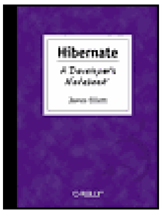
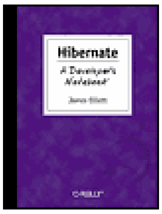


Hibernate: A Developer's Notebook

By James Elliott

Publisher: O'Reilly
Pub Date: May 2004
ISBN: 0-596-00696-9
Pages: 190

Hibernate: A Developer's Notebook shows you how to use Hibernate to automate persistence: you write natural Java objects and some simple configuration files, and Hibernate automates all the interaction between your objects and the database. You don't even need to know the database is there, and you can change from one database to another simply by changing a few statements in a configuration file. If you've needed to add a database backend to your application, don't put it off. It's much more fun than it used to be, and Hibernate: A Developer's Notebook shows you why.



Hibernate: A Developer's Notebook

By James Elliott

Publisher: O'Reilly
Pub Date: May 2004
ISBN: 0-596-00696-9
Pages: 190

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Preface

Hibernate is a lightweight object/relational mapping service for Java. What does that mean? It's a way to easily and efficiently work with information from a relational database in the form of natural Java objects. But that description doesn't come close to conveying how useful and exciting the technology is. I'm not the only person who thinks so: Hibernate 2.1 just won Software Development magazine's 14th annual Jolt Award in the 'Libraries, Frameworks, and Components' category.

So, what's great about Hibernate? Every nontrivial application (and even many trivial ones) needs to store and use information, and these days this usually involves a relational database. Databases are a very different world than Java objects, and they often involve people with different skills and specializations. Bridging between the two worlds has been important for a while, but it used to be quite complex and tedious.

Most people start out struggling to write a few SQL queries, embedding these awkwardly as strings within Java code, and working with JDBC to run them and process the results. JDBC has evolved into a rich and flexible database communication library, which now provides ways to simplify and improve on this approach, but there is still a fair degree of tedium involved. People who work with data a great deal need more power, some way of moving the queries out of the code, and making them act more like wellbehaved components in an object-oriented world.

Such capabilities have been part of my own (even more) lightweight object/relational layer for years. It began with a Java database connection and query pooling system written by my colleague Eric Knapp for the Lands' End e-commerce site. Our pooler introduced the idea of external SQL templates that could be accessed by name and efficiently combined with runtime data to generate the actual database queries. Only later did it grow to include the ability to bind these templates directly to Java objects, by adding simple mapping directives to the templates.

Although far less powerful than a system like Hibernate, this approach proved valuable in many projects of different sizes and in widely differing environments. I've continued to use it to this day, most recently in building IP telephony applications for Cisco's CallManager platform. But I'm going to be using Hibernate instead from now on. Once you work through this book, you'll understand why, and will probably make the same decision yourself. Hibernate does a tremendous amount for you, and does it so easily that you can almost forget you're working with a database. Your objects are simply there when you need them. This is how technology should work.

You may wonder how Hibernate relates to Enterprise JavaBeans TM . Is it a competing solution? When would you use one over the other? In fact, you can use both. Most applications have no need for the complexity of EJBs, and they can simply use Hibernate directly to interact with a database. On the other hand, EJBs are indispensable for very complex threetier application environments. In such cases, Hibernate may be used by an EJB Session bean to persist data, or it might be used to persist BMP entity beans.

How to Use This Book

The Developer's Notebook TM series is a new approach to helping readers rapidly come up to speed with useful new technologies. This book is not intended to be a comprehensive reference manual for Hibernate. Instead, it reflects my own exploration of the system, from initial download and configuration through a series of projects that demonstrate how to accomplish a variety of practical goals.

By reading and following along with these examples, you'll be able to get your own Hibernate environment set up quickly and start using it for realistic tasks right away. It's as if you can 'walk with me' through terrain I've mapped out, while I point out useful landmarks and tricky pitfalls along the way.

Although I certainly include some background materials and explanations of how Hibernate works and why, this is always in the service of a focused task. Sometimes I'll refer you to the reference documentation or other online resources if you'd like more depth about one of the underlying concepts or details about a related but different way to use Hibernate.

Once you're past the first few chapters, you don't need to read the rest in order; you can jump to topics that are particularly interesting or relevant to you. The examples do build on each other, but you can download the finished source code from the book's web site (you may want to start with the previous chapter's files and follow along making changes yourself to implement the examples you're reading). You can always jump back to the earlier examples if they turn out to be interesting because of how they relate to what you've just learned.

Font Conventions

This book follows certain conventions for font usage. Understanding these conventions up-front makes it easier to use this book.

Italic

Used for filenames, file extensions, URLs, application names, emphasis, and new terms when they are first introduced.

Constant width

Used for Java class names, methods, variables, properties, data types, database elements, and snippets of code that appear in text.

Constant width bold

Used for commands you enter at the command line and to highlight new code inserted in a running example.

Constant width italic

Used to annotate output.

On the Web Site

The web site for this book, www.oreilly.com/catalog/hibernate, offers some important materials you'll want to know about. All the examples for this book can be found there, organized by chapter.

The examples are available as a ZIP archive and a compressed TAR archive.

In many cases, the same files are used throughout a series of chapters, and they evolve to include new features and capabilities from example to example. Each chapter folder in the downloadable archive contains a snapshot of the state of the example system, reflecting all the changes and new content introduced in that chapter.

How to Contact Us

Please address comments and questions concerning this book to the publisher:


```
O'Reilly & Associates, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (in the United States or Canada)

(707) 829-0515 (international or local)

(707) 829-0104 (fax)
```


O'Reilly's web page for this book, where we list errata, examples, or any additional information. You can access this page at:

www.oreilly.com/catalog/hibernate/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

www.oreilly.com/

Acknowledgments

Any list of thanks has to start with my parents for fostering my interest in computing even when we were living in countries that made that a major challenge, and with my partner Joe for putting up with it today when it has flowered into a major obsession. I'd also like to acknowledge my employer, Berbee, for giving me an opportunity to delve deeply into Java and build skills as an architect of reusable APIs; for letting me stay clear of the proprietary, platform-specific tar pit that is engulfing so much of the programming world; for surrounding me with such incredible colleagues; and for being supportive when I wanted to leverage these experiences in writing this book.

Marc Loy got me connected with the wonderful folks at O'Reilly by inviting me to help with the second edition of Java Swing, and Mike Loukides has been patiently working on me ever since—encouraging me to write a book of my own. In Hibernate he found the perfect topic to get me started. Deb Cameron, our revisions editor for the Swing effort, has played a big role in turning my tentative authorial ambitions into a rewarding reality. I'm also grateful she was willing to 'loan me out' from helping with the third edition of Learning Emacs to take on the Hibernate project.

I'm particularly indebted to my technical reviewers: Adrian Kellor and Curt Pederson. They looked at some very early drafts and helped set my tone and direction, as well as reinforcing my enthusiasm about the value of this project. As the book came together, Bruce Tate provided an important sanity check from someone actively using and teaching Hibernate, and he offered some great advice and even more encouragement. Eric Knapp reviewed a large portion with an eye toward using the book in an instructional setting at a technical college, and reminded me to keep my feet on the ground. Tim Cartwright jumped in at the end, working with a nearly complete draft in an effort to understand Hibernate as a potential platform for future work, and providing a great deal of useful feedback about the content and presentation.

I'd also like to thank the many members of O'Reilly's production department who put in lots of work under an unusually tight schedule.

Chapter 1. Installation and Setup

NOTE

In this chapter:

It continues to amaze me how many great, free, open source Java™ tools are out there. When I needed a lightweight object/relational mapping service for a JSP e-commerce project several years ago, I had to build my own. It evolved over the years, developed some cool and unique features, and we've used it in a wide variety of different contexts. But now that I've discovered Hibernate, I expect that I'll be using it on my next project instead of my own familiar system (toward which I'll cheerfully admit bias). That should tell you how compelling it is!

Since you're looking at this book, you're interested in a powerful and convenient way to bridge between the worlds of Java objects and relational databases. Hibernate fills that role very nicely, without being so big and complicated that learning it becomes a daunting challenge in itself. To demonstrate that, this chapter guides you to the point where you can play with Hibernate and see for yourself why it's so exciting.

1.1 Getting an Ant Distribution

If you're not already using Ant to manage the building, testing, running, and packaging of your Java projects, now is the time to start. The examples in this book are Ant driven, so you'll need a working Ant installation to follow along and experiment with variations on your own system, which is the best way to learn.

First of all, get an Ant binary and install it.

1.1.1 Why do I care?

We chose to structure our examples around Ant for several reasons. It's convenient and powerful, it's increasingly (almost universally) the standard build tool for Java-based development, it's free and it's crossplatform. Many of the example and helper scripts in the current Hibernate distribution are Windows batch files, which don't do any good for people like me who live in a Unix world. Using Ant means our examples can work equally well anywhere there's a Java environment, which means we don't have to frustrate or annoy any readers of this book. Happily, it also means we can do many more cool things with less effort—especially since several Hibernate tools have explicit Ant support, which we'll show how to leverage.

To take advantage of all this, you need to have Ant installed and working on your system.

NOTE

I used to wonder why people bothered with Ant when they could use Make. Now that I've seen how well it manages Java builds, I feel lost without it.

1.1.2 How do I do that?

Download a binary release of Ant from ant.apache.org/bindownload.cgi . Scroll down to find the current release of Ant, and download the archive in a format that's convenient for you to work with. Pick an appropriate place for it to live, and expand the archive there. The directory into which you've expanded the archive is referred to as ANT_HOME. Let's say you've expanded the archive into the directory /usr/ local/apache-ant-1.5.1; you may want to create a symbolic link to make it easier to work with, and to avoid the need to change any environment configuration when you upgrade to a newer version:

```
/usr/local $ ln -s apache-ant-1.5.1 ant
```

Once Ant is situated, you need to do a couple of things to make it work right. You need to add its bin directory in the distribution (in our example, /usr/local/ant/bin) to your command path. You also need to set the environment variable ANT_HOME to the top-level directory you installed (in this example, /usr/local/ant). Details about how to perform these steps under different operating systems can be found in the Ant manual, ant.apache.org/manual/ , if you need them.

Of course, we're also assuming you've got a Java SDK. Because some of Hibernate's features are available only in Java 1.4, you'd be best off upgrading to the latest 1.4 SDK. It's also possible to use most of Hibernate with Java 1.3, but you may have to rebuild the Hibernate JAR file using your 1.3 compiler. Our examples are written assuming you've got Java 1.4, and they will need tweaking if you don't.

Once you've got this set up, you should be able to fire up Ant for a test run and verify that everything's right:

```
~ $ ant -version
Apache Ant version 1.5.1 compiled on February 7 2003
```

1.1.3 What just happened?

1.2 Getting the HSQLDB Database Engine

Hibernate works with a great many relational databases; chances are, it will work with the one you are planning to use for your next project. We need to pick one to focus on in our examples, and luckily there's an obvious choice. The free, open source, 100% Java HSQLDB project is powerful enough that it forms the backing storage for several of our commercial software projects. Surprisingly, it's also incredibly self-contained and simple to install, so it's perfect to discuss here. (If you've heard of HypersonicSQL, this is its current incarnation. Much of the Hibernate documentation uses the older name.)

> Don't panic if you end up at hsql.sourceforge.net/ and it seems like the project has been shut down. That's the wrong address—it's talking about the predecessor to the current HSQLDB project. Use the address below to find the current version of the database engine.

