Spring Persistence with Hibernate

Build robust and reliable persistence solutions for your enterprise Java application

Ahmad Reza Seddighi

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Spring Persistence with Hibernate

First published: November 2009

Production Reference: 1201109

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849510-56-1

`www.packtpub.com`

Cover Image by Vinayak Chittar (`vinayak.chittar@gmail.com`)

Credits

Author
Ahmad Reza Seddighi

Reviewer
Luca Masini

Acquisition Editor
Sarah Cullington

Development Editor
Rakesh Shejwal

Technical Editor
Pallavi Kachare

Indexer
Hemangini Bari

Editorial Team Leader
Akshara Aware

Project Team Leader
Priya Mukherji

Project Coordinator
Zainab Bagasrawala

Proofreader
Joel T. Johnson

Graphics
Nilesh Mohite

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Ahmad Reza Seddighi is an author, speaker, and consultant in architecting and developing enterprise software systems. He is an IT graduate from the University of Isfahan, Iran, and has ten years experience in software development. He currently lives in Tehran, where he works with a number of small but growing IT companies. He loves teaching so he grabs any teaching opportunities. He is also the author of three other books: Core Java Programming, Java Web Development, and Open Source J2EE Development, all in Farsi.

As the book was going to publish by Apress, and prepared and made ready by that, I'd like to thank the Apress crew at first. In particular, Steve Anglin, acquisitions editor; Kylie Johnston, project manager; Matthew Moodie, reviewing editor; Sumit Pal, technical reviewer; Nancy Holzner, copyeditor; Elizabeth Berry, production editor; Brenda Miller, indexer; Gina Rexrode, compositor; April Eddy, proofreader; and April Milne, artist.

I should also thank the Packt crew, Sarah Cullington, acquisition editor; Priya Mukherji, project team leader; Rakesh Shejwal, development editor; Luca Massini, technical reviewer; Pallavi Kachare, technical editor; Zainab Bagasrawala, project coordinator; Akshara Aware, editorial team leader; Joel T. Johnson, proofreader; Nilesh Mohite, graphic organizer; Shantanu Zagade, production coordinator; Hemangini Bari, indexer; and Shantanu Zagade, cover work.

Special thanks to Sepehr Fatemi for his support in writing and reviewing the book, and sincere regards to my family, and specially my brother, Rohollah, who has always offered excellent support and encouragement.

In the end, I express my gratitude to all my friends, whom I've not named here and who have always supported me. Thank you all.

About the Reviewer

Luca Masini is a Senior Software Engineer and Architect, born as a game developer for Commodore 64 (Footbal Manager) and Commodore Amiga (Ken il guerriero), soon he converted to Object-Oriented programming and for that, from his beginning in 1995, he was attracted by the Java language.

He worked on this passion as a consultant for the major Italian banks, developing and integrating the main software projects for which he has often taken the technical leadership. He was able to lead adoption of Java Enterprise in an environment where COBOL was the flagship platform, converting them from mainframe centric to distributed.

He then shifted his eyes toward open-source, starting from Linux and then with enterprise frameworks, with which he was able to introduce, with low impact, some concept like IoC, ORM, MVC. For that he was an early adopter of Spring, Hibernate, Struts, and an entire host of other technologies that in the long run have given his customers a technological advantage, and therefore development costs cuts.

Lately, however, his attention is completely directed towards the simplification and standardization of development with Java EE, and for this he is working at the ICT of a large Italian company to introduce advanced build tools (Maven and Continuous Integration), archetypes of project and "Agile Development" with plain standards.

Dedicated to my friend Enzo, and our common passion.

To my parents

Table of Contents

Preface

Hibernate is a popular open-source Java framework. It aims to solve problems associated with persistence in the Java world. Whether you are developing a simple stand-alone application, or a full-blown, server-side Java EE application, you can use and benefit from Hibernate. Although Hibernate has competitors, no other persistence framework is as flexible and as easy to learn.

Spring is another popular framework. It aims to simplify Java development in many areas, including persistence. However, Spring does not provide a persistence framework similar to Hibernate. Instead, it provides an abstraction layer over Hibernate to offer more flexibility, produce more effective code, and reduce maintenance costs.

What this book covers

Chapter 1, *An Introduction to Hibernate and Spring* introduces Spring and Hibernate, explaining what persistence is, why it is important, and how it is implemented in Java applications. It provides a theoretical discussion of Hibernate and how Hibernate solves problems related to persistence. Finally, we take a look at Spring and the role of Spring in persistence.

Chapter 2, *Preparing an Application to Use Spring with Hibernate* guides you, step-by-step, down the path of preparing your application to use Hibernate and Spring. The prerequisites to developing with Hibernate and Spring, including getting Hibernate and Spring distributions, setting up a database, and adding extra tools and frameworks to your application, are all discussed here.

Chapter 3, *A Quick Tour of Hibernate with Spring* provides a quick tour of developing with Hibernate and Spring. Here, a simple example illustrates the basic concepts behind Hibernate and Spring.

Chapter 4, *Hibernate Configuration* shows you how to configure and set up Hibernate. It discusses the basic configuration settings that are always required in any application. (Some optional settings are covered in the book's appendix.)

Chapter 5, *Hibernate Mappings* explains basic issues related to persistent objects and their mappings. It starts with basic mapping concepts and then moves on to advanced and practical issues.

Chapter 6, *More on Mappings* continues the mapping discussion with some advanced mapping topics. It explains how to map complex objects and create complicated mapping files.

Chapter 7, *Hibernate Types* discusses how Hibernate types help to define which Java types are mapped to which SQL database types. It explores the built-in Hibernate types. It also looks at custom type implementation when these built-in types do not satisfy the application's requirements, or when you want to change the default behavior of a built-in type.

Chapter 8, *Hibernate Persistence Behavior* discusses the life cycle of persistent objects within the application's lifetime. This chapter explains the basic persistence operations provided by the Session API at the heart of the Hibernate API. The chapter also discusses how persistence operations are cascaded between persistent objects, and how cascading behavior is defined in mapping files

Chapter 9, *Querying in Hibernate* explains the different approaches that Hibernate provides for querying persistent objects. It investigates HQL, a Hibernate-specific query language; native SQL, a database-relevant query language; and the Criteria API, a Hibernate API to express query statements.

Chapter 10, *Inversion of Control with Spring* starts developing with Spring, introducing the Inversion of Control (IoC) pattern that is implemented at the heart of Spring.

Chapter 11, *Spring AOP* investigates Aspect-Oriented Programming (AOP) as another Spring feature. Here, you'll learn what AOP means, how AOP simplifies application architecture, and how to implement AOP in Spring.

Chapter 12, *Transaction Management* discusses transaction management. It explains transaction concepts and how transactions are managed in native and Spring-based Hibernate applications. It also discusses caching as a persistence aspect that involves reliability of data manipulation.

Chapter 13, *Integrating Hibernate with Spring* explains how Hibernate and Spring are integrated and introduces the Data Access Object (DAO) pattern. It shows how Spring and Hibernate combine to implement this pattern.

Chapter 14, *Web Development with Hibernate and Spring* provides a quick discussion of web development with Spring and Hibernate. It does not provide a detailed discussion of web development. Instead, it takes Spring and Struts as sample web frameworks to illustrate how you might use Spring and Hibernate to develop web applications.

Chapter 15, *Testing* looks into testing persistence code, with a focus on unit testing. It introduces JUnit as an open-source unit-testing framework and discusses which aspects of persistence code with Hibernate require testing.

Appendix, *Hibernate's Advanced Features* looks at some advanced Hibernate topics, including some useful Hibernate properties, the event/listener model implemented by Hibernate, and Hibernate filters.

What you need for this book

In this book, I assume that you have a good understanding of the Java programming language, preferably version 1.5 or later, including the Java syntax and basic APIs. You are also expected to have a basic understanding of the JDBC API, relational databases, and the SQL query language. For Chapters 14, you should have a basic understanding of web development with Java, including HTML, JSP, servlets, and a web container such as Tomcat.

Who this book is for

The book is primarily for Spring developers and users who want to persist using the popular Hibernate persistence framework. Java, Hibernate, JPA, Spring, and open source developers in general will also find the book useful.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "For instance, the `==` operator can be used as follows to check whether `object1` and `object2` are identical."

A block of code is set as follows:

```
package com.packtpub.springhibernate.ch01;

import java.util.Date;

public class Student {
  private int id;
  private String firstName;
  private String lastName;
  private String ssn;
  private Date birthdate;
  private String stdNo;
  private Date entranceDate ;

}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
package com.packtpub.springhibernate.ch01;

public class Course {
  private int id;
  private String courseName;
  private int units;

  //setter and getter methods
}
```

Any command-line input or output is written as follows:

```
[Method=setId|Old Value=0|New Value=41]

[Method=setFirstName|Old Value=null|New Value=John]
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "A **relational database** is an application that provides the persistence service".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit http://www.packtpub.com/files/code/0561_Code.zip to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
An Introduction to Hibernate and Spring

Hibernate and Spring are open-source Java frameworks that simplify developing Java/JEE applications from simple, stand-alone applications running on a single JVM, to complex enterprise applications running on full-blown application servers. Hibernate and Spring allow developers to produce scalable, reliable, and effective code. Both frameworks support declarative configuration and work with a **POJO (Plain Old Java Object)** programming model (discussed later in this chapter), minimizing the dependence of application code on the frameworks, and making development more productive and portable.

Although the aim of these frameworks partially overlap, for the most part, each is used for a different purpose. The Hibernate framework aims to solve the problems of managing data in Java: those problems which are not fully solved by the Java persistence API, **JDBC (Java Database Connectivity)**, persistence providers, **DBMS (Database Management Systems)**, and their mediator language, **SQL (Structured Query Language)**.

In contrast, Spring is a multitier framework that is not dedicated to a particular area of application architecture. However, Spring does not provide its own solution for issues such as persistence, for which there are already good solutions. Rather, Spring unifies preexisting solutions under its consistent API and makes them easier to use. As mentioned, one of these areas is persistence. Spring can be integrated with a persistence solution, such as Hibernate, to provide an abstraction layer over the persistence technology, and produce more portable, manageable, and effective code.

Furthermore, Spring provides other services spread over the application architecture, such as inversion of control and aspect-oriented programming (explained later in this chapter), decoupling the application's components, and modularizing common behaviors.

This chapter looks at the motivation and goals for Hibernate and Spring. The chapter begins with an explanation of why Hibernate is needed, where it can be used, and what it can do. We'll take a quick look at Hibernates alternatives, exploring their advantages and disadvantages. I'll outline the valuable features that Hibernate offers and explain how it can solve the problems of the traditional approach to Java persistence. The discussion continues with Spring. I'll explain what Spring is, what services it offers, and how it can help to develop a high-quality data-access layer with Hibernate.

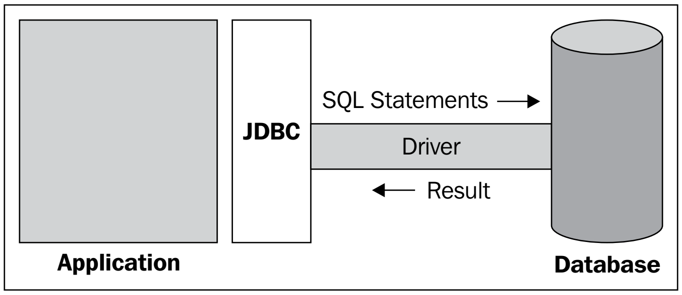
If you already have enough background to start learning Hibernate and Spring, you can skip this chapter and jump to the next one.

Persistence management in Java

Persistence has long been a challenge in the enterprise community. Many persistence solutions from primitive, file-based approaches, to modern, object-oriented databases have been presented. For any of these approaches, the goal is to provide reliable, efficient, flexible, and scalable persistence.

Among these competing solutions, relational databases (because of certain advantages) have been most widely accepted in the IT world. Today, almost all enterprise applications use relational databases. A **relational database** is an application that provides the persistence service. It provides many persistence features, such as indexing data to provide speedy searches; solves the relevant problems, such as protecting data from unauthorized access; and handles many complications, such as preserving relationships among data. Creating, modifying, and accessing relational databases is fairly simple. All such databases present data in two-dimensional tables and support SQL, which is relatively easy to learn and understand. Moreover, they provide other services, such as transactions and replication. These advantages are enough to ensure the popularity of relational databases.

To provide support for relational databases in Java, the JDBC API was developed. JDBC allows Java applications to connect to relational databases, express their persistence purpose as SQL expressions, and transmit data to and from databases. The following screenshot shows how this works:



Using this API, SQL statements can be passed to the database, and the results can be returned to the application, all through a driver.

The mismatch problem

JDBC handles many persistence issues and problems in communicating with relational databases. It also provides the needed functionality for this purpose. However, there remains an unsolved problem in Java applications: Java applications are essentially object-oriented programs, whereas relational databases store data in a relational form. While applications use object-oriented forms of data, databases represent data in two-dimensional table forms. This situation leads to the so-called object-relational paradigm mismatch, which (as we will see later) causes many problems in communication between object-oriented and relational environments.

For many reasons, including ease of understanding, simplicity of use, efficiency, robustness, and even popularity, we may not discard relational databases. However, the mismatch cannot be eliminated in an effortless and straightforward manner.

Identity and equality mismatch

The first and most significant mismatch involves the concepts of data equality. Java provides two definitions for object identity and equality. According to Java, two objects are called identical when they point to the same reference in memory. In contrast, two objects are considered equal when they contain similar data, as determined by the developer, regardless of the memory locations to which the objects point.

Java offers the `equals()` method and `==` operator to support equality and identity, respectively. For instance, the `==` operator can be used as follows to check whether `object1` and `object2` are identical:

```
object1==object2
```

The `equals()` method, used as follows, determines whether two objects are equal:

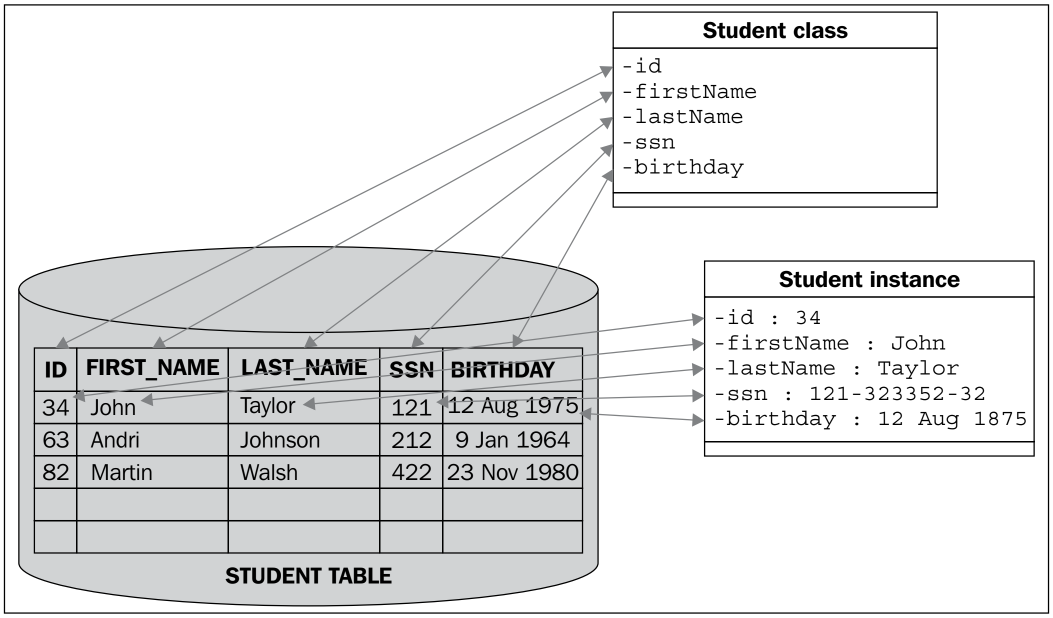
```
object1.equals(object2)
```

When two objects are identical, they refer to the same memory location. Therefore, they have the same value and are definitely equal. However, two objects that are equal may not be identical since they may point to different locations in memory.

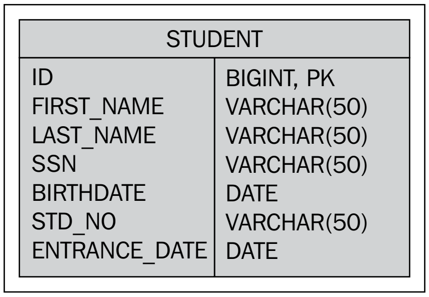
> The `hashCode()` method must be overridden in every class which overrides the `equals()` method. The `hashCode()` method returns an integer as the hash code value for the object on which this method is invoked. This code is supported for the benefit of hashing based collection classes such as `Hashtable`, `HashMap`, `HashSet`, and so on. Equal objects must produce the same hash code, as long as they are equal. However, unequal objects do not need to produce distinct hash codes. See the JDK documentation for more details.

While Java offers two distinct definitions for object identity and equality, databases do not have any corresponding definitions for these terms. In a database, the data is represented as table rows, and each table row is identified based on the content it holds. A significant mismatch occurs when we map from the object-oriented world to the relational world. Although two objects are not identical because they refer to different locations in memory, in the database, they may be considered identical because they hold the same content.

The common approach to eliminating this mismatch is to use an extra field in the object's class, and an extra identifier column in the respective table of the object. This approach identifies objects based on the identifier values they hold in either object or relational form, instead of identifying them based on their references in memory (in the object-oriented world) and based on the content they hold (and in the relational world). Therefore, as the following screenshot shows, each object and its corresponding row in the table can be identified through the same strategy, that is, by considering the identifier value:



Let's look at a simple example. Suppose that our application has a `Student` class. We may typically use a `STUDENT` table in the database to store `Student` objects. We may also use an extra field in the class, called an object identifier, and a corresponding column in the table, called primary key, to allow objects to be recognized when they are stored, retrieved, or updated. The following screenshot shows the `STUDENT` table:



STUDENT	
ID	BIGINT, PK
FIRST_NAME	VARCHAR(50)
LAST_NAME	VARCHAR(50)
SSN	VARCHAR(50)
BIRTHDATE	DATE
STD_NO	VARCHAR(50)
ENTRANCE_DATE	DATE

This table would hold the objects of the class shown in the following code listing. Note that, we have just shown the skeleton of the class with its properties and without its other details:

```
package com.packtpub.springhibernate.ch01;

import java.util.Date;

public class Student {
  private int id;
  private String firstName;
  private String lastName;
  private String ssn;
  private Date birthdate;
  private String stdNo;
  private Date entranceDate ;

}
```

We may also use the following JDBC code in our application to store a `Student` object in the `STUDENT` table:

```
Connection c = null;
PreparedStatement p = null;

try {
  Student std = …//a ready to store Student object
  c = …//obtaining a Connection object
  String q = "INSERT INTO STUDENT " +
    "(ID, FIRST_NAME, LAST_NAME, SSN, BIRTHDATE, STD_NO,
      ENTRANCE_DATE)" +
    " VALUES (?, ?, ?, ?, ?, ?, ?)";
  p = c.prepareStatement(q);
  p.setInt(1, std.getId());
  p.setString(2, std.getFirstName());
  p.setString(3, std.getLastName());
  p.setString(4, std.getSsn());
  p.setDate(5, new java.sql.Date(std.getBirthdate().getTime()));
  p.setString(6, std.getSsn());
  p.setDate(7, new java.sql.Date(std.getEntranceDate().getTime()));

  p.executeUpdate();
} catch(Exception ex) {
  //handling the exception
} finally {
  if (p != null) {
    try {
      p.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
  if (c != null) {
    try {
      c.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
}
```

As you can see, to store a `Student` object, we need to obtain a `PreparedStatement` object with an appropriate SQL query. We then need to put the properties of the `Student` object in the `PreparedStatement`, and finally execute the update query by calling the `executeUpdate()` method of `PreparedStatement`. With all of these, we should handle exceptions when there is any problem in communicating to the database or executing the query.

Similarly, we may use code such as the following to load a `Student` object:

