isn't required for all component classes. However, we recommend it for any component class because the implementation is straightforward and "better safe than sorry" is a good motto.

The `Item` class hasn't changed: it still has a `Set` of images. Of course, the objects in this collection are no longer `String`s. Let's map this to the database.

Mapping the collection

Collections of components are mapped similarly to other collections of value type instances. The only difference is the use of `<composite-element>` in place of the familiar `<element>` tag. An ordered set of images could be mapped like this:

```
<set name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <property name="name" column="IMAGE_NAME" not-null="true"/>
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX" not-null="true"/>
        <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

This is a set, so the primary key consists of the key column and all element columns: `ITEM_ID`, `IMAGE_NAME`, `FILENAME`, `SIZEX`, and `SIZEY`. Since these columns all appear in the primary key, we declare them with `not-null="true"`. (This is clearly a disadvantage of this particular mapping.)

Bidirectional navigation

The association from `Item` to `Image` is unidirectional. If the `Image` class also declared a property named `item`, holding a reference back to the owning `Item`, we'd add a `<parent>` tag to the mapping:

```
<set name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <parent name="item"/>
        <property name="name" column="IMAGE_NAME" not-null="true"/>
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX" not-null="true"/>
        <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

True bidirectional navigation is impossible, however. You can't retrieve an `Image` independently and then navigate back to its parent `Item`. This is an important issue: You'll be able to load `Image` instances by querying for them, but components, like all value types, are retrieved by value. The `Image` objects won't have a reference to the parent (the property is `null`). You should use a full parent/child entity association, as described in chapter 3, if you need this kind of functionality.

Still, declaring all properties as `not-null` is something you should probably avoid. We need a better primary key for the `IMAGE` table.

Avoiding not-null columns

If a set of `Image`s isn't what we need, other collection styles are possible. For example, an `<idbag>` offers a surrogate collection key:

```
<idbag name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="IMAGE_NAME asc">
    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <property name="name" column="IMAGE_NAME"/>
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX"/>
        <property name="sizeY" column="SIZEY"/>
    </composite-element>
</idbag>
```

This time, the primary key is the `ITEM_IMAGE_ID` column, and it isn't important that we implement `equals()` and `hashCode()` (at least, Hibernate doesn't require it). Nor do we need to declare the properties with `not-null="true"`. They may be nullable in the case of an idbag, as shown in figure 6.6.

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGE_NAME	FILENAME
1	1	Foo Image 1	fooimage1.jpg
2	1	Foo Image 1	fooimage1.jpg
3	2	Bar Image 1	barimage1.jpg

Figure 6.6 Collection of `Image` components using a bag with a surrogate key

We should point out that there isn't a great deal of difference between this bag mapping and a standard parent/child entity relationship. The tables are identical, and even the Java code is extremely similar; the choice is mainly a matter of taste. Of course, a parent/child relationship supports shared references to the child entity and true bidirectional navigation.

We could even remove the `name` property from the `Image` class and again use the image name as the key of a map:

```
<map name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
```

```
        <index type="string" column="IMAGE_NAME"/>
        <composite-element class="Image">
            <property name="filename" column="FILENAME" not-null="true"/>
            <property name="sizeX" column="SIZEX"/>
            <property name="sizeY" column="SIZEY"/>
        </composite-element>
    </map>
```

As before, the primary key is composed of `ITEM_ID` and `IMAGE_NAME`.

A composite element class like `Image` isn't limited to simple properties of basic type like `filename`. It may contain components, using the `<nested-composite-element>` declaration, and even `<many-to-one>` associations to entities. It may not own collections, however. A composite element with a many-to-one association is useful, and we'll come back to this kind of mapping later in this chapter.

We're finally finished with value types; we'll continue with entity association mapping techniques. The simple parent/child association we mapped in chapter 3 is just one of many possible association mapping styles. Most of them are considered exotic and are rare in practice.

6.3 Mapping entity associations

When we use the word *associations*, we're always referring to relationships between entities. In chapter 3, we demonstrated a unidirectional many-to-one association, made it bidirectional, and finally turned it into a parent/child relationship (one-to-many and many-to-one).

One-to-many associations are easily the most important kind of association. In fact, we go so far as to discourage the use of more exotic association styles when a simple bidirectional many-to-one/one-to-many will do the job. In particular, a many-to-many association may always be represented as two many-to-one associations to an intervening class. This model is usually more easily extensible, so we tend not to use many-to-many associations in our applications.

Armed with this disclaimer, let's investigate Hibernate's rich association mappings starting with one-to-one associations.

6.3.1 One-to-one associations

We argued in chapter 3 that the relationships between `User` and `Address` (the user has both a `billingAddress` and a `homeAddress`) were best represented using `<component>` mappings. This is usually the simplest way to represent one-to-one relationships, since the lifecycle of one class is almost always dependent on the lifecycle of the other class, and the association is a composition.