1.2.1 Why do I care?

Examples based on a database that everyone can download and easily experiment with mean you won't have to translate any of the SQL dialects or operating system commands to work with your available databases (and may even mean you can save a day or two learning how to download, install, and configure one of the more typical database environments). Finally, if hsqldb is new to you, chances are good you'll be impressed and intrigued, and may well end up using it in your own projects. As it says on the project home page (at hsqldb.sourceforge.net):

hsqldb is a relational database engine written in Java, with a JDBC driver, supporting a rich subset of ANSI-92 SQL (BNF tree format). It offers a small (less than 160k), fast database engine which offers both in memory and disk based tables. Embedded and server modes are available. Additionally, it includes tools such as a minimal web server, in-memory query and management tools (can be run as applets), and a number of demonstration examples.

NOTE

Go on, download HSQLDB. Heck, take two, they're small!

1.2.2 How do I do that?

Getting the database is simply a matter of visiting the project page at hsqldb.sourceforge.net and clicking the link to download the current stable version. This will take you to a typical SourceForge downloads page with the current release highlighted. Pick your mirror and download the zip archive. There's nothing to install or configure; we'll show you how to use it shortly.

1.2.3 What about...

...MySQL, PostgreSQL, Oracle, DB2, Sybase, Informix, or some other common database? Don't worry, Hibernate can work with all these and others. We'll talk about how you specify 'dialects' for different databases later on. And if you really want, you can try to figure out how to work with your favorite from the start, but it will mean extra work for you in following along with the examples, and you'll miss out on a great opportunity to discover HSQLDB.

1.3 Getting Hibernate

This doesn't need much motivation! You picked up this book because you wanted to learn how to use Hibernate.

1.3.1 How do I do that?

Go to the Hibernate home page, www.hibernate.org , and click on the 'Download' link. The Binary Releases section will tell you which version is recommended for downloading; follow that advice. Make a note of the version you want and proceed to the 'Download: SourceForge' link. It takes you to a SourceForge downloads page. Scroll down until you find the recommended release version of Hibernate itself (which will look something like hibernate-2.x.y.zip or hibernate-2.x.y.tar.gz). Choose the archive format that is most convenient for you and download it.

Pick a place that is suitable for keeping such items around, and expand the archive. We will use part of it in the next step, and investigate more of it later on. You may also want to poke around in there some yourself.

While you're on the Hibernate downloads page, also pick up the Hibernate Extensions. They contain several useful tools which aren't necessary for an application running Hibernate, but are very helpful for developers creating such applications. We'll be using one to generate Java code for our first Hibernate experiment in the next chapter. This filename will look like hibernate-extensions-2.x.y.zip (it won't necessarily have the same version as Hibernate itself). Once again, pick your favorite archive format, download this file, and expand it next to where you put Hibernate.

1.4 Setting Up a Project Hierarchy

Although we're going to start small, once we start designing data structures and building Java classes and database tables that represent them, along with all the configuration and control files to glue them together and make useful things happen, we're going to end up with a lot of files. So let's start out with a good organization from the beginning. As you'll see in this process, between the tools you've downloaded and their supporting libraries, there are already a significant number of files to organize.

1.4.1 Why do I care?

If you end up building something cool by following the examples in this book, and want to turn it into a real application, you'll be in good shape from the beginning. More to the point, if you set things up the way we describe here, the commands and instructions we give you throughout the examples will make sense and actually work. Many examples also build on one another throughout the book, so it's important to get on the right track from the beginning.

If you want to skip ahead to a later example, or just avoid typing some of the longer sample code and configuration files, you can download 'finished' versions of the chapter examples from the book's web site. These downloads will all be organized as described here.

1.4.2 How do I do that?

Here's how:

1. Pick a location on your hard drive where you want to play with Hibernate, and create a new folder, which we'll refer to from now on as your project directory.

2. Move into that directory, and create subdirectories called src, lib, and data. The hierarchy of Java source and related resources will be in the src directory. Our build process will compile it into a classes directory it creates, as well as copy any runtime resources there. The data directory is where we'll put the HSQLDB database, and any Data Definition Language (DDL) files we generate in order to populate it.

   The lib directory is where we'll place third-party libraries we use in the project. For now, copy the HSQLDB and Hibernate JAR files into the lib directory.

3. If you haven't already done so, expand the HSQLDB distribution file you downloaded earlier in this chapter. You'll find hsqldb.jar in its lib directory; copy this to your own project lib directory (the lib directory you just created in step 2).

4. Similarly, locate the lib directory in the Hibernate directory you expanded in the previous section, and copy all of its contents into your own project lib directory (you'll notice that Hibernate relies on a lot of other libraries; conveniently, they're included in its binary distribution so you don't have to hunt them all down yourself).

5. Then copy Hibernate itself, in the form of the hibernate2.jar file found at the top level of the distribution, into your project lib directory.

6. Installing the Hibernate Extensions is very similar. Locate the tools/lib directory inside the Hibernate Extensions directory you expanded, and copy its contents into your own lib directory, so the extensions will be able to access the libraries they rely on.

7.

Chapter 2. Introduction to Mapping

NOTE

In this chapter:

Now that we're in a position to work with Hibernate, it's worth pausing to reflect on why we wanted to in the first place, lest we remain lost in the details of installation and configuration. Object-oriented languages like Java provide a powerful and convenient abstraction for working with information at runtime in the form of objects that instantiate classes. These objects can link up with each other in a myriad of ways, and they can embody rules and behavior as well as the raw data they represent. But when the program ends, all the objects swiftly and silently vanish.

For information we need to keep around between runs, or share between different programs and systems, relational databases have proven to be hard to beat. They're scalable, reliable, efficient, and extremely flexible. So what we need is a means of taking information from a SQL database and turning it into Java objects, and vice versa.

There are many different ways of doing this, ranging from completely manual database design and coding, to highly automated tools. The general problem is known as Object/Relational Mapping, and Hibernate is a lightweight O/R mapping service for Java.

The 'lightweight' designation means it is designed to be fairly simple to learn and use, and to place reasonable demands on system resources, compared to some of the other available tools. Despite this, it manages to be broadly useful and deep. The designers have done a good job of figuring out the kinds of things that real projects need to accomplish, and supporting them well.

You can use Hibernate in many different ways, depending on what you're starting with. If you've got a database that you need to interact with, there are tools that can analyze the existing schema as a starting point for your mapping, and help you write the Java classes to represent the data. If you've got classes that you want to store in a new database, you can start with the classes, get help building a mapping document, and generate an initial database schema. We'll look at some of these approaches later.

For now, we're going to see how you can start a brand new project, with no existing classes or data, and have Hibernate help you build both. When starting from scratch like this, the most convenient place to begin is in the middle, with an abstract definition of the mapping we're going to make between program objects and the database tables that will store them.

In our examples we're going to be working with a database that could power an interface to a large personal collection of music, allowing users to search, browse, and listen in a natural way. (You might well have guessed this from the names of the database files that were created at the end of the first chapter.)

2.1 Writing a Mapping Document

Hibernate uses an XML document to track the mapping between Java classes and relational database tables. This mapping document is designed to be readable and hand-editable. You can also start by using graphical CASE tools (like Together, Rose, or Poseidon) to build UML diagrams representing your data model, and feed these into AndroMDA (www.andromda.org/), turning them into Hibernate mappings.

NOTE

Don't forget that Hibernate and its extensions let you work in other ways, starting with classes or data if you've got them.

We'll write one by hand, showing it's quite practical.

We're going to start by writing a mapping document for tracks, pieces of music that can be listened to individually or as part of an album or play list. To begin with, we'll keep track of the track's title, the path to the file containing the actual music, its playing time, the date on which it was added to the database, and the volume at which it should be played (in case the default volume isn't appropriate because it was recorded at a very different level than other music in the database).

2.1.1 Why do I care?

You might not have any need for a new system to keep track of your music, but the concepts and process involved in setting up this mapping will translate to the projects you actually want to tackle.

2.1.2 How do I do that?

Fire up your favorite text editor, and create the file Track.hbm.xml in the src/com/oreilly/hh directory you set up in the previous Chapter. (If you skipped that chapter, you'll need to go back and follow it, because this example relies on the project structure and tools we set up there.) Type in the mapping document as shown in Example 2-1. Or, if you'd rather avoid all that typing, download the code examples from this book's web site, and find the mapping file in the directory for Chapter 2.

Example 2-1. The mapping document for tracks, Track.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
           PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
           "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>

 <class name="com.oreilly.hh.Track" table="TRACK">
    <meta attribute="class-description">
      Represents a single playable track in the music database.
      @author Jim Elliott (with help from Hibernate)
    </meta>
```

2.2 Generating Some Class

Our mapping contains information about both the database and the Java class between which it maps. We can use it to help us create both. Let's look at the class first.

2.2.1 How do I do that?

The Hibernate Extensions you installed in Chapter 1 included a tool that can write Java source matching the specifications in a mapping document, and an Ant task that makes it easy to invoke from within an Ant build file. Edit build.xml to add the portions shown in bold in Example 2-2.

Example 2-2. The Ant build file updated for code generation

```
<project name="Harnessing Hibernate: The Developer's Notebook"
        default="db" basedir=".">
<!-- Set up properties containing important project directories -->
<property name="source.root" value="src"/>
<property name="class.root" value="classes"/>
<property name="lib.dir" value="lib"/>
<property name="data.dir" value="data"/>

<!-- Set up the class path for compilation and execution -->
<path id="project.class.path">
    <!-- Include our own classes, of course -->
    <pathelement location="${class.root}" />
    <!-- Include jars in the project library directory -->
    <fileset dir="${lib.dir}">
      <include name="*.jar"/>
    </fileset>
</path>

<target name="db" description="Runs HSQLDB database management UI
against the database file--use when application is not running">
    <java classname="org.hsqldb.util.DatabaseManager"
          fork="yes">
       <classpath refid="project.class.path"/>
       <arg value="-driver"/>
       <arg value="org.hsqldb.jdbcDriver"/>
```

2.3 Cooking Up a Schema

That was pretty easy, wasn't it? You'll be happy to learn that creating database tables is a very similar process. As with code generation, you've already done most of the work in coming up with the mapping document. All that's left is to set up and run the schema generation tool.

2.3.1 How do I do that?

The first step is something we alluded to in Chapter 1. We need to tell Hibernate the database we're going to be using, so it knows the specific 'dialect' of SQL to use. SQL is a standard, yes, but every database goes beyond it in certain directions and has a specific set of features and limitations that affect real-life applications. To cope with this reality, Hibernate provides a set of classes that encapsulate the unique features of common database environments, in the package net.sf.hibernate.dialect. You just need to tell it which one you want to use. (And if you want to work with a database that isn't yet supported 'out of the box,' you can implement your own dialect.)

In our case, we're working with HSQLDB, so we want to use HSQLDialect. The easiest way to configure Hibernate is to create a properties file named hibernate.properties and put it at the root level somewhere in the class path. Create this file at the top level of your src directory, and put the lines shown in Example 2-4 into it.

NOTE

You can use an XML format for the configuration information as well, but for the simple needs we have here, it doesn't buy you anything.

Example 2-4. Setting up hibernate.properties

