Hibernate

A Developer's Notebook™

James Elliott

O'REILLY®

CHAPTER 3

Harnessing Hibernate

In this chapter:

- *Creating Persistent Objects*
- *Finding Persistent Objects*
- *Better Ways to Build Queries*

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.

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.

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.

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.

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.

Example 3-1. *CreateTest.java*

```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
        // in the standard place.
        Configuration config = new Configuration();

        // Tell it about the classes we want mapped, taking advantage of
        // the way we've named their mapping documents.
        config.addClass(Track.class);

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory();

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession();
        Transaction tx = null;
        try {
            // Create some data and persist it
            tx = session.beginTransaction();

            Track track = new Track("Russian Trance",
                                    "vol2/album610/track02.mp3",
                                    Time.valueOf("00:03:30"), new Date(),
                                    (short)0);
            session.save(track);

            track = new Track("Video Killed the Radio Star",
                              "vol2/album611/track12.mp3",
                              Time.valueOf("00:03:49"), new Date(),
                              (short)0);
            session.save(track);


```

Example 3-1. *CreateTest.java* (continued)

```
             track = new Track("Gravity's Angel",
                                  "vol2/album175/track03.mp3",
                                  Time.valueOf("00:06:06"), new Date(),
                                  (short)0);
            session.save(track);

            // We're done; make our changes permanent
            tx.commit();

        } catch (Exception e) {
            if (tx != null) {
                // Something went wrong; discard all partial changes
                tx.rollback();
            }
            throw e;
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close();
    }
}
```

With all we've got in place by now it's quite easy to tell Ant how to run this test. Add the target shown in Example 3-2 right before the closing `</project>` tag at the end of *build.xml*.

Example 3-2. Ant target to invoke our data creation test

```
<target name="ctest" description="Creates and persists some sample data"
        depends="compile">
  <java classname="com.oreilly.hh.CreateTest" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

All right, we're ready to create some data! Example 3-3 shows the results of invoking the new `ctest` target. Its dependency on the `compile` target ensures the `CreateTest` class gets compiled before we try to use it. The output for `ctest` itself shows the logging emitted by Hibernate as the environment and mappings are set up and the connection is shut back down.

Example 3-3. Invoking the CreateTest class

```
% ant ctest
Buildfile: build.xml

prepare:
```

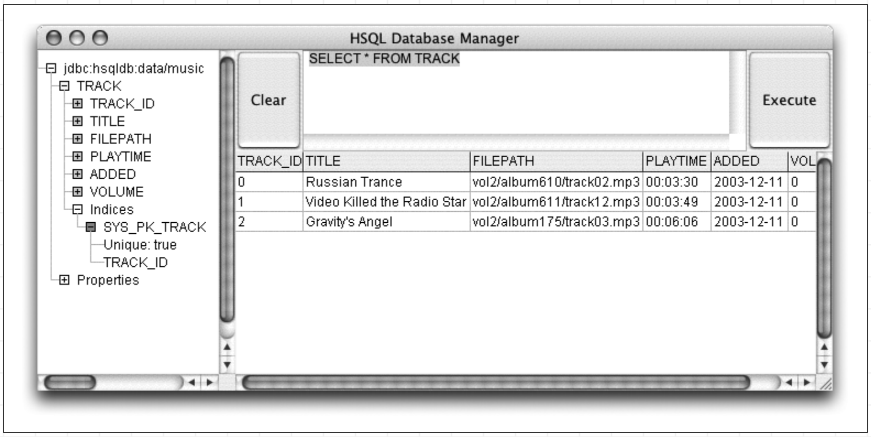
Example 3-3. Invoking the CreateTest class (continued)

```
compile:
    [javac] Compiling 1 source file to /Users/jim/Documents/Work/OReilly/
Hibernate/Examples/ch03/classes