But what if we want a dedicated table for `Address` and to map both `User` and `Address` as entities? Then, the classes have a true one-to-one association. In this case, we start with the following mapping for `Address`:

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator class="native"/>
    </id>
    <property name="street"/>
    <property name="city"/>
    <property name="zipcode"/>
</class>
```

Note that `Address` now requires an identifier property; it's no longer a component class. There are two different ways to represent a one-to-one association to this `Address` in Hibernate. The first approach adds a foreign key column to the `USER` table.

Using a foreign key association

The easiest way to represent the association from `User` to its `billingAddress` is to use a `<many-to-one>` mapping with a unique constraint on the foreign key. This may surprise you, since *many* doesn't seem to be a good description of either end of a one-to-one association! However, from Hibernate's point of view, there isn't much difference between the two kinds of foreign key associations. So, we add a foreign key column named `BILLING_ADDRESS_ID` to the `USER` table and map it as follows:

```
<many-to-one name="billingAddress"
    class="Address"
    column="BILLING_ADDRESS_ID"
    cascade="save-update"/>
```

Note that we've chosen `save-update` as the cascade style. This means the `Address` will become persistent when we create an association from a persistent `User`. Probably, `cascade="all"` makes sense for this association, since deletion of the `User` should result in deletion of the `Address`. (Remember that `Address` now has its own entity lifecycle.)

Our database schema still allows duplicate values in the `BILLING_ADDRESS_ID` column of the `USER` table, so two users could have a reference to the same address. To make this association truly one-to-one, we add `unique="true"` to the `<many-to-one>` element, constraining the relational model so that there can be only one user per address:

```
<many-to-one name="billingAddress"
    class="Address"
```

```
    column="BILLING_ADDRESS_ID"
    cascade="all"
    unique="true"/>
```

This change adds a unique constraint to the `BILLING_ADDRESS_ID` column in the DDL generated by Hibernate—resulting in the table structure illustrated by figure 6.7.

But what if we want this association to be navigable from `Address` to `User` in Java? From chapter 3, you know how to turn it into a bidirectional one-to-many collection—but we've decided that each `Address` has just one `User`, so this can't be the right solution. We don't want a collection of users in the `Address` class. Instead, we add a property named `user` (of type `User`) to the `Address` class, and map it like so in the mapping of `Address`:

```
<one-to-one name="user"
    class="User"
    property-ref="billingAddress"/>
```

This mapping tells Hibernate that the `user` association in `Address` is the reverse direction of the `billingAddress` association in `User`.

In code, we create the association between the two objects as follows:

```
Address address = new Address();
address.setStreet("646 Toorak Rd");
address.setCity("Toorak");
address.setZipcode("3000");

Transaction tx = session.beginTransaction();
User user = (User) session.get(User.class, userId);
address.setUser(user);
user.setBillingAddress(address);
tx.commit();
```

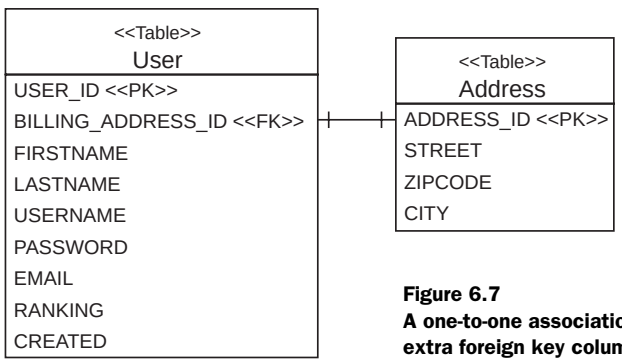

Figure 6.7
A one-to-one association with an extra foreign key column

To finish the mapping, we have to map the `homeAddress` property of `User`. This is easy enough: we add another `<many-to-one>` element to the `User` metadata, mapping a new foreign key column, `HOME_ADDRESS_ID`:

```
<many-to-one name="homeAddress"
    class="Address"
    column="HOME_ADDRESS_ID"
    cascade="save-update"
    unique="true"/>
```

The `USER` table now defines two foreign keys referencing the primary key of the `ADDRESS` table: `HOME_ADDRESS_ID` and `BILLING_ADDRESS_ID`.

Unfortunately, we can't make both the `billingAddress` and `homeAddress` associations bidirectional, since we don't know if a particular address is a billing address or a home address. (We can't decide which property name—`billingAddress` or `homeAddress`—to use for the `property-ref` attribute in the mapping of the `user` property.) We *could* try making `Address` an abstract class with subclasses `HomeAddress` and `BillingAddress` and mapping the associations to the subclasses. This approach would work, but it's complex and probably not sensible in this case.

Our advice is to avoid defining more than one one-to-one association between any two classes. If you must, leave the associations unidirectional. If you don't have more than one—if there really is exactly one instance of `Address` per `User`—there is an alternative approach to the one we've just shown. Instead of defining a foreign key column in the `USER` table, you can use a *primary key association*.

Using a primary key association

Two tables related by a primary key association share the same primary key values. The primary key of one table is also a foreign key of the other. The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the objects are saved. Before we try to solve this problem, let's see how we would map the primary key association.

For a primary key association, *both* ends of the association are mapped using the `<one-to-one>` declaration. This also means that we can no longer map both the billing and home address, only one property. Each row in the `USER` table has a corresponding row in the `ADDRESS` table. Two addresses would require an additional table, and this mapping style therefore wouldn't be adequate. Let's call this single address property `address` and map it with the `User`:

```
<one-to-one name="address"
    class="Address"
    cascade="save-update"/>
```

Next, here's the `user` of `Address`:

```
<one-to-one name="user"
    class="User"
    constrained="true"/>
```

The most interesting thing here is the use of `constrained="true"`. It tells Hibernate that there is a foreign key constraint on the primary key of `ADDRESS` that refers to the primary key of `USER`.

Now we must ensure that newly saved instances of `Address` are assigned the same identifier value as their `User`. We use a special Hibernate identifier-generation strategy called `foreign`:

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
                class="User"
                constrained="true"/>
</class>
```

The `<param>` named `property` of the `foreign` generator allows us to name a one-to-one association of the `Address` class—in this case, the `user` association. The `foreign` generator inspects the associated object (the `User`) and uses its identifier as the identifier of the new `Address`. Look at the table structure in figure 6.8.

The code to create the object association is unchanged for a primary key association; it's the same code we used earlier for the many-to-one mapping style.

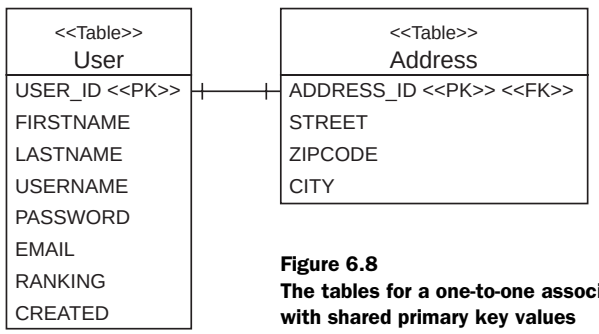

Figure 6.8
The tables for a one-to-one association with shared primary key values

There is now just one remaining entity association multiplicity we haven't discussed: many-to-many.

6.3.2 Many-to-many associations

The association between `Category` and `Item` is a many-to-many association, as you can see in figure 6.9.

In a real system, we might not use a many-to-many association. In our experience, there is almost always other information that must be attached to each link between associated instances (for example, the date and time when an item was set in a category), and the best way to represent this information is via an intermediate *association class*. In Hibernate, we could map the association class as an entity and use two one-to-many associations for either side. Perhaps more conveniently, we could also use a composite element class, a technique we'll show you later.

Nevertheless, it's the purpose of this section to implement a real many-to-many entity association. Let's start with a unidirectional example.

A unidirectional many-to-many association

If you only require unidirectional navigation, the mapping is straightforward. Unidirectional many-to-many associations are no more difficult than the collections of value type instances we covered previously. For example, if the `Category` has a set of `Items`, we can use this mapping:

```
<set name="items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

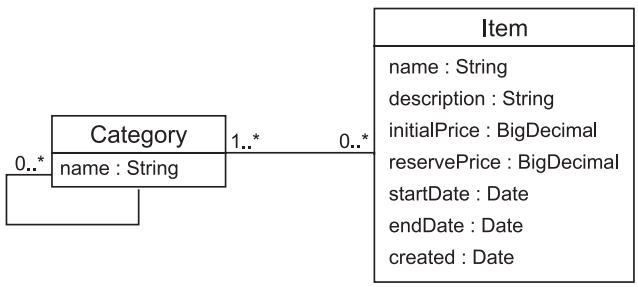

Figure 6.9 A many-to-many valued association between `Category` and `Item`

Just like a collection of value type instances, a many-to-many association has its own table, the *link table* or *association table*. In this case, the link table has two columns: the foreign keys of the `CATEGORY` and `ITEM` tables. The primary key is composed of both columns. The full table structure is shown in figure 6.10.

We can also use a bag with a separate primary key column:

```
<idbag name="items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

As usual with an `<idbag>` mapping, the primary key is a surrogate key column, `CATEGORY_ITEM_ID`. Duplicate links are therefore allowed; the same `Item` can be added twice to a particular `Category`. (This doesn't seem to be a very useful feature.)

We can even use an indexed collection (a map or list). The following example uses a list:

```
<list name="items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
    <key column="CATEGORY_ID"/>
    <index column="DISPLAY_POSITION"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</list>
```

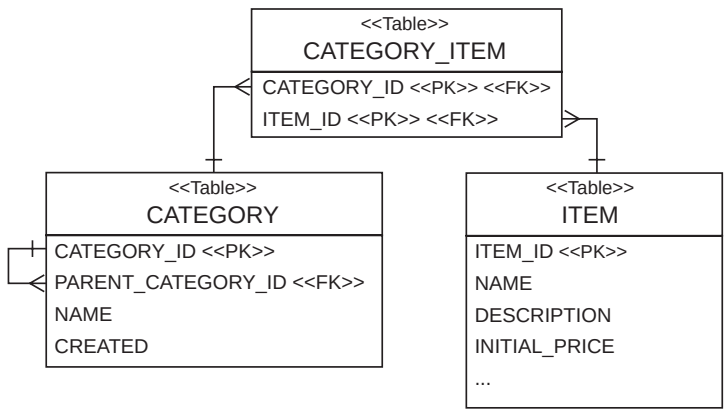

Figure 6.10 Many-to-many entity association mapped to an association table

The primary key consists of the CATEGORY_ID and DISPLAY_POSITION columns. This mapping guarantees that every Item knows its position in the Category.

