2 Introducing and integrating Hibernate

This chapter covers

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

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

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

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

2.1 "Hello World" with Hibernate

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The code fragment prints

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

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

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

Listing 2.2 A simple Hibernate XML mapping

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

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

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

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

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

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

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

Listing 2.3 Updating a message

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

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

This code calls three SQL statements inside the same transaction:

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

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

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

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

If we run "Hello World" again, it prints

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

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