```
hibernate.dialect=net.sf.hibernate.dialect.HSQLDialect

hibernate.connection.driver_class=org.hsqldb.jdbcDriver

hibernate.connection.url=jdbc:hsqldb:data/music

hibernate.connection.username=sa

hibernate.connection.password=
```

In addition to establishing the SQL dialect we are using, this tells Hibernate how to establish a connection to the database using the JDBC driver that ships as part of the HSQLDB database JAR archive, and that the data should live in the data directory we've created—in the database named music. The username and empty password (indeed, all these values) should be familiar from the experiment we ran at the end of Chapter 1.

> Notice that we're using a relative path to specify the database filename. This works fine in our examples—we're using ant to control the working directory. If you copy this for use in a web application or other environment, though, you'll likely need to be more explicit about the location of the file.

You can put the properties file in other places, and give it other names, or use entirely different ways of getting the properties into Hibernate, but this is the default place it will look, so it's the path of least resistance (or, I guess, least runtime configuration).

We also need to add some new pieces to our build file, shown in Example 2-5. This is a somewhat substantial addition, because we need to compile our Java source in order to use the schema generation tool, which relies on reflection to get its details right. Add these targets right before the closing </project> tag at the end of build.xml.

2.4 Connecting Hibernate to MySQL

If you were skimming through this chapter (or, more likely, the table of contents) you may not have even noticed that Hibernate connected to and manipulated a database in the previous section, 'Cooking Up a Schema.' Since working with databases is the whole point of Hibernate, it makes this as easy as possible. Once you've set up a configuration file like the one in Example 2-4, the schema generation tool can get in and work with your database, and your Java code can use it for persistence sessions as demonstrated in Chapter 3.

NOTE

This example assumes you've already got a working MySQL instance installed and running, since explaining how to do that would be quite a detour.

In the interest of further clarifying this aspect of working with Hibernate, let's take a look at what we'd change in that example to set up a connection with the popular, free, and open source MySQL database (available from www.mysql.com).

2.4.1 How do I do that?

Connect to your MySQL server and set up a new database to play with, along the lines of Example 2-8.

Example 2-8. Setting up the MySQL database notebook_db as a Hibernate playground

```
% mysql -u root -p

Enter password:

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 764 to server version: 3.23.44-Max-log


Type 'help;' or '\h' for help. Type '\c' to clear the buffer.


mysql> CREATE DATABASE notebook_db;

Query OK, 1 row affected (0.00 sec)


mysql> GRANT ALL ON notebook_db.* TO jim IDENTIFIED BY "s3cret";

Query OK, 0 rows affected (0.20 sec)


mysql> quit;

Bye
```

NOTE

Hopefully you'll use a less guessable password than this in your real databases!

Make a note of the database name you create, as well as the username and password that can access to it. These will need to be entered into hibernate.properties, as shown in Example 2-9.

Chapter 3. Harnessing Hibernate

NOTE

In this chapter:

All right, we've set up a whole bunch of infrastructure, defined an object/ relational mapping, and used it to create a matching Java class and database table. But what does that buy us? It's time to see how easy it is to work with persistent data from your Java code.

3.1 Creating Persistent Objects

Let's start by creating some objects in Java and persisting them to the database, so we can see how they turn into rows and columns for us. Because of the way we've organized our mapping document and properties file, it's extremely easy to configure the Hibernate session factory and get things rolling.

To get started, set up the Hibernate environment and use it to turn some new Track instances into corresponding rows in the database table.

3.1.1 How do I do that?

This discussion assumes you've created the schema and generated Java code by following the examples in Chapter 2. If you haven't, you can start by downloading the examples archive from this book's web site, jumping in to the ch03 directory, and copying in the third-party libraries as instructed in Chapter 1. Once you've done that, use the commands ant codegen followed by ant schema to set up the generated Java code and database schema on which this example is based. As with the other examples, these commands should be issued in a shell/command window whose current working directory is the top of your project tree, containing Ant's build.xml file.

NOTE

The examples in most chapters build on the previous ones, so if you are skipping around, you'll really want to download the sample code.

We'll start with a simple example class, CreateTest, containing the necessary imports and housekeeping code to bring up the Hibernate environment and create some Track instances that can be persisted using the XML mapping document we started with. The source is shown in Example 3-1.

Example 3-1. CreateTest.java

```
package com.oreilly.hh;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

import java.sql.Time;
import java.util.Date;

/**
 * Create sample data, letting Hibernate persist it for us.
 */
public class CreateTest {

  public static void main(String args[]) throws Exception {
      // Create a configuration based on the properties file we've put
```

3.2 Finding Persistent Objects

It's time to throw the giant lever into reverse and look at how you load data from a database into Java objects.

Use Hibernate Query Language to get an object-oriented view of the contents of your mapped database tables. These might have started out as objects persisted in a previous session, or they might be data that came from completely outside your application code.

3.2.1 How do I do that?

Example 3-5 shows a program that runs a simple query using the test data we just created. The overall structure will look very familiar, because all the Hibernate setup is the same as the previous program.

Example 3-5. QueryTest.java

```
package com.oreilly.hh;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

import java.sql.Time;
import java.util.*;

/**
* Retrieve data as objects
*/
public class QueryTest {

    /**
     * Retrieve any tracks that fit in the specified amount of time.
     *
     * @param length the maximum playing time for tracks to be returned.
     * @param session the Hibernate session that can retrieve data.
     * @return a list of {@link Track}s meeting the length restriction.
     * @throws HibernateException if there is a problem.
     */
    public static List tracksNoLongerThan(Time length, Session session)
```

3.3 Better Ways to Build Queries

As mentioned earlier, HQL lets you go beyond the use of JDBC-style query placeholders to get parameters conveniently into your queries. The features discussed in this section can make your programs much easier to read and maintain.

Use named parameters to control queries and move the query text completely outside of your Java source code.

3.3.1 Why do I care?

Well, I've already promised that this will make your programs easier to write, read, and update. In fact, if these features weren't available in Hibernate, I would have been less eager to adopt it, because they've been part of my own (even more) lightweight O/R layer for years.

Named parameters make code easier to understand because the purpose of the parameter is clear both within the query itself and within the Java code that is setting it up. This self-documenting nature is valuable in itself, but it also reduces the potential for error by freeing you from counting commas and question marks, and it can modestly improve efficiency by letting you use the same parameter more than once in a single query.

NOTE

If you haven't yet had to deal with this, trust me, it's well worth avoiding.

Keeping the queries out of Java source code makes them much easier to read and edit because they aren't giant concatenated series of Java strings spread across multiple lines and interleaved with extraneous quotation marks, backslashes, and other Java punctuation. Typing them the first time is bad enough, but if you've ever had to perform significant surgery on a query embedded in a program in this way, you will have had your fill of moving quotation marks and plus signs around to try to get the lines to break in nice places again.

3.3.2 How do I do that?

The key to both of these capabilities in Hibernate is the Query interface. We'll start by changing our query to use a named parameter (Example 3-8). (This isn't nearly as big a deal for a query with a single parameter like this one, but it's worth getting into the habit right away. You'll be very thankful when you start working with the light-dimming queries that power your real projects!)

Example 3-8. Revising our query to use a named parameter

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    Query query = session.createQuery("from com.oreilly.hh.Track as track " +
                                      "where track.playTime <= :length");
    query.setTime("length", length);
    return query.list();
}
```

Chapter 4. Collections and Associations

NOTE

In this chapter:

No, this isn't about taxes or politics. Now that we've seen how easy it is to get individual objects into and out of a database, it's time to see how to work with groups and relationships between objects. Happily, it's no more difficult.

4.1 Mapping Collections

In any real application you'll be managing lists and groups of things. Java provides a healthy and useful set of library classes to help with this: the Collections utilities. Hibernate provides natural ways for mapping database relationships onto Collections, which are usually very convenient. You do need to be aware of a couple semantic mismatches, generally minor. The biggest is the fact that Collections don't provide 'bag' semantics, which might frustrate some experienced database designers. This gap isn't Hibernate's fault, and it even makes some effort to work around the issue.

NOTE

Bags are like sets, except that the same value can appear more than once.

Enough abstraction! The Hibernate reference manual does a good job of discussing the whole bag issue, so let's leave it and look at a working example of mapping a collection where the relational and Java models fit nicely. It might seem natural to build on the Track examples from Chapter 3 and group them into albums, but that's not the simplest place to start, because organizing an album involves tracking additional information, like the disc on which the track is found (for multi-disc albums), and other such finicky details. So let's add artist information to our database.

NOTE

As usual, the examples assume you followed the steps in the previous chapters. If not, download the example source as a starting point.

The information we need to keep track of for artists is, at least initially, pretty simple. We'll start with just the artist's name. And each track can be assigned a set of artists, so we know who to thank or blame for the music, and you can look up all tracks by someone we like. (It really is critical to allow more than one artist to be assigned to a track, yet so few music management programs get this right. The task of adding a separate link to keep track of composers is left as a useful exercise for the reader after understanding this example.)

4.1.2 How do I do that?

For now, our Artist class doesn't need anything other than a name property (and its key, of course). Setting up a mapping document for it will be easy. Create the file Artist.hbm.xml in the same directory as the Track mapping document, with the contents shown in Example 4-1.

Example 4-1. Mapping document for the Artist class

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

  <class name="com.oreilly.hh.Artist" table="ARTIST">
    <meta attribute="class-description">
      Represents an artist who is associated with a track or album.
```

4.2 Persisting Collections

Our first task is to beef up the CreateTest class to take advantage of the new richness in our schema, creating some artists and associating them with tracks.

4.2.1 How do I do that?

To begin with, add some helper methods to CreateTest.java to simplify the task, as shown in Example 4-5 (with changes and additions in bold).

Example 4-5. Utility methods to help find and create artists, and to link them to tracks

```
package com.oreilly.hh;

import net.sf.hibernate.*;

import net.sf.hibernate.cfg.Configuration;

import java.sql.Time;
import java.util.*;

/**
 * Create more sample data, letting Hibernate persist it for us.
 */
public class CreateTest {

/**
* Look up an artist record given a name.
* @param name the name of the artist desired.
* @param create controls whether a new record should be created if
* the specified artist is not yet in the database.
* @param session the Hibernate session that can retrieve data
* @return the artist with the specified name, or <code>null</code> if no
* such artist exists and <code>create</code> is <code>false</code>.
* @throws HibernateException if there is a problem.
*/
public static Artist getArtist(String name, boolean create,
```

4.3 Retrieving Collections

You might expect that getting the collection information back out of the database is similarly easy. You'd be right! Let's enhance our QueryTest class to show us the artists associated with the tracks it displays. Example 4-8 shows the appropriate changes and additions in bold. Little new code is needed.

Example 4-8. QueryTest.java enhanced in order to display artists associated with tracks

```
package com.oreilly.hh;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

import java.sql.Time;
import java.util.*;

/**
 * Retrieve data as objects
 */
public class QueryTest {

   /**
    * Retrieve any tracks that fit in the specified amount of time.
    *
    * @param length the maximum playing time for tracks to be returned.
    * @param session the Hibernate session that can retrieve data.
    * @return a list of {@link Track}s meeting the length restriction.
    * @throws HibernateException if there is a problem.
    */
   public static List tracksNoLongerThan(Time length, Session session)
  throws HibernateException
   {
  Query query = session.getNamedQuery(
    "com.oreilly.hh.tracksNoLongerThan");
  query.setTime("length", length);
  return query.list();
```

4.4 Using Bidirectional Associations

In our creation code, we established links from tracks to artists, simply by adding Java objects to appropriate collections. Hibernate did the work of translating these associations and groupings into the necessary cryptic entries in a join table it created for that purpose. It allowed us with easy, readable code to establish and probe these relationships. But remember that we made this association bidirectional—the Artist class has a collection of Track associations too. We didn't bother to store anything in there.

The great news is that we don't have to. Because of the fact that we marked this as an inverse mapping in the Artist mapping document, Hibernate understands that when we add an Artist association to a Track, we're implicitly adding that Track as an association to the Artist at the same time.

> This convenience works only when you make changes to the 'primary' mapping, in which case they propagate to the inverse mapping. If you make changes only to the inverse mapping, in our case the Set of tracks in the Artist object, they will not be persisted. This unfortunately means your code must be sensitive to which mapping is the inverse.

Let's build a simple interactive graphical application that can help us check whether the artist to track links really show up. It will let you type in an artist's name, and show you all the tracks associated with that artist. A lot of the code is very similar to our first query test. Create the file QueryTest2.java and enter the code shown in Example 4-10 .

Example 4-10. Source for QueryTest2.java