ctest:
     [java] 00:07:46,376  INFO Environment:432 - Hibernate 2.1.1
     [java] 00:07:46,514  INFO Environment:466 - loaded properties from resource
hibernate.properties: {hibernate.connection.username=sa, hibernate.connection.
password=, hibernate.cglib.use_reflection_optimizer=true, hibernate.dialect=net.
sf.hibernate.dialect.HSQLDialect, hibernate.connection.url=jdbc:hsqldb:data/
music, hibernate.connection.driver_class=org.hsqldb.jdbcDriver}
     [java] 00:07:46,644  INFO Environment:481 - using CGLIB reflection optimizer
     [java] 00:07:46,691  INFO Configuration:318 - Mapping resource: com/oreilly/
hh/Track.hbm.xml
     [java] 00:07:50,686  INFO Binder:225 - Mapping class: com.oreilly.hh.Track
-> TRACK
     [java] 00:07:51,620  INFO Configuration:584 - processing one-to-many
association mappings
     [java] 00:07:51,627  INFO Configuration:593 - processing one-to-one
association property references
     [java] 00:07:51,628  INFO Configuration:618 - processing foreign key
constraints
     [java] 00:07:51,869  INFO Dialect:82 - Using dialect: net.sf.hibernate.
dialect.HSQLDialect
     [java] 00:07:51,886  INFO SettingsFactory:62 - Use outer join fetching: false
     [java] 00:07:51,966  INFO DriverManagerConnectionProvider:41 - Using
Hibernate built-in connection pool (not for production use!)
     [java] 00:07:52,036  INFO DriverManagerConnectionProvider:42 - Hibernate
connection pool size: 20
     [java] 00:07:52,117  INFO DriverManagerConnectionProvider:71 - using driver:
org.hsqldb.jdbcDriver at URL: jdbc:hsqldb:data/music
     [java] 00:07:52,135  INFO DriverManagerConnectionProvider:72 - connection
properties: {user=sa, password=}
     [java] 00:07:52,171  INFO TransactionManagerLookupFactory:33 - No
TransactionManagerLookup configured (in JTA environment, use of process level
read-write cache is not recommended)
     [java] 00:07:53,497  INFO SettingsFactory:89 - Use scrollable result sets:
true
     [java] 00:07:53,504  INFO SettingsFactory:99 - Query language substitutions: {}
     [java] 00:07:53,507  INFO SettingsFactory:110 - cache provider:
net.sf.ehcache.hibernate.Provider
     [java] 00:07:53,528  INFO Configuration:1057 - instantiating and configuring
caches
     [java] 00:07:54,533  INFO SessionFactoryImpl:119 - building session factory
     [java] 00:07:56,721  INFO SessionFactoryObjectFactory:82 - no JNDI name
configured
     [java] 00:07:57,357  INFO SessionFactoryImpl:527 - closing
     [java] 00:07:57,370  INFO DriverManagerConnectionProvider:137 - cleaning up
connection pool: jdbc:hsqldb:data/music

BUILD SUCCESSFUL
Total time: 23 seconds
```

What Just Happened?

Our test class fired up Hibernate, loaded the mapping information for the `Track` class, opened a persistence session to the associated HSQLDB database, and used that to create some instances and persist them in the `TRACK` table. Then it shut down the session and closed the database connection, ensuring the data was saved.

After running this test, you can use **`ant db`** to take a look at the contents of the database. You should find three rows in the `TRACK` table now, as shown in Figure 3-1. (Type your query in the text box at the top of the window and click the Execute button. You can get a command skeleton and syntax documentation by choosing Command → Select in the menu bar.)