```
Connection c = null;
PreparedStatement p = null;
Student std = null;
try {
  int id = ...//the identifier of the student to  load
  c = …//obtaining a Connection object
  String q = "SELECT FIRST_NAME, LAST_NAME, SSN, BIRTHDATE, STD_NO,
  ENTRANCE_DATE " + "FROM STUDENT WHERE ID = ?";
  p = c.prepareStatement(q);
  p.setInt(1, id);
  ResultSet rs = p.executeQuery();
  if (rs.next()) {
    String firstName = rs.getString("FIRST_NAME");
    String lastName = rs.getString("LAST_NAME");
    String ssn = rs.getString("SSN");
    Date birthdate = rs.getDate("BIRTHDATE");
    String stdNo = rs.getString("STD_NO");
    Date entranceDate = rs.getDate("ENTRANCE_DATE");
    std = new Student(id, firstName, lastName, ssn, birthdate, stdNo,
    entranceDate);
    if (rs.next()) {
            //write a message warning about multiple objects
    }
  }
} catch(Exception ex) {
  //handling the exception
} finally {
  if (p != null) {
    try {
      p.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
  if (c != null) {
    try {
      c.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
}
return std;
```

To load an object from the database, we need to obtain a `PreparedStatment` with an appropriate SQL query, set the required values, execute the query, iterate over the result, and produce the `Student` object. We should also handle the exceptions correspondingly.

> The most difficult and important parts of the above codes are the SQL statements. These statements are expressed in a different language than the language that object-oriented languages such as Java use. Besides, their syntax can't be checked in the compiled time since they are expressed in the raw `String`.

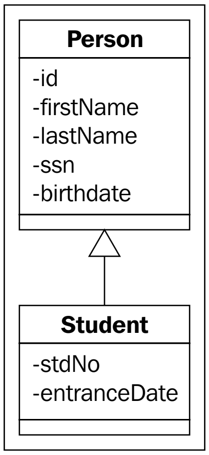
As it can be concluded from both the previous examples, converting the object-oriented and relational forms of data to one another cannot be accomplished in an effortless and straightforward manner. After all, any change in the object model or the database schema can affect the converting code.

This example demonstrates a primary difficulty in mapping objective and relational forms of data. However, this simple case requires only minimal effort to map a typical entity object. The mapping of objects could easily be more complicated than in the example, especially when an entity object is inherited from or associated with another object. Here, we are not going to discuss issues behind persisting inheritance and such in detail. However, a quick look at how pure JDBC deals with them offers insight into how Hibernate can ease object mapping, as we'll see in this book.

Mapping object inheritance mismatch

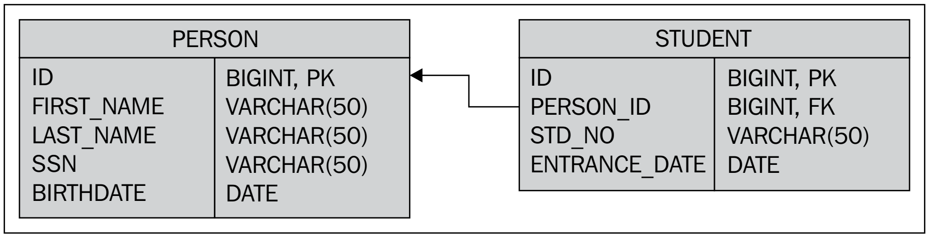
Another point where a mismatch occurs is in the mapping of object inheritance. While Java lets an object be inherited by another object, relational databases do not support the notion of inheritance. This means we need to define our own strategy to translate class hierarchy to database schema.

Let's elaborate inheritance by extending our simple example using a general class, `Person`, as a superclass of `Student`. The class diagram shown in the following screenshot shows the `Student` class, which is now a subclass of `Person`:



The question here is, how can the object inheritance be persisted? Can it be persisted in one table? If not, how should we establish the relationship between tables?

One solution would be to use individual tables for individual classes in the hierarchy. According to this solution, the objects of type superclass are stored directly in the superclass's table, but the subclass objects are persisted in both superclass and subclass tables. If we chose this strategy for our example, we would have two tables, `PERSON` and `STUDENT`, as shown in the following figure:



Because objects of the subclass are spread over two tables, we need a mechanism to recognize the association between rows when an object is retrieved, updated, or removed. Fortunately, many databases support foreign keys which are used to establish a relationship between database tables. This is what I have used in our example. As you can see, the `STUDENT` table takes the ID column as its primary key and a foreign key onto the `PERSON` table, meaning it holds the identifier of the associated row in the `STUDENT` table. When a `Student` object is stored, updated, or removed, the relevant student's data should be inserted in, updated in, or removed from the two tables. Moreover, to load a `Student` object, we need to query both tables with SQL joins, like this:

```
Connection c = null;
PreparedStatement p = null;
Student std = null;
try {
  int id =...//the identifier of the student that is loaded
  c = …//obtaining a Connection object
  c.setAutoCommit(false);
  String q = "SELECT P.FIRST_NAME, P.LAST_NAME, P.SSN, " +
             "P.BIRTHDATE, S.STD_NO,
              S.ENTRANCE_DATE "+
             "FROM PERSON P INNER JOIN STUDENT S " +
              " ON P.ID = S.PERSON_ID WHERE P.ID=?; ";
  p = c.prepareStatement(q);
  p.setInt(1, id);
  ResultSet rs = p.executeQuery();
  if (rs.next()) {
    String firstName = rs.getString("FIRST_NAME");
    String lastName = rs.getString("LAST_NAME");
    String ssn = rs.getString("SSN");
    Date birthday = rs.getDate("BIRTHDATE");
    String stdNo = rs.getString("STD_NO");
    Date entranceDate = rs.getDate("ENTRANCE_DATE");
    std = new Student(id, firstName, lastName, ssn,
                      birthday, stdNo, entranceDate);
    if (rs.next()) {
      //making a message warning about multiple objects existence
    }
  }
  p.close();
  c.commit();
} catch (Exception ex) {
  //handling the exception
} finally {
  if (p != null) {
    try {
      p.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
  if (c != null) {
    try {
      c.close();
    } catch (SQLException e) {
      //handling the exception
    }
  }
}
return std;
```

As you can see, using SQL joins makes the query expressions more complex, and consequently harder to develop, test, and maintain.

Mapping more complicated objects

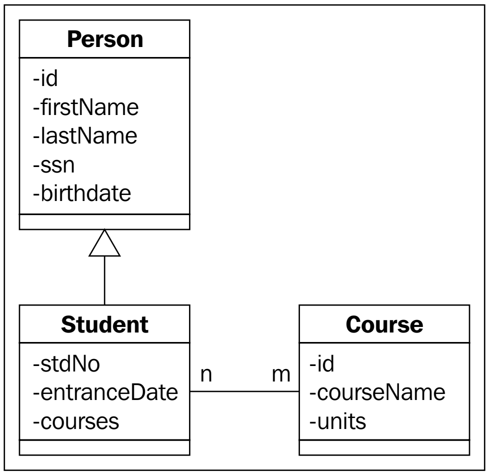
Many Java objects are associated with other objects. The associated objects may be values or entities. **Entities** are objects that have their own persistent identity and are stored as discussed before. In contrast, **values** are objects that do not define some kind of persistent identity. If an entity object is associated with a value object, no real problem arises since the value object can be stored with the entity object in the same table. However, this is not true in the case of associations between two entity objects. Unfortunately, databases do not offer a way to persist object associations by default.

The next mismatch happens when a graph of entity objects must be persisted in the database. In this case, the persistence should be accomplished in such a way that allows the object graph to be restored to its original form at a later time. As a common strategy, for each entity class a database table is used and when an object is stored, each object is persisted in its own database table. The next question here is, how can object associations be persisted? As with inheritance, foreign keys can be taken to establish inter-table relationships and provide table associations.

Let's dig a bit deeper into object associations and extend our example to see how JDBC and SQL deal with associations in practice.

Mapping a many-to-many association

Let's assume that each student is associated with an array of courses. If each course is represented by another class, `Course`, then we have an object relationship like the one shown in the following screenshot:



The following code shows the `Course` class:

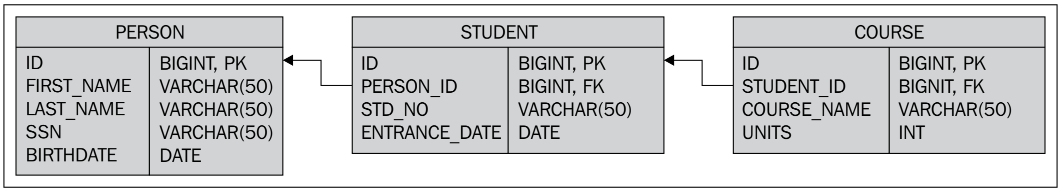
```
package com.packtpub.springhibernate.ch01;
public class Course {
  private int id;
  private String courseName;
  private int units;
  //setter and getter methods
}
```

And this shows the `Student` class:

```
package com.packtpub.springhibernate.ch01;
import java.util.Date;
import java.util.List;
public class Student extends Person {
  private String stdNo;
  private Date entranceDate;
  private List courses;
  //setter and getter methods
}
```

When we work with Hibernate, we always use simple classes which do not have any special behaviors. It is recommended that these classes be expressed with private properties and with setter and getter methods to access the properties. This way of class definition has many advantages that will be fully discussed in the coming chapters.

Note that, we have designed our classes as simple to be as possible to keep our example simple, as well. In this case, the `COURSE` table needs a foreign key column referring to the associated row in the `STUDENT` table. The tables may be related as shown in the following screenshot:



PERSON	
ID	BIGINT, PK
FIRST_NAME	VARCHAR(50)
LAST_NAME	VARCHAR(50)
SSN	VARCHAR(50)
BIRTHDATE	DATE

STUDENT	
ID	BIGINT, PK
PERSON_ID	BIGINT, FK
STD_NO	VARCHAR(50)
ENTRANCE_DATE	DATE

COURSE	
ID	BIGINT, PK
STUDENT_ID	BIGNIT, FK
COURSE_NAME	VARCHAR(50)
UNITS	INT

This kind of relationship indicates that when a `Student` object is persisted, the associated `Course` objects should be persisted as well. However, we may use a different strategy for updating or removing associated objects when the object is updated or removed. For example, we may decide that the associated objects are not to be updated when the object is updated, but they should be erased from the database when the object is removed. This is what we call a cascade operation, indicating whether the operation, or operations, should be propagated to associated entities.

To load an entity object, we must query the appropriate table and any others associated with it. The following snippet shows the query to load a `Student` object with its associated courses:

```
SELECT PERSON.FIRST_NAME, PERSON.LAST_NAME, PERSON.SSN, PERSON.
BIRTHDATE, STUDENT.STD_NO, STUDENT.ENTRANCE_DATE, COURSE.ID, COURSE.
COURSE_NAME, COURSE.UNITS FROM PERSON INNER JOIN (STUDENT INNER JOIN
COURSE ON STUDENT.ID = COURSE.STUDENT_ID) ON PERSON.ID = STUDENT.
PERSON_ID WHERE STUDENT.ID=?;
```

It is very difficult, tedious, and error prone to use only pure JDBC and SQL to store the object graph in multiple tables, restore the object-oriented form of data, search object associations, and handle object inheritance. The SQL statements you use may not be optimized, and may be very difficult to test and maintain.

This is the main reason to use a persistence framework such as Hibernate. The next section looks at Hibernate's background, as well as its advantages, alternatives, and architecture.

Object relational mapping

As the previous discussion shows, we are looking for a solution that enables applications to work with the object representation of the data in database tables, rather than dealing directly with that data. This approach isolates the business logic from any relational issues that might arise in the persistence layer. The strategy to carry out this isolation is generally called object/relational mapping (O/R Mapping, or simply ORM).

A broad range of ORM solutions have been developed. At the basic level, each ORM framework maps entity objects to JDBC statement parameters when the objects are persisted, and maps the JDBC query results back to the object representation when they are retrieved. Developers typically implement this framework approach when they use pure JDBC. Furthermore, ORM frameworks often provide more sophisticated object mappings, such as the mapping of inheritance hierarchy and object association, lazy loading, and caching of the persistent objects. Caching enables ORM frameworks to hold repeatedly fetched data in memory, instead of being fetched from the database in the next requests, causing deficiencies and delayed responses, the objects are returned to the application from memory. **Lazy loading**, another great feature of ORM frameworks, allows an object to be loaded without initializing its associated objects until these objects are accessed.

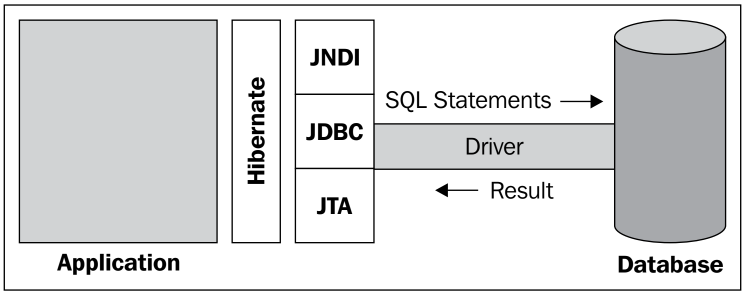
ORM frameworks usually use mapping definitions, such as metadata, XML files, or Java annotations, to determine how each class and its persistent fields should be mapped onto database tables and columns. These frameworks are usually configured declaratively, which allows the production of more flexible code.

Many ORM solutions provide an object query language, which allows querying the persistent objects in an object-oriented form, rather than working directly with tables and columns through SQL. This behavior allows the application to be more isolated from the database properties.

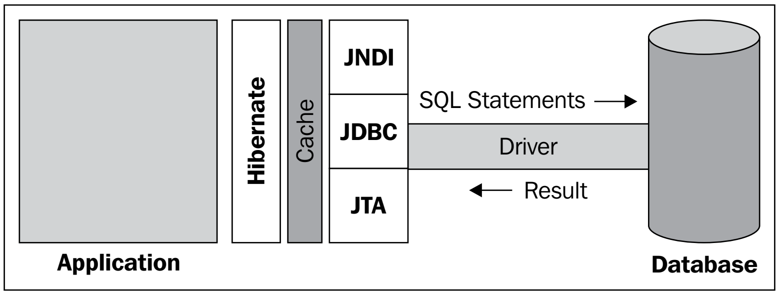
Hibernate as an O/R Mapping solution

For a long time, Hibernate has been the most popular persistence framework in the Java community. Hibernate aims to overcome the already mentioned impedance mismatch between object-oriented applications and relational databases.

With Hibernate, we can treat the database as an object-oriented store, thereby eliminating mapping of the object-oriented and relational environments. Hibernate is a mediator that connects the object-oriented environment to the relational environment. It provides persistence services for an application by performing all of the required operations in the communication between the object-oriented and relational environments. Storing, updating, removing, and loading can be done regardless of the objects persistent form. In addition, Hibernate increases the application's effectiveness and performance, makes the code less verbose, and allows the code to be more focused on business rules than persistence logic. The following screenshot depicts Hibernates role in persistence:



Hibernate fully supports object orientation, meaning all aspects of objects, such as association and inheritance, are properly persisted. Hibernate can also persist object navigation, that is, how an object is navigable through its associated objects. It caches data that is fetched repeatedly and provides lazy loading, which notably enhances database performance. As you will see, Hibernate provides caches in two levels: first-level built-in, and second-level pluggable cache strategies. The first-level cache is a required property for any ORM to preserve object consistency. It guaranties that the application always works with consistent objects. This is originated from the fact that many threads in the application use the ORM to persist the objects which might potentially be associated to the same table rows in the database. The following screenshot depicts the role of a cache when using Hibernate:



Hibernate provides its own query language, which is **Hibernate Query Language (HQL)**. At runtime, HQL expressions are transformed to their corresponding SQL statements, based on the database used. Because databases may use different versions of SQL and may expose different features, Hibernate presents a new concept, called an *SQL dialect*, to distinguish how databases differ. Furthermore, Hibernate allows SQL expressions to be used either declaratively or programmatically, which is useful in specific situations when Hibernate does not satisfy application persistence requirements.

Hibernate keeps track of object changes through snapshot comparisons to prevent unnecessary updating.

Other O/R Mapping solutions

Although Hibernate is the most popular persistence framework, many other frameworks do exist. Some of these are explained as follows:

- **Enterprise JavaBeans (EJB)**: It is a standard **J2EE (Java 2 Enterprise Edition)** technology that defines a different type of persistence by presenting entity beans. Mostly, for declarative middleware services that are provided by the application server, such as transactions, EJB may be preferred for architecture. However, due to its complexity, nontransparent persistence, and need for a container (all of which make it difficult to implement, test, and maintain), EJB is less often used than other persistence frameworks.
- **iBatis SQL Map**: It is a result set–mapping framework which works at the SQL level, allowing SQL string definitions with parameter placeholders in XML files. At runtime, the placeholders are filled with runtime values, either from simple parameter objects, JavaBeans properties, or a parameter map. To their advantage, SQL maps allow SQL to be fully customized for a specific database. To their disadvantage, however, these maps do not provide an abstraction from the specific features of the target database.
- **Java Data Objects (JDO):** It is a specification for general object persistence in any kind of data store, including relational databases and object-oriented databases. Most JDO implementations support using metadata mapping definitions. JDO provides its own query language, JDOQL, and its own strategy for change detection.
- **TopLink**: It provides a visual mapping editor (Mapping Workbench) and offers a particularly wide range of object, relational mappings, including a complete set of direct and relational mappings, object-to-XML mappings, and JAXB (Java API for XML Binding) support. TopLink provides a rich query framework that supports an object-oriented expression framework, EJB QL, SQL, and stored procedures. It can be used in either a JSE or a JEE environment.

Hibernate designers has borrowed many Hibernate concepts and useful features from its ancestors.

Hibernate versus other frameworks

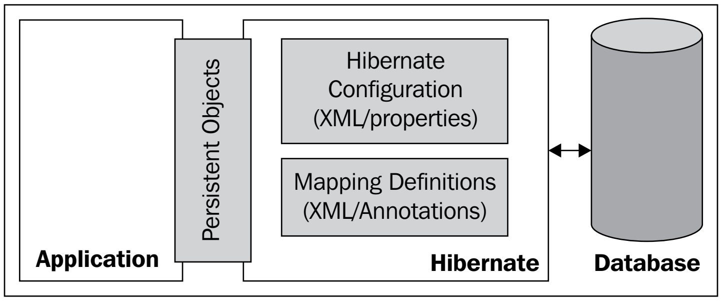
Unlike the frameworks just mentioned, Hibernate is easy to learn, simple to use, comprehensive, and (unlike EJB) does not need an application server. Hibernate is well documented, and many resources are available for it. Downloaded more than three million times, Hibernate is used in many applications around the world. To use Hibernate, you need only J2SE 1.2 or later, and it can be used in stand-alone or distributed applications.

The current version of Hibernate is 3, but the usage and configuration of this version are very similar to version 2. Most of the changes in Hibernate 3 are compatible with Hibernate 2.

Hibernate solves many of the problems of mapping objects to a relational environment, isolating the application from getting involved in many persistence issues. Keep in mind that Hibernate is not a replacement for JDBC. Rather, it can be thought of as a tool that connects to the database through JDBC and presents an object-oriented, application-level view of the database.

Hibernate architecture

The following screenshot depicts the main participants in the Hibernate architecture:


Persistent Objects
Hibernate Configuration (XML/properties)
Mapping Definitions (XML/Annotations)
Application
Hibernate
Database

As the screenshot shows, the main players are Hibernate configuration file(s), mapping definitions, and persistent objects. At the heart of Hibernate is its configuration. This configuration is always presented by an XML, or a properties file and includes the relevant database information, such as database username, password, URL, driver class, and SQL dialect that Hibernate needs for connecting to the database, communicating with it, and performing persistence operations.

Persistent objects form another part of the Hibernate architecture. These objects are what we will persist in the database. These entity objects and their classes, as upcoming chapters explain, do not need to exhibit any special behavior, except that they must follow some POJO rules.

In addition to the Hibernate configuration file and the persistent objects, Hibernate's architecture uses other XML documents, which define how application objects should be mapped to database tables. These documents specify the respective table of each entity class, the mapping of each class's field to its respective table column, and sometimes other mapping information, such as object associations and inheritance. These files have a simple syntax, making them easy to develop and maintain. However, some utility tools that ship with Hibernate let you automatically generate the mapping files, based on the application classes or database schema, and also allow you to modify them in a graphic tool.

Although these objects are the main players in the Hibernate architecture, a Hibernate application's runtime architecture is not limited to them. As we will see in the chapters that follow, the most significant runtime objects are `Configuration`, `SessionFactory`, `Session`, and `Transaction`.

Hibernate can be used as simply as follows to store or retrieve a `Student` object:

```
Configuraion cfg = new Configuration();
cfg.configure();
SessionFactory sessionFactory = cfg.buildSessionFactory();
Student student = …//a new instantiated student object
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(student);
tx.commit();
session.close();
```

Here, we have just configured Hibernate, started a transaction, stored the student object, committed the transaction, and finally disconnected from Hibernate. These are actually the required operations to interact with Hibernate. The configuration includes setting up Hibernate to work with a particular database with special behavior. Every Hibernate interaction should be done inside a transaction, which justifies the next step. Don't worry, all of these concepts with their usages will be explained in the future chapters.

What is Spring

Now that you've read a bit about Hibernate's background, it is time to look at Spring and what it has to offer. Here, we'll take a quick look at Spring, saving the details for later chapters.

Spring is an ambitious framework that aims to be a complete solution for entire JEE applications. Unlike Hibernate, which works merely on the persistence tier, Spring is a multitier framework which offers a wide range of services. It makes JEE development easier, by providing a clean separation of concerns, decoupling an application's components, and minimizing complexities typically encountered in sophisticated JEE environments.

Choosing an appropriate solution for a Java application, particularly when it is built with open-source tools, is a common challenge in application design. This is another challenge that Spring aims to address. The challenge starts when you encounter a large number of open-source technologies that may be used for the same purpose, such as Struts, WebWork, JSF, or Tapestry for web, and Hibernate, JDO, and iBatis for the persistence tier. Spring lets you use a large variety of open-source tools behind the scenes, without needing large amounts of code or coupling the application too closely to the underlying frameworks.

Spring is also called a lightweight framework, since it replaces frameworks that are restrictive and cumbersome to use, such as EJB, that are already offered by JEE.

Spring is modularized with several components. Each component provides a particular service. The following sections summarize some of these.

Inversion of Control container

Inversion of Control (IoC) is the technology most identified with Spring. With the IoC core container, Spring enables the management of object dependencies by pushing dependencies into objects at runtime, instead of letting the objects pull their dependencies from their environment. This approach has many advantages:

- All application classes are designed as simple as possible with minimum behaviors, and with their only required properties they will be well documented.
- All application classes are self-documented, and the documentation is always up-to-date.
- No class has its own configuration management, which allows more manageable code.
- The application leaves configuration management to the framework.

- IoC increases consistency in configuration management, since such management is accomplished by the framework.
- The application has no need for any configuration management code since the framework handles this common aspect of every application.

Chapter 10 discusses Spring's Inversion of Control in depth.

Aspect-oriented programming framework

Aspect-oriented programming (AOP) is the perfect complementary approach to IoC, solving common problems related to J2EE design. AOP allows us to consolidate functionality, which would be otherwise scattered in different places, in a single place. Managing transactions is an example of this functionality. With Spring, transaction management occurs in a single place and is not scattered in persistence methods.

AOP complements **object-oriented programming (OOP)** by introducing a new concept, called **concerns** or **aspects**, to model real-world objects. Concerns are processes that are not directly related to the object hierarchy. Instead, they spread over sets of operations. For example, logging, security, and transaction are common examples for a concern, since they should be applied to sets of methods without any relationship to the object hierarchy.

Spring AOP has the following key benefits:

- It prevents code duplication and provides more manageable code.
- It allows declarative enabling or disabling concerns.

Aspect-oriented programming with Spring is fully discussed in Chapter 11.

Data access abstraction

Spring allows consistent data access to be implemented with solid abstraction. This is carried out through a rich hierarchy of exceptions and a set of helper classes, for working with a wide range of persistence technologies.

We'll look at Spring's Data Access Abstraction and how Spring is integrated with Hibernate in Chapter 13.

Transaction abstraction

Spring provides a transaction abstraction layer over **JTA (Java Transaction API)** global transactions, which span multiple transactional resources. They are managed by an application server, or local transactions managed by JDBC, Hibernate, JDO, or any other persistence technology. It allows coding transactions, either programmatically or declaratively. These topics are discussed in Chapter 12.

MVC web framework

Spring provides a rich, powerful web framework based on the **Model-View-Controller (MVC)** pattern. The aim of the framework, like any other web framework, is to simplify web development. It allows a variety of different view technologies, such as JavaServer Pages, Velocity, and iText PDF to be used. You can use MVC with Spring's other services, such as AOP and IoC. Furthermore, Spring can be integrated with other web frameworks, including Struts, WebWork, Tapestry, and JSF. Discussing Spring's web framework is beyond this book's scope. However, we will take a quick look at it, along with its alternatives, to give you a sense of how, in practice, Spring can be used to make web development easier and solve typical problems related to web applications.

Chapter 14 explains in detail the different strategies Spring offers for supporting web frameworks.

Testing support

Spring applications are more readily testable than other applications. This is because Spring applications rely on POJOs, which do not call any Spring APIs, and their dependencies are normally expressed in the form of interfaces that are easy to stub or mock. Moreover, Spring provides some useful helper classes for implementing test classes that test an application's interaction with Spring.

Moreover, Spring provides a lightweight container against traditional full-blown JEE containers. The container provided by Spring can easily be started from the JUnit test itself, which is not easy to do with EJB-3 and a JEE container.

Chapter 15 looks at using Spring's testing abilities to test Hibernate and Spring applications.

Summary

This chapter looked at Hibernate's background and explored how the framework provides a bridge between the object-oriented and relational worlds. This is why Hibernate is called an O/R mapping tool.

Enterprise JavaBeans are server-side components which provide another approach to persisting Java objects. In addition, enterprise beans provide declarative middleware services, such as transaction and security. Developing with enterprise beans is not as easy as developing with Hibernate, and using entity beans also requires an application server, which is difficult to set up, run, and test.

Hibernate architecture consists of three contributors: persistent objects, configuration file(s), and mapping definitions. The Hibernate API comes with three Java interfaces which are always involved in persisting the objects. These are `org.hibernate.Session`, `org.hibernate.SessionFactory`, and `org.hibernate.Transaction`.

2
Preparing an Application to Use Spring with Hibernate

Hibernate, like many other frameworks, must be configured before you can use it. However, its configuration is very simple and straightforward. Besides configuration, some other settings are required to use Hibernate. These include setting up a database, creating the application structure, and adding extra frameworks, such as Log4j and Ant to the project.

In this chapter, we will discuss the prerequisites to developing with Hibernate. The chapter explains the configuration steps, based on the priority you should take in a typical application. Note that, not all steps are mandatory when using Hibernate. However, taking the optional steps is highly recommended.

We'll start our discussion by setting up a database. Choosing a database may depend on your application requirements, such as robustness, performance, and price, and on other special features that your application needs. Fortunately, as a persistence service provider, Hibernate supports a range of databases, allowing you to choose a database that you prefer, or as your application requirements dictate. This book uses a simple, lightweight database, HSQLDB, for its examples.

After you've set up a database, the next step is getting a Hibernate distribution. Next, set up the project hierarchy, which includes creating source and build directories, installing a build framework (such as Ant), configuring Hibernate, and finally adding the Hibernate libraries, the database driver, and any other frameworks to the application classpath.

Lets discuss these steps in more detail.

Setting up the database

Whether or not you are using Hibernate in your application, you need a database to store your application data. Therefore, your first step is to set up and run a database. Although there are no restrictions in choosing a database when you use Hibernate, we will use a simple, lightweight database called HSQLDB in this book. This lets us concentrate on Hibernate and explore its features, without getting involved in database details.

Getting and installing HSQLDB

HSQLDB is an open-source database that has been fully developed in the Java language. It's a small, lightweight database that is easy to install and use, and it is supported by Hibernate. To get HSQLDB, go to its home page at `http://www.hsqldb.org`, and download the latest version in the form of a compressed file. To install, you only need to extract the file anywhere in your file system.

As the next sections explain, you must also put the database driver, `hsqldb.jar`, in the application classpath.

Configuring and running HSQLDB

Although HSQLDB supports different types of databases, and has many features that make it flexible and reliable, discussing HSQLDB in detail is beyond this book's scope. However, to proceed, we need some primary information about HSQLDB and its configuration. Interested readers can get more information by studying the documentation packed with its download.

Extract the compressed file somewhere on your file system. After extracting the compressed file, you need to configure HSQLDB before you can use it. Configuring HSQLDB is easy and straightforward. It is simply performed through a properties file, which includes all of the configuration information.

Create the `server.properties` file with the following content in `C:/hsqldb`, the directory where you've installed HSQLDB:

```
# Filename: C:\hsqldb\server.properties
# Hibernate examples database - create a database on the default port.
# Specifies the path to the database files –
# note that the trailing slash IS required.
server.database.0=file:/hsqldb/hibernate/
# Specifies the name of the database
server.dbname.0=hiberdb
```

The properties file entries represent the configuration arguments of the HSQLDB database. In our case, the file includes only two entries:

- `server.database`: This entry determines the relative path of the database inside the HSQLDB root directory.
- `server.dbname`: This entry assigns a name to the specified database, by which we can then refer to the database.

Note the use of `0` at the end of the `server.database` and `server.dbname` properties. This number specifies the index of the database being configured and is incremented for the second database, and so on.

HSQLDB server modes

HSQLDB can run in different modes based on the protocol used for communications between the client and server. The following list gives a brief review on the different server modes that HSQLDB supports:

- **Hsqldb server**: The fastest way for running a database server, this mode uses a proprietary communications protocol.
- **Hsqldb web server**: This mode is used when access to the computer hosting the database server is restricted to the HTTP protocol. The only reason for using this mode is restrictions imposed by firewalls on the client or server machines. It should not be used where there are no such restrictions.
- **Hsqldb servlet**: This uses the same protocol as the web server. It is used when a separate servlet engine (or application server), such as Tomcat or Resin, provides access to the database. Since the database server sits on the servlet engine to be used, this mode cannot be used without a servlet engine.
- **In-process (stand-alone) mode**: This mode runs the database engine as part of your application program in the same Java Virtual Machine. For most applications, this mode can be faster, as the data is not converted and sent over the network. The main drawback is that, by default, it is not possible to connect to the database from outside your application. As a result, you cannot check the contents of the database with external tools while your application is running.
- **Memory-only databases**: In this mode, HSQLDB holds the data entirely in memory without persisting it to disk. Since this mode does not use the disk, it is useful only for internal processing of application data, such as applets.

The HSQLDB documentation, which is packed with its distribution, is the best reference for more detailed information about these modes.

After preparing the `server.properties` file, run the following command from the same directory as the `server.properties` file, in order to run HSQLDB in the mode of server:

```
java -classpath c:/hsqldb/lib/hsqldb.jar org.hsqldb.Server
```

This command starts HSQLDB, and the following output is displayed in the command window:

```
C:\hsqldb>java -classpath c:/hsqldb/lib/hsqldb.jar org.hsqldb.Server
[Server@1a758cb]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@1a758cb]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@1a758cb]: Startup sequence initiated from main() method
[Server@1a758cb]: Loaded properties from [C:\hsqldb\server.properties]
[Server@1a758cb]: Initiating startup sequence...
[Server@1a758cb]: Server socket opened successfully in 211 ms.
[Server@1a758cb]: Database [index=0, id=0, db=file:/hsqldb/hibernate/,
alias=hiberdb] opened sucessfully in 1702 ms.
[Server@1a758cb]: Startup sequence completed in 2053 ms.
[Server@1a758cb]: 2008-09-03 11:42:23.327 HSQLDB server 1.8.0 is online
[Server@1a758cb]: To close normally, connect and execute SHUTDOWN SQL
[Server@1a758cb]: From command line, use [Ctrl]+[C] to abort abruptly
```

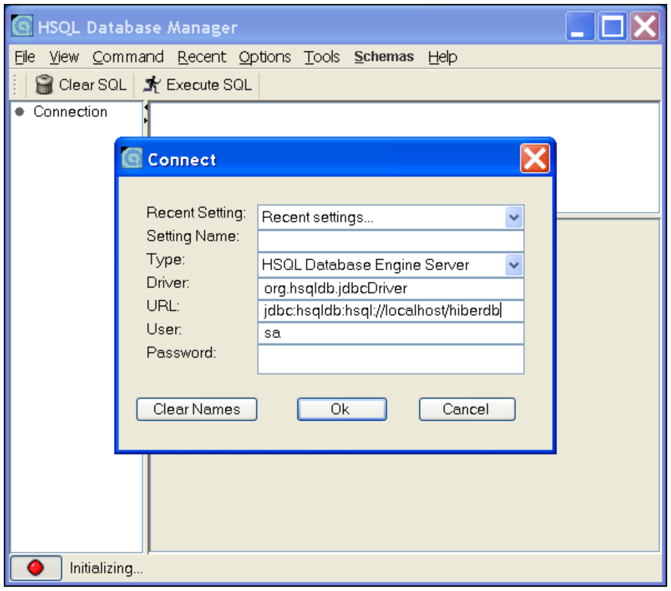
You can shut down the server with the `SHUTDOWN` SQL command, issued as an SQL query, in the Database Manager (covered in the next section).

You can also shut down the HSQLDB server by terminating its process, done easily by simultaneously pressing the `Ctrl` and `C` keys. However, this method is clean and safe, because HSQLDB is not allowed to complete its tasks (for example, storing all data in memory to disk and closing the database files) in a persistent and safe way.

Now that we have successfully executed HSQLDB, we can continue our work by creating the tables that we need.

Managing HSQLDB

The HSQLDB distribution ships with a utility application called **Database Manager**. Database Manager is a tool for managing the database schemas to create, edit, or delete tables, and perform many other database operations. To run this application, go to the `demo` directory and execute the `runManagerSwing.bat` batch file. You'll then see a window like the following one:



Recent Setting: Recent settings...
Setting Name:
Type: HSQL Database Engine Server
Driver: org.hsqldb.jdbcDriver
URL: jdbc:hsqldb:hsql://localhost/hiberdb
User: sa
Password:

In the application's input dialog, enter the driver class name, URL, username, and password of the database, as shown in the following table:

Option	**Value**
Setting Name	`hibernate-exercise`
Type	**HSQL Database Engine Server**
Driver	`org.hsqldb.jdbcDriver`
URL	`jdbc:hsqldb:hsql://localhost/hiberdb`
User	`sa`
Password	`--`

Notice that we've chosen **HSQL Database Engine Server** as the value for the **Type** option. The **Type** option determines what kind of server you want to run.

`sa` and blank are the default values for username and password when you create a schema in HSQLDB. Although these are enough for our application, you can create new accounts, as well. Please see HSQLDB's website for more information.

`hiberdb` is the name of the database schema that we have already defined in the `server.properties` file (and we use that for the examples in this book). After you've entered the information, click **OK** and connect to the specified database schema.

The manager then lets you enter the SQL statements in the top section of this window. You can then execute them by clicking the **Execute** button.

Getting a Hibernate distribution

To use Hibernate in your application, you first need to get it. Therefore, browse to the homepage of Hibernate at `http://www.hibernate.org`, choose the Hibernate Core package, and download the latest version in the form of a `.zip` or `.tar.gz` compressed file.

Extract the compressed file in a directory on your system. This directory will include the Hibernate library `hibernate3.jar`, optional and required JAR files in the `lib` subdirectory, and Hibernate documentation in the `doc` subdirectory.

Getting a Spring distribution

As with Hibernate, you must download the full distribution version of Spring from its website: `http://www.springframework.org`. The file, which downloads as a `.zip` or `.tar.zip` file, contains separate JAR files as Spring libraries in the `dist/modules` subdirectory. (The number of JAR files depends on the version of Spring.) Each JAR file contains classes of a particular Spring module. For instance, `spring-aop.jar` contains all of the classes you need to use Spring's AOP features in your application. Although not all of these JAR files are required when you are using Spring with Hibernate, you can simply add all of them to the project `lib` directory.

Setting up the project hierarchy

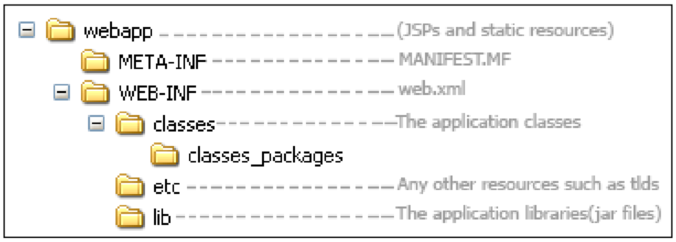
Now that all of the prerequisites for application development are ready, it's time to set up the project. In this section, we'll look at a general project structure that you can apply to your Spring and Hibernate projects. We'll use this structure in the examples throughout this book.

As the first step, we'll set up the project hierarchy, which includes building the basic directories and creating the needed files. The project hierarchy depends on your application type and the frameworks that your application uses. As a developer, you should note that the basis of your development is providing a clean environment, in which all files are classified, and where you always have full control of changes as the application grows. Here, we'll see a typical application structure that satisfies the most typical application requirements.

First, create and name the application work directory. This directory contains all of the application source files, build files, script files, and so on. This directory can include the following subdirectories:

- `src` for the application source files
- `out` for the compiled classes
- `test` for unit tests
- `lib` for libraries used by the application

If you are working on a web application, such as the sample application discussed in this book, you may need an additional directory in `src`. The following figure shows a sample layout for this directory:


webapp
(JSPs and static resources)
META-INF
MANIFEST.MF
WEB-INF
web.xml
classes
The application classes
classes_packages
etc
Any other resources such as tlds
lib
The application libraries(jar files)

The `build.xml` and `pom.xml` files, which are the Ant and Maven files respectively, and other configuration files (such as `log4j.properties` for configuring logging) should be placed in the application's root directory. You must also add all of the necessary JAR library files, such as `hibernate.jar`, `spring.jar`, the database driver, and so on, to the application classpath.

Put the required libraries in the lib directory

Since you are developing a Hibernate with Spring project, you need to put their libraries, and also their dependencies, in your project. Therefore, when you create the structure of your project, put `Hibernate.jar` with its dependencies, and `spring.jar` with its required dependencies. Please note that with Hibernate you need almost all dependencies, but since Spring is a general and throughout project, you may just need a subset of Spring dependencies. A typical `lib` directory of a Hibernate with Spring project may look like the following:

```
+lib
    antlr.jar
    asm.jar
    asm-attrs.jars
    c3p0.jar
    cglib.jar
    commons-collections.jar
    commons-logging.jar
    dom4j.jar
```

```
        hibernate3.jar
        hsqldb.jar
        jta.jar
        junit-3.8.2.jar
        log4j-1.2.15.jar
        slf4j-api-1.5.0.jar
        slf4j-log4j12-1.5.0.jar
        spring.jar
```

Setting up Ant or Maven in the project

As the next step, you need a build framework in your project, to automatic the build and test process. The common solutions for this are the Ant and Maven projects. **Ant** is one of the most popular frameworks in the Java world, and plays an essential role in Java applications. It provides an automatic and reliable build process for your application, allowing you to define a repeatable and reliable build process in XML format. **Maven** is another build framework with a somewhat similar role in Java applications, but does more than what Ant does. Maven can manage the project dependencies and resolve them in sophisticated projects, which may consist of many other libraries and projects. Almost all of today's projects use Maven as the build framework, so using Maven is preferred over Ant. Here, adding and setting up both frameworks are briefly discussed. First, let's look at Ant.

To get Ant, go to `http://ant.apache.org` and download the latest version. Then, extract the compressed zip file in a directory in your file system.

To complete the installation process, you need to perform the following two tasks:

1. Define the `ANT_HOME` variable (which points to the Ant's root directory) as an environment variable.
2. Add Ant's `bin` subdirectory to the system or command line path.

To check whether Ant has been installed correctly, run the ant command in a command window. You should see the following output:

```
> ant -version
Apache Ant version 1.6.1 compiled on February 12 2004
```

The message identifies your version of Ant.

The build process for any application accomplished by Ant is defined in an XML file. This file, usually called the build file or the Ant file, starts with the XML definition and is followed by a `<project>` element as root. All of the process steps are defined inside the project through target elements, one of which is the default.

Targets contain task elements and can depend on other targets. For example, you might have a target for compiling, and a target for creating a distributable archive file. As you can only build the distributable file after you have compiled the source code, the distribute target depends on the compile target. Ant resolves these dependencies.

I won't go into detail here, since Ant is fairly straightforward and widely used. To get more information about Ant, refer to its excellent documentation at `http://ant.apache.org/manual`.

To get Maven, go to `http://maven.apache.org/run-maven/index.html` and download the latest version. Similar to Ant, you need to perform two other tasks as follows to complete the installation:

1. Add the Maven's `bin` directory to the system or command line path.
2. Increase the memory available to Maven with setting `MAVEN_OPTS -Xms128m -Xmx512m=`

> Whether you are using Ant or Maven, you need to define a system property, `JAVA_HOME`, which refers to the root of the JDK installation directory.

To get more information about Maven and its usage, refer to Maven website at `http://maven.apache.org/run-maven/index.html`.

Summary

This chapter demonstrated the basic steps for preparing to use Hibernate. The first step is to install and run a database engine. HSQLDB is a simple, lightweight, and easy-to-use open-source database. Therefore, it lets us learn Hibernate without getting involved in database details.

After setting up a database, the next steps are getting Hibernate and setting up the project hierarchy. Although there are some templates for the application structure, this structure mostly depends on the application you are working on.

Getting Ant or Maven and adding them to the project are the next steps. Ant is a utility project that provides an automatic build process. To start using Ant, extract Ant in a directory, set up the `ANT_HOME` variable in your system properties, add its `bin` subdirectory to the System PATH, and create and set up a build file. Maven is another project which is used to build the project and manage project dependencies. Like Ant, to use Maven, you need to set up the `MAVEN_HOME` variable in the system properties and add its `bin` subdirectory to the PATH of the system.

3
A Quick Tour of Hibernate and Spring

Getting started with a new framework is often the most difficult part of learning it. In our case, there are two frameworks to learn, both integrated with each other, containing many details, and used for different purposes in the application architecture. Therefore, before we dig into the details, it's important to understand how to use these frameworks within the context in which they operate. With this understanding in place, you'll learn the details more smoothly, with a less steep learning curve.

This chapter provides a quick look at different aspects of persistence that typically arise during application development. We'll discuss how Hibernate handles these cases and simplifies development of the persistence layer. We'll then look at the aspects of Spring that are involved in the persistence area of the application architecture. Because the main goal of this chapter is to demonstrate the basic usage of Hibernate and Spring, simple examples will explain Hibernate's and Spring's basic features without presenting anything complicated. First, I'll use a simple Java class, a simple database table, and a simple mapper class to explain the core Hibernate API and the basic steps in using Hibernate. After that, when we discuss the fundamental concepts behind Hibernate, we'll look at other aspects, including the mapping of objects association and inheritance, querying persistent objects, object relations and navigability, managing transactions, lazy loading, caching, and practical Hibernate configuration. At the end of this chapter, I'll review two basic features of Spring: **Inversion of Control (IoC)** and **Aspect-Oriented Programming (AOP)**, looking at how these are used in mixed Hibernate-Spring applications.

Getting started with Hibernate

Hibernate may be used in many different situations. It can be used when you're developing a new application from scratch, extending an existing application, or developing a new application which works with an existing database schema.

Usually, it's best to develop a new Hibernate application from scratch since you can design your database schema flexibly, based on application requirements and Hibernate best practices. When you use an existing schema in developing an application, you must work with the schema, which is not necessarily consistent with new requirements, so you must make "dirty" changes simply to make the application work. However, when the database schema exists, utility tools can help you to analyze the schema and guide you in designing appropriate Java classes that represent the data.

Here, we'll start with the simplest case and discuss how to start a new Hibernate application from scratch. When you're developing a new Hibernate application, after you have configured it, you should do the following three things:

- **Design and implement persistent classes, which represent the application's live data**: The business processes inside the application design and produce persistent objects. In fact, they contain significant information to be kept in a persistent store for future use. As the first step in developing a new Hibernate application, you need to design the object model with which the application works.
- **Design and create database tables**: This step involves designing the database schema that encompasses the live data. Most of the time, tables in the database schema correspond to each class in the object model and are designed to hold the data contained by these objects.
- **Creating mapping metadata:** Mapping metadata is the data, defined in the Java code as annotations or in XML documents as individual files, that describes the relationship of persistent classes to database tables. This metadata tells Hibernate how to map the object model to its respective database schema.

Upcoming chapters discuss all of these steps in detail. Throughout this book, I present a sample application, a comprehensive educational system, to demonstrate how to perform these steps in a real application. This demo application manages information related to all schools, teachers, students, and courses in an educational system. It provides add, edit, and delete operations for teachers, students, and courses, and handles the associated business rules. In this and the subsequent chapters, individual parts of this application will explain working with Hibernate in detail.

Designing and implementing persistent classes

As you remember from Chapter 1, `Student` objects can be persisted into the `STUDENT` table. Although there would be other persistent classes such as `Course`, `Teacher`, and `School` in our application, we will not involve those here in order to keep our example as simple as possible.

The `Student` class looks like the code here:

```
package com.packtpub.springhibernate.ch03;
public class Student {
  private int id;
  private String firstName;
  private String lastName;
  //zero-arguement constructor
  public Student() {
  }
  public Student(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public boolean equals(Object o) {
    if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
```

```
        Student student = (Student) o;

        if (id != student.id) return false;
        if (firstName != null ? !firstName.equals(student.firstName) :
                    student.firstName != null) return false;
        if (lastName != null ? !lastName.equals(student.lastName) :
                    student.lastName != null) return false;

        return true;
      }

      public int hashCode() {
        int result;
        result = id;
        result = 29 * result + (firstName != null ? firstName.hashCode()
         : 0);
        result = 29 * result + (lastName != null ? lastName.hashCode()
         : 0);
        return result;
      }
    }
```

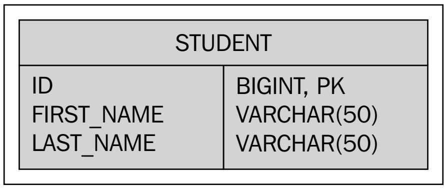
This class has simple properties: `id`, `firstName`, and `lastName`, all private, and accessed by their setter and getter methods.

Classes with no special behavior, only containing some properties with their setter and getter methods, are called **Plain Old Java Object (POJO)**. POJO classes have critical roles in many new frameworks such as Hibernate and Spring. All of the persistent objects that are stored or retrieved by Hibernate are POJOs. POJO rules, and characteristics of Hibernate persistent classes, are discussed in detail in Chapter 5.

This class also has a zero-argument constructor. This is an optional (and recommended) requirement that persistent classes in Hibernate should meet. Chapter 5 discusses this and other requirements for persistent classes in Hibernate.

Creating Database Tables

Now that the `Student` class has been designed and implemented, we need a table in the database to store the `student` objects. Usually, there should be a database table corresponding to each persistent class for the persistence of related objects. The following screenshot shows the `STUDENT` table with the columns in the `hiberdb` database schema, which was set up in Chapter 2.

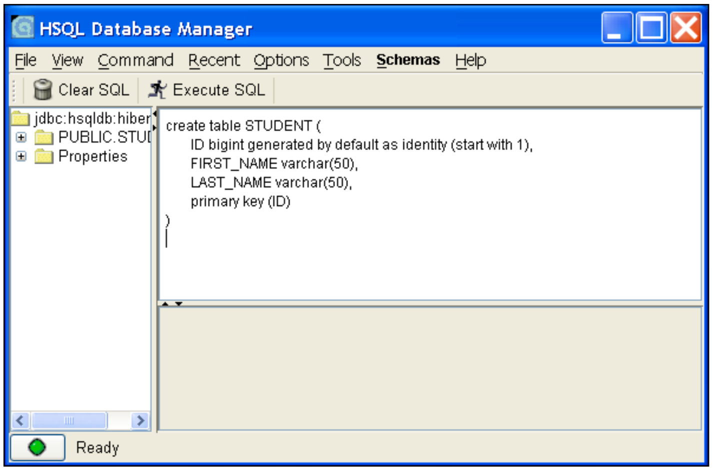


STUDENT	
ID	BIGINT, PK
FIRST_NAME	VARCHAR(50)
LAST_NAME	VARCHAR(50)

The SQL expression used to create this table is shown below:

```
create table STUDENT (
        ID bigint generated by default as identity (start with 1),
        FIRST_NAME varchar(50),
        LAST_NAME varchar(50),
        primary key (ID)
)
```

We may use the HSQL Database Manager to create the table, as shown here:



```
create table STUDENT (
    ID bigint generated by default as identity (start with 1),
    FIRST_NAME varchar(50),
    LAST_NAME varchar(50),
    primary key (ID)
)
```

If your query syntax is invalid, or if the manager application is unable to execute the query, a dialog box reports the query execution failure.

If you are developing an application from scratch, you can use SchemaExport, shipped with Hibernate, to generate your schema from the mappings. Additionally, it is also possible to use SchemaUpdate to update an existing schema based on the modified mappings.

Creating mapping metadata

After you've designed the `Student` class and created the `STUDENT` table, you need to give Hibernate information about the mapping between the `Student` class fields and the `STUDENT` table columns. For this purpose, Hibernate lets you use either XML documents, as individual mapping files, or annotations in the Java code. The mapping metadata specifies how to map a persistent class to a database table. In this book, I discuss using both approaches: annotations are newer and are used more, but the XML approach isn't still deprecated.

In our example, we can use an XML file to define how each `Student` property is stored, retrieved, updated, and removed from a `STUDENT` column when a `Student` instance is stored, retrieved, updated, or removed from the database.

As a mapping file, `Student.hbm.xml` tells Hibernate that `id`, `firstName`, and `lastName` of the `Student` class should be stored, in the respective `ID`, `FIRST_NAME`, and `LAST_NAME` columns of the `STUDENT` table. The following listing shows this file:

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

<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch03.Student"
    table="STUDENT">
    <id name="id" type="int" column="id">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <property name="lastName" column="LAST_NAME" type="string"/>
  </class>
</hibernate-mapping>
```

As a convention, it's recommended to name the mapping files as `className.hbm.xml`, in which `className` is the name of the matching persistent class. It's also recommended that each mapping file be placed in the same package as its persistent class. This simplifies the use of Hibernate and helps with application maintenance. For instance, if the `Student` class's package is `com.packtpub.springhibernate.ch03`, then `Student.hbm.xml` would be in the `com/packtpub/springhibernate/ch03` directory with the `Student` class. This approach lets you manage mapping files based on the packages of their respective persistent classes.

The content of all mapping files begins with the XML version and encoding identifiers, followed by a `DOCTYPE` declaration:

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

The `DOCTYPE` instruction identifies the document type definition of the mapping document. As we use Hibernate 3 in this book, the `DOCTYPE` instruction refers to `hibernate-mapping-3.0.dtd`. If you use a different version of Hibernate, specify the corresponding DTD document in the same way.

After `DOCTYPE`, the `<hibernate-mapping>` element is the root element and specifies the actual mappings. Inside the root element, at least one `<class>` element should exist. Each `<class>` element defines the mapping of a persistent class to a database table. For our sample application, we use a nested `<class>` element inside the root to determine the mapping of our persistent class, `Student`, to its respective table, `STUDENT`:

```
<?xml version="1.0"?>
<!DOCTYPE … >
<hibernate-mapping>
  <class>
  </class>
</hibernate-mapping>
```

As we currently have only one class, `Student`, we have used only one `<class>` element inside the mapping document. However, when there are more persistent classes in the application, you can use other `<class>` elements inside the root element to map them to their respective tables.

The `<class>` element comes with two attributes: `name` and `table`. These attributes specify, respectively, the fully qualified class name you want to map, and the database tables to which persistent objects are mapped:

```
<class name="com.packtpub.springhibernate.ch03.Student"
 table="STUDENT">
```

The `<class>` element and its nested elements specify all relevant information needed to map the persistent class to its database table.

The `<id>` element, coming as the first element inside `<class>`, uses the `name`, `type`, and `column` attributes to indicate the object identifier name, the corresponding Hibernate type for that identifier, and the table's primary key column which holds the object identifier respectively. Additionally, the `<id>` element uses a nested `<generator>` element to determine the Hibernate strategy for generating identifier values. The following code shows the `<id>` element with its attributes and nested `<generator>` element:

```
<id name="id" type="int" column="id">
  <generator class="increment"/>
</id>
```

A Hibernate type is a representation of the SQL type of the column in which the property's value is stored. Hibernate uses these values when it generates an SQL query to insert, update, delete, or perform any other persistence operations. These types may also be used when the database schema is generated from the mapping definitions. Hibernate types let the application be isolated from the underlying database's actual SQL types. At runtime, Hibernate types may be transformed to different SQL types when different database engines are used with Hibernate. Hibernate types are fully discussed in Chapter 7.

The nested `<generator>` element specifies how to generate an identifier value for a new instance when the instance is stored. The value of the `class` attribute refers to a Hibernate-included algorithm, according to which the identifier is generated. In this case, we have chosen `increment` as the identifier generation strategy. This means every identifier must be produced by incrementing the preceding one. Chapter 5 explains this and other algorithms.

After `<id>`, `<property>` elements appear as the next elements inside `<class>`, and define the mapping of other class properties to table columns. Every `<property>` element commonly comes with three attributes: `name`, `column`, and `type`, which specify the name of the property, the table column, and the column's Hibernate type respectively:

```
<property name="firstName" column="FIRST_NAME" type="string"/>
<property name="lastName" column="LAST_NAME" type="string"/>
```

Chapter 5 and Chapter 6 offer in-depth discussions of mapping concepts.

A simple client

Now that we've completed all three steps in designing and developing our Hibernate project, which currently consists of only the `Student` class, the `STUDENT` table, and the `Student.hbm.xml` file, the Hibernate API can interact with the database and persist the objects.

The following listing shows an immature use of Hibernate to store an instance of the `Student` class into the `STUDENT` table:

```
package com.packtpub.springhibernate.ch03;
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class PersistByHibernate {
  public static void main(String[] args) {
    //configuring Hibernate
    Configuration config = new Configuration();
    config.setProperty("hibernate.dialect",
                       "org.hibernate.dialect.HSQLDialect");
    config.setProperty("hibernate.connection.driver_class",
                       "org.hsqldb.jdbcDriver");
    config.setProperty("hibernate.connection.url",
                       "jdbc:hsqldb:hsql://localhost/hiberdb");
    config.setProperty("hibernate.connection.username", "sa");
    config.setProperty("hibernate.connection.password", "");
    //introducing persistent classes to Hibernate
    config.addClass(Student.class);
    //obtaining a session object
    SessionFactory factory = config.buildSessionFactory();
    Session session = factory.openSession();
    //starting a transaction
    Transaction tx = session.beginTransaction();
    //persisting…
    Student student = new Student("Andrew", "White");
    session.save(student);
    //commiting the transaction
    tx.commit();
    session.close();
  }
}
```

The class shown in the previous code uses Hibernate to save an instance of `Student`. The process for all persisting operations, including save, load, update, and delete, consists of the following steps:

1. Configuring Hibernate.
2. Obtaining a Session object.
3. Starting the transaction.
4. Performing persistence operation(s).
5. Committing or rolling back the transaction.

The next several sections cover these steps.

Configuring Hibernate

Before you can interact with Hibernate and perform any persistence operations, you need to configure it. The basic configuration includes specifying the database driver, URL, username, password, and dialect (discussed in upcoming chapters). It also includes introducing the persistent class names or their mappings. All of these settings are represented by a configuration object of type `org.hibernate.cfg.Configuration`.

To set each property value, the `Configuration` class provides a `setProperty()` method, which takes two arguments: the property name and the property value, seen here:

```
public Configuration setProperty(String propertyName, String
  propertyValue)
```

For instance, invoking:

```
config.setProperty("hibernate.connection.driver_class",
  "org.hsqldb.jdbcDriver");
```

sets `org.hsqldb.jdbcDriver` as the value of `hibernate.connection.driver_class`. The `Configuration` class also provides the `addClass()` method, which introduces a persistent class to Hibernate:

```
public Configuration addClass(Class aClass) throws MappingException
```

Note that, both `setProperty()` and `addClass()` return an object of type `Configuration`, allowing you to set properties by invoking a chain of `setProperty()` or `addClass()` methods. I cover configuration in more detail in Chapter 4.

Obtaining a session object

With Hibernate, all persisting operations, including save, load, update, and delete, are performed through instances of `org.hibernate.Session`. As you will see, these instances are not thread-safe. `Session` objects are not instantiated directly. Instead, they are constructed by invoking the `openSession()` method of another object called a **session factory**. A session factory is a heavyweight object of type `org.hibernate.SessionFactory`. This object is thread-safe and responsible for creating and managing session objects. It's analogous to a connection pool, which is responsible for creating and managing database connection objects. Note that typical applications need only one `SessionFactory` object for each database they use. Chapter 4 gives an example of how to maintain this condition.

Like `Session` objects, `SessionFactory` is not instantiated directly. Instead, it is constructed by invoking the `buildSessionFactory()` method of the `Configuration` object that has already been created.

The following snippet shows the process of obtaining instances of `SessionFactory` and `Session`:

```
//obtaining a session object
SessionFactory factory = config.buildSessionFactory();
Session session = factory.openSession();
```

Starting a transaction

Transaction management is an essential requirement in any application. It lets you reliably perform a set of operations, as a unit of work. Both Hibernate and JDBC let you mark the boundaries of a set of persistence operations to be performed as a unit of work, whether all operations are done successfully or all are failed. In Hibernate, you can call the `beginTransaction()` method of `Session` to obtain an instance of type `org.hibernate.Transaction`. The `Transaction` object, in fact, is a handler which lets you control the transaction execution, commit, or roll back when persistence operations are done.

Hibernate automatically starts the transaction when the `beginTransaction()` method of `Session` is invoked as follows:

```
Transaction tx = session.beginTransaction();
//performing operations
```

As the next section discusses, the started transaction should be committed or rolled back after the persistence operation is done.

Performing the operation

After the transaction has begun, you can perform any desired persistence operations with Hibernate. For this purpose, you can use the `Session` instance, which provides distinct persistence methods, including `save()`, `update()`, `saveOrUpdate()`, and so on. The Session API and persistence methods exposed through it are discussed in Chapter 8.

Committing/rolling back the transaction

When persistence operations are performed, you can decide to commit or roll back the already started transaction by invoking the `commit()` or `rollback()` method of `Transaction`, respectively, as follows:

```
//decide to commit
if(every_thing_is_ok)
  tx.commit();
else //roll back
  tx.rollback();
```

Although, database operations always happen inside transactions (so you need to manage transactions in your Hibernate application when persistence operations are performed), you can leave transaction management to the database engine by setting the `connection.autocommit` property to `true` in the configuration file, or invoking the `setAutoCommit(true)` of the associated `Connection` object to `Session` as follows:

```
session.connection().setAutoCommit(true);
```

Then, you will not need to start and then commit or roll back the transaction. The database automatically handles transactions and performs every operation in a single transaction.

Using auto-commit is mostly impossible. There are common situations in a typical project that a persistent operation in the application is performed with two or more database operations together. Therefore, we need to manage transactions in the application for each application-level operation.

Chapter 10 discusses transaction management in detail.

Hibernate declarative configuration

As with many other frameworks, you must configure Hibernate before you can use it. The configuration process consists of two parts:

- **Object Mappings:** Determine the object mappings that define how persistent objects are mapped to their respective database tables.
- **Database-Relevant Information:** Specify database-relevant information to tell Hibernate how to connect to the database and perform persistence operations. The basic configuration information is database driver, URL, username, and password. However, configuration options are not limited to these. Other significant configuration settings include using a container-managed versus a non-container-managed data source, cache configuration, transaction settings, minimum and maximum pool size, and connection timeout value.

Earlier, we configured Hibernate programmatically. However, that approach is not flexible enough to be used in a real project. Hibernate supports a declarative approach for configuration, which makes it more flexible. You can configure Hibernate declaratively, either through a properties file, or an XML file. The following code is a sample configuration file named `Hibernate.cfg.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- database related configurations -->
    <property name="connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">
      jdbc:hsqldb:hsql://localhost/hiberdb
    </property>
    <property name="connection.username">sa</property>
    <property name="connection.password"> </property>
    <property name="pool_size">5</property>
    <property name="connection.autocommit">true</property>
    <property name="show_sql">true</property>
    <property name="dialect">
        org.hibernate.dialect.HSQLDialect</property>
    <!-- mapping files-->
    <mapping
     resource="com/packtpub/springhibernate/ch03/Student.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

As you can see, configuration values are determined through `<property>` and `<mapping>` elements. Each `<property>` element specifies a particular property of Hibernate.

The following table explains the properties in the configuration file shown in the listing above.

Property name	**Meaning**
`connection.driver_class`	refers to the database driver class name
`connection.url`	specifies the database's URL
`connection.username` `connection.password`	determines the database username and password
`pool_size`	specifies the number of connection objects held in the connection pool
`connection.autocommit`	specifies whether each database operation is performed in an individual transaction and is committed automatically
`show_sql`	determines whether Hibernate-generated SQL statements are shown in either the application console or the logging files
`dialect`	specifies the database dialect used to connect to the database (the dialect concept is discussed in the next chapter)

The last part of the configuration file configures the mapping files. As mentioned, Hibernate uses these files to map persistent classes to database tables. Each `<mapping>` element takes a `resource` attribute, which refers to the relative path of the mapping file. For example, the following mapping element specifies `Student.hbm.xml`, which is located in the `com.packtpub.springhibernate.ch03` relative path:

```
<mapping
 resource="com/packtpub/springhibernate/ch03/Student.hbm.xml"/>
```

The application's root directory commonly holds the Hibernate configuration file, although you don't have to follow this practice. To configure Hibernate when this file is in the root of the classpath, simply instantiate an object of the `org.hibernate.cfg.Configuration` class and invoke its `configure()` method:

```
Configuration config = new Configuration();
config.configure();
```

Hibernate then automatically loads the defined settings in the configuration file.

When the configuration file is not in the root, specify the relative path of the file as the `configure()` method's argument:

```
Configuration config = new Configuration();
config.configure("relative path");
```

Configuration details are discussed in Chapter 4.

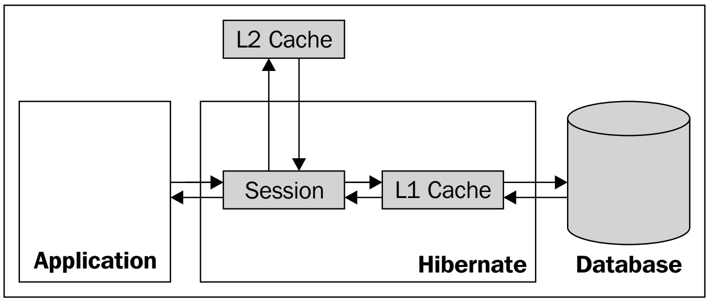
Some issues in mapping

So far, we have used a simple persistent class, `Student`, in our example. As mentioned in Chapter 1, most persistent classes are not so simple. Many are not plain as they may be inherited from or associated with others. As a persistence provider, Hibernate should handle these cases. We'll see more about this in Chapter 5. For now, let's look at how Hibernate caches objects, and how to query the objects we've mapped.

Caching

Regarding performance enhancement, Hibernate provides two caches, each in a different level. As the *first-level cache*, each `Session` object has its own cache, which is also called *transactional cache*. This cache holds the persistent objects of `Session`, and minimizes database interactions, thereby enhancing efficiency and performance.