```
package com.oreilly.hh;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

import java.sql.Time;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Provide a user interface to enter artist names and see their tracks.
 */
public class QueryTest2 extends JPanel {

    JList list; // Will contain tracks associated with current artist
```

4.5 Working with Simple Collections

The collections we've been looking at so far have all contained associations to other objects, which is appropriate for a chapter titled 'Collections and Associations,' but isn't the only kind you can use with Hibernate. You can also define mappings for collections of simple values, like strings, numbers, and nonpersistent value classes.

4.5.1 How do I do that?

Suppose we want to be able to record some number of comments about each track in the database. We want a new property called comments to contain the String values of each associated comment. The new mapping in Tracks.hbm.xml looks a lot like what we did for artists, only a bit simpler:

```
<set name="comments" table="TRACK_COMMENTS">

   <key column="TRACK_ID"/>

   <element column="COMMENT" type="string"/>

</set>
```

Since we're able to store an arbitrary number of comments for each Track, we're going to need a new table to put them in. Each comment will be linked to the proper Track through the track's id property.

Rebuilding the databases with ant schema shows how this gets built in the database:

```
[schemaexport] create table TRACK_COMMENTS (

[schemaexport]    TRACK_ID INTEGER not null,

[schemaexport]    COMMENT VARCHAR(255)

[schemaexport] )

[schemaexport] alter table TRACK_COMMENTS add constraint FK105B26884C5F92B

foreign key (TRACK_ID) references TRACK
```

NOTE

Data modeling junkies will recognize this as a 'one-to-many' relationship.

After updating the Track class via ant codegen , we need to add another Set at the end of each constructor invocation in CreateTest.java, for the comments. For example:

```
track = new Track("Test Tone 1",

                  "vol2/singles/test01.mp3",

                  Time.valueOf("00:00:10"), new Date(),

                  (short)0, new HashSet(), new HashSet());
```

Chapter 5. Richer Associations

NOTE

In this chapter:

Yes, wealthy friends would be nice. But I can't propose an easy way to get any, so let's look at relationships between objects that carry more information than simple grouping. In this chapter we'll look at the tracks that make up an album. We had put that off in Chapter 4 because organizing an album involves more than simply grouping some tracks; you also need to know the order in which the tracks occur, as well as things like which disc they're on, in order to support multi-disc albums. That goes beyond what you can achieve with an automatically generated join table, so we'll design our own AlbumTrack object and table, and let albums link to these.

Before diving in, there's an important concept called 'laziness' we need to explore.

5.1 Using Lazy Associations

First rich, then lazy? I suppose that could be a plausible story about someone, as long as it happened in that order. But this really is an object relational mapping topic of some importance. As your data model grows, adding associations between objects and tables, your program gains power, which is great. But you often end up with a large fraction of your objects somehow linked to each other. So what happens when you load one of the objects that is part of a huge interrelated cluster? Since, as you've seen, you can move from one object to its associated objects just by traversing properties, it seems you'd have to load all the associated objects when you load any of them. For small databases this is fine, but in general your database can't hold a lot more than the memory available to your program. Uh oh! And even if it does all fit, rarely will you actually access most of those objects, so it's a waste to load them all.

Luckily, this problem was anticipated by the designers of object/relational mapping software, including Hibernate. The trick is to configure some associations to be 'lazy,' so that associated objects aren't loaded until they're actually referenced. Hibernate will instead make a note of the linked object's identity and put off loading it until you actually try to access it. This is often done for collections like those we've been using.

5.1.1 How do I do that?

With collections, all you need to do is set the lazy attribute in the mapping declaration. For example, our track artists mapping could look like Example 5-1.

Example 5-1. Lazily initializing the track artist associations

```
<set name="artists" table="TRACK_ARTISTS" lazy="true">

  <key column="TRACK"/>

  <many-to-many class="com.oreilly.hh.Artist" column="ARTIST"/>

</set>
```

This would tell Hibernate to use a special lazy implementation of Set that doesn't load its contents from the database until you actually try to use them. This is done completely transparently, so you don't even notice it's taking place in your code.

Well, if it's that simple, and avoids problems with loading giant snarls of interrelated objects, why not do it all the time? The problem is that the transparency breaks down once you've closed your Hibernate session. At that point, if you try to access content from a lazy collection that hasn't been initialized (even if you've assigned the collection to a different variable, or returned it from a method call), the Hibernate-provided proxy collection can no longer access the database to perform the deferred loading of its contents, and it is forced to throw a LazyInitializationException.

NOTE

Conservation of complexity seems almost like a law of thermodynamics.

Because this can lead to unexpected crashes far away from the Hibernatespecific code, lazy initialization is turned off by default. It's your responsibility to think carefully about situations in which you need to use it, and ensure that you are doing so safely. The Hibernate reference manual goes into a bit of detail about strategies to consider.

5.1.2 What about...

...Laziness outside of collections? Caching and clustering?

5.2 Ordered Collections

Our first goal is to store the tracks that make up an album, keeping them in the right order. Later we'll add information like the disc on which a track is found, and its position on that disc, so we can gracefully handle multi-disc albums.

5.2.1 How do I do that?

The task of keeping a collection in a particular order is actually straightforward. If that's all we cared about in organizing album tracks, we'd need only tell Hibernate to map a List or array. In our Album mapping we'd use something like Example 5-2.

Example 5-2. Simple ordered mapping of tracks for an album

```
<list name="tracks" table="ALBUM_TRACKS">

  <key column="ALBUM_ID"/>

  <index column="POSITION"/>

  <many-to-many class="com.oreilly.hh.Track" column="TRACK_ID"/>

</list>
```

This is very much like the set mappings we've used so far (although it uses a different tag to indicate it's an ordered list and therefore maps to a java.util.List). But notice that we also need to add an index tag to establish the ordering of the list, and we need to add a column to hold the value controlling the ordering in the database. Hibernate will manage the contents of this column for us, and use it to ensure that when we get the list out of the database in the future, its contents will be in the same order in which we stored them. The column is created as an integer, and if possible, it is used as part of a composite key for the table. The mapping in Example 5-2, when used to generate a HSQLDB database schema, produces the table shown in Example 5-3.

Example 5-3. Our simple track list realized as an HSQLDB schema

```
 [schemaexport] create table ALBUM_TRACKS (

[schemaexport]    ALBUM_ID INTEGER not null,

[schemaexport]    TRACK_ID INTEGER not null,

[schemaexport]    POSITION INTEGER not null,

[schemaexport]    primary key (ALBUM_ID, POSITION)

[schemaexport] )
```

It's important to understand why the POSITION column is necessary. We need to control the order in which tracks appear in an album, and there aren't any properties of the tracks themselves we can use to keep them sorted in the right order. (Imagine how annoyed you'd be if your jukebox system could only play the tracks of an album in, say, alphabetical order, regardless of the intent of the artists who created it!) The fundamental nature of relational database systems is that you get results in whatever order the system finds convenient, unless you tell it how to sort them. The POSITION column gives Hibernate a value under its control that can be used to ensure that our list is always sorted in the order in which we created it. Another way to think about this is that the order of the entries is one of the independent pieces of information we want to keep track of, so Hibernate needs a place to store it.

The corollary is also important. If there are values in your data that provide a natural order for traversal, there is no need for you to provide an index column; you don't even have to use a list. The set and map collection mappings can

5.3 Augmenting Associations in Collections

All right, we've got a handle on what we need to do if we want our albums' tracks to be kept in the right order. What about the additional information we'd like to keep, such as the disc on which the track is found? When we map a collection of associations, we've seen that Hibernate creates a join table in which to store the relationships between objects. And we've just seen how to add an index column to the ALBUM_TRACKS table to maintain an ordering for the collection. Ideally, we'd like the ability to augment that table with more information of our own choosing, in order to record the other details we'd like to know about album tracks.

As it turns out, we can do just that, and in a very straightforward way.

5.3.1 How do I do that?

Up until this point we've seen two ways of getting tables into our database schema. The first was by explicitly mapping properties of a Java object onto columns of a table. The second was defining a collection of associations, and specifying the table and columns used to manage that collection. As it turns out, there's nothing that prevents us from using a single table in both ways. Some of its columns can be used directly to map to our own objects' properties, while the others can manage the mapping of a collection. This lets us achieve our goals of recording the tracks that make up an album in an ordered way, augmented by additional details to support multi-disc albums.

NOTE

This flexibility took a little getting used to but it makes sense, especially if you think about mapping objects to an existing database schema.

We'll want a new data object, AlbumTrack, to contain information about how a track is used on an album. Since we've already seen several examples of how to map full-blown entities with independent existence, and there really isn't a need for our AlbumTrack object to exist outside the context of an Album entity, this is a good opportunity to look at mapping a component. Recall that in Hibernate jargon an entity is an object that stands on its own in the persistence mechanism: it can be created, queried, and deleted independently of any other objects, and therefore has its own persistent identity (as reflected by its mandatory id property). A component, in contrast, is an object that can be saved to and retrieved from the database, but only as a subordinate part of some other entity. In this case, we'll define a list of AlbumTrack objects as a component part of our Album entity. Example 5-4 shows a mapping for the Album class that achieves this.

Example 5-4. Album.hbm.xml, the mapping definition for an Album

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
  <class name="com.oreilly.hh.Album" table="ALBUM">
    <meta attribute="class-description">
      Represents an album in the music database, an organized list of tracks.
      @author Jim Elliott (with help from Hibernate)
    </meta>
```

5.4 Lifecycle Associations

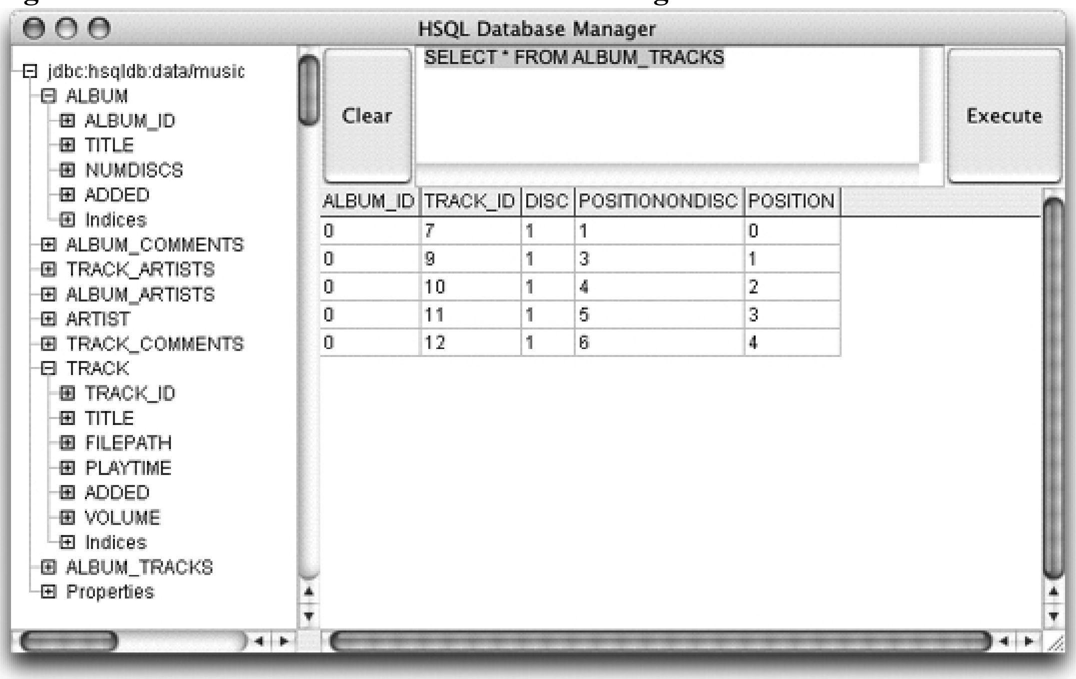
Hibernate is completely responsible for managing the ALBUM_TRACKS table, adding and deleting rows (and, if necessary, renumbering POSITION values) as entries are added to or removed from Album beans' tracks properties. You can test this by writing a test program to delete the second track from our test album and see the result. A very quick and dirty way to do this would be to add the following four lines (see Example 5-11) right after the existing tx.commit() line in Example 5-7 and then run ant schema ctest atest db .