Creating an object association is easy:

```
Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, categoryId);
Item item = (Item) session.get(Item.class, itemId);

cat.getItems().add(item);

tx.commit();
```

Bidirectional many-to-many associations are slightly more difficult.

A bidirectional many-to-many association

When we mapped a bidirectional one-to-many association in chapter 3 (section 3.7, "Introducing associations"), we explained why one end of the association must be mapped with inverse="true". We encourage you to review that explanation now.

The same principle applies to bidirectional many-to-many associations: each row of the link table is represented by two collection elements, one element at each end of the association. An association between an Item and a Category is represented in memory by the Item instance belonging to the items collection of the Category but also by the Category instance belonging to the categories collection of the Item.

Before we discuss the mapping of this bidirectional case, you must be aware that the code to create the object association also changes:

```
cat.getItems.add(item);
item.getCategories().add(category);
```

As always, a bidirectional association (no matter of what multiplicity) requires that you set both ends of the association.

When you map a bidirectional many-to-many association, you must declare one end of the association using inverse="true" to define which side's state is used to update the link table. You can choose for yourself which end that should be.

Recall this mapping for the items collection from the previous section:

```
<class name="Category" table="CATEGORY">
    ... <
    set name="items"
           table="CATEGORY_ITEM"
           lazy="true"
           cascade="save-update">
       <key column="CATEGORY_ID"/>
```

```
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
</class>
```

We can reuse this mapping for the `Category` end of the bidirectional association. We map the `Item` end as follows:

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
            table="CATEGORY_ITEM"
            lazy="true"
            inverse="true"
            cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Item" column="CATEGORY_ID"/>
    </set>
</class>
```

Note the use of `inverse="true"`. Once again, this setting tells Hibernate to ignore changes made to the `categories` collection and use the other end of the association (the `items` collection) as the representation that should be synchronized with the database if we manipulate the association in Java code.

We've chosen `cascade="save-update"` for both ends of the collection; this isn't unreasonable. On the other hand, `cascade="all"`, `cascade="delete"`, and `cascade="all-delete-orphans"` aren't meaningful for many-to-many associations, since an instance with potentially many parents shouldn't be deleted when just one parent is deleted.

What kinds of collections may be used for bidirectional many-to-many associations? Do you need to use the same type of collection at each end? It's reasonable to use, for example, a list at the end not marked `inverse="true"` (or explicitly set `false`) and a bag at the end that is marked `inverse="true"`.

You can use any of the mappings we've shown for unidirectional many-to-many associations for the noninverse end of the bidirectional association. `<set>`, `<idbag>`, `<list>`, and `<map>` are all possible, and the mappings are identical to those shown previously.

For the inverse end, `<set>` is acceptable, as is the following bag mapping:

```
<class name="Item" table="ITEM">
    ...
    <bag name="categories"
            table="CATEGORY_ITEM"
            lazy="true"
            inverse="true" cascade="save-update">
```

```
        <key column="ITEM_ID"/>
        <many-to-many class="Item" column="CATEGORY_ID"/>
    </bag>
</class>
```

This is the first time we've shown the `<bag>` declaration: It's similar to an `<idbag>` mapping, but it doesn't involve a surrogate key column. It lets you use a `List` (with bag semantics) in a persistent class instead of a `Set`. Thus it's preferred if the non-inverse side of a many-to-many association mapping is using a map, list, or bag (which all permit duplicates). Remember that a bag doesn't preserve the order of elements, despite the `List` type in the Java property definition.

No other mappings should be used for the inverse end of a many-to-many association. Indexed collections (lists and maps) can't be used, since Hibernate won't initialize or maintain the index column if `inverse="true"`. This is also true and important to remember for all other association mappings involving collections: an indexed collection (or even arrays) can't be set to `inverse="true"`.

We already frowned at the use of a many-to-many association and suggested the use of composite element mappings as an alternative. Let's see how this works.

Using a collection of components for a many-to-many association

Suppose we need to record some information each time we add an `Item` to a `Category`. For example, we might need to store the date and the name of the user who added the item to this category. We need a Java class to represent this information:

```
public class CategorizedItem {
    private String username;
    private Date dateAdded;
    private Item item;
    private Category category;
    ....
}
```

(We omitted the accessors and `equals()` and `hashCode()` methods, but they would be necessary for this component class.)

We map the `items` collection on `Category` as follows:

```
<set name="items" lazy="true" table="CATEGORY_ITEMS">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>
        <many-to-one name="item"
                     class="Item"
                     column="ITEM_ID"
                     not-null="true"/>
        <property name="username" column="USERNAME" not-null="true"/>
```