A third-party cache provider provides the *second-level cache*, which is configured as a plug-in in Hibernate. The second-level cache holds the repetitive persistent objects associated with all session objects. The following screenshot depicts the role of these two cache levels in Hibernate:



When the application queries a persistent object, the `Session` instance initially checks the first-level cache. If the object is not found, the instance checks the second-level cache. If the object is not found in either the first-level or the second-level cache, it is loaded from the database. Hibernate does not provide any way to enable or disable the first-level cache. However, the second-level cache is optional, so you can ignore it or configure Hibernate using any third-party cache provider. You can enable or disable the second-level cache through general Hibernate configuration (for all persistent objects), or in the mapping definitions (for particular objects).

Chapter 12 discusses Hibernate caching.

Querying objects

You can use the `load()` and `get()` methods of the `Session` instance to load a particular persistent object. For example, you can use the `load()` method as follows to load a `Student` instance with identifier value 41:

```
Student std = (Student) session.load(Student.class, new Long(41));
```

You can also use HQL to load persistent objects. With HQL, you can describe the objects you are looking for with the `from` and `where` clauses, as with SQL. The following statement simply queries the `Student` object with the identifier value 41:

```
List result = session.find("from Student as s where s.id=41");
```

The `result` is a `java.util.List` object, which contains the `Student` instance associated with identifier value 41. Using HQL, you can specify conditions other than the object identifier for the objects being loaded, or you can load many objects instead of just one.

Hibernate also provides a rich query API, called the Criteria API, to programmatically describe the objects being loaded, and then load them. The following snippet shows the Criteria approach to load the `Student` instance with the identifier value 41:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.idEq(new Integer(41)));
List result = criteria.list();
```

As with HQL, the `result` object is a `java.util.List` object, which contains the `Student` instance associated with the identifier value 41.

Although HQL and the Criteria API are interchangeable, in most cases, HQL is a better fit than Criteria, and in particular cases Criteria may fit better than HQL.

All of these approaches are discussed in detail in Chapter 9.

Getting started with Spring

Chapter 1 introduced the Spring services that are typically involved in an application's persistence tier. At its core, Spring provides Inversion of Control (IoC). This is the main functionality with which Spring aims to simplify application architecture. IoC affects all other services provided by Spring.

Here, we'll take a quick look at Spring IoC.

A simple case

In the educational system application, assume that a nonfunctional requirement indicates that before execution of any of the application classes setter methods, the setter method name, the old value, and the new value being set to the property are printed in the application console, a database table, or a log file, based on administrator preferences.

To handle this, the application uses three different classes, shown as follows;

`SetterInfoConsolePrinter` as this:

```
package com.packtpub.springhibernate.ch03;
public class SetterInfoConsolePrinter {
  public void print(String methodName, Object oldValue, Object
  newValue){
    System.out.println("[Method=" + methodName +
                       "|Old Value=" + oldValue +
                       "|New Value=" + newValue+"]");
  }
}
```

`SetterInfoDBPrinter` as this:

```
package com.packtpub.springhibernate.ch03;
public class SetterInfoDBPrinter {
  public void print(String methodName, Object oldValue, Object
  newValue){
    //connect to database and insert the information
  }
}
```

`SetterInfoLogPrinter` as this:

```
package com.packtpub.springhibernate.ch03;
public class SetterInfoLogPrinter {
  public void print(String methodName, Object oldValue, Object
  newValue){
    //insert the method information in the log file
  }
}
```

As you can see, all of these classes contain a common `print()` method, which prints the setter method runtime arguments. (To keep the example simple, I have shown only the implementation of `SetterInfoConsolePrinter`.) All of the application classes setter methods use one of these three printer classes to print their information. For instance, the `Student` class that uses the `SetterInfoConsolePrinter` class looks like this:

```
package com.packtpub.springhibernate.ch03;

public class Student {

  private int id;
  private String firstName;
  private String lastName;
  private SetterInfoConsolePrinter printer = new
  SetterInfoConsolePrinter();

  //zero-argument
  public Student() {
  }

  public Student(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public void setId(int id) {
  printer.print("setId", this.id, id);
    this.id = id;
  }

  public void setFirstName(String firstName) {
  printer.print("setFirstName", this.firstName, firstName);
    this.firstName = firstName;
  }

  public void setLastName(String lastName) {
  printer.print("setLastName", this.lastName, lastName);
    this.lastName = lastName;
  }
  //getter methods
  //hashCode() and equals() methods

}
```

Note that, we have intentionally not shown the getter, `hashCode()`, and `equals()` methods, which do not affect our example. In this case, it is obvious that if we want to change the printing strategy to use a different destination, we must change all classes and change the `SetterInfoConsolePrinter` class with an appropriate printer class.

The code we have just implemented can be refactored by adding a `SetterInfoPrinter` interface and a `SetterInfoPrinterFactory` class. The `SetterInfoPrinter` outlines the `print()` method as the common method for all printer classes, and the `SetterInfoPrinterFactory` class acts as a factory for printer classes, allowing centralized control of the printer class instantiation. This is `SetterInfoPrinter`:

```
package com.packtpub.springhibernate.ch03;

public interface SetterInfoPrinter {
  public void print(String methodName, Object oldValue, Object
  newValue);
}
```

And here is `SetterInfoPrinterFactory`:

```
package com.packtpub.springhibernate.ch03;

public class SetterInfoPrinterFactory {

  public static SetterInfoPrinter getSetterInfoPrinter(){
    return new SetterInfoConsolePrinter();
  }
}
```

All printer classes now need to implement the `SetterInfoPrinter` interface, so we will have `SetterInfoConsolePrinter` as follows:

```
package com.packtpub.springhibernate.ch03;

public class SetterInfoConsolePrinter implements SetterInfoPrinter{
  public void print(String methodName, Object oldValue, Object
  newValue){
    System.out.println("[Method=" + methodName +
                       "|Old Value=" + oldValue +
                       "|New Value=" + newValue+"]");
  }
}
```

And `SetterInfoDBPrinter` as this:

```
package com.packtpub.springhibernate.ch03;

public class SetterInfoDBPrinter implements SetterInfoPrinter{
  public void print(String methodName, Object oldValue, Object
  newValue){
    //connect to database and insert the information
  }
}
```

And `SetterInfoLogPrinter` as this:

```
package com.packtpub.springhibernate.ch03;
public class SetterInfoLogPrinter implements SetterInfoPrinter{
  public void print(String methodName, Object oldValue, Object
  newValue){
    //insert the method information in the log file
  }
}
```

The `Student` class, and other classes, can now use the `SetterInfoPrinterFactory` class to obtain a printer class. Here is the refactored `Student` class that uses `SetterInfoPrinterFactory`:

```
package com.packtpub.springhibernate.ch03;
public class Student {
  private int id;
  private String firstName;
  private String lastName;
  private SetterInfoPrinter printer;
  //zero-argument
  public Student() {
    printer = SetterInfoPrinterFactory.getSetterInfoPrinter();
  }
  public Student(String firstName, String lastName) {
    this();
    this.firstName = firstName;
    this.lastName = lastName;
  }
  public void setId(int id) {
    printer.print("setId", this.id, id);
    this.id = id;
  }
  public void setFirstName(String firstName) {
    printer.print("setFirstName", this.firstName, firstName);
    this.firstName = firstName;
  }
  public void setLastName(String lastName) {
    printer.print("setLastName", this.lastName, lastName);
    this.lastName = lastName;
  }
  //getter methods
  //hashCode() and equals() methods
}
```

In this class, we have replaced the instance of the concrete `SetterInfoConsolePrinter` with an instance of `SetterInfoPrinter`. The class constructor uses the `SetterInfoPrinterFactory` class to obtain a `SetterInfoPrinter` instance when the `Student` class is instantiated. The only other constructor class calls the default constructor to guarantee that the printer instance is always instantiated.

Now, our code is more manageable, that is, if the printing strategy changes, only `SetterInfoPrinterFactory` requires modification. The following code snippet shows a simple Java program that instantiates a `Student` object:

```
package com.packtpub.springhibernate.ch03;

public class Main {
  public static void main(String[] args) {
    Student student = new Student();
    student.setId(41);
    student.setFirstName("John");
    student.setLastName("White");
  }
}
```

The program's execution result is as follows:

```
[Method=setId|Old Value=0|New Value=41]
[Method=setFirstName|Old Value=null|New Value=John]
[Method=setLastName|Old Value=null|New Value=White]
```

Although the code we have developed so far works very well, there are still two maintenance problems:

- Developers must worry about the `SetterInfoPrinter` instantiation in the constructor of the classes
- Developers should be careful about invoking the `print()` method before setting a new value for each property

Let's see how Spring IoC can solve the first problem. The second problem is solved with Spring AOP, which is outside the scope of this chapter. However, you can read about the Spring AOP in Chapter 11.

Applying IoC

You can use IoC to eliminate the first problem: the `SetterInfoPrinter` instantiation when the `Student` instance is created. According to IoC, object instantiation can be accomplished by an outside object, instead of by the owner object. In our case, this means we can leave the `SetterInfoPrinter` instantiation to the Spring IoC container as an outside object, and always look up the Spring IoC container to obtain the `Student` instances.

In our example, applying IoC requires the following three steps:

1. Remove the `SetterInfoPrinter` instantiation from the `Student` constructor, and implement the setter method for the instance to be provided by the outside object.
2. Configure the `Student` object to be created and managed by the Spring container, so that Spring knows how to create `Student` instances and do the `SetterInfoPrinter` instantiation.
3. Obtain the configured `Student` instances from the Spring container instead of initializing them directly.

Let's see how to implement each step.

Remove object instantiation and implement the setter method

In the first step, you need to refactor the `Student` class so that the `SetterInfoPrinter` is not instantiated in the constructor and, instead, the setter method is implemented for the `SetterInfoPrinter` instance. Spring uses the setter method to provide an instantiated `SetterInfoPrinter` for the `Student` instance. The following code snippet shows the refactored `Student` class:

```
package com.packtpub.springhibernate.ch03;

import java.io.Serializable;

public class Student {

  private int id;
  private String firstName;
  private String lastName;
  private SetterInfoPrinter printer;

  //zero-arguement constructor
  public Student() {
    //no SetterInfoPrinter instantiation
  }
```

```
    public Student(String firstName, String lastName) {
      this();
      this.firstName = firstName;
      this.lastName = lastName;
    }
    public void setId(int id) {
      printer.print("setId", this.id, id);
      this.id = id;
    }
    public void setFirstName(String firstName) {
      printer.print("setFirstName", this.firstName, firstName);
      this.firstName = firstName;
    }
    public void setLastName(String lastName) {
      printer.print("setLastName", this.lastName, lastName);
      this.lastName = lastName;
    }
    public SetterInfoPrinter getPrinter() {
      return printer;
    }
    public void setPrinter(SetterInfoPrinter printer) {
      this.printer = printer;
    }
    //getter methods
    //hashCode() and equals() methods
  }
```

Configure the Student object

Next, configure the `Student` object so that Spring knows how to create and configure `Student` instances.

Spring uses an XML file called **Spring context**. This file includes information about all of the objects that are created and managed by Spring. Each object in this file is called a bean and is defined through the `<bean>` element. A simple Spring context, called `spring-context.xml`, which declares information only about the `Student` object, is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
      xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
    <bean id="student"
     class="com.packtpub.springhibernate.ch03.Student">
      <property name="printer">
        <bean
   class="com.packtpub.springhibernate.ch03.SetterInfoConsolePrinter"/>
      </property>
    </bean>
  </beans>
```

Like any other XML file, this file starts with the XML declaration, specifying the XML version and the document encoding being used. The `<beans>` element is the root element in this file, which introduces the namespaces and the schema document according to how the document should be validated. The `<beans>` element includes the entire object configuration defined through individual `<bean>` elements. Each `<bean>` element declares an object that is created and managed by Spring. The `<bean>` element can come with two attributes: `id` and `name`. `id` assigns an identifier to the object, by which the object is identified in the Spring context and which the application uses to look up the object. The `name` attribute determines the concrete class from which the object must be instantiated. The `<bean>` element can nest with an arbitrary number of `<property>` elements; each one determines a property of the object that is created, managed, and set by Spring. The `<property>` element may either refer to another managed object inside the Spring context through the nested `<ref>` element, or may use a nested `<bean>` element to define the class from which the property is instantiated.

In our example, we have used the nested `<bean>` element, so we cannot reuse the `SetterInfoConsolePrinter` definition. However, it is possible to use the `<ref>` element if we want the object to be reused by other instances, as follows:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="student"
     class="com.packtpub.springhibernate.ch03.Student">
      <property name="printer">
        <ref local="printer"/>
      </property>
    </bean>

    <bean id="printer"
    class="com.packtpub.springhibernate.ch03.SetterInfoConsolePrinter">
    </bean>
  </beans>
```

Obtain the Student instance from Spring

Finally, replace direct instantiation of the `Student` object with looking up and obtaining the object from Spring. The following shows the `Main` class, which now uses Spring to obtain the `Student` instance:

```
package com.packtpub.springhibernate.ch03;

import org.springframework.context.support.
ClassPathXmlApplicationContext;
import org.springframework.context.ApplicationContext;

public class Main {
  public static void main(String[] args) {

    ApplicationContext ctx = new ClassPathXmlApplicationContext(
                           "com/packtpub/springhibernate/ch03/
                            spring-context.xml");
    Student student = (Student)ctx.getBean ("student");
    student.setId(41);
    student.setFirstName("John");
    student.setLastName("White");
  }
}
```

As you can see, we have used the `ApplicationContext` interface and an implementation of this interface, that is, `ClassPathXmlApplicationContext`, to interact with Spring. `ClassPathXmlApplicationContext` fires up the Spring IoC container with an XML context file in the application classpath. The constructor of `ClassPathXmlApplicationContext` takes the relative path of the Spring context file as a `String` argument. When the `ClassPathXmlApplicationContext` object is created, the `getBean()` method of the `ApplicationContext` instance, with the name of the object configured in the Spring context, is called to obtain the configured object. By executing the `Main` class, you can see the same result as when you do not use IoC.

As mentioned before, IoC allows you to manage the object dependencies at the development or maintenance time, and inject the dependencies at runtime. This functionality made IoC to be renamed with another descriptive name **Dependency Injection (DI)**.

IoC is discussed more in depth in Chapter 10.

Hibernate with Spring

As mentioned so far, Hibernate and Spring essentially are used for different purposes. Although there are many Spring features, including IoC, which make the Hibernate application more maintainable, the main feature of Spring in Hibernate applications is **Aspect-Oriented Programming(AOP)**. AOP is an added feature to the traditional Object-Oriented programming. As you will see, this feature allows us to define common properties for a set of methods. In the case of Hibernate, suppose that there are a range of persistence methods that all need transaction management. AOP lets us to start transactions at the beginning of those methods and commit at the end, or roll back if any exception occurs during the method executions. The main benefit of this style of programming is using a declarative approach that, like IoC, makes the application clean, simple, and more maintainable.

We will discuss Spring's AOP in-depth in an individual chapter in this book.

Summary

In this chapter, you learned the fundamental concepts behind using Hibernate and Spring. We started with a simple `Student` class. Next, we used an XML mapping document to define mapping of the class properties to the table columns. Then, we called Hibernate to persist a `Student` object.

You must configure Hibernate before you can use it. You can configure Hibernate either programmatically or declaratively. However, for flexibility and maintenance, the declarative approach is highly recommended.

Our discussion continued with other issues in object persistence, including object querying, caching, and transaction management.

Finally, we looked at the Spring IoC container. Spring provides IoC functionality in its core API, meaning that all other Spring services sit around IoC. IoC allows us to create and manage objects using the IoC container, instead of using the owner objects themselves.

4 Hibernate Configuration

In Chapter 3, we took a quick tour of the fundamental concepts behind Hibernate and Spring. In fact, the steps in using Hibernate do not require much beyond what you have already learned. However, since Hibernate provides a persistence service for many different environments, it has been designed to be flexible enough for use in different environments, under different circumstances, supplying any persistence requirements.

Hibernate allows many configuration options, based on an application's persistence requirements. However, most of these parameters have default values, which relieve us of detailed configuration in most situations.

In this chapter, I will show how Hibernate is configured and examine Hibernate's configuration settings. To begin, let's look at the basic configuration information that's commonly required. You can find more information about Hibernate configuration in the Appendix.

Basic configuration information

Through its configuration, Hibernate allows us to choose either Hibernate-managed JDBC connections or a container-managed data source. If you decide to use Hibernate-managed connections, you need to tell Hibernate about the database properties, such as the name of the driver class, the database JDBC URL, and the database username and password. These are the basic configuration settings for Hibernate-managed connections. Each of these settings is represented by a name, as explained in the following table:

Property name	Property value
`hibernate.connection.driver_class`	The fully qualified classname of the database's JDBC driver
`hibernate.connection.url`	The database's JDBC URL
`hibernate.connection.username`	The database username
`hibernate.connection.password`	The database password
`hibernate.dialect`	The Hibernate dialect class corresponding to the used database (see the next section)

To use a container-managed data source, you need to give Hibernate the JNDI information of the configured datasource object. The following table shows the configuration properties for container-managed connections:

Property name	Property value
`hibernate.connection.datasource`	Datasource JNDI name
`hibernate.jndi.class`	The fully qualified name of the initial context factory class for JNDI
`hibernate.jndi.url`	URL of the JNDI provider
`hibernate.connection.username`	Database username
`hibernate.connection.password`	Database password
`hibernate.jndi.<JNDIpropertyname>`	Used for specifying other JNDI properties
`hibernate.dialect`	The Hibernate dialect class corresponding to the database used (see the next section)

Between all of these properties, usually `hibernate.connection.datasource` and `hibernate.dialect` are used.

Hibernate uses many more properties than listed in these tables, but we'll put that discussion aside and look at dialect, because dialect is a very important aspect of Hibernate. Other properties are presented in the following chapters as we discuss relevant topics.

You already know a little about Hibernate dialect. Here, let's dig deeper into Hibernate dialect and see what exactly a dialect does.

Hibernate dialect

Although JDBC and SQL allow all Java applications to connect to all databases, we can't use special database features relying only on standard JDBC and SQL. In other words, although most interactions with all databases are treated similarly through the standard ANSI SQL, many database properties are essentially used differently, and we need to use a slightly different version of SQL to interact with them.

For example, databases may differ in the following ways:

- Different databases may define identity columns differently
- Different databases may support different column types
- Different databases may use SQL syntaxes
- Different databases may define foreign key columns differently
- Not all databases support cascading delete

To solve this problem, Hibernate dialect allows us to use database features which are provided differently by different databases, without having to worry about the underlying details. We use a common syntax, and Hibernate translates our commands into the language of the database we are using, by applying the appropriate dialect. This guarantees application portability.

From an implementation perspective, each dialect is a subclass of `org.hibernate.dialect.Dialect`, defining the use of a database's special features in a standard way. Each database has its own, individual dialect class. To use a database with Hibernate, the dialect class for the database must be specified through the mandatory `hibernate.dialect` entry in the configuration file. You can learn more about Hibernate at `http://docs.jboss.org/hibernate/stable/core/reference/en/html/session-configuration.html#configuration-optional-dialects`. The following table shows the supported databases and their respective dialect classes. You can find the updated list of supported database names on the Hibernate website.

> When you use Hibernate, don't consider the special features of the underlying database. In other words, all applications should view the database only as much as Hibernate exposes it and not rely on special features of the database that are not exposed by Hibernate. Otherwise, the application will depend on the database and will not be portable, which may reduce scalability and cause maintenance problems.

Dialect (All in `org.hibernate.dialect` package)	**Database name**
`DB2390Dialect`	DB2/390
`DB2400Dialect`	DB2/400
`DB2Dialect`	DB2
`DerbyDialect`	Derby
`FirebirdDialect`	Firebird
`FrontBaseDialect`	FrontBase
`HSQLDialect`	HSQLDB
`InformixDialect`	Informix
`IngresDialect`	Ingres
`InterbaseDialect`	Interbase
`MckoiDialect`	Mckoi
`MySQLDialect`	MySQL
`MySQLInnoDBDialect`	MySQL with InnoDB tables
`MySQLMyISAMDialect`	MySQL with MyISAM tables
`Oracle9Dialect`	Oracle 9
`OracleDialect`	Oracle
`PointbaseDialect`	PointBase
`PostgreSQLDialect`	PostgreSQL
`ProgressDialect`	Progress
`SAPDBDialect`	SAP DB
`SQLServerDialect`	SQL Server
`Sybase11Dialect`	Sybase 11
`SybaseAnywhereDialect`	Sybase Anywhere
`SybaseDialect`	Sybase

If you want to use a database that does not have any built-in dialect in Hibernate, you must implement an appropriate dialect class, according to the database.

Let's now see how to use Hibernate's basic properties.

Configuring Hibernate

As you saw in Chapter 3, you must configure Hibernate before you can use it. The configuration is represented by an object of type `org.hibernate.Configuration`. This object wraps all of the configuration settings and is used in the next step for building a `SessionFactory` object. (We'll discuss a useful strategy for creating the `SessionFactory` at the end of this chapter.) To configure Hibernate, you first need to instantiate a configuration object:

```
Configuration cfg = new Configuration();
```

The actual configuration settings can be divided into two parts:

- **Hibernate properties***:* These include all of the database-related properties that Hibernate should consider while connecting to the database, as we saw earlier.
- **Hibernate mappings***:* These are XML documents that define the mapping of the entity classes to the database tables. You'll see a lot more on Hibernate mappings in Chapter 5 and Chapter 6.

Once you have instantiated a `Configuration` object, there are two ways to configure Hibernate, either programmatically and declaratively. Let's first look at programmatic configuration.

Programmatic configuration

Hibernate properties are set in the configuration object through its `setProperty()` method:

```
public Configuration setProperty(String propertyName, String
propertyValue)
```

The first argument determines the name of a Hibernate property that we want to configure. The second indicates the actual value for that property. The following snippet shows an example:

```
cfg.setProperty("hibernate.connection.url",
               "jdbc:hsqldb:hsql://localhost/hiberdb");
```

Note that the `setProperty()` method returns a `Configuration` object. This allows method chaining to set other properties. For instance, you can set all required properties in one single line of code, as follows:

```
Configuration cfg = new Configuration()
    .setProperty("hibernate.connection.url",
                 "jdbc:hsqldb:hsql://localhost/hiberdb")
    .setProperty("hibernate.dialect", "org.hibernate.dialect.
                  HSQLDialect")
    .setProperty("hibernate.connection.username", "sa")
    .setProperty("hibernate.connection.password", "")
    .setProperty("hibernate.connection.driver_class",
                 "org.hsqldb.jdbcDriver");
```

Another way to set up the configuration object is by passing an instance of `java.util.Properties` to the `setProperties()` method:

```
public Configuration setProperties(Properties properties)
```

To set an XML mapping file, use the `addResource()` method of the configuration object:

```
public Configuration addResource(String resourcePath) throws
MappingException
```

The parameter of this method refers to the relative path of the mapping file in the classpath.

The following snippet shows an example:

```
config.addResource("com/packtpub/springhibernate/ch04/Student.hbm.
xml");
```

In this example, the mapping file is in the `com.packtpub.springhibernate.ch04` package in the classpath, next to its respective entity class, `com.packtpub.springhibernate.ch04.Student`. Although it's a good idea to place each mapping file next to its respective entity class, you don't need to do that.

An alternative way to configure mapping files, when they are in the same package as their respective entity classes, is using the `addClass()` method:

```
public Configuration addClass(Class aClass) throws MappingException
```

This method takes the entity class as an argument to find its mapping file, as in this example:

```
config.addClass(com.packtpub.springhibernate.ch04.Student.class);
```

Hibernate then looks for a mapping file named `com/packtpub/springhibernate/ch04/Student.hbm.xml` in the classpath.

This following snippet shows the programmatic configuration of Hibernate:

```
Configuration config = new Configuration();
config.setProperty("hibernate.dialect",
                   "org.hibernate.dialect.HSQLDialect");
config.setProperty("hibernate.connection.driver_class",
                   "org.hsqldb.jdbcDriver");
config.setProperty("hibernate.connection.url",
                   "jdbc:hsqldb:hsql://localhost/hiberdb");
config.setProperty("hibernate.connection.username", "sa");
config.setProperty("hibernate.connection.password", "");
config.addClass(Student.class);
config.addClass(Teacher.class);
config.addClass(Course.class);
```

Note that all of these configuration settings are set only once, when the first interaction with Hibernate is performed.

Because the actual configuration information is specified in the deployment phase, or may change in during the lifetime of the application, Hibernate, like other frameworks, should be configured declaratively, instead of programmatically as we have done in this section.

Declarative configuration

The declarative approach is an alternative way to configure Hibernate. In this approach, an XML or a properties file sets up Hibernate properties. This file is then loaded in a `Configuration` object at runtime.

Hibernate supports both XML and properties files as the Hibernate configuration file, and there is actually no difference between them in terms of performance. However, the XML variant is preferred as this approach is much more flexible when specifying `hbm` mapping files. Let's see how each approach is used in practice.

Using a properties file

As you know, properties files are text files with the `properties` suffix. Their contents are keys and values:

```
key=value
```

Every key is a unique word or phrase that corresponds to another word or phrase as the value. Therefore, every key leads you or the application to the corresponding value.

The following code shows a properties file's contents, specifying the basic configuration information of Hibernate:

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

To configure Hibernate with a properties file, you need to only create a `Configuration` object as follows:

```
Configuration config = new Configuration();
```

In this approach, Hibernate searches for a file named `hibernate.properties` in the root of the classpath. If this file exists, all `hibernate.*` properties are loaded and added to the `Configuration` object. When using this approach, you need to introduce mapping files to Hibernate programmatically, like this:

```
config.addClass(Student.class);
config.addClass(Teacher.class);
config.addClass(Teacher.class);
```

Note that Hibernate allows neither a different name for this file nor any location other than the root of the classpath. If you want to use a different file, or the file in a different location than the root of the classpath, you must use the XML approach.

Using an XML file

As mentioned earlier, another variant of declaratively configuring Hibernate is the XML approach. Because an XML file allows greater flexibility in specifying `hbm` mapping files, it's preferred to using a properties file and programmatic configuration. The following code shows a simple XML configuration file with the basic configuration entries:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
          "http://hibernate.sourceforge.net/hibernate-configuration
          -3.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- Hibernate Properties -->
    <property name="connection.driver_class">
```

```
      org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">
      jdbc:hsqldb:hsql://localhost/hiberdb
    </property>
    <property name="connection.username">sa</property>
    <property name="connection.password"> </property>
    <property name="pool_size">5</property>
    <property name="show_sql">false</property>
    <property name="dialect">
      org.hibernate.dialect.HSQLDialect
    </property>

    <!-- Mapping files -->
    <mapping resource="com/packtpub/springhibernate/ch04/
                                  Student.hbm.xml"/>
    <mapping resource="com/packtpub/springhibernate/ch04/
                                  Teacher.hbm.xml"/>
    <mapping resource="com/packtpub/springhibernate/ch04/
                                   Course.hbm.xml"/>

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

As this shows, the configuration entry names in the XML file are similar to the corresponding names in the properties file, only without the `hibernate` prefix. For example, the `hibernate.connection.username` entry in the properties file corresponds to the `connection.username` entry in the XML file.

To configure Hibernate with an XML file, create a `Configuration` object and then call its `configure()` method:

```
Configuration cfg = new Configuration();
cfg.configure();
```

When the `configure()` method is called, Hibernate searches the root of the classpath for a file named `hibernate.cfg.xml`. If this file exists, Hibernate loads all of the settings and then adds them to the `Configuration` object. If the file doesn't exist, Hibernate throws a `HibernateException`.

If both `hibernate.properties` and `hibernate.cfg.xml` exist in an application, Hibernate first loads `hibernate.properties`, then overrides the loaded settings with the values defined in `hibernate.cfg.xml`.

If you want to use a file with a different name, or a file in a different location than the root of the classpath, you need to call the `configure()` method of the `Configuration` object and pass the file path as the argument. Here is an example:

```
Configuration cfg = new Configuration();
cfg.configure("/conf/hibernate.cfg.xml");
```

Using a single instance of SessionFactory

As mentioned in Chapter 3, the application uses session objects to interact with Hibernate and to request Hibernate to perform a persistence operation. This means the application needs to obtain a session object before issuing a persistence request. These objects are not thread-safe, so they cannot be shared. They undertake transactions, manage caching, and provide other controls on the persisting of the entity objects. However, session objects are not directly instantiated. Instead, they are constructed by another object called `SessionFactory`, which is thread-safe and acts as a factory for sessions.

As you saw in Chapter 3, the `SessionFactory` object is also not directly instantiated directly. Instead, it is constructed through the `Configuration` object by invoking the `buildSessionFactory()` method:

```
Configuration cfg = new Configuration();
…
SessionFactory sessions = cfg.buildSessionFactory();
```

There are many reasons to use only one instance of `SessionFactory` for an entire application. For example, `SessionFactory` is an expensive object. Instantiating it is a time-consuming operation and uses system resources. A `SessionFactory` is roughly analogous to a JDBC connection pool, which holds JDBC connections. However, it does more. It instantiates and prepares `Session` objects and binds them to the JDBC connections. It should be noted that only a single instance of `SessionFactory` is used throughout the application for each database that the application uses. However, situations when the application uses more than one database schema, you must have one `SessionFactory` for each database.

The following code shows the `HibernateHelper` class designed for creating a single instance of `SessionFactory` throughout the application. As you can see, the constructor of the class is `private`, meaning no objects of the class can be instantiated by other objects.

As you will see in the coming chapters, you will not worry about this class anymore if you are using Hibernate with Spring. You can omit this class from the application and let Spring configure Hibernate and provide a single instance of `SessionFactory`, and you can even let Spring provide `Session` instances behind the scene.

The `sessionFactory` object is a private, static member of the class. This member represents the single instance of `SessionFactory` used for the entire application. The class also includes the `getSession()` method, which provides the session objects for the rest of the application. Actually, none of the classes that interact with Hibernate needs to configure Hibernate or obtain the `SessionFactory` object itself. A class should only invoke the `getSession()` method of this class to get a session object and then perform its desired persisting operation:

```
package com.packtpub.springhibernate.ch04;

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

public class HibernateHelper {

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

  //inaccessible constructor
  private HibernateHelper() {
  }

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

JPA configuration

The JPA compatible EntityManager is configured through the persistence.xml file located in the META-INF directory of the classpath. The persistence.xml file starts with the XML declaration and the persistence element as root. The root defines the schema, determining the version of JPA according to which the document is created. Inside the root element, an unlimited number of persistence-unit elements can be present. In most situations, only a single persistence-unit element is used. However, in rare situations when you are using two or more persistence strategies together in your application (for example, Hibernate for persisting some classes and iBatis for others), you can distinct them with individual persistence-unit elements. Here is an example of persistence-unit element with possible attributes and subelements:

```
<persistence-unit name="AUniqueName" transaction-type="JTA|RESOURCE_
LOCAL">
    <provider>FullyQualifiedClassName</provider>
    <jta-data-source>JNDIName</jta-data-source>
    <jar-file>JARFileName</jar-file>
    <class>EntityClassName</class>
    <properties>
        <property name="propertyName" value="propertyValue"/>
    </properties>
</persistence-unit>
```

The meaning of these attributes and subelements are as follows:

- name is the identifier of the persistence unit. The EntityManger object for this unit is configured by the persistence unit identified by this name.
- transaction-type specifies the transaction type of this persistence unit. The valid values are JTA and RESOURCE_LOCAL. When a jta-datasource is used, the default is JTA, and if non-jta-datasource is used, RESOURCE_LOCAL is used. The default value in a JavaEE environment is JTA and in a JavaSE environment is RESOURCE_LOCAL.
- provider specifies the fully qualified class name of the JPA provider. In case of Hibernate, it's value is org.hibernate.ejb.HibernatePersistence.
- jta-data-source or non-jta-data-source specifies the JNDI name of the datasource being used. You may use none of these and use <property> elements, as described below, letting Hibernate create and manage connections by itself.

- `jar-file` refers to a JAR file which includes annotated classes, annotated packages, and all `hbm.xml` files to be added to the persistence unit configuration. This element is mainly used in Java EE environment.
- `exclude-unlisted-classes` determines if the annotated classes should explicitly be listed with the `<class>` elements or all should be loaded from the JAR file. The default value for this element is `true` in Java SE, and `false` in Java EE.
- `class` specifies a fully qualified class name of a persistent class. By default, if `exclude-unlisted-classes` is set to `true`, all properly annotated classes and all `hbm.xml` files found inside the archive are added to the persistence unit configuration. This element also allows you to add some external entities to the persistence unit.
- `properties` lets you to specify vendor-specific properties. This is where you will define your Hibernate-specific configurations, JDBC connection information, and so on.

To start up Hibernate with JPA, you should set up the EntityManager with the just configured persistence unit. This is done through the static `createEntityManagerFactory()` method of the `Persistence` class, as follows:

```
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("instituteWebApp");
```

`instituteWebApp` is the value of the `name` attribute of the used `persistence-unit`.

> If you are deploying in Java SE environment, you will need a utility class, like `HibernateHelper`, to provide only a single instance of the `EntityManagerFactory` object throughout the application.

It is also possible to configure `EntityManagerFactory` programmatically. Here is an example:

```
Map properties= new HashMap();
properties.put("hibernate.show_sql", "true");
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("instituteWebApp",
properties);
```

It then overrides any property already set in the `persistence.xml` file.

Alternatively, you can use a Hibernate-native API to create and configure the `EntityManagerFactory` instance. Hibernate provides the `org.hibernate.ejb.Ejb3Configuration` class for this purpose. It can be used as follows:

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
cfg.configure("/hibernate.cfg.xml")
    .setProperty("hibernate.show_sql", "true")
    .addAnnotatedClass( hello.Message.class)
    .addResource( "/Student.hbm.xml")
    .buildEntityManagerFactory();
```

After the `EntityManagerFactory` is created, it can be used as follows to obtain `EntityManager` objects to perform unit of works:

```
EntityManager em = emf.createEntityManager();
```

The `EntityManagerFactory` object should be closed by invocation of it's `close()` method when the application has finished using this object, or the application shutdowns. Once the `EntityManagerFactory` has been closed, all of its `EntityManager`s are considered to be in the closed state. You will learn more about the `EntityManager` class in Chapter 8.

Summary

In this chapter, we discussed Hibernate configuration settings and how Hibernate is configured. Hibernate lets us use a container-managed data source or Hibernate-managed connections. In either case, you need to specify the database username, password, and dialect. Additionally, when you use a container-managed data source, you must also specify the JNDI name of the data source. When you use a Hibernate-managed data source, you need to indicate the database URL and the classname of the database driver. Although this is the basic configuration information for Hibernate, Hibernate allows more configuration settings, making it an appropriate solution for any application deployed in any environment with any persistence requirement.

You have two options when configuring Hibernate, that is, the programmatic approach and the declarative approach. Although there is no difference between these two approaches as far as Hibernate is concerned, I prefer the declarative variant, because of its greater flexibility and lower maintenance cost.

Hibernate also allows you to use an XML or a properties file as its configuration file. If you choose the XML variant, you can also specify the mapping files through it, in addition to other properties. Hibernate uses `hibernate.properties` and `hibernate.cfg.xml` as the default names for the configuration files. If both files exist in the application classpath, the `hibernate.properties` file is read first, then all of the read settings are overridden by the values defined in `hibernate.cfg.xml`.

In Hibernate, every persisting operation is performed through session objects. These are not instantiated directly. Instead, they are obtained by a `SessionFactory` object. This object is built by a `Configuration` object which wraps all configuration information.

You only need one instance of `SessionFactory` for each database in the application.

5
Hibernate Mappings

In earlier chapters, you have learned how to set up a project from scratch to use with Hibernate. You also learned how Hibernate is configured after it has been installed. After that, the application is ready to use Hibernate for persisting your objects.

Obviously, we are not going to persist all application objects in the database. Only the business entities of the problem domain (such as students and courses in an educational system application) are persistent. The classes that implement these entities are called *entity classes*. The persistence with Hibernate is called *transparent*, because the persistent classes never use or call Hibernate APIs and are not influenced by the persistent logic. These classes are merely simple data holders. We will discuss the implementation details and the persistence capability of these classes in this chapter.

For the next phase in Hibernate development, you'll need to tell Hibernate how it should map each entity class to its respective database table. This process is called *object mapping* and may involve setting up XML mapping files and/or annotating Java classes with Hibernate annotations, to determine the mapping information. To complete the process of object mapping, you introduce implemented mapping files and annotated Java classes to Hibernate through the Hibernate configuration file. Mapping implementations are the subject of this chapter.

Effective database schema design, proper object design, and proper object mapping are usually the most critical parts of application development. Any inappropriate design of the database's tables, inappropriate object design, or inappropriate object mapping hurts the application's performance and may limit its scalability.

This chapter discusses basic issues related to persistent objects and their mappings. Our discussion begins with simple cases, moving on to advanced and practical ones. The discussion continues into Chapter 6, which presents some advanced concepts behind Hibernate mappings.

Persistent entity classes

Entity classes are absolutely unaware of any Hibernate APIs, the underlying database and communication language (SQL), and any persistence issues, such as transaction. It is the role of Hibernate to work with these simple classes, handle persistence issues related to them, and provide a transparent persistence as the result.

The structure of persistent classes in Hibernate follows the **Plain Old Java Object (POJO)** programming model. POJOs are a customized and simplified form of JavaBeans, a component model for user interface development in Java. However, unlike JavaBeans, POJOs can be used in any layer of application architecture. They are the essence of many new Java frameworks that simply aim for JEE development. In this book, I use POJO to refer to any object that follows the POJO rules, regardless of whether it is persistent. I use the term *persistent class* to refer to any class that is persistent with Hibernate.

Let's see what entity classes look like in practice, before we dig more deeply into the POJO model and its Hibernate requirements to make these entity classes persistent.

POJOs are simply defined as data holders. They have properties, which hold data accessed through setter/getter accessor methods. A simple Hibernate persistent class, `Student`, which was introduced in earlier chapters, is shown below:

```
package com.packtpub.springhibernate.ch05;

import java.util.Date;
import java.util.Calendar;
import java.io.Serializable;

public class Student implements Serializable{
  private int id;
  private String firstName;
  private String lastName;
  private String ssn;
  private Date birthday;
  private String stdNo;
  private Date entranceDate ;

  //zero-argument constructor
  public Student() {
  }

  public Student(int id,
                 String firstName,
                 String lastName,
                 String ssn,
                 Date birthday,
                  String stdNo,
                  Date entranceDate) {
    setId(id);
    setFirstName(firstName);
    setLastName(lastName);
    setSsn(ssn);
    setBirthday(birthday);
    setStdNo(stdNo);
    setEntranceDate(entranceDate);
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getSsn() {
    return ssn;
  }
  public void setSsn(String ssn) {
    this.ssn = ssn;
  }
  public Date getBirthday() {
    return birthday;
  }
  public void setBirthday(Date birthday) {
    this.birthday = birthday;
  }
  public String getStdNo() {
    return stdNo;
  }
  public void setStdNo(String stdNo) {
    this.stdNo = stdNo;
  }
  public Date getEntranceDate() {
    return entranceDate;
  }
  public void setEntranceDate(Date entranceDate) {
    this.entranceDate = entranceDate;
  }
  public int getAge(){
    Calendar c = Calendar.getInstance();
    c.setTime(new java.util.Date(getBirthday().getTime()));
    int birthYear = c.get(Calendar.YEAR);
    c.setTime(new java.util.Date());
    int now = c.get(Calendar.YEAR);
    return now-birthYear;
  }
  public String toString(){
    return id+":"+firstName+":"+lastName+":"+ssn+":["+birthday+"]:";
  }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Student)) return false;

    final Student student = (Student) o;

    if (id == 0) return false;
    if (birthday != null ? !birthday.equals(student.birthday) :
            student.birthday != null) return false;
    if (entranceDate != null ? !entranceDate.equals(student.
        entranceDate) :student.entranceDate != null) return false;
    if (firstName != null ? !firstName.equals(student.firstName) :
            student.firstName != null) return false;
    if (lastName != null ? !lastName.equals(student.lastName) :
            student.lastName != null) return false;
    if (ssn != null ? !ssn.equals(student.ssn) :
            student.ssn != null) return false;
    if (stdNo != null ? !stdNo.equals(student.stdNo) :
            student.stdNo != null) return false;