Example 5-11. Deleting our album's second track

```
 tx = session.beginTransaction();

album.getTracks().remove(1);

session.update(album);

tx.commit();
```

Doing so changes the contents of ALBUM_TRACKS as shown in Figure 5-4 (compare this with the original contents in Figure 5-3). The second record has been removed (remember that Java list elements are indexed starting with zero), and POSITION has been adjusted so that it retains its consecutive nature, corresponding to the indices of the list elements (the values you'd use when calling tracks.get()).

Figure 5-4. Album track associations after deleting our album's second track



This happens because Hibernate understands that this list is 'owned' by the Album record, and that the 'lifecycles' of the two objects are intimately connected. This notion of lifecycle becomes more clear if you consider what happens if the entire Album is deleted: all of the associated records in ALBUM_TRACKS will be deleted as well. (Go ahead and modify the test program to try this if you're not convinced.)

Contrast this with the relationship between the ALBUM table and the TRACK table. Tracks are sometimes associated with albums, but they are sometimes independent. Removing a track from the list got rid of a row in ALBUM_TRACKS, eliminating the link between the album and track, but didn't get rid of the row in TRACK, so it didn't delete the persistent Track object itself. Similarly, deleting the Album would eliminate all the associations in the collection, but none of the actual Tracks. It's the responsibility of our code to take care of that when appropriate (probably after consulting the user, in case any of the track records might be shared across multiple albums, as discussed above).

5.5 Reflexive Associations

It's also possible for objects and tables to have associations back to themselves. This supports persistent recursive data structures like trees, in which nodes link to other nodes. Tracing through a database table storing such relationships using a SQL query interface is a major chore. Luckily, once it's mapped to Java objects, the process is much more readable and natural.

One way we might use a reflexive link in our music database is to allow alternate names for artists. This is useful more often than you might expect, because it makes it very easy to let the user find either 'The Smiths' or 'Smiths, The' depending on how they're thinking of the group, with little code, and in a language-independent way.

NOTE

I mean human language here, English versus Spanish or something else. Put the links in the data rather than trying to write tricky code to guess when an artist name should be permuted.

5.5.1 How do I do that?

All that's needed is to add another field to the Artist mapping in Artist.hbm.xml, establishing a link back to Artist. Example 5-13 shows one option.

Example 5-13. Supporting a reflexive association in the Artist class

```
<many-to-one name="actualArtist" class="com.oreilly.hh.Artist">

  <meta attribute="use-in-tostring">true</meta>

</many-to-one>
```

This gives us an actualArtist property that we can set to the id of the 'definitive' Artist record when we're setting up an alternate name. For example, our 'The Smiths' record might have id 5, and its actualArtist field would be null since it is definitive. Then we can create an 'alias' Artist record with the name 'Smiths, The' at any time, and set the actualArtist field in that record to point to record 5.

> This kind of reflexive link is one instance where a column containing a foreign key can't be named the same as the key column to which it is a link. We are associating a row in ARTIST with another row in ARTIST, and of course the table already has a column named ARTIST_ID.

Why is this association set up as many-to-one? There might be many alias records that point to one particular definitive Artist. So each nickname needs to store the id of the actual artist record for which it is an alternative name. This is, in the language of data modeling, a many-to-one relationship.

Code that looks up artists just needs to check the actualArtist property before returning. If it's null, all is well. Otherwise it should return the record indicated by actualArtist. Example 5-14 shows how we could extend the getArtist() method in CreateTest to support this new feature (additions are in bold). Notice that the Artist constructor gets a new argument for setting actualArtist.

Example 5-14. Artist lookup method supporting resolution of alternate names

```
public static Artist getArtist(String name, boolean create,

                               Session session)

    throws HibernateException
```

Chapter 6. Persistent Enumerated Types

In this chapter:

An enumerated type is a common and useful programming abstraction allowing a value to be selected from a fixed set of named choices. These were originally well represented in Pascal, but C took such a minimal approach (essentially just letting you assign symbolic names to interchangeable integer values) that early Java releases reserved C's enum keyword but declined to implement it. A better, object-oriented approach known as the "typesafe enum pattern" evolved and was popularized in Joshua Bloch's Effective Java Programming Language Guide (Addison- Wesley). This approach requires a fair amount of boilerplate coding, but it lets you do all kinds of interesting and powerful things. The Java 1.5 specification resuscitates the enum keyword as an easy way to get the power of typesafe enumerations without all the tedious boilerplate coding, and it provides other nifty benefits.

Regardless of how you implement an enumerated type, you're sometimes going to want to be able to persist such values to a database.

6.1 Defining a Persistent Enumerated Type

NOTE

C-style enumerations still appear too often in Java. Older parts of the Sun API contain many of them.

Hibernate has been around for a while and (at least as of this writing) Java 1.5 isn't yet released, so the support for enumerations in Hibernate can't take advantage of its new enum keyword. Instead, Hibernate lets you define your own typesafe enumeration classes however you like, and it provides a mechanism to help you get them into and out of a database, by translating them to and from small integer values. This is something of a regression to the world of C, but it is useful nonetheless.

In our music database, for example, we might want to add a field to our Track class that tells us the medium from which it was imported.

6.1.1 How do I do that?

The key to adding persistence support for our enumeration is to have it implement Hibernate's PersistentEnum interface. This interface has two methods, toInt() and fromInt(), that Hibernate uses to translate between the enumeration constants and values that represent them in a database.

Let's suppose we want to be able to specify whether our tracks came from cassette tapes, vinyl, VHS tapes, CDs, a broadcast, an internet download site, or a digital audio stream. (We could go really nuts and distinguish between Internet streams and satellite radio services like Sirius or XM, or radio versus television broadcast, but this is plenty to demonstrate the important ideas.)

Without any consideration of persistence, our typesafe enumeration class might look something like Example 6-1. (The JavaDoc has been compressed to take less printed space, but the downloadable version is formatted normally.)

Example 6-1. SourceMedia.java, our initial typesafe enumeration

```
package com.oreilly.hh;


import java.util.*;

import java.io.Serializable;


/**

 * This is a typesafe enumeration that identifies the media on which an

 * item in our music database was obtained.

 **/

public class SourceMedia implements Serializable {


   /** Stores the external name of this instance, by which it can be retrieved. */

   private final String name;


   /**
```

6.2 Working with Persistent Enumerations

If you were thinking about it, you may have noticed that we never defined a persistence mapping for the SourceMedia class in the first part of this chapter. That's because our persistent enumerated type is a value that gets persisted as part of one or more entities, rather than being an entity unto itself.

In that light, it's not surprising that we've not yet done any mapping. That happens when it's time to actually use the persistent enumeration.

6.2.1 How do I do that?

Recall that we wanted to keep track of the source media for the music tracks in our jukebox system. That means we want to use the SourceMedia enumeration in our Track mapping. We can simply add a new property tag to the class definition in Track.hbm.xml, as shown in Example 6-3.

Example 6-3. Adding the sourceMedia property to the Track mapping document

```
 ...

<property name="volume" type="short">

  <meta attribute="field-description">How loud to play the track</meta>

</property>



<property name="sourceMedia" type="com.oreilly.hh.SourceMedia">

  <meta attribute="field-description">Media on which track was obtained</meta>

  <meta attribute="use-in-tostring">true</meta>

</property>



</class>

...
```

Because the type of our sourceMedia property names a class that implements the PersistentEnum interface, Hibernate knows to persist it using its built-in enumeration support.

With this addition in place, running ant codegen updates our Track class to include the new property. The signature of the full-blown Track constructor now looks like this:

```
public Track(String title, String filePath, Date playTime, Date added,

             short volume, com.oreilly.hh.SourceMedia sourceMedia,

             Set artists, Set comments) { ... }
```

We need to make corresponding changes in CreateTest.java:

```
Track track = new Track("Russian Trance",

                        "vol2/album610/track02.mp3",

                        Time.valueOf("00:03:30"), new Date(),

                        (short)0, SourceMedia.CD,

                        new HashSet(), new HashSet());
```

Chapter 7. Custom Value Types

NOTE

In this chapter:

Hibernate supports a wealth of Java types, be they simple values or objects, as you can see by skimming Appendix A. By setting up mapping specifications, you can persist even highly complex, nested object structures to arbitrary database tables and columns. With all this power and flexibility, you might wonder why you'd ever need to go beyond the built-in type support.

One situation that might motivate you to customize Hibernate's type support is if you want to use a different SQL column type to store a particular Java type than Hibernate normally chooses. The reference documentation cites the example of persisting Java BigInteger values into VARCHAR columns, which might be necessary to accommodate a legacy database schema.

Another scenario that requires the ability to tweak the type system is when you have a single property value that needs to get split into more than one database column—maybe the Address object in your company's mandated reuse library stores ZIP+4 codes as a single string, but the database to which you're integrating contains a required five digit column and a separate nullable four digit column for the two components. Or maybe it's the other way around, and you need to separate a single database column into more than one property.

Luckily, in situations like this, Hibernate lets you take over the details of the persistence mapping so you can fit square pegs into round holes when you really need to.

NOTE

Continuing in the spirit of making simple things easy and complex things possible...

You might also want to build a custom value type even in some cases where it's not strictly necessary. If you've got a composite type that is used in many places throughout your application (a vector, complex number, address, or the like), you can certainly map each of these occurrences as components, but it might be worth encapsulating the details of the mapping in a shared, reusable Java class rather than propagating the details throughout each of the mapping documents. That way, if the details of the mapping ever need to change for any reason, you've only got one class to fix rather than many individual component mappings to hunt down and adjust.

7.1 Defining a User Type

In all of these scenarios, the task is to teach Hibernate a new way to translate between a particular kind of in-memory value and its persistent database representation.

Hibernate lets you provide your own logic for mapping values in situations that need it, by implementing one of two interfaces: net.sf.hibernate.UserType or net.sf.hibernate.CompositeUserType.

It's important to realize that what is being created is a translator for a particular kind of value, not a new kind of value that knows how to persist itself. In other words, in our ZIP code example, it's not the ZIP code property that would implement UserType. Instead, we'd create a new class implementing UserType, and in our mapping document specify this class as the Java type used to map the ZIP code property. Because of this, I think the terminology of 'user types' is a little confusing.

Let's look at a concrete example. In Chapter 6 we saw how to use Hibernate's built-in enumeration support to persist a typesafe enumeration to an integer column, and we had to work around the fact that many object-oriented enumerations have no natural integer representation. While we can hope that Java 1.5 will allow Hibernate to resolve this tension in a universal way, we don't have to wait for that to happen, nor do we necessarily have to make the kind of compromises we did in Example 6-2. We can define our own custom value type that persists the SourceMedia class on its own terms. Later in the chapter we'll look at a more complex example involving multiple properties and columns.

7.1.1 How do I do that?

We'll work with the verson of SourceMedia.java shown in Example 6-1. Our custom type will allow this class to be persisted without any changes from its original form. In other words, the design of our data classes can be dictated by the needs and semantics of the application alone, and we can move the persistence support into a separate class focused on that sole purpose. This is a much better division of labor.