```
SELECT * FROM TRACK
```

TRACK_ID	TITLE	FILEPATH	PLAYTIME	ADDED	VOL
0	Russian Trance	vol2/album610/track02.mp3	00:03:30	2003-12-11	0
1	Video Killed the Radio Star	vol2/album611/track12.mp3	00:03:49	2003-12-11	0
2	Gravity's Angel	vol2/album175/track03.mp3	00:06:06	2003-12-11	0

Figure 3-1. Test data persisted into the TRACK table

The first part of Example 3-1's code needs little explanation. Lines 3 and 4 import some useful Hibernate classes, including `Configuration`, which is used to set up the Hibernate environment. The `Time` and `Date` classes are used in our data objects to represent track playing times and creation timestamps. The only method we implement in `CreateTest` is the `main()` method that supports invocation from the command line.

When run, we start by creating a Hibernate `Configuration` object on line 17. Since we don't tell it otherwise, it looks for a file named *hibernate.properties* at the root level in the class path. It finds the one we created in the previous chapter (see Example 2-4), which tells it we're using HSQLDB, and how to find the database. Next line 21 requests mapping services for the `Track` class. Because we've placed the mapping file *Track.hbm.xml* in the same package, and followed the standard naming convention, Hibernate is able to find and load it without requiring an explicit path. This approach is particularly handy when you want to distribute your application as a Jar archive, or when you are operating in a web application environment.

That's all the configuration we need in order to create and persist track data, so we're ready to create the `SessionFactory` on line 24. Its purpose is to provide us with `Session` objects, the main avenue for interaction with Hibernate. The `SessionFactory` is thread safe, and you only need one for your entire application. (To be more precise, you need one for each database environment for which you want persistence services; most applications therefore need only one.) Creating the session factory is a pretty expensive and slow operation, so you'll definitely want to share it throughout your application. It's trivial in a one-class application like this one, but the reference documentation provides some good examples of ways to do it in more realistic scenarios.

When it comes time to actually perform persistence, we ask the `SessionFactory` to open a `Session` for us (line 27), which establishes a JDBC connection to the database, and provides us with a context in which we can create, obtain, manipulate, and delete persistent objects. As long as the session is open, a connection to the database is maintained, and changes to the persistent objects associated with the session are tracked so they can be applied to the database when the session is closed. Conceptually you can think of a session as a "large scale transaction" between the persistent objects and the database, which may encompass several database-level transactions. Like a database transaction, though, you should not think about keeping your Hibernate session open over long periods of application existence (such as while you're waiting for user input). A single session is used for a specific and bounded operation in the application, something like populating the user interface or making a change that has been committed by the user. The next operation will use a new session. Also note that `Session` objects are not thread safe, so they cannot be shared between threads. Each thread needs to obtain its own session from the factory.

It's worth getting a solid understanding of the purposes and lifecycles of these objects. This notebook gives you just enough information to get started; you'll want to spend some time with the reference documentation and understand the examples in depth.

We need to look more closely at the lifecycle of mapped objects in Hibernate, and how this relates to sessions, because the terminology is rather specific and the concepts are quite important. A mapped object such as an instance of our `Track` class moves back and forth between two states with respect to Hibernate: *transient* and *persistent*. An object that is transient is not associated with any session. When you first create a `Track` instance using `new()`, it is transient; unless you tell Hibernate to persist it, the object will vanish when it is garbage collected or your application terminates.

Passing a transient mapped object to a `Session`'s `save()` method causes it to become persistent. It will survive garbage collection and termination of the Java VM, staying available until it is explicitly deleted. (There is a related distinction between *entities* and *values* discussed at the beginning of Appendix A. Mapped objects that have been persisted are called entities, even if they do not currently exist as an instance in any virtual machine.) If you've got a persistent object and you call `Session`'s `delete()` method on it, the object transitions back to a transient state. The object still exists as an instance in your application, but it is no longer going to stay around unless you change your mind and save it again; it's ceased being an entity.

On the other hand, and this point is worth extra emphasis, if you haven't deleted an object (so it's still persistent), when you change its properties there is no need to save it again for those changes to be reflected in the database. Hibernate automatically tracks changes to any persistent objects and flushes those changes to the database at appropriate times. When you close the session, any pending changes are flushed.

Hang in there, we'll be back to the example soon!

An important but subtle point concerns the status of persistent objects you worked with in a session that has been closed, such as after you run a query to find all entities matching some criteria (you'll see how to do this in the upcoming section, "Finding Persistent Objects"). As noted above, you don't want to keep this session around longer than necessary to perform the database operation, so you close it once your queries are finished. What's the deal with the mapped objects you've loaded at this point? Well, they were persistent while the session was around, but once they are no longer associated with an active session (in this case because the session has been closed) they are not persistent any longer. Now, this doesn't mean that they no longer exist in the database; indeed, if you run the query again (assuming nobody has changed the data in the meantime), you'll get back the same set of objects; they're still entities. It simply means that there is not currently an active correspondence being maintained between the state of the objects in your virtual machine and the database. It is perfectly reasonable to carry on working with the objects. If you later need to make changes to the objects and you want the changes to "stick," you will open a new session and use it to save the changed objects. Because each entity has a unique ID, Hibernate has no problem figuring out how to link the transient objects back to the appropriate persistent state in the new session.

TIP

Of course, as with any environment in which you're making changes to an offline copy of information backed by a database, you need to think about application-level data integrity constraints. You may need to devise some higher-level locking or versioning protocol to support them. Hibernate can offer help with this task too, but the design and detailed implementation is up to you. The reference manual does strongly recommend the use of a version field, and there are several approaches available.

Armed with these concepts and terms, the remainder of the example is easy enough to understand. Line 31 sets up a database transaction using our open session. Within that, we create a few `Track` instances containing sample data and save them in the session (lines 33–50), turning them from transient instances into persistent entities. Finally, line 53 commits our transaction, atomically (as a single, indivisible unit) making all the database changes permanent. The `try`/`catch`/`finally` block wrapped around all this shows an important and useful idiom for working with transactions. If anything goes wrong, lines 56–60 will roll back the transaction and then bubble out the exception, leaving the database the way we found it. The session is closed in the `finally` portion at line 63, ensuring that this takes place whether we exit through the "happy path" of a successful commit, or via an exception that caused rollback. Either way, it gets closed as it should.

At the end of our method we also close the session factory itself on line 67. This is something you'd do in the "graceful shutdown" section of your application. In a web application environment, it would be in the appropriate lifecycle event handler.* In this simple example, when the `main()` method returns, the application is ending.

* If you're not familiar with these, read about the `ServletContextListener` interface in the Servlet 2.3 specification.

WARNING

One pitfall I discovered in working with this test program is that it is necessary to explicitly close both the `SessionFactory` and `Session`. If you don't, the application won't exit when `main()` returns, and you have to kill it with Ctrl-C. This turns out to be caused by a non-daemon thread used in the current HSQLDB logging implementation. The HSQLDB developers have said they may fix this in the next release, in which logging is being heavily revamped. Even if they do, similar issues may exist with other databases; it's better to be safe and close things gracefully when you know you're done using them.

At this point it's worth pausing a moment to reflect on the fact that we wrote no code to connect to the database or to issue SQL commands. Looking back to the preceding chapter, we didn't even have to create the table ourselves, nor the `Track` object that encapsulates our data. Yet the query in Figure 3-1 shows nicely readable data representing the Java objects created and persisted by our short, simple test program. Hopefully you'll agree that this reflects very well on the power and convenience of Hibernate as a persistence service. For being free and lightweight, Hibernate can certainly do a lot for you, quickly and easily.

WARNING

Although the Hibernate reference documentation states that the use of transactions is optional, I found that (at least through Version 2.1.1, with the HSQLDB database) any changes I made outside the context of a Hibernate transaction would disappear when the application ended. Analyzing the SQL emitted by Hibernate revealed that even though I didn't request a transaction, auto-commit was being turned off, so my changes were getting rolled back when the application ended. So for now it's always necessary to use explicit transactions in your code, a good habit in any case.

An alternate way of seeing the results of our persistence test is to simply look at the database itself. Since we're using HSQLDB, the database is stored in a highly human-readable format: the file *music.script* contains a series of SQL statements that are used to reconstruct the in-memory database structures when the database is opened. The end of the file contains our persisted objects, as shown in Example 3-4.

Example 3-4. Looking at the raw database file