```
        <property name="dateAdded" column="DATE_ADDED" not-null="true"/>
    </composite-element>
</set>
```

We use the `<many-to-one>` element to declare the association to `Item`, and we use the `<property>` mappings to declare the extra association-related information. The link table now has four columns: `CATEGORY_ID`, `ITEM_ID`, `USERNAME`, and `DATE_ADDED`. The columns of the `CategorizedItem` properties should never be `null`: otherwise we can't identify a single link entry, because they're all part of the table's primary key. You can see the table structure in figure 6.11.

In fact, rather than mapping just the `username`, we might like to keep an actual reference to the `User` object. In this case, we have the following *ternary association* mapping:

```
<set name="items" lazy="true" table="CATEGORY_ITEMS">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>
        <many-to-one name="item"
                     class="Item"
                     column="ITEM_ID"
                     not-null="true"/>
        <many-to-one name="user"
                     class="User"
                     column="USER_ID"
                     not-null="true"/>
        <property name="dateAdded" column="DATE_ADDED" not-null="true"/>
    </composite-element>
</set>
```

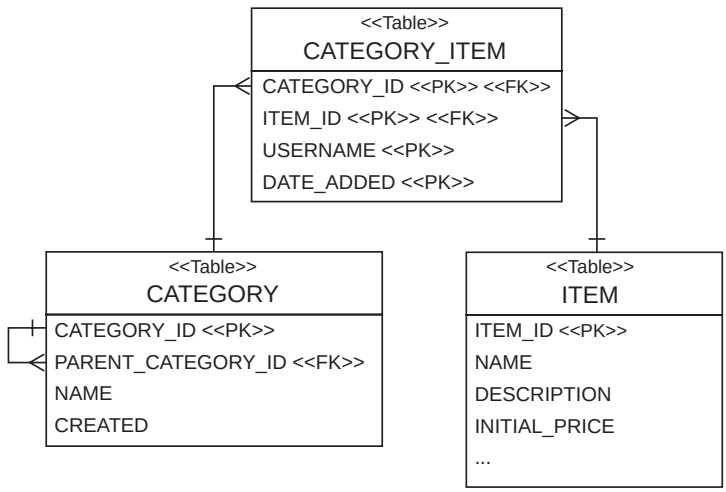

Figure 6.11 Many-to-many entity association table using a component

This is a fairly exotic beast! If you find yourself with a mapping like this, you should ask whether it might be better to map `CategorizedItem` as an entity class and use two one-to-many associations. Furthermore, there is no way to make this mapping bidirectional: a component (such as `CategorizedItem`) can't, by definition, have shared references. You can't navigate from `Item` to `CategorizedItem`.

We talked about some limitations of many-to-many mappings in the previous section. One of them, the restriction to nonindexed collections for the inverse end of an association, also applies to one-to-many associations, if they're bidirectional. Let's take a closer look at one-to-many and many-to-one again, to refresh your memory and elaborate on what we discussed in chapter 3.

One-to-many associations

You already know most of what you need to know about one-to-many associations from chapter 3. We mapped a typical parent/child relationship between two entity persistent classes, `Item` and `Bid`. This was a bidirectional association, using a `<one-to-many>` and a `<many-to-one>` mapping. The "many" end of this association was implemented in Java with a `Set`; we had a collection of `bids` in the `Item` class. Let's reconsider this mapping and walk through some special cases.

Using a bag with set semantics

For example, if you absolutely need a `List` of children in your parent Java class, it's possible to use a `<bag>` mapping in place of a set. In our example, first we have to replace the type of the `bids` collection in the `Item` persistent class with a `List`. The mapping for the association between `Item` and `Bid` is then left essentially unchanged:

```
<class
    name="Bid"
    table="BID">
    ...
    <many-to-one
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>

</class>

<class
    name="Item"
    table="ITEM">
    ...
    <bag
        name="bids"
```

```
        inverse="true"
        cascade="all-delete-orphan">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </bag>

</class>
```

We renamed the `<set>` element to `<bag>`, making no other changes. Note, however, that this change isn't useful: the underlying table structure doesn't support duplicates, so the `<bag>` mapping results in an association with set semantics. Some tastes prefer the use of `Lists` even for associations with set semantics, but ours doesn't, so we recommend using `<set>` mappings for typical parent/child relationships.

The obvious (and wrong) solution would be to use a real `<list>` mapping for the `bids` with an additional column holding the position of the elements. Remember the Hibernate limitation we introduced earlier in this chapter: you can't use indexed collections on an *inverse* side of an association. The `inverse="true"` side of the association isn't considered when Hibernate saves the object state, so Hibernate will ignore the index of the elements and not update the position column.

However, if your parent/child relationship will only be unidirectional (navigation is only possible from parent to child), you could even use an indexed collection type (because the "many" end would no longer be inverse). Good uses for unidirectional one-to-many associations are uncommon in practice, and we don't have one in our auction application. You may remember that we started with the `Item` and `Bid` mapping in chapter 3, making it first unidirectional, but we quickly introduced the other side of the mapping.

Let's find a different example to implement a unidirectional one-to-many association with an indexed collection.

Unidirectional mapping

For the sake of this section, we now suppose that the association between `Category` and `Item` is to be remodeled as a one-to-many association (an item now belongs to at most one category) and further that the `Item` doesn't own a reference to its current category. In Java code, we model this as a collection named `items` in the `Category` class; we don't have to change anything if we don't use an indexed collection. If `items` is implemented as a `Set`, we use the following mapping:

```
<set name="items" lazy="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="Item"/>
</set>
```

Remember that one-to-many association mappings don't need to declare a table name. Hibernate already knows that the column names in the collection mapping (in this case, only `CATEGORY_ID`) belong to the `ITEM` table. The table structure is shown in figure 6.12.

The other side of the association, the `Item` class, has no mapping reference to `Category`. We can now also use an indexed collection in the `Category`—for example, after we change the `items` property to `List`:

```
<list name="items" lazy="true">
    <key>
        <column name="CATEGORY_ID" not-null="false"/>
    </key>
    <index column="DISPLAY_POSITION/>
    <one-to-many class="Item"/>
</list>
```

Note the new `DISPLAY_POSITION` column in the `ITEM` table, which holds the position of the `Item` elements in the collection.

There is an important issue to consider, which, in our experience, puzzles many Hibernate users at first. In a unidirectional one-to-many association, the foreign key column `CATEGORY_ID` in the `ITEM` must be nullable. An `Item` could be saved without knowing anything about a `Category`—it's a stand-alone entity! This is a consistent model and mapping, and you might have to think about it twice if you deal with a `not-null` foreign key and a parent/child relationship. Using a bidirectional association (and a `Set`) is the correct solution.

Now that you know about all the association mapping techniques for normal entities, we still have to consider inheritance and associations to the various levels of an inheritance hierarchy. What we really want is *polymorphic* behavior. Let's see how Hibernate deals with polymorphic entity associations.

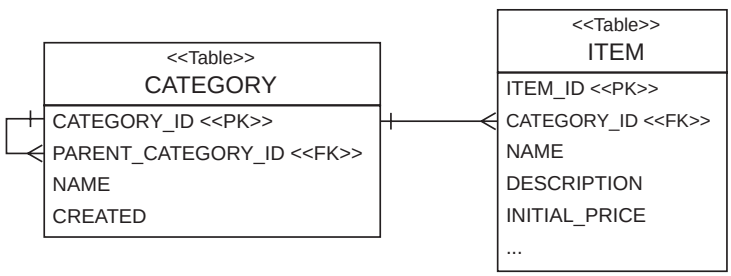

Figure 6.12
A standard one-to-many association using a foreign key column

6.4 Mapping polymorphic associations

Polymorphism is a defining feature of object-oriented languages like Java. Support for polymorphic associations and polymorphic queries is a basic feature of an ORM solution like Hibernate. Surprisingly, we've managed to get this far without needing to talk much about polymorphism. Even more surprisingly, there isn't much to say on the topic—polymorphism is so easy to use in Hibernate that we don't need to spend a lot of effort explaining this feature.

To get an overview, we'll first consider a many-to-one association to a class that might have subclasses. In this case, Hibernate guarantees that you can create links to any subclass instance just as you would to instances of the superclass.

6.4.1 Polymorphic many-to-one associations

A *polymorphic association* is an association that may refer to instances of a subclass of the class that was explicitly specified in the mapping metadata. For this example, imagine that we don't have many `BillingDetails` per `User`, but only one, as shown in figure 6.13.

We map this association to the abstract class `BillingDetails` as follows:

```
<many-to-one name="billingDetails"
    class="BillingDetails"
    column="BILLING_DETAILS_ID"
    cascade="save-update"/>
```

But since `BillingDetails` is abstract, the association must refer to an instance of one of its subclasses—`CreditCard` or `BankAccount`—at runtime.

All the association mappings we've introduced so far in this chapter support polymorphism. You don't have to do anything special to use polymorphic associations in Hibernate; specify the name of any mapped persistent class in your

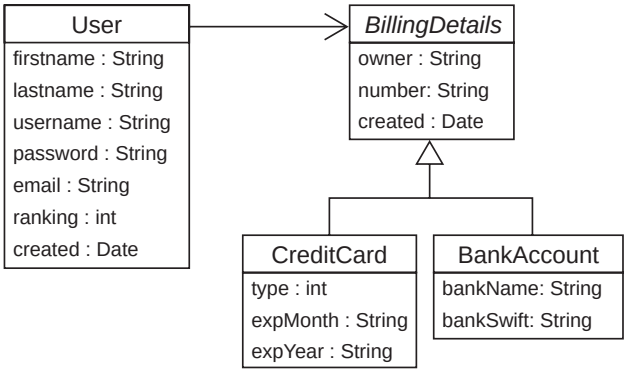

Figure 6.13
The user has only one billing information object.

association mapping (or let Hibernate discover it using reflection); then, if that class declares any `<subclass>` or `<joined-subclass>` elements, the association is naturally polymorphic.

The following code demonstrates the creation of an association to an instance of the `CreditCard` subclass:

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
user.setBillingDetails(cc);

tx.commit();
session.close();
```

Now, when we navigate the association in a second transaction, Hibernate automatically retrieves the `CreditCard` instance:

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
// Invoke the pay() method on the actual subclass
user.getBillingDetails().pay(paymentAmount);

tx.commit();
session.close();
```

There is one thing to watch out for: if `BillingDetails` was mapped with `lazy="true"`, Hibernate would proxy the `billingDetails` association. In this case, we wouldn't be able to perform a typecast to the concrete class `CreditCard` at runtime, and even the `instanceof` operator would behave strangely:

```
User user = (User) session.get(User.class, uid);
BillingDetails bd = user.getBillingDetails();
System.out.println( bd instanceof CreditCard ); // prints "false"
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

In this code, the typecast fails because `bd` is a proxy instance. When a method is invoked on the proxy, the call is delegated to an instance of `CreditCard` that is fetched lazily. To perform a proxysafe typecast, use `Session.load()`:

```
User user = (User) session.get(User.class, uid);
BillingDetails bd = user.getBillingDetails();
// Get a proxy of the subclass, doesn't hit the database
CreditCard cc =
```

```
    (CreditCard) session.load( CreditCard.class, bd.getId() );
expiryDate = cc.getExpiryDate();
```

After the call to load, `bd` and `cc` refer to two different proxy instances, which both delegate to the same underlying `CreditCard` instance.

Note that you can avoid these issues by avoiding lazy fetching, as in the following code, using a query technique discussed in the next chapter:

```
User user = (User) session.createCriteria(User.class)
    .add( Expression.eq("id", uid) )
    .setFetchMode("billingDetails", FetchMode.EAGER)
    .uniqueResult();
// The user's billingDetails were fetched eagerly
CreditCard cc = (CreditCard) user.getBillingDetails();
expiryDate = cc.getExpiryDate();
```

Truly object-oriented code shouldn't use `instanceof` or numerous typecasts. If you find yourself running into problems with proxies, you should question your design, asking whether there is a more polymorphic approach.

One-to-one associations are handled the same way. What about many-valued associations?

6.4.2 Polymorphic collections

Let's refactor the previous example to its original form in CaveatEmptor. If `User` owns many `BillingDetails`, we use a bidirectional one-to-many. In `BillingDetails`, we have the following:

```
<many-to-one name="user"
    class="User"
    column="USER_ID"/>
```

In the `User`s mapping, we have this:

```
<set name="billingDetails"
        lazy="true"
        cascade="save-update"
        inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

Adding a `CreditCard` is easy:

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
```

```
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
// Call convenience method that sets both "ends"
user.addBillingDetails(cc);

tx.commit();
session.close();
```

As usual, `addBillingDetails()` calls `getBillingDetails().add(cc)` and `cc.setUser(this)`.

We can iterate over the collection and handle instances of `CreditCard` and `BankAccount` polymorphically (we don't want to bill users multiple times in our final system, though):

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
Iterator iter = user.getBillingDetails().iterator();
while ( iter.hasNext() ) {
    BillingDetails bd = (BillingDetails) iter.next();
    // Invoke CreditCard.pay() or BankAccount.pay()
    bd.pay(ccPaymentAmount);
}

tx.commit();
session.close();
```

In the examples so far, we've assumed that `BillingDetails` is a class mapped explicitly in the Hibernate mapping document, and that the inheritance mapping strategy is table-per-hierarchy or table-per-subclass. We haven't yet considered the case of a table-per-concrete-class mapping strategy, where `BillingDetails` wouldn't be mentioned explicitly in the mapping file (only in the Java definition of the subclasses).

6.4.3 Polymorphic associations and table-per-concrete-class

In section 3.6.1, "Table per concrete class," we defined the *table-per-concrete-class* mapping strategy and observed that this mapping strategy makes it difficult to represent a polymorphic association, because you can't map a foreign key relationship to the table of the abstract superclass. There is no table for the superclass with this strategy; you only have tables for concrete classes.

Suppose that we want to represent a polymorphic many-to-one association from `User` to `BillingDetails`, where the `BillingDetails` class hierarchy is mapped using this table-per-concrete-class strategy. There is a `CREDIT_CARD` table and a

BANK_ACCOUNT table, but no BILLING_DETAILS table. We need two pieces of information in the USER table to uniquely identify the associated CreditCard or BankAccount:

- The name of the table in which the associated instance resides
- The identifier of the associated instance

The USER table requires the addition of a BILLING_DETAILS_TYPE column, in addition to the BILLING_DETAILS_ID. We use a Hibernate <any> element to map this association:

```
<any name="billingDetails"
        meta-type="string"
        id-type="long"
        cascade="save-update">
    <meta-value value="CREDIT_CARD" class="CreditCard"/>
    <meta-value value="BANK_ACCOUNT"class="BankAccount"/>
    <column name="BILLING_DETAILS_TYPE"/>
    <column name="BILLING_DETAILS_ID"/>
</any>
```

The meta-type attribute specifies the Hibernate type of the BILLING_DETAILS_TYPE column; the id-type attribute specifies the type of the BILLING_DETAILS_ID column (CreditCard and BankAccount must have the same identifier type). Note that the order of the columns is important: first the type, then the identifier.

The <meta-value> elements tell Hibernate how to interpret the value of the BILLING_DETAILS_TYPE column. We don't need to use the full table name here—we can use any value we like as a type discriminator. For example, we can encode the information in two characters:

```
<any name="billingDetails"
        meta-type="string"
        id-type="long"
        cascade="save-update">
    <meta-value value="CC" class="CreditCard"/>
    <meta-value value="CA" class="BankAccount"/>
    <column name="BILLING_DETAILS_TYPE"/>
    <column name="BILLING_DETAILS_ID"/>
</any>
```

An example of this table structure is shown in figure 6.14.

Here is the first major problem with this kind of association: we can't add a foreign key constraint to the BILLING_DETAILS_ID column, since some values refer to the BANK_ACCOUNT table and others to the CREDIT_CARD table. Thus, we need to come up with some other way to ensure integrity (a trigger, for example).

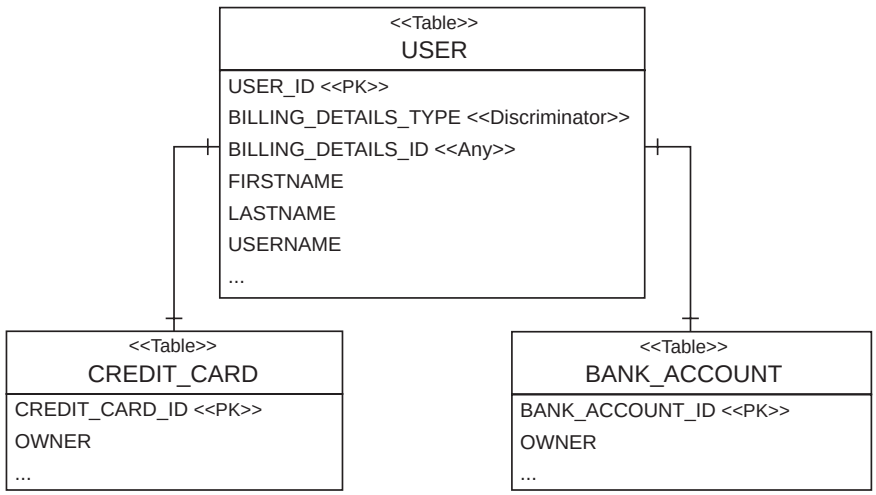

Figure 6.14 Using a discriminator column with an *any* association

Furthermore, it's difficult to write SQL table joins for this association. In particular, the Hibernate query facilities don't support this kind of association mapping, nor may this association be fetched using an outer join. We discourage the use of `<any>` associations for all but the most special cases.

As you can see, polymorphism is messier in the case of a table-per-concrete-class inheritance mapping strategy. We don't usually use this mapping strategy when polymorphic associations are required. As long as you stick to the other inheritance-mapping strategies, polymorphism is straightforward, and you don't usually need to think about it.

6.5 Summary

This chapter covered the finer points of ORM and techniques needed to solve the structural mismatch problem. We can now fully map all the entities and associations in the CaveatEmptor domain model.

The Hibernate type system distinguishes *entities* from *value types*. An entity instance has its own lifecycle and persistent identity; an instance of a value type is completely dependant on an owning entity.

Hibernate defines a rich variety of built-in value mapping types. When the predefined types are insufficient, you can easily extend them using custom types or

component mappings and even implement arbitrary conversions from Java to SQL data types.

Collection-valued properties are considered to be of value type. A collection doesn't have its own persistent identity and belongs to a single owning entity. You've seen how to map collections, including collections of value-typed instances and many-valued entity associations.

Hibernate supports one-to-one, one-to-many, and many-to-many associations between entities. In practice, we recommend against the overuse of many-to-many associations. Associations in Hibernate are naturally polymorphic. We also talked about bidirectional behavior of such relationships.

JAVA

HIBERNATE IN ACTION

Christian Bauer and Gavin King

"The Bible of Hibernate"

—Ara Abrahamian, XDoclet Lead Developer

Hibernate practically exploded on the Java scene. Why is this open-source tool so popular? Because it automates a tedious task: persisting your Java objects to a relational database. The inevitable mismatch between your object-oriented code and the relational database requires you to write code that maps one to the other. This code is often complex, tedious and costly to develop. Hibernate does the mapping for you.

Not only that, Hibernate makes it easy. Positioned as a layer between your application and your database, Hibernate takes care of loading and saving of objects. Hibernate applications are cheaper, more portable, and more resilient to change. And they perform better than anything you are likely to develop yourself.

Hibernate in Action carefully explains the concepts you need, then gets you going. It builds on a single example to show you how to use Hibernate in practice, how to deal with concurrency and transactions, how to efficiently retrieve objects and use caching.

The authors created Hibernate and they field questions from the Hibernate community every day—they know how to make Hibernate sing. Knowledge and insight seep out of every pore of this book.

What's Inside

- ORM concepts
- Getting started
- Many real-world tasks
- The Hibernate application development process

Ask the Authors

Ebook edition

www.manning.com/bauer

A member of the core Hibernate developer team, **Christian Bauer** maintains the Hibernate documentation and website. He is a senior software engineer in Frankfurt, Germany. **Gavin King** is the Hibernate founder and principal developer. He is a J2EE consultant based in Melbourne, Australia.

$44.95 US/$62.95 Canada

ISBN 1-932394-15-X