We'll call our new class SourceMediaType. Our next decision is whether it needs to implement UserType or CompositeUserType. The reference documentation doesn't provide much guidance on this question, but the API documentation confirms the hint contained in the interface names: the CompositeUserType interface is only needed if your custom type implementation is to expose internal structure in the form of named properties that can be accessed individually in queries (as in our ZIP code example). For SourceMedia, a simple UserType implementation is sufficient. The source for a mapping manager meeting our needs is shown in Example 7-1.

Example 7-1. SourceMediaType.java, our custom type mapping handler

```
package com.oreilly.hh;


import java.io.Serializable;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Types;


import net.sf.hibernate.UserType;

import net.sf.hibernate.Hibernate;
```

7.2 Using a Custom Type Mapping

All right, we've created a custom type persistence handler, and it wasn't so bad! Now it's time to actually use it to persist our enumeration data the way we want it.

7.2.1 How do I do that?

This is actually almost embarrassingly easy. Once we've got the value class, SourceMedia, and the persistence manager, SourceMediaType, in place, all we need to do is modify any mapping documents that were previously referring to the raw value type to refer instead to the custom persistence manager.

NOTE

That's it. No, really!

In our case, that means we change the mapping for the mediaSource property in Track.hbm.xml so it looks like Example 7-2 rather than Example 6-3.

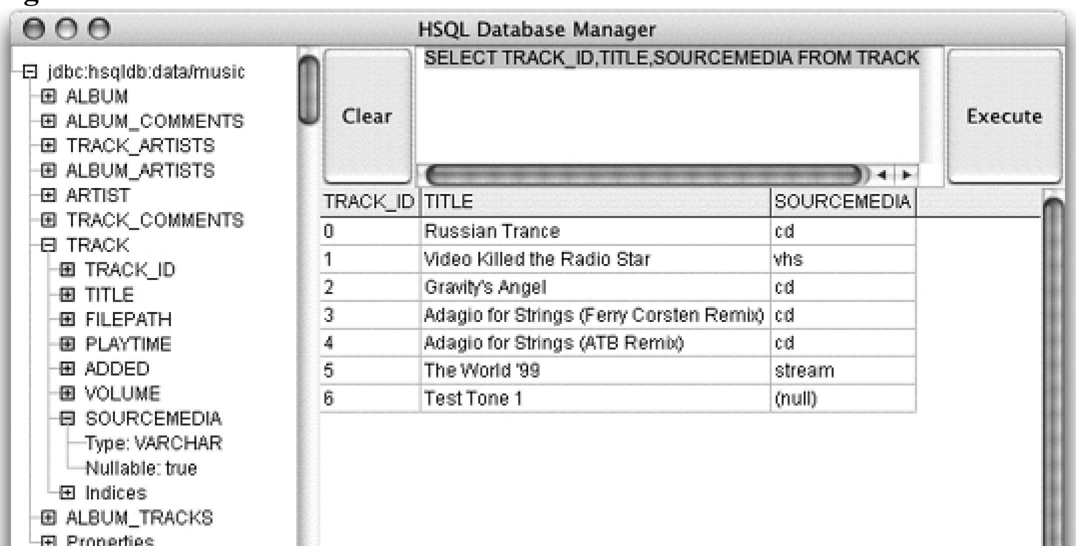
Example 7-2. Custom type mapping for the sourceMedia property

```
<property name="sourceMedia" type="com.oreilly.hh.SourceMediaType">

   <meta attribute="field-description">Media on which track was obtained</meta>

   <meta attribute="use-in-tostring">true</meta>

</property>
```

At this point, running ant schema will rebuild the database schema, changing the SOURCEMEDIA column in the TRACK table from integer to VARCHAR (as specified by SourceMediaType's sqlTypes() method).

Thanks to the beauty of letting the object/relational mapping layer handle the details of how data is stored and retrieved, we don't need to change any aspect of the example or test code that we were using in Chapter 6. You can run ant ctest to create sample data. It will run with no complaint. If you fire up ant db to look at the way it's stored, you'll find that our goal of storing semantically meaningful enumeration symbols has been achieved, as shown in Figure 7-1.

Figure 7-1. Nicer source media information in the TRACK table



7.3 Building a Composite User Type

Recall that in our Track object we have a property that determines our preferred playback volume for the track. Suppose we'd like the jukebox system to be able to adjust the balance of tracks for playback, rather than just their volume. To accomplish this we'd need to store separate volumes for the left and right channels. The quick solution would be to edit the Track mapping to store these as separate mapped properties.

If we're serious about object-oriented architecture, we might want to encapsulate these two values into a StereoVolume class. This class could then simply be mapped as a composite-element, as we did with the AlbumTrack component in lines 38-45 of Example 5-4. This is still fairly straightforward.

There is a drawback to this simple approach. It's likely we will discover other places in our system where we want to represent StereoVolume values. If we build a playlist mechanism that can override a track's default playback options, and also want to be able to assign volume control to entire albums, suddenly we have to recreate the composite mapping in several places, and we might not do it consistently everywhere (this is more likely to be an issue with a more complex compound type, but you get the idea). The Hibernate reference documentation says that it's a good practice to use a composite user type in situations like this, and I agree.

7.3.1 How do I do that?

Let's start by defining the StereoVolume class. There's no reason for this to be an entity (to have its own existence independent of some other persistent object), so we'll write it as an ordinary (and rather simple) Java object. Example 7-4 shows the source.

NOTE

The JavaDoc in this example has been compressed to take less space. I'm trusting you not to do this in real projects... the downloadable version is more complete.

Example 7-4. StereoVolume.java, which is a value class representing a stereo volume level

```
package com.oreilly.hh;

import java.io.Serializable;

/**
 * A simple structure encapsulating a stereo volume level.
 */
public class StereoVolume implements Serializable {

  /** The minimum legal volume level. */
  public static final short MINIMUM = 0;

  /** The maximum legal volume level. */
```

Chapter 8. Criteria Queries

NOTE

In this chapter:

Relational query languages like HQL (and SQL, on which it's based) are extremely flexible and powerful, but they take a long time to truly master. Many application developers get by with a rudimentary understanding, cribbing similar examples from past projects, and calling in database experts when they need to come up with something truly new, or to understand a particularly cryptic query expression.

It can also be awkward to mix a query language's syntax with Java code. The section 'Better Ways to Build Queries' in Chapter 3 showed how to at least keep the queries in a separate file so they can be seen and edited in one piece, free of Java string escape sequences and concatenation syntax. Even with that technique, though, the HQL isn't parsed until the mapping document is loaded, which means that any syntax errors it might harbor won't be caught until the application is running.

Hibernate offers an unusual solution to these problems in the form of criteria queries. They provide a way to create and connect simple Java objects that act as filters for picking your desired results. You can build up nested, structured expressions. The mechanism also allows you to supply example objects to show what you're looking for, with control over which details matter and which properties to ignore.

As you'll see, this can be quite convenient. To be fair, it has its own disadvantages. Expanding long query expressions into a Java API makes them take more room, and they'll be less familiar to experienced database developers than a SQL-like query. There are also some things you simply can't express using the current criteria API, such as projection (retrieving a subset of the properties of a class, e.g., 'select title, id from com.oreilly.hh.Track' rather than 'select * from com.oreilly. hh.Track') and aggregation (summarizing results, e.g., getting the sum, average, or count of a property). The next chapter shows how to accomplish such tasks using Hibernate's object-oriented query language.

8.1 Using Simple Criteria

Let's start by building a criteria query to find tracks shorter than a specified length, replacing the HQL we used in Example 3-9 and updating the code of Example 3-10.

8.1.1 How do I do that?

The first thing we need to figure out is how to specify the kind of object we're interested in retrieving. There is no query language involved in building criteria queries. Instead, you build up a tree of Criteria objects describing what you want. The Hibernate Session acts as a factory for these criteria, and you start, conveniently enough, by specifying the type of objects you want to retrieve.

Edit QueryTest.java, replacing the contents of the tracksNoLongerThan() method with those shown in Example 8-1.

Example 8-1. The beginnings of a criteria query

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    Criteria criteria = session.createCriteria(Track.class);
    return criteria.list();
}
```

NOTE

These examples assume the database has been set up as described in the preceding chapters. If you don't want to go through all that, download the sample code, then jump into this chapter and run the 'codegen', 'schema', and 'ctest' targets.

The session's createCriteria() method builds a criteria query that will return instances of the persistent class you supply as an argument. Easy enough. If you run the example at this point, of course, you'll see all the tracks in the database, since we haven't gotten around to expressing any actual criteria to limit our results yet (Example 8-2).

Example 8-2. Our fledgling criteria query returns all tracks

```
% ant qtest
...
qtest:
    [java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
    [java] Track: "Video Killed the Radio Star" (The Buggles) 00:03:49, from VHS
Videocassette Tape
    [java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
    [java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
    [java] Track: "Adagio for Strings (ATB Remix)" (ATB, William Orbit, Samuel
```

8.2 Compounding Criteria

As you might expect, you can add more than one Criterion to your query, and all of them must be satisfied for objects to be included in the results. This is equivalent to building a compound criterion using Expression.conjunction(), as described in Appendix B. As in Example 8-8, we can restrict our results so that the tracks also have to contain a capital 'A' somewhere in their title by adding another line to our method.

Example 8-8. A pickier list of short tracks

```
Criteria criteria = session.createCriteria(Track.class);

criteria.add(Expression.le("playTime", length));

criteria.add(Expression.like("title", "%A%"));

criteria.addOrder(Order.asc("title"));

return criteria.list();
```

With this in place, we get fewer results (Example 8-9).

Example 8-9. Tracks of seven minutes or less containing a capital A in their titles

```
qtest:

    [java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,

William Orbit, Samuel Barber) 00:06:35, from Compact Disc

    [java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
```

If you want to find any objects matching any one of your criteria, rather than requiring them to fit all criteria, you need to explicitly use Expression.disjunction() to group them. You can build up combinations of such groupings, and other complex hierarchies, using the built-in criteria offered by the Expression class. Check Appendix B for the details. Example 8-10 shows how we'd change the sample query to give us tracks that either met the length restriction or contained a capital A.

NOTE

Criteria queries are a surprising mix of power and convenience.

Example 8-10. Picking tracks more leniently

```
Criteria criteria = session.createCriteria(Track.class);

Disjunction any = Expression.disjunction();

any.add(Expression.le("playTime", length));

any.add(Expression.like("title", "%A%"));

criteria.add(any);
```

8.3 Applying Criteria to Associations

So far we've been looking at the properties of a single class in forming our criteria. Of course, in our real systems, we've got a rich set of associations between objects, and sometimes the details we want to use to filter our results come from these associations. Fortunately, the criteria query API provides a straightforward way of performing such searches.

8.3.1 How do I do that?

Let's suppose we're interested in finding all the tracks associated with particular artists. We'd want our criteria to look at the values contained in each Track's artists property, which is a collection of associations to Artist objects. Just to make it a bit more fun, let's say we want to be able to find tracks associated with artists whose name property matches a particular SQL string pattern.

Let's add a new method to QueryTest.java to implement this. Add the method shown in Example 8-13 after the end of the tracksNoLongerThan() method.

Example 8-13. Filtering tracks based on their artist associations

```
1   /**
2    * Retrieve any tracks associated with artists whose name matches a
3    * SQL string pattern.
4    *
5    * @param namePattern the pattern which an artist's name must match
6    * @param session the Hibernate session that can retrieve data.
7    * @return a list of {@link Track}s meeting the artist name restriction.
8    * @throws HibernateException if there is a problem.
9    */
10  public static List tracksWithArtistLike(String namePattern, Session session)
11      throws HibernateException
12  {
13      Criteria criteria = session.createCriteria(Track.class);
14      Criteria artistCriteria = criteria.createCriteria("artists");
15      artistCriteria.add(Expression.like("name", namePattern));
16      criteria.addOrder(Order.asc("title"));
17      return criteria.list();
18  }
```

Line 14 creates a second Criteria instance, attached to the one we're using to select tracks, by following the tracks' artists property. We can add constraints to either criteria (which would apply to the properties of the Track itself), or to artistCriteria, which causes them to apply to the properties of the Artist entities associated with the track. In this case, we are only interested in features of the artists, so line 15 restricts our results to tracks associated with at least

8.4 Querying by Example

If you don't want to worry about setting up expressions and criteria, but you've got an object that shows what you're looking for, you can use it as an example and have Hibernate build the criteria for you.

8.4.1 How do I do that?

Let's add another query method to QueryTest.java. Add the code of Example 8-16 to the top of the class where the other queries are.

Example 8-16. Using an example entity to populate a criteria query