```
% tail data/music.script
CREATE ALIAS ATAN FOR "java.lang.Math.atan"
CREATE ALIAS UPPER FOR "org.hsqldb.Library.ucase"
CREATE ALIAS ASCII FOR "org.hsqldb.Library.ascii"
CREATE ALIAS RAND FOR "java.lang.Math.random"
CREATE ALIAS LENGTH FOR "org.hsqldb.Library.length"
CREATE ALIAS ROUND FOR "org.hsqldb.Library.round"
CREATE ALIAS REPLACE FOR "org.hsqldb.Library.replace"
INSERT INTO TRACK VALUES(0,'Russian Trance','vol2/album610/track02.mp3',
'00:03:30','2003-12-11',0)
INSERT INTO TRACK VALUES(1,'Video Killed the Radio Star','vol2/album611/
track12.mp3','00:03:49','2003-12-11',0)
INSERT INTO TRACK VALUES(2,'Gravity''s Angel','vol2/album175/track03.mp3',
'00:06:06','2003-12-11',0)
```

The final three statements show our TRACK table rows. The aliases that come before them are part of the normal HSQLDB environment you get when creating a new database.

Tempted to learn more about HSQLDB? I won't try to stop you!

What About...

...Objects with relationships to other objects? Collections of objects? You're right, these are cases where persistence gets more challenging (and, if done right, valuable). Hibernate can handle associations like this just fine. In fact, there isn't any special effort involved on our part. We'll discuss this in Chapter 4. For now, let's look at how to retrieve objects that were persisted in earlier sessions.

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.

How Can 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*

```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)
        throws HibernateException
    {
        return session.find("from com.oreilly.hh.Track as track " +
                            "where track.playTime <= ?",
                            length, Hibernate.TIME);
    }