```
1  /**
2   * Retrieve any tracks that were obtained from a particular source
3   * media type.
4   *
5   * @param sourceMedia the media type of interest.
6   * @param session the Hibernate session that can retrieve data.
7   * @return a list of {@link Track}s meeting the media restriction.
8   * @throws HibernateException if there is a problem.
9   */
10 public static List tracksFromMedia(SourceMedia media, Session session)
11     throws HibernateException
12 {
13     Track track = new Track();
14     track.setSourceMedia(media);
15     Example example = Example.create(track);
16
17     Criteria criteria = session.createCriteria(Track.class);
18     criteria.add(example);
19     criteria.addOrder(Order.asc("title"));
20     return criteria.list();
21 }
```

Lines 13 and 14 create the example Track and set the sourceMedia property to represent what we're looking for. Lines 15 wraps it in an Example object. This object gives you some control over which properties will be used in building criteria and how strings are matched. The default behavior is that null properties are ignored, and that strings are compared in a case-sensitive and literal way. You can call example's excludeZeroes() method if you want properties with a value of zero to be ignored too, or excludeNone() if even null properties are to be matched. An excludeProperty() method lets you explicitly ignore specific properties by name, but that's starting to get a lot like

Chapter 9. A Look at HQL

In this chapter:

HQL queries have already been used a few times in previous chapters. It's worth spending a little time looking at how HQL differs from SQL and some of the useful things you can do with it. As with the rest of this notebook, our intention is to provide a useful introduction and some examples, not a comprehensive reference.

9.1 Writing HQL Queries

We've already shown that you can get by with fewer pieces in an HQL query than you might be used to in SQL (the queries we've been using, such as those in Chapter 3, have generally omitted the "select" clause). In fact, the only thing you really need to specify is the class in which you're interested. Example 9-1 shows a minimal query that's a perfectly valid way to get a list of all Track instances persisted in the database.

NOTE

HQL stands for Hibernate Query Language. And SQL? It depends who you ask.

Example 9-1. The simplest HQL query

```
from Track
```

There's not much to it, is there? This is the HQL equivalent of Example 8-1, in which we built a criteria query on the Track class and supplied no criteria.

By default, Hibernate automatically "imports" the names of each class you map, which means you don't have to provide a fully qualified package name when you want to use it, the simple class name is enough. As long as you've not mapped more than one class with the same name, you don't need to use fully qualified class names in your queries. You are certainly free to do so if you prefer—as I have in this book to help readers remember that the queries are expressed in terms of Java data beans and their properties, not database tables and columns (like you'd find in SQL). Example 9-2 produces precisely the same result as our first query.

Example 9-2. Explicit package naming, but still pretty simple

```
from com.oreilly.hh.Track
```

If you do have more than one mapped class with the same name, you can either use the fully qualified package name to refer to each, or you can assign an alternate name for one or both classes using an import tag in their mapping documents. You can also turn off the auto-import facility for a mapping file by adding auto-import="false" to the hibernatemapping tag's attributes.

> You're probably used to queries being case-insensitive, since SQL behaves this way. For the most part, HQL acts the same, with the important exceptions of class and property names. Just as in the rest of Java, these are case-sensitive, so you must get the capitalization right.

Let's look at an extreme example of how HQL differs from SQL, by pushing its polymorphic query capability to its logical limit.

9.1.1 How do I do that?

A powerful way to highlight the fundamental difference between SQL and HQL is to consider what happens when you query "from java.lang.Object". At first glance this might not even seem to make sense! In fact, Hibernate supports queries that return polymorphic results. If you've got mapped classes that extend each other, or have some shared ancestor or interface, whether you've mapped the classes to the same table or to different tables, you can query the superclass. Since every Java object extends Object, this query asks Hibernate to return every single entity it knows about in the database.

We can test this by making a quick variant of our query test. Copy QueryTest.java to QueryTest3.java, and make the changes shown in Example 9-3 (most of the changes, which don't show up in the example, involve deleting example queries we don't need here).

9.2 Selecting Properties and Pieces

The queries we've been using so far have returned entire persistent objects. This is the most common use of an object/relational mapping service like Hibernate, so it should come as no surprise. Once you've got the objects, you can use them in whatever way you need to within the familiar realm of Java code. There are circumstances where you might want only a subset of the properties that make up an object, though, such as producing reports. HQL can accommodate such needs, in exactly the same way you'd use ordinary SQL—projection in a select clause.

9.2.1 How do I do that?

Suppose we want to change QueryTest.java to display only the titles of the tracks that meet our search criteria, and we want to extract only that information from the database in the first place. We'd start by changing the query of Example 3-9 to retrieve only the title property. Edit Track.hbm.xml to make the query look like Example 9-6.

Example 9-6. Obtaining just the titles of the short tracks

```
<query name="com.oreilly.hh.tracksNoLongerThan">

  <![CDATA[

     select track.title from com.oreilly.hh.Track as track

        where track.playTime <= :length

    ]]>

</query>
```

Make sure the tracksNoLongerThan() method in QueryTest.java is set up to use this query. (If you edited it to use criteria queries in Chapter 8, change it back to the way it was in Example 3-10. To save you the trouble of hunting that down, it's reproduced as Example 9-7.)

Example 9-7. HQL-driven query method, using the query mapped in Example 9-6

```
public static List tracksNoLongerThan(Time length, Session session)

    throws HibernateException

{

    Query query = session.getNamedQuery(

                        "com.oreilly.hh.tracksNoLongerThan");

    query.setTime("length", length);

    return query.list();

}
```

Finally, the main() method needs to be updated, as shown in Example 9-8, to reflect the fact that the query method is now returning the title property rather than entire Track records. This property is defined as a String, so the method now returns a List of Strings.

Example 9-8. Changes to QueryTest's main() method to work with the title query

```
// Print the titles of tracks that will fit in five minutes

List titles = tracksNoLongerThan(Time.valueOf("00:05:00"),

                                 session);

for (ListIterator iter = titles.listIterator() ;
```

9.3 Sorting

It should come as no surprise that you can use a SQL-style "order by" clause to control the order in which your output appears. I've alluded to this several times in earlier chapters, and it works just like you'd expect. You can use any property of the objects being returned to establish the sort order, and you can list multiple properties to establish sub-sorts within results for which the first property values are the same.

9.3.1 How do I do that?

Sorting is very simple: you list the values that you want to use to sort the results. As usual, where SQL uses columns, HQL uses properties. For Example 9-13, let's update the query in Example 9-10 so that it displays the results in reverse alphabetical order.

NOTE

As in SQL, you specify an ascending sort using "asc" and a descending sort with "desc".

Example 9-13. Addition to Track.hbm.xml that sorts the results backwards by title

```
<query name="com.oreilly.hh.tracksNoLongerThan">

  <![CDATA[

      select track.id, track.title from com.oreilly.hh.Track as track

      where track.playTime <= :length

    order by track.title desc

    ]]>

</query>
```

The output from running this is as you'd expect (Example 9-14).

Example 9-14. Titles and IDs in reverse alphabetical order

```
% ant qtest

Buildfile: build.xml


prepare:

     [copy] Copying 1 file to /Users/jim/Documents/Work/OReilly/Hibernate/

Examples/ch09/classes


compile:


qtest:

     [java] Track: Video Killed the Radio Star [ID=1]

     [java] Track: Test Tone 1 [ID=6]

     [java] Track: Russian Trance [ID=0]
```

9.4 Working with Aggregate Values

Especially when writing reports, you'll often want summary information from the database: "How many? What's the average? The longest?" HQL can help with this, by offering aggregate functions like those in SQL. In HQL, of course, these functions apply to the properties of persistent classes.

9.4.1 How do I do that?

Let's try some of this in our query test framework. First, add the query in Example 9-15 after the existing query in Track.hbm.xml.

Example 9-15. A query collecting aggregate information about tracks

```
 <query name="com.oreilly.hh.trackSummary">

  <![CDATA[

      select count(*), min(track.playTime), max(track.playTime)

      from com.oreilly.hh.Track as track

    ]]>

</query>
```

I was tempted to try asking for the average playing time as well, but unfortunately HSQLDB doesn't know how to calculate averages for nonnumeric values, and this property is stored in a column of type date.

Next we need to write a method to run this query and display the results. Add the code in Example 9-16 to QueryTest.java, after the tracksNoLongerThan() method.

Example 9-16. A method to run the trackSummary query

```
 /**

 * Print summary information about all tracks.

 *

 * @param session the Hibernate session that can retrieve data.

 * @throws HibernateException if there is a problem.

 **/

public static void printTrackSummary(Session session)

    throws HibernateException

{

    Query query = session.getNamedQuery("com.oreilly.hh.trackSummary");

    Object[] results = (Object[])query.uniqueResult();

    System.out.println("Summary information:");

    System.out.println("       Total tracks: " + results[0]);

    System.out.println("     Shortest track: " + results[1]);

    System.out.println("      Longest track: " + results[2]);

}
```

9.5 Writing Native SQL Queries

Given the power and convenience of HQL, and the way it dovetails so naturally with the objects in your Java code, why wouldn't you want to use it? Well, there might be some special feature supported by the native SQL dialect of your project's database that HQL can't exploit. If you're willing to accept the fact that using this feature will make it harder to change databases in the future, Hibernate will let you write queries in that native dialect while still helping you write expressions in terms of properties and translate the results to objects. (If you didn't want this help, you could just use a raw JDBC connection to run a plain SQL query, of course.)

Another circumstance in which it might be nice to meet your database halfway is if you're in the process of migrating an existing JDBC-based project to Hibernate, and you want to take small steps rather than thoroughly rewriting each query right away.

9.5.1 How do I do that?

If you're embedding your query text inside your Java source code, you use the Session method createSQLQuery() instead of Example 3-8's createQuery(). Of course, you know better than to code like that, so I won't even show you an example. The better approach is to put the query in a mapping document like Example 3-9. The difference is that you use a sql-query tag rather than the query tag we've seen up until now. You also need to tell Hibernate the mapped class you want to return, and the alias that you're using to refer to it (and its properties) in the query.

As a somewhat contrived example, suppose we want to know all the tracks that end exactly halfway through the last minute they're playing (in other words, the time display on the jukebox would be h:mm:30). An easy way to do that would be to take advantage of HSQLDB's built-in SECOND function, which gives you the seconds part of a Time value. Since HQL doesn't know about functions that are specific to HSQLDB's SQL dialect, this will push us into the realm of a native SQL query. Example 9-23 shows what it would look like; add this after the HQL queries in Track.hbm.xml.

Example 9-23. Embedding a native SQL dialect query in a Hibernate mapping

```
 <sql-query name="com.oreilly.hh.tracksEndingAt">

  <return alias="track" class="com.oreilly.hh.Track"/>

  <![CDATA[

       select {track.*}

      from TRACK as {track}

      where SECOND({track}.PLAYTIME) = :seconds

    ]]>

</sql-query>
```

The return tag tells Hibernate we're going to be using the alias track in our query to refer to a Track object. That allows us to use the shorthand {track.*} in the query body to refer to all the columns from the TRACK table we need in order to create a Track instance. (Notice that everywhere we use the alias in the query body we need to enclose it in curly braces. This gets us "out of" the native SQL environment so we can express things in terms of Hibernate-mapped classes and properties.)

The where clause in the query uses the HSQLDB SECOND function to narrow our results to include only tracks whose length has a specified number in the seconds part. Happily, even though we're building a native SQL query, we can still make use of Hibernate's nice named query parameters. In this case we're passing in a value named "seconds" to control the query. (You don't use curly braces to tell Hibernate you're using a named parameter even in an SQL query; its parser is smart enough to figure this out.)

The code that uses this mapped SQL query is no different than our previous examples using HQL queries. The

Appendix A. Hibernate Types

Hibernate makes a fundamental distinction between two different kinds of data in terms of how they relate to the persistence service: entities and values.

An entity is something with its own independent existence, regardless of whether it's currently reachable by any object within a Java virtual machine. Entities can be retrieved from the database through queries, and they must be explicitly saved and deleted by the application. (If cascading relationships have been set up, the act of saving or deleting a parent entity can also save or delete its children, but this is still explicit at the level of the parent.)

Values are stored only as part of the persistent state of an entity. They have no independent existence. They might be primitives, collections, enumerations, and custom user types. Since they are entirely subordinated to the entity in which they exist, they cannot be independently versioned, nor can they be shared by more than one entity or collection.

Notice that a particular Java object might be either an entity or a value—the difference is in how it is designed and presented to the persistence service. Primitive Java types are always values.

A.1 Basic Types

Hibernate's basic types fall into a number of groupings:

Simple numeric and Boolean types

These correspond to the primitive Java types that represent numbers, characters and Boolean values, or their wrapper classes. They get mapped to appropriate SQL column types (based on the SQL dialect in use). They are: boolean, byte, character, double, float, integer, long, short, true_false, and yes_no. The last two are alternate ways to represent a Boolean value within the database; true_false uses the values 'T' and 'F', while yes_no uses 'Y' and 'N'.

String type

The Hibernate type string maps from java.lang.String to the appropriate string column type for the SQL dialect (usually VARCHAR, but in Oracle VARCHAR2 is used).

Time types

Hibernate uses date, time, and timestamp to map from java.util.Date (and subclasses) to appropriate SQL types (e.g., DATE, TIME, TIMESTAMP).

Arbitrary precision numeric

The Hibernate type big_decimal provides a mapping between java.math.BigDecimal to the appropriate SQL type (usually NUMERIC, but Oracle uses NUMBER).

Localization values

The types locale, timezone, and currency are stored as strings (VARCHAR or VARCHAR2 as noted above), and mapped to the Locale, TimeZone, and Currency classes in the java.util package. Locale and Currency are stored using their ISO codes, while TimeZone is stored using its ID property.

Class names

The type class maps instances of java.lang.Class using their fully qualified names, stored in a string column (VARCHAR, or VARCHAR2 in Oracle).

Byte arrays

The type binary stores byte arrays in an appropriate SQL binary type.

Any serializable object

The type serializable can be used to map any serializable Java object into a SQL binary column. This is the fallback type used when attempting to persist an object that doesn't have a more specific appropriate mapping (and does not implement PersistentEnum; see the next section).

A.2 Persistent Enumerated Types

Hibernate provides a mechanism to help map the common Java type-safe enumeration pattern to a database column. Unfortunately, the approach taken requires your enumerations to have an integer representation to store in the database, forcing them back to the lowest common denominator semantics of the enum type in the C language. I hope that a richer, string-based storage mechanism will eventually be supported, to dovetail nicely with the built-in support for this idiom that is coming in Tiger (Java Version 1.5). Storing enumerations as strings would also make them more readable to users of the raw database, a form of self-documenting storage.

To work with the current Hibernate implementation, your enumeration classes need only implement the net.sf.hibernate.PersistentEnum interface, and its fromInt() and toInt() methods. This is demonstrated in Chapter 6.

A.3 Custom Value Types

In addition to mapping your objects as entities, you can also create classes that are mapped to the database as values within other entities, without their own independent existence. This can be as simple as changing the way an existing type is mapped (because you want to use a different column type or representation), or as complex as splitting a value across multiple columns.

Although you can do this on a case-by-case basis within your mapping documents, the principle of avoiding repeated code argues for encapsulating types you use in more than one place into an actual reusable class. Your class will implement either net.sf.hibernate.UserType or net.sf.hibernate.CompositeUserType. This technique is illustrated in Chapter 7.

A.4 'Any' Type Mappings

This final kind of mapping is very much a free-for-all. Essentially, it allows you to map references to any of your other mapped entities interchangeably. This is done by providing two columns, one which contains the name of the table to which each reference is being made, and another which provides the ID within that table of the specific entity of interest.

You can't maintain any sort of foreign key constraints in such a loose relationship. It's rare to need this kind of mapping at all. One situation in which you might find it useful is if you want to maintain an audit log that can contain actual objects. The reference manual also mentions web application session data as another potential use, but that seems unlikely in a well-structured application.

A.5 All Types

The following table shows each of the type classes in the net.sf.hibernate.types package, along with the type name you would use for it in a mapping document, the SQL type used in columns storing mapped values, and any relevant comments about its purpose. In many cases, more detailed discussion can be found earlier. To save space, the 'Type' which appears at the end of each class name has been removed, except in the case of the Type interface implemented by all the others.

Type class	Type name	SQL type	Notes
Abstract-Component	N/A	N/A	Abstract ancestor of Component, DynaBean, and Object types
Abstract	N/A	N/A	Abstract skeleton used by the built-in types
Array	N/A	N/A	Maps a Java array as a Persistent-Collection
Association	N/A	N/A	Interface used by all associations between entities
Bag	N/A	N/A	Maps collections with bag semantics
BigDecimal	big_decimal	NUMERIC	In Oracle, SQL type is NUMBER
Binary	binary	VARBINARY	Basic type for byte arrays
Blob	blob	BLOB	Not all drivers support this
Boolean	boolean	BIT	A basic type
Byte	byte	TINYINT	A basic type
CalendarDate	calendar_date	DATE	A basic type
Calendar	calendar	TIMESTAMP	A basic type
CharBoolean	N/A	CHAR	Abstract skeleton used to implement yes_no and true_false types
Character	character	CHAR	A basic and primitive type
Class	class	VARCHAR or VARCHAR2	Basic type that stores a class' name
Clob	clob	CLOB	Not all drivers support this
Component	N/A	N/A	Maps the properties of a contained value class on to a group of columns
Composite-Custom	N/A	N/A	Adapts CompositeUserType implementations to the Type interface
Currency	currency	VARCHAR or VARCHAR2	Stores ISO code for a currency

Appendix B. Standard Criteria

B.1 The Expression Factory

Hibernate provides the class net.sf.hibernate.expression.Expression as a factory for creating the Criterion instances you use to set up criteria queries. Expression defines a bunch of static methods you can invoke to conveniently create each of the standard Criterion implementations available in Hibernate, using parameters you supply. These criteria are used to determine which persistent objects from the database are included in the results of your query. Here is a summary of the available options.

Method	Parameters	Purpose
allEq	Map properties	A shortcut for requiring several properties to have particular values. The keys of the supplied map are the names of the properties you want to constrain, while the values in the map are the target values each property must equal if an entity is to be included in the query results. The returned Criterion ensures that each named property has the corresponding value.
and	Criterion lhs, Criterion rhs	Builds a compound Criterion that requires both halves to be met in order for the whole to succeed.
between	String property, Object low, Object high	Requires the value of the named property to fall between the values of low and high.
conjunction	None	Creates a Conjunction object which can be used to build an "and" criterion with as many pieces as you need. Simply call its add() method with each of the Criterion instances you want to check. The conjunction will be true if and only if all its component criteria are true. This is more convenient than building a tree of and() criteria "by hand." The add() method of the Criteria interface acts as though it contains a Conjunction.
disjunction	None	Creates a Disjunction object that can be used to build an "or" criterion with as many pieces as you need. Simply call its add() method with each of the Criterion instances you want to check. The disjunction will be true if any of its component criteria are true. This is more convenient than building a tree of or() criteria "by hand." See Example 8-10.
eq	String property, Object value	Requires the named property to have the specified value.
eqProperty	String property1, String property2	Requires the two named properties to have the same value.

Appendix C. Hibernate SQL Dialects

C.1 Getting Fluent in the Local SQL

Hibernate ships with detailed support for many commercial and free relational databases. While most features will work properly without doing so, it's important to set the hibernate.dialect configuration property to the right subclass of net.sf.hibernate.dialect.Dialect, especially if you want to use features like native or sequence primary key generation or session locking. Choosing a dialect is also a very convenient way of setting up a whole raft of Hibernate configuration parameters you'd otherwise have to deal with individually.

Database system	Appropriate hibernate.dialect setting
DB2	net.sf.hibernate.dialect.DB2Dialect
FrontBase	net.sf.hibernate.dialect.FrontbaseDialect
HSQLDB	net.sf.hibernate.dialect.HSQLDialect
Informix	net.sf.hibernate.dialect.InformixDialect
Ingres	net.sf.hibernate.dialect.IngresDialect
Interbase	net.sf.hibernate.dialect.InterbaseDialect
Mckoi SQL	net.sf.hibernate.dialect.MckoiDialect
Microsoft SQL Server	net.sf.hibernate.dialect.SQLServerDialect
MySQL	net.sf.hibernate.dialect.MySQLDialect
Oracle (any version)	net.sf.hibernate.dialect.OracleDialect
Oracle 9 (specifically)	net.sf.hibernate.dialect.Oracle9Dialect
Pointbase	net.sf.hibernate.dialect.PointbaseDialect
PostgreSQL	net.sf.hibernate.dialect.PostgreSQLDialect
Progress	net.sf.hibernate.dialect.ProgressDialect
SAP DB	net.sf.hibernate.dialect.SAPDBDialect
Sybase	net.sf.hibernate.dialect.SybaseDialect
Sybase Anywhere	net.sf.hibernate.dialect.SybaseAnywhereDialect

If you don't see your target database here, check whether support has been added to the latest Hibernate release. The dialects are listed in the 'SQL Dialects' section of the Hibernate reference documentation. If that doesn't pan out, see if you can find a third-party effort to support the database, or consider starting your own!

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The Developer's Notebook series is modeled on the tradition of laboratory notebooks. Laboratory notebooks are an invaluable tool for researchers and their successors.

The purpose of a laboratory notebook is to facilitate the recording of data and conclusions as the work is being conducted, creating a faithful and immediate history. The notebook begins with a title page that includes the owner's name and the subject of research. The pages of the notebook should be numbered and prefaced with a table of contents. Entries must be clear, easy to read, and accurately dated; they should use simple, direct language to indicate the name of the experiment and the steps taken. Calculations are written out carefully and relevant thoughts and ideas recorded. Each experiment is introduced and summarized as it is added to the notebook. The goal is to produce comprehensive, clearly organized notes that can be used as a reference. Careful documentation creates a valuable record and provides a practical guide for future developers.

Colleen Gorman was the production editor and Marlowe Shaeffer was the proofreader for Hibernate: A Developer's Notebook. Mary Agner and Jamie Peppard provided production support. Claire Cloutier provided quality control. Tom Dinse wrote the index.

Edie Freedman designed the cover of this book. Emma Colby produced the cover layout with QuarkXPress 4.1 using the Officina Sans and JuniorHandwriting fonts.

Edie Freedman and David Futato designed the interior layout. This book was converted by Julie Hawks to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Adobe Boton; the heading font is ITC Officina Sans; the code font is LucasFont's TheSans Mono Condensed, and the handwriting font is a modified version of JRHand made by Tepid Monkey Fonts and modified by O'Reilly. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Colleen Gorman.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.