    /**
     * Look up and print some tracks when invoked from the command line.
     */
    public static void main(String args[]) throws Exception {
        // Create a configuration based on the properties file we've put
        // in the standard place.
        Configuration config = new Configuration();

        // Tell it about the classes we want mapped, taking advantage of
        // the way we've named their mapping documents.
        config.addClass(Track.class);

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory();

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession();
        try {
            // Print the tracks that will fit in five minutes
            List tracks = tracksNoLongerThan(Time.valueOf("00:05:00"),
                                             session);
            for (ListIterator iter = tracks.listIterator() ;
                 iter.hasNext() ; ) {
                Track aTrack = (Track)iter.next();
```

Example 3-5. *QueryTest.java* (continued)

```
                System.out.println("Track: \"" + aTrack.getTitle() +
                                   "\", " + aTrack.getPlayTime());
            }
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close();
    }
}
```

Again, add a target (Example 3-6) at the end of *build.xml* to run this test.

Example 3-6. Ant target to invoke our query test

```
<target name="qtest" description="Run a simple Hibernate query"
        depends="compile">
  <java classname="com.oreilly.hh.QueryTest" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

With this in place, we can simply type **`ant qtest`** to retrieve and display some data, with the results shown in Example 3-7. To save space in the output, we've edited our *log4j.properties* to turn off all the "Info" messages, since they're no different than in the previous example. You can do this yourself by changing the line:

```
    log4j.logger.net.sf.hibernate=info
```

to replace the word `info` with `warn`:

```
    log4j.logger.net.sf.hibernate=warn
```

Example 3-7. Running the query test

```
% ant qtest
Buildfile: build.xml

prepare:

compile:
    [javac] Compiling 1 source file to /Users/jim/Documents/Work/OReilly/
Hibernate/Examples/ch03/classes

qtest:
     [java] Track: "Russian Trance", 00:03:30
     [java] Track: "Video Killed the Radio Star", 00:03:49

BUILD SUCCESSFUL
Total time: 11 seconds
```

What Just Happened?

If you don't want to deal with any query syntax at all, check out criteria queries in Chapter 8!

In Example 3-5, we started out by defining a utility method, `tracksNoLongerThan()` on lines 14–28, which performs the actual Hibernate query. It retrieves any tracks whose playing time is less than or equal to the amount specified as a parameter. Hibernate queries are written in HQL (Hibernate Query Language—explored in Chapter 9). This is a SQL-inspired but object-oriented query language with important differences from SQL. HQL is explored in Chapter 9. For now, note that it supports parameter placeholders, much like `PreparedStatement` in JDBC. And, just like in that environment, using them is preferable to putting together queries through string manipulation. As you'll see below, however, Hibernate offers even better ways of working with queries in Java.

The query itself looks a little strange. It starts with "`from`" rather than "`select` *something*" as you might expect. While you can certainly use the more familiar format, and will do so when you want to pull out individual properties from an object in your query, if you want to retrieve entire objects you can use this more abbreviated syntax.

When you're working with pre-existing databases and objects, it's important to realize that HQL queries refer to object properties rather than database table

Also note that the query is expressed in terms of the mapped Java *objects* and *properties* rather than the tables and columns. Since the object being used in the query is in the same package as the mapping document, we don't actually need to use a fully qualified class name like this, but it helps remind readers of the query that it's expressed in terms of objects. Keeping the names consistent is a fairly natural choice, and this will always be the case when you're using Hibernate to generate the schema and the data objects, unless you tell it explictly to use different column names.

Also, in HQL—just as in SQL—you can alias a column or table to another name; in fact, you *must* use aliases to refer to specific JavaBeans properties of mapped classes in your `where` clauses.

With all that in mind, the meaning of the HQL query "`from com.oreilly.hh.Track as track where track.playTime <= ?`" breaks down like this:

- We're retrieving instances of our persistent Java data bean `Track`. (This happens to be stored in the database table called `TRACK`, but that is managed by the mapping document for class `Track`; the table could be called anything.) Since we started the query with the `from` clause (rather than a `select` clause), we want to obtain entire `Track` instances as the result of the query.
- Within the query, we are defining an alias "`track`" to refer to one particular instance of the `Track` class at a time, so we can specify the properties of the instances we'd like to retrieve.

- The instances we want are those where the `playTime` property has a value that's less than or equal to the first (and only) query parameter we're passing in at runtime. (Once again, the fact that the `playTime` property is mapped to the `PLAYTIME` column is a detail that is buried in the `Track` mapping document, and it need not be true.)

HQL reads a lot like SQL, but the semantics are different—they refer to the world of the Java program rather than the world of the relational database. This will be more clear in later examples and in Chapter 9 where we perform more complex queries.

The rest of the program should look mighty familiar from the previous example. Our `try` block is simplified because we don't need a transaction, as we're not changing any data. We still use one so that we can have a `finally` clause to close our session cleanly. The body is quite simple, calling our query method to request any tracks whose playing time is five minutes or less, and then iterating over the resulting `Track` objects, printing their titles and playing times.

What About...

...Deleting objects? If you've made changes to your data creation script and want to start with a "clean slate" in the form of an empty database so you can test them, all you need to do is run **`ant schema`** again. This will drop and recreate the `Track` table in a pristine and empty state. Don't do it unless you mean it!

If you want to be more selective about what you delete, you can either do it through SQL commands in the HSQLDB UI (**`ant db`**), or you can make a variant of the query test example that retrieves the objects you want to get rid of. Once you've got a reference to a persistent object, passing it to the `Session`'s `delete()` method will remove it from the database:

```
session.delete(aTrack);
```

You've still got at least one reference to it in your program, until `aTrack` goes out of scope or gets reassigned, so conceptually the easiest way to understand what `delete()` does is to think of it as turning a persistent object back into a transient one.

Another way to use the `delete()` method is to pass it an HQL query string that matches multiple objects. This lets you delete many persisted objects at once, without writing your own loop. A Java-based alternative to **`ant schema`**, and a slightly less violent way of clearing out all the tracks, would therefore be something like this:

```
session.delete("from com.oreilly.hh.Track");
```

TIP

Don't forget that regardless of which of these approaches you use, you'll need to wrap the data manipulation code inside a Hibernate transaction, and commit the transaction if you want your changes to "stick."

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.

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.

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.

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();
}
```

Named parameters are identified within the query body by prefixing them with a colon. Here, we've changed the "?" to "`:length`". The `Session` object provides a `createQuery()` method that gives us back an implementation of the `Query` interface with which we can work. `Query` has a full complement of type-safe methods for setting the values of named parameters. Here we are passing in a `Time` value, so we use `setTime()`. Even in a simple case like this, the syntax is more natural and readable than the original version of our query. If we had been passing in anonymous arrays of values and types (as would have been necessary with more than one parameter), the improvement would be even more significant. And we've added a layer of compile-time type checking, always a welcome change.

Running this version produces the same output as our original program.

So how do we get the query text out of the Java source? Again, this query is short enough that the need to do so isn't as pressing as usual in real projects, but it's the best way to do things, so let's start practicing! As you may have predicted, the place we can store queries is inside the mapping document. Example 3-9 shows what it looks like (we have to use the somewhat clunky `CDATA` construct since our query contains characters—like "<"—that could otherwise confuse the XML parser).

Example 3-9. Our query in the mapping document

```
<query name="com.oreilly.hh.tracksNoLongerThan">
  <![CDATA[
      from com.oreilly.hh.Track as track
```

Example 3-9. Our query in the mapping document (continued)

```
      where track.playTime <= :length
    ]]>
</query>
```

Put this just after the closing tag of the class definition in *Track.hbm.xml* (right before the `</hibernate-mapping>` line). Then we can revise *QueryTest.java* one last time, as shown in Example 3-10. Once again, the program produces exactly the same output as the initial version. It's just better organized now, and we're in great shape if we ever want to make the query more complex.

Example 3-10. The final version of our query method

```
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();
}
```

The `Query` interface has other useful capabilities beyond what we've examined here. You can use it to control how many rows (and which specific rows) you retrieve. If your JDBC driver supports scrollable `ResultSet`s, you can access this capability as well. Check the JavaDoc or the Hibernate reference manual for more details.

What About...

...Avoiding a SQL-like language altogether? Or diving in to HQL and exploring more complex queries? These are both options that are covered later in the book.

Chapter 8 discusses criteria queries, an interesting mechanism that lets you express the constraints on the entities you want, using a natural Java API. This gives you the benefits of compile-time syntax checking, and easy dynamic configuration, all in the language you're already using to code your application. It also supports a form of "query by example," where you can supply objects that are similar to the ones you're searching for.

SQL veterans who'd like to see more tricks with HQL can jump to Chapter 9, which explores more of its capabilities and unique features.

For now, we'll continue our examination of mapping by seeing how to represent groups and links between objects.