    return true;
  }
  public int hashCode() {
    int result;
    result = 29 * result + (firstName != null ?
                    firstName.hashCode() : 0);
    result = 29 * result + (lastName != null ?
                     lastName.hashCode() : 0);
    result = 29 * result + (ssn != null ? ssn.hashCode() : 0);
    result = 29 * result + (birthday != null ?
                    birthday.hashCode() : 0);
    result = 29 * result + (stdNo != null ? stdNo.hashCode() : 0);
    result = 29 * result + (entranceDate != null ?
                    entranceDate.hashCode() : 0);
    return result;
  }
}
```

Hibernate does not require persistent classes to extend or implement any Hibernate-specific classes, or call any other particular API. Every persistent class can be part of any class hierarchy, or may call or use any arbitrary API.

> If the persistent objects are distributed over the network, or are stored in an `HttpSession`, they must implement the `java.io.Serializable` interface.

Although most POJOs are compatible with Hibernate, they should meet a few mandatory and optional requirements to be persistent with Hibernate. Upcoming subsections discuss these requirements.

Having a zero-argument constructor (mandatory)

As Hibernate uses the Reflection API through the `newInstance()` method with no-arguments to instantiate persistent objects, every persistent class must have a zero-argument (empty) constructor. You can either define this constructor yourself, or not write any constructors at all, which produces a default empty constructor.

The zero-argument constructor can be nonpublic. However, it is highly recommended that it is at least package-visible. This allows Hibernate to generate runtime proxies to optimize performance. Generating runtime proxies is a mechanism Hibernate uses to avoid unnecessary loading of persistent objects, thereby enhancing performance. This is the basis of Hibernate's lazy loading feature. Obviously, the subclass cannot be created if the persistent class has a private constructor or is defined as final. We'll discuss runtime proxies in more depth in Chapter 8.

Providing accessors to access class properties (optional)

Each persistent class may have properties that represent the data which the class holds, defined as simple values, or other persistent classes, or both. These properties are usually defined as nonpublic instance variables, and are accessed through public accessor methods called **setters** and **getters** with the name of `getXxx` and `setXxx`. For example, for the `Student` class with the `firstName` property of type `String`, the getter and setter methods are respectively named `getFirstName()` (which takes no arguments and returns a `String`) and `setFirstName()` (which takes a `String` and has a `void` return type). Notice the lower-case `f` in the property name, and the upper-case `F` in the method names).

> If a class has a `getXxx` method, but no corresponding `setXxx`, then the class is said to have a *read-only property* named xxx. The read-only property is useful when there is a property in the class but no corresponding value for that property in the database. In that case, the class may have its own logic to determine the property value, in practice, from other properties, or Hibernate may load the property value from a column which is valued by a database trigger instead of the application. In the `Student` class above, age is a read-only property calculated from the persisted birthdate. You can get more information about JavaBeans at `http://java.sun.com/docs/books/tutorial/javabeans/`.

The exception to this naming convention is Boolean properties, which uses a method called `isXxx` to look up their values. For example, the `Student` class might have methods called `isGraduated()` (which takes no arguments and returns a Boolean) and `setGraduated()` (which takes a Boolean and has a `void` return type), and would be said to have a Boolean property named `graduated`. (Again, notice the lower-case first letter in the property name.)

For a configuration setting, you can specify the strategy that Hibernate uses to access class properties, either through accessor methods or by direct access. However, using accessor methods is recommended, because this allows you to do more than accessing properties (such as imposing constraints on property values or validating them).

> These requirements for accessor methods are rare. They mostly get and set values to the properties.

Defining nonfinal classes (optional)

As explained before, Hibernate generates runtime proxies to enhance performance. To enable Hibernate to generate runtime proxies, the persistent classes either must be declared as nonfinal, letting Hibernate extend them at runtime, or must implement a Hibernate interface. Implementing the Hibernate interface is not recommended, because this method couples the persistent class to Hibernate. Instead, define the persistent classes as nonfinal. Chapter 8 discusses proxy generation in more detail.

Implementing equals() and hashCode() (optional)

Hibernate uses snapshot comparison to automatically detect changes, checking to see if the objects in memory are dirty and need updating. Hibernate does this comparison through the `equals()` and `hashCode()` methods implemented in the persistent classes. The `hashCode()` and `equals()` methods are required if the application uses detached objects (the objects which have already persisted and are being changed). When you implement the `equals()` method, you need to implement the `hashCode()` method as well. When two objects are equal, they must have the same hash code value.

Object/relational mapping metadata

You now have persistent classes that meet Hibernate requirements for persistence. To do its job, Hibernate (or any other object/relational mapping tool) needs information about the persistent classes and the target database tables in which the objects are persisted. It needs detailed information about the mapping of object properties to and from table columns, and of Java types to and from SQL types. It may require additional information about establishing foreign keys when the persistent class is associated with other(s). This is typical information that object/relational mapping tools need. Moreover, Hibernate provides features, such as lazy loading, cascading, caching, and others, that must be configured. This information is represented as **object/relational mapping metadata**.

Hibernate supports the following three different strategies to describe this metadata:

- Setting up XML mapping files
- Annotating persistent classes with Hibernate XDoclet tags
- Annotating persistent classes with Hibernate annotations

The first approach, which is also the traditional approach, maps metadata in Hibernate through XML documents. XML documents are easy to edit and maintain, even for administrators who do not know much about Java. Developing XML-based metadata is not effortless, especially when the XML schema is complex and no element or attribute has a default value.

Hibernate has tried to make using XML as simple and convenient as possible. XML-based metadata in Hibernate is well structured, with simple, understandable elements and attributes. Many elements and attributes have default values. Therefore, they can be dropped if they are worthless. For some elements (and even attributes), Hibernate can find the best value by investigating the persistent classes, if those elements or attributes are omitted.

In addition to XML, Hibernate supports two other methods for determining mapping metadata:

- Annotating persistent classes with XDoclet markup
- Annotating persistent classes with Hibernate annotations

Hibernate started supporting annotations with XDoclet when there was no support for annotations in Java. In this approach, persistent classes are annotated with special Hibernate-specific Java doclet tags, and then fed into XDoclet to generate XML mapping metadata. With JDK 5.0 annotations, Hibernate has provided its own, proprietary annotation API for supporting source-level metadata. This book covers both the XML and JPA annotation API approaches to describe metadata.

Next, we'll discuss how XML documents can determine mapping metadata.

Metadata in XML

Let's look at a simple example of object mapping to explore the XML approach. The following code shows the `Student.hbm.xml` file, which represents the mapping metadata for the `Student` class already discussed:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch05.Student"
table="STUDENT">
    <id name="id" column="id" type="int">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <property name="lastName" column="LAST_NAME"   type="string"/>
    <property name="ssn" column="SSN"   type="string"/>
    <property name="birthday" column="BIRTHDAy"   type="date"/>
    <property name="stdNo" column="STD_NO"  type="string"/>
    <property name="entranceDate" column="ENTRANCE_DATE"  type="date"/>
  </class>
</hibernate-mapping>
```

As you can see in this code, the XML-based metadata, like any other XML document, starts with XML and Doctype definitions.

All mapping information is declared inside the `<hibernate-mapping>` element, coming as the root. Within the root, the `<class>` element specifies the mapping information of the persistent class and its respective database table. Other elements are defined inside the `<class>` element to determine the mapping of class properties and their respective table columns, in addition to their corresponding Hibernate types.

The upcoming sections have a more in-depth discussion of these elements and their meanings.

Doctype

All XML mappings should start with the Doctype declaration, referring to the actual path of the DTD document, which is either a file path in the classpath, or an Internet URL.

> The Doctype element is mandatory. Hibernate's XML parser uses this element to validate the mapping document.

The <hibernate-mapping> root element

All mapping definitions in an XML mapping file are enclosed by a `<hibernate-mapping>` element. Typically, the mapping of each persistent class is defined in an individual XML file. However, you may specify many class mappings in a single file, all inside the `<hibernate-mapping>` element.

This element can come with attributes such as the following:

```
<hibernate-mapping auto-import="true|false"
               package="defaultPackage"
               default-cascade="none|all|delete|persist|merge|
               save-update|evict|replicate|lock|refresh|delete-orphan"
               default-lazy="true|false"
               default-access="field|property|PropertyAccessorClass">
          <!-- nested subelements -->
</hibernate-mapping>
```

In this book, underlined words show the default values for each element. All attributes are optional, unless otherwise noted.

All attributes specify the global properties values for all enclosed definitions. If these values have not been specified, the default value will be used. However, as you will see, each enclosed mapping definition can use its own, individual attributes, and consequently override the default values or the defined global values. The previously mentioned attributes are explained as follows:

- `auto-import` allows you to name persistent classes in Hibernate queries with their unqualified names. You may set this to `false` if you think the class name would be ambiguous for Hibernate.
- `package` indicates a package prefix that should be considered for unqualified class names in the mapping document.
- `default-cascade` defines how changes made to an object affect its associated objects when the object is stored, updated, or removed. Using this attribute, you can specify that when the object is stored, updated, or removed, the associated objects are also stored, updated, or removed. In addition to `all`, `delete-orphan`, and `none`, Hibernate provides various cascade values for each basic operation it supports. The default is `none`, meaning that, by default, no persistence operation on an object is propagated to its associated objects. Cascading operations are discussed in detail in Chapter 8.
- `default-lazy` determines whether associated objects are loaded when the object is loaded. The default is `true`, meaning that, by default, every object is loaded with all of its associated objects.
- `default-access` indicates how object fields are accessed, whether directly (`field`) or by getter and setter methods (`property`). Alternatively, you may use the name of a `org.hibernate.property.PropertyAccessor` implementation, which defines a customized access mechanism.

The <class> element

The `class` element is the first nested element inside the root, which defines the mapping of an entity class. The most common attributes of this element are as follows:

```
<class name="className"
       table="tableName"
       abstract="true|false"
       dynamic-update="true|false"
       dynamic-insert="true|false"
```

```
        polymorphism="implicit|explicit"
        lazy="true|false"
        select-before-update="true|false"
        where="SQLWhereCondition">
…
</class>
```

The meanings of these attributes and their possible values are as follows:

- `abstract` specifies whether the class being mapped is an abstract class. This is used when this class in on the top of a class hierarchy, and if any concrete object of this class exists.
- `name` indicates the fully qualified name of the persistent class.
- `table` determines the name of the target database table in which the objects of this entity class are persisted.
- `lazy` enables or disables lazy fetching of associated objects. This attribute is usually used to change the enclosing mapping's default.
- `select-before-update` determines whether Hibernate must load each object before updating it. If the attribute is set to `true`, Hibernate does this to make a comparison and prevent unnecessary updates. However, loading objects before any updating is often inefficient. Sometimes when the database contains triggers, it can prevent unnecessary updates (which may hit triggers).
- `where` specifies an arbitrary SQL `WHERE` condition to be applied on selecting the objects of this class. This attribute allows Hibernate to work with a subset of objects.
- `dynamic-update` indicates that Hibernate should update only modified columns when it issues a `SQL UPDATE` statement for updating an object. The default is `false`, which means Hibernate updates all columns whether or not the corresponding values have changed. When you use `dynamic-update`, Hibernate must have a current snapshot of the object to make a comparison. Providing the snapshot and making a comparison have a cost in performance, so it's almost wise to avoid them. Using `dynamic-update="true"` is recommended only when there are sufficient, convincing reasons, such as when you want to use version-based optimistic locking (explained in Chapter 12), or when you are working with a table that has an extremely large number of columns and only a few columns are updated.

- `dynamic-insert` determines that whether Hibernate should only insert not-null values in columns when it issues a `SQL INSERT` statement for inserting an object or not. The default is `false`, meaning that Hibernate inserts all values whether or not any of these values is null. Using `dynamic-insert="true"` may be useful when columns have defaults and writing null values would override the defaults. Note that using `dynamic-insert="true"` may produce a performance hit, since Hibernate cannot utilize prepared statement caching as efficiently as when null and not-null values are included in the `SQL INSERT` statements.
- `polymorphism` determines if by querying a persistent class, all instances of that class and subclasses of that class should be loaded (`implicit`), or only the instances of that class that are mapped explicitly with Hibernate mapping metadata is loaded (`explicit`). This attribute will be discussed later in this chapter.

All attributes of the `<class>` element are optional.

The <id> element

The `<id>` element determines the object identifier and its corresponding primary key column in the table. This element also specifies how identifiers are generated for new instances of the class. The general form of the `<id>` element is as follows:

```
<id  name="indentifierProperty"
     type="identifierType"
     column="identifierColumn"
     unsaved-value=" null|any|none|undefined|id_value">
     <generator class="identifierGeneratorClass"/>
</id>
```

Here are the meanings of these attributes and their possible values:

- `name` specifies the object identifier's name.
- `column` sets the name of the table column that contains the primary key, and therefore holds the object identifier. If the `column` attribute is not specified, the value of the `name` attribute is considered instead.
- `type` specifies the corresponding Hibernate type for the column. Hibernate uses this value with the selected dialect to pick the correct SQL data type when Hibernate exports the database schema from the mapping metadata. For example, the `long` and `int` values in most databases are transformed to `BIGINT` and `INTEGER` SQL types, respectively. We'll discuss Hibernate types in more depth in Chapter 7.

- `unsaved-value` indicates the identifier value of the newly instantiated objects, which have not yet been persisted. Depending on the value you set, Hibernate determines whether an `UPDATE` or `INSERT` SQL statement is needed when the object is persisted. This attribute is mandatory. Commonly, zero (0) is used as the value when the identifier type is a number such as integer, long, and so on.

The `<generator>` element, nested within `<id>`, specifies how identifiers are generated for new instances of the class. This element takes a `class` attribute, that is, a Java class that generates unique identifiers for instances of the persistent class. Hibernate provides many built-in generators which satisfy most applications requirements. Still, when you need a particular kind of generator not provided by Hibernate (such as when you use a particular class instead of `string`, `integer`, `long`, and so on for an identifier), you can define a custom generator by implementing the `IdentifierGenerator` interface.

The following table shows Hibernate's built-in ID generators, which can easily be referred to by their short names:

Generator Short name	**Description**
`sequence`	This generator works with the sequence column type supported by DB2, PostgreSQL, Oracle, SAP DB, Mckoi, or a generator in Interbase. Both `sequence` and `generator` column types hint at the database to generate the ID values. The returned identifier is of type `long`, `short`, or `int`.
`increment`	This generator picks up the maximum primary key column value of the table at Hibernate startup, and then produces a series of identifier values by incrementing the preceding ones. This generator cannot be used in situations in which multiple processes access the database. The returned identifier value is of type `long`, `short`, or `int`.
`identity`	This generator works with identity columns in DB2, MySQL, MS SQL Server, Sybase, and HypersonicSQL. The returned identifier is of type `long`, `short`, or `int`.
`native`	This generator selects another identifier generator, such as `sequence`, `identity`, and `hilo`, depending upon the underlying database. The main advantage of this generator is that it keeps your application portable to many different databases.

Generator Short name	Description
`hilo`	This generator uses a **High/Low** algorithm to generate identifiers efficiently. According to this algorithm, each identifier value is made of two parts: a high value, which comes from a source common to all object identifier generators, and a low value, which is generated by your local object-identifier generator. The high values are more expensive, since they must be fetched from a central source available to all, but each of these values is unique to an object-identifier generator. The low value is initialized and incremented by the generator itself, locally, which makes this value easy and fast to obtain and to manage. The concatenation of the high and the low makes a unique key. To use this generator, you need a table and column (by default, `hibernate_unique_key` and `next_hi`, respectively) as the source of high values.
`seqhilo`	This generator uses the **High/Low** algorithm, in which the high values are generated by a named database sequence. Note that you can use this generator if the database sequence is supported.
`guid`	This generator uses a database-generated identifier that is unique in any context. For this reason, this generator is called **global unique identifier (guid)**. The guid generator only works with databases that support a guid type, including MS SQL Server and MySQL.
`uuid`	This generator uses a 128-bit **Universally Unique Identifier (UUID)** algorithm to generate identifiers of type string, unique within a network. The UUID algorithm uses the local IP address, in combination with the startup time of JVM, the current time, and a unique static counter in JVM, to generate the unique identifier. The generated identifier is encoded as a 32-digit hexadecimal string.
`assigned`	This does not refer to a built-in generator for identifiers. If this is used, the application must generate an identifier itself and assign it to the object before `save()` is called. This is the default strategy if no `<generator>` element is specified.
`select`	This generator performs a select query to read back the primary key value that has been assigned by a database trigger to the just-inserted row. This generator uses a key option, which refers to an additional, unique identifier key column, and allows Hibernate to select the just-inserted row.
`foreign`	This generator uses the identifier of another associated object. Usually, this generator is used when there is a one-to-one relation.

The <property> element

The `<property>` element maps a primitive property of the class, except for the identifier, to a particular column. The common attributes of this element come in the following form:

```
<property  name="propertyName"
           type="propertyType"
           lazy="true|false"
           optimistic-lock="true|false"
           generated="never|insert|always"
           column="columnName"
           not-null="true|false"
           unique="true|false"  />
```

Here are the meanings for these attributes and their possible values:

- `name` refers to the property name, which starts with a lower-case letter.
- `type` specifies the corresponding Hibernate type of the column.
- `lazy` indicates whether the property value should be loaded lazily when an instance is first accessed.
- `optimistic-lock` determines whether to use the optimistic lock when this property is updated.
- `generated` indicates whether the property value is assigned by the application, or generated by the database. The valid values are `never`, `insert`, and `always`. The default value `never`, specifies that the database never generates the value. The `insert` value means that the database generates the value when the object is inserted, but never regenerates it in subsequent updates. `always` indicates that the database always generates the value, whether an insert or update statement is performed.
- `column` specifies the table column in which the property is persisted. However, `column` may be omitted. When it is, the property name, given by the `name` property, is considered instead.
- `not-null` indicates whether the column may maintain `null` values.
- `unique` specifies whether to allow duplicate values for the column.

Except for `name`, all of these attributes are optional.

The formula attribute

The `formula` attribute can be used with the `<property>` element to represent a computed property. In this case, formula refers to an HQL expression (discussed in Chapter 9) by which the property is dynamically computed, instead of being loaded from a particular column. For example, consider the `count` property in the `Student` class, representing the number of all students persisted in the database:

```
public class Student {
  private int id;
  private int count;
  //other fields and getter&setter methods
}
```

The `formula` attribute can indicate how the count property should be initialized with the number of all student objects when each `Student` object is loaded:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch05.Student"
table="STUDENT">
    <id name="id" type="int" column="id">
      <generator class="increment"/>
    </id>
    <property name="count" formula="(SELECT COUNT(*) FROM Student)"/>
    <!--other properties -->
<hibernate-mapping>
```

The insert and update attributes

It is possible to use `insert` and `update` attributes for the `<property>` element, to specify if the mapped property should appear in the SQL `INSERT` or SQL `UPDATE` statements, when the object is inserted (saved for the first time) or updated. For instance, you may use `insert="false"` and `update="false"` for a property that is valued by a database trigger. Therefore, the application never assigns a value to it. You may also use `insert="true"` and `update="false"` for a property which is just inserted and never updated. The `insert="false"` and `update="false"` are useful for read-only properties that do not have corresponding columns in the database and are computed from other class properties.

Metadata in annotations

Let's now see how annotation mappings can be created for our simple class.

Before going ahead, make sure the `hibernate-annotations.jar` file, and `ejb3-persistence.jar`, in the application classpath. They are the required libraries to EJB-3 and Hibernate proprietary annotation API.

The following code shows the `Student` class which is now annotated:

```
package com.packtpub.springhibernate.ch05;

import java.util.Date;
import java.util.Calendar;
import java.io.Serializable;

import javax.persistence.*;

@Entity
@Table(name="STUDENT")
public class Student implements Serializable{
  private int id;
  private String firstName;
  private String lastName;
  private String ssn;
  private Date birthday;
  private String stdNo;
  private Date entranceDate ;

  //constructors

  @Id
  @Column(name = "ID")
  public int getId() {
    return id;
  }

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

  @Basic
  @Column(name="FIRST_NAME")
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @Basic
```

```
@Column(name="LAST_NAME")
public String getLastName() {
  return lastName;
}
public void setLastName(String lastName) {
  this.lastName = lastName;
}
@Basic
@Column(name="SSN")
public String getSsn() {
  return ssn;
}
public void setSsn(String ssn) {
  this.ssn = ssn;
}
@Basic
@Column(name="BIRTHDAY")
public Date getBirthday() {
  return birthdate;
}
public void setBirthdate(Date birthday) {
  this.birthdate = birthdate;
}
@Basic
@Column(name="STD_NO")
public String getStdNo() {
  return stdNo;
}
public void setStdNo(String stdNo) {
  this.stdNo = stdNo;
}
@Basic
@Column(name="ENTRANCE_DATE")
public Date getEntranceDate() {
  return entranceDate;
}
public void setEntranceDate(Date entranceDate) {
  this.entranceDate = entranceDate;
}
public int getAge(){
        Calendar c = Calendar.getInstance();
        c.setTime(new java.util.Date(getBirthdate().getTime()));
        int birthYear = c.get(Calendar.YEAR);
        c.setTime(new java.util.Date());
        int now = c.get(Calendar.YEAR);
        return now-birthYear;
      }

      //hashCode() and equals() methods
    }
```

As you can see, the class imports the `javax.persistence` package. This package includes the standard EJB-3 annotation API. This API allows us to develop a portable application.

Hibernate annotations may be used to annotate the class definition, the properties, and the methods. Let's explore what each annotation means and how it can be used.

@Entity

The `@Entity` annotation, which comes before the class definition, is used to mark a class as an entity class. This is done as follows:

```
import javax.persistence.*;
@Entity
public class Student implements Serializable{
    …
}
```

`@Entity` is located in `javax.persistence` and is used immediately before the class definition.

@Table

Similar to `@Entity`, `@Table` is set at the class level and provides information about the target table, in which the objects of the entity class are stored. This annotation is optional. Therefore, if it is omitted, Hibernate maps the entity class to a table with the same name as the entity class. However, you may use the `name` attribute of this annotation to change this default behavior. Here is an example:

```
import javax.persistence.*;
@Entity
@Table(name="STUDENT_TBL")
public class Student implements Serializable{
    …
}
```

The objects of this class will be persisted in a table named `STUDENT_TBL`. `@Table` has another important attribute, named `UniqueConstraint`. It will be discussed in the coming sections.

@Id and @GeneratedValue

The `@Id` annotation is used to mark a property as the class identifier. This annotation can be put either on the property or the corresponding getter method. This placement determines the default access strategy to the class properties, either field or property.

By default, the `@Id` annotation chooses the best key generation strategy. However, it is possible to override this default behavior through another annotation, named `@GeneratedValue` which has two attributes, `strategy` and `generator`.

The strategy attribute can have one of the four values specified by the `javax.persistence.GeneratorType` enumeration. The values are as follows:

- `AUTO` makes Hibernate choose the appropriate generator based on the database.
- `IDENTITY` makes the database be responsible for key generation.
- `SEQUENCE` makes Hibernate use a database sequence as the generator for primary key values.
- `TABLE` tells Hibernate to use a proprietary table in the database to hold the key values.

`AUTO` is the default value for `@GeneratedValue`, if this annotation is omitted.

When `SEQUENCE` or `TABLE` is used, you need to use the `@SequenceGenerator` or `@TableGenerator` to determine the details of the key generation strategy. Here is an example of `@SequenceGenerator`:

```
@Id
@SequenceGenerator(name="seqId",sequenceName=" SEQ")
@GeneratedValue(strategy= GeneratorType.SEQUENCE,generator="seqId")
public int getId() {
     return id;
}
```

As you can see, `@SequenceGenerator` comes with two attributes, `name` and `sequenceName`. The `name` attribute only provides an identifier to the sequence generator, which can be used by the `@GeneratodValue`. The `sequenceName` attribute determines the database sequence object.

Similarly, `@TableGenerator` is used to specify the details of the table sequence generator. Here is an example:

```
@Id
@TableGenerator(name="keys", table="KEYS_TBL")
@GeneratedValue(strategy= GeneratorType.TABLE,generator="keys")
public int getId() {
      return id;
}
```

The `name` attribute provides an identifier for the generator. The `table` attribute specifies the name of the table which holds the primary key values. `@TableGenerator` can be used with a variety of other attributes. Most importantly, the `pkColumnName` and `pkColumnValue` attributes are used when the table is used by more than one entity object, and each one has its own primary key column and primary key column value.

@Basic

All simple properties of an entity class are stored and retrieved by Hibernate, even if they are not marked up with annotations. These simple properties can be Java primitives, primitive wrappers, array of primitives, or any `Serializable` class.

@Lob

Any Java type that is mapped to a target column with the type of `java.sql.Blob` or `java.sql.Clob` is annotated with `@Lob`. These types include `java.sql.Clob`, `Character[]`, `char[]`, `String` for `Clob`, and `java.sql.Blob`, `byte[]`, or `Byte[]` for `Blob`. The following example shows this:

```
@Lob
public String getDescription() {
    return description;
}
@Lob
public byte[] getEncryptedCode() {
    return encryptedCode;
}
```

As you can see, both the `String` and `byte[]` types are annotated with `@Lob`.

@Transient

The `@Transient` annotation is used to mark a property as transient, so that the property will not be persisted.

@Column

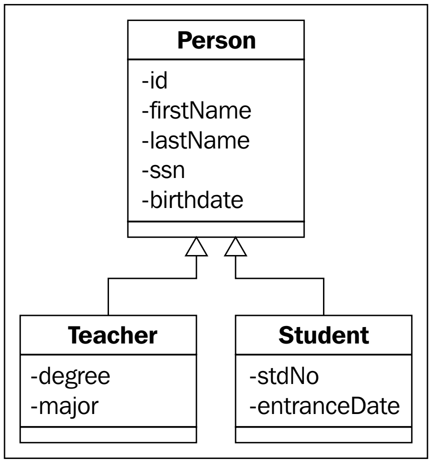
By default, Hibernate stores each persistent property in a column with the same name as the property. The `@Column` annotation can be used in cases where the column name is not the same as the property name. This annotation also allows information to be provided for the DDL.

Mapping inheritance hierarchy

An entity class may be derived from another entity class. Although the subclass has its own properties and associations, it also inherits the superclass properties and associations. As a result, to persist a subclass object, we need to persist its own properties and associations, as well as those it inherits. The persisting should occur in a way that allows reconstruction of the original object later.

Databases, however, do not naturally provide a solution for persisting inheritance hierarchies. Therefore, each application may use its own solution for this purpose.

Let's continue our discussion with a simple example. The following figure shows a class diagram of an inheritance hierarchy with the `Person`, `Student`, and `Teacher` classes. Through this example, we will see the different approaches Hibernate allows for mapping an inheritance hierarchy.



Hibernate has three distinct ways to map an inheritance relationship:

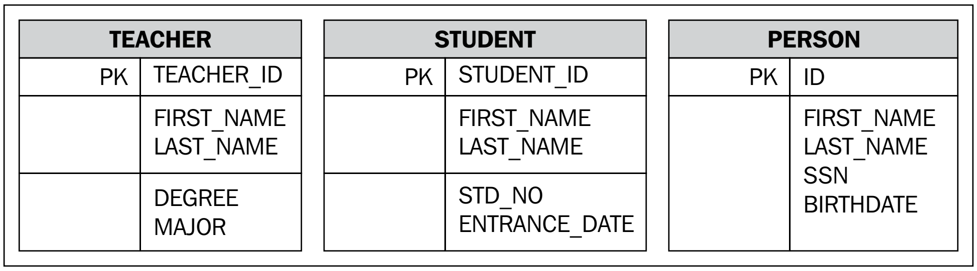
- Use one table for each concrete (nonabstract) class
- Use one table for each subclass, including interfaces and abstract classes
- Use one table for each class hierarchy

The sections that follows discuss these approaches, along with their advantages and disadvantages.

One table for each concrete class

In this approach, a single table is used for each concrete class. All persisted properties of each concrete class, including its own properties and its inherited properties, are mapped to a single table. Since there are no instances of interfaces or abstract classes, there is no need for mapping definitions.

To map our example class hierarchy using this approach, the database needs the tables shown in the following figure. Each table persists its respective entity class regardless of any inheritance considerations:



This approach has four shortcomings, which are explained below:

- **Imperfect support for polymorphic associations**: The problem occurs when a parent class is associated with another persistent class. In our example, suppose `Person` is associated with an `Address` class, so both `Student` and `Teacher` are associated with `Address`, as well. To map this class hierarchy according to this approach, we need these four tables in our database schema: `PERSON`, `STUDENT`, `TEACHER`, and `ADDRESS`. If `Address` has a many-to-one relationship with `Person` (more than one person may have the same address), then the `ADDRESS` table should provide a foreign key reference to all `PERSON`, `STUDENT`, and `TEACHER` tables, in order to establish database-level relationship, but this is not possible.
- **Low-performance capability**: Another shortcoming of this approach is that all objects persisted in subclass tables are naturally instances of a superclass. Therefore, to query all superclass objects, you need to query both the superclass and all subclass tables. This results in a negative impact on performance, especially when the class hierarchy is complex.

- **Verbose query statements***:* For the same reason as in the previous bullet, to query objects of a superclass, you need to write verbose query statements which include both subclass and superclass objects. This would be very difficult in practice when you are testing the database behavior through hand-written queries, or when the database is used by another application through pure JDBC.
- **Difficult to maintain***:* Another shortcoming of this approach is its maintenance cost. It produces a complex and dirty schema as several columns are duplicated across many tables. Therefore, any changes to the parent class may cause changes to a large number of tables, including its respective table and all of its subclass tables. Verbose query statements can also increase the maintenance cost.

The mapping of this strategy is very easy and straightforward. If you are using XML mappings, you just need to use an individual `hbm.xml` file for each concrete class regardless of their relationship.

To use annotations, you need to use the `@Inheritance` annotation with its `strategy` attribute. Here is an example:

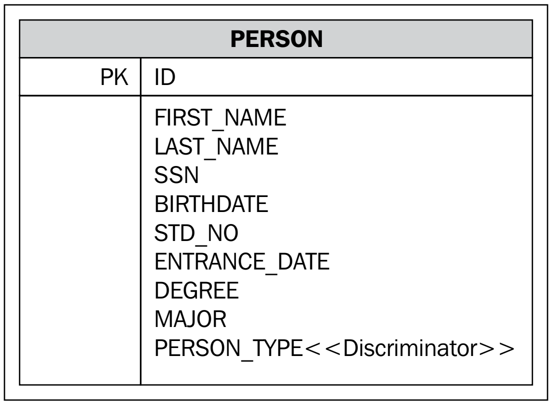
```
@Entity
@Inheritance(strategy = TABLE_PER_CLASS)
public class Student implements Serializable {
    ...
}
```

As you can see, the `strategy` attribute determines the type of inheritance mapping, which is `TABLE_PER_CLASS` for this case.

This approach is not recommended, except when the object model is not very complex and polymorphism is not required.

One table for class hierarchy

In this approach, the entire class hierarchy can be mapped to a single table. This table should have proper columns for all properties of all classes in the class hierarchy. The table uses an extra column, called the **discriminator column**, which allows recognition of the class to which each row belongs. The following figure shows the `PERSON` table, which maintains all of the objects of our hierarchy. In this, the `PERSON_TYPE` is the discriminator column:



The benefits of this approach are simplicity and efficiency. Polymorphic associations are implemented simply by foreign key constraints, and you can easily retrieve desired objects by simple queries without requiring any `JOIN` or `UNION` clause. However, this approach also has its disadvantages. Some columns are shared between derived classes, and each column must be defined nullable or not-nullable, the classes can use different nullable status for their shared columns.

To use this approach, first determine the `discriminator` column through the `<discriminator>` element, which defines the column name and type. Second, use the `<class>` and its nested `<subclass>` elements inside the mapping document for mapping the superclass and its derived classes. Third, each `<class>` or `<subclass>` element should use its desired discriminator value. Specify this value through the `discriminator-value` attribute.

The following code shows this approach for mapping the `Person`, `Student`, and `Teacher` classes:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Person" table="PERSON"
        discriminator-value="PE">
    <id name="id" column="ID" type="long">
      <generator class="native"/>
    </id>
```

```
        <discriminator column="PERSON_TYPE" type="string"/>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <property name="lastName" column="LAST_NAME" type="string"/>
        <property name="ssn" column="SSN" type="string"/>
        <property name="birthdate" column="BIRTHDATE" type="date"/>
        <subclass name="Student" discriminator-value="ST">
          <property name="stdNo" column="STD_NO" type="string"/>
          <property name="entranceDate" column="ENTRANCE_DATE"
                                                  type="date"/>
        </subclass>
        <subclass name="Teacher" discriminator-value="TE">
          <property name="degree" column="DEGREE"   type="int"/>
          <property name="major" column="MAJOR" type="int"/>
        </subclass>
      </class>
    </hibernate-mapping>
```

Note that the `discriminator` column is used merely for mapping. It has no corresponding values in the entity classes.

You can nest a `<subclass>` element within the `<subclass>` element to specify the mapping of a derived class from the subclass.

As for the table per class strategy, the `@Inheritance` annotation is used to mark up the entity class. But now `SINGLE_TABLE` is used as the value for the strategy attribute. It is also needed to use the `@DiscriminatorColumn` and `@DiscriminatorValue` annotations to specify the discriminator column, as well as the value in that column which is used for the entity class. The following code shows how annotations are used to map our example class hierarchy:

```
@Entity
@Inheritance(strategy = SINGLE_TABLE)
@DiscriminatorColumn(
    name="PERSON_TYPE",
    discriminatorType=STRING
)
@DiscriminatorValue("PE")

public class Person implements Serializable {
...
}
```

The `@DiscriminatorColumn` has two attributes, `name` and `discriminatorType`, which specify the name and the type of the discriminator column. For mapping the subclasses of `Person`, you just need to use `@Inheritance` and `@DiscriminatorValue` as follows:

```
@Entity
@Inheritance(strategy = SINGLE_TABLE)
@DiscriminatorValue("ST")
public class Student extends Person{
    ...
}
```

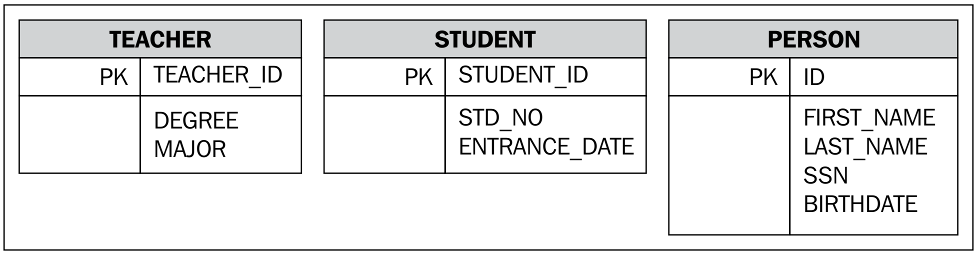
As you can see, `ST` now is used as the value of `@DiscriminatorValue`.

One table per subclass

You may use distinct tables for persisting each class in the hierarchy. The inheritance relationships between objects are established by foreign key constraints between tables.

This approach is analogous to the first approach, in which all of the properties of each class (its own and those that are inherited) are persisted into one table. However, unlike that approach, the one-table-per-subclass approach lets you use one table for each class' own properties. With this approach, you can imagine the database schema like the object model, a table corresponding to each class.

The following figure shows tables and their relations for mapping the `Person`, `Student`, and `Teacher` classes using this approach:



The `STUDENT_ID` and `TEACHER_ID` columns are foreign keys to the `ID` column of the `PERSON` table. While each object of the `Person` class is persisted with a row in the `PERSON` table, each `Student` or `Teacher` object is represented with a row in the `STUDENT` or `TEACHER` table, as well as a row in the `PERSON` table. This approach is simple to implement since each persistent class and its properties correspond to one table and its columns, respectively. The approach is also easy to manage, as any changes to a single persistent class makes a single change to the database schema. The data is fully normalized.

Use the `<joined-subclass>` elements in the mapping file to map the class hierarchy. The following code shows this approach in persisting our class hierarchy:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Person" table="PERSON">
    <id name="id" column="ID" type="long">
      <generator class="native"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <property name="lastName" column="LAST_NAME" type="string"/>
    <property name="ssn" column="SSN" type="string"/>
    <property name="birthdate" column="BIRTHDATE" type="date"/>
    <joined-subclass name="Student" table="STUDENT">
      <key column="STUDENT_ID"/>
      <property name="stdNo" column="STD_NO" type="string"/>
      <property name="entranceDate" column="ENTRANCE_DATE"
                                            type="date"/>
    </joined-subclass>
    <joined-subclass name="Teacher" table="TEACHER">
      <key column="TEACHER_ID"/>
      <property name="degree" column="DEGREE" type="int"/>
      <property name="major" column="MAJOR" type="int"/>
    </joined-subclass>
  </class>
</hibernate-mapping>
```

As you can see, each `<joined-subclass>` element has a nested `<key>` element, which represents the subclass table's primary key column that is a foreign key constraint to the parent table. In this example, `STUDENT_ID` and `TEACHER_ID` are the primary keys, and they are also foreign keys to the `PERSON` table.

To map this strategy with annotations, `strategy=JOINED` is used for the `@Inheritance` annotation in the superclass. Without any other annotations, Hibernate joins subclass tables with the superclass table using the same primary key names. The following code shows using the annotation mapping of the join strategy:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person implements Serializable {
     ...
}
@Entity
public class Student extends Person{
 ...
}
```

If tables are joined by columns with different names, the `@PrimaryKeyJoinColumn` and `@PrimaryKeyJoinColumns` annotations can be used. Here is an example:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person implements Serializable {
     ...
}
@Entity
@PrimaryKeyJoinColumn(name="STUDENT_ID")
public class Student extends Person{
 ...
}
```

In the example above, the `STUDENT` table is joined with `PERSON` using the `join` condition `PERSON.id = STUDENT.STUDENT_ID`.

The main shortcoming of this approach appears when the class hierarchy grows vertically: class `A` has a `B` subclass, and class `B` has a `C` subclass, and so on. When a child class is queried, Hibernate must join the child table and all of its parent tables to construct the target objects. This has a negative effect on performance. Furthermore, tables in the database are not self-explained. In other words, the bounds of the data in those tables are not clear for everyone who uses the data directly without Hibernate.

> The `dynamic-update` and `dynamic-insert` attributes, which are used with the class element to avoid inserting `null` property values and to avoid updating unchanged property values, respectively, are not inherited by subclasses. For that reason, they should be duplicated in the `<subclass>` or `<joined-subclass>` elements.

Implicit polymorphism versus explicit polymorphism

Hibernate allows polymorphic relations to be handled by providing the `polymorphism` attribute through the `<class>`, `<subclass>`, and `<joined-subclass>` elements. With this attribute, you can determine whether to use `implicit` or `explicit` query polymorphism.

Implicit polymorphism allows querying of all instances of a class and its subclasses by a query which only names the class itself. **Explicit polymorphism** allows querying only of instances of the class and its subclasses mapped inside this `<class>` declaration as a `<subclass>` or a `<joined-subclass>`. For most purposes, the default, `polymorphism="implicit"`, is appropriate. Explicit polymorphism is useful when the class hierarchy is mapped to the same table. In that case, we are not interested in all instances when querying the superclass.

Summary

In this chapter, you learned the principles of entity classes and their mapping definitions. Hibernate defines only a few requirements for persistent classes. The structure of all persistent classes must follow a POJO programming model, a customized and simplified form of the JavaBeans model, with a bit of additional requirements. The transparent persistence feature of Hibernate indicates that these classes do not depend on the Hibernate API or any persistence logic. This increases the application's portability and keeps it consistent with any persistence technology.

After designing the object model, you need to give Hibernate information about the target tables and their columns, in which the objects properties are persisted. You accomplish this through the mapping documents. Every mapping document is written in XML, starting with a mandatory `DOCTYPE` declaration, referring to the DTD document, and followed by `<hibernate-mapping>` as the root element. Inside the root, the `<class>` element is used to map the entity class. This element comes with `name` and `table` attributes, indicating the entity class name and the respective target table.

Inside `<class>`, the `<id>` element comes first. This element associates the identifier property of the class with the primary key column of the table. It also specifies how to generate an identifier value for new instances of the class. After `<id>`, `<property>` elements appear to map primitive properties of the class.

Hibernate allows inheritance hierarchies to be mapped in three distinct ways: using one table for each concrete class, using one table for each subclass, and using one table for each class hierarchy.

6
More on Mappings

In Chapter 5, you learned the fundamental concepts behind Hibernate mappings. In this chapter, we will look at some other issues in mapping definitions, including the mapping of collections and associations. We will see how persistent classes can be divided into value types and entity types, and how these may affect the mapping definition. As you probably know, there are different types of object associations: one-to-one, one-to-many, many-to-one, and many-to-many. The many side of the relationship is represented by an object of a `java.util.Collection` or a `java.util.Map`.

The mapping definitions introduced in this chapter are explained with simple examples from the educational system application. Let's begin with the simplest form of class mapping, called component mapping.

> Most examples in this chapter are not listed in complete form. Setter and getter methods are always omitted, and each persistent class includes only the properties relevant to the current discussion.

Mapping components

Commonly, each persistent class is stored in its own database table. This means we normally use a database table which corresponds to each persistent class. Although this is a common approach, it is not always taken. In practice, there is a situation where you may want to store more than one persistent class in a single database table.

This situation is where you have two persistent classes, one of them is always represented as a property of the other one and is used nowhere else. In other words, no object of one class that is represented as the property of the other class lives independently in the application. For instance, suppose our application includes two persistent classes: `Student` and `Phone`. If any `Phone` instance is always a property of a `Student` instance and not of any other class, we can say no `Phone` object can live independently of a `Student` object in the application. In this case, `Phone` is called a *component* of `Student`, or simply a *value object*.

Why then don't we merge the two classes into a single persistent class? The answer is simple: these classes model our domain more comprehensibly. You may then ask why we don't treat them normally and persist them in individual tables. This is because the two classes together form a single item of data, and it is neither attractive nor reasonable to present a single piece of data in two distinct tables. We use one identifier for both persistent classes for exactly the same reason. In our example, the `Phone` class does not have an identifier. Instead, it can use its associated `Student` identifier.

Hibernate provides the `<component>` element for such situations. This element is analogous to the `<property>` element, but unlike `<property>` which maps a primitive type, `<component>` maps an associated persistent object to extra columns in a table. This element is commonly used in the following form:

```
<component name="propertyName"
           class="className"
           lazy="true|false"
           unique="true|false">
```

The meanings of these attributes and their possible values are as follows:

- `name` (mandatory) refers to the name of the object field persisted.
- `class` (optional) specifies the classname of the mapped field. If `class` is not specified, Hibernate uses the reflection API to find the field's class. Use the qualified name for duplicate classes.
- `lazy` (optional) determines whether lazy loading is used for the field.
- `unique` (optional) indicates whether all columns to which the associated object is mapped are unique.
- The `<component>` element can come with other attributes, such as `insert`, `update`, and `access`. These have the same meanings when they are used for the `<class>` element.

In this chapter, just like in the previous chapter, underlined words show the default values.

The following code shows an example of using the `<component>` element, in which the `Student` class and its composite field, `Phone`, are mapped to a single table:

```
public class Student {
  private Phone phone;
  //other fields and getter/setter methods
}
public class Phone {
  private String comment;
  private String number;
  //other fields and getter/setter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <component name="phone"
class="com.packtpub.springhibernate.ch06.Phone">
      <property name="comment" column="COMMENT"/>
      <property name="number" column="PHONE_NUMBER"/>
    </component>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

Persistent classes are called **value types** if they do not live independently in the application and are always represented without an identifier value. Other normal persistent classes are called **entity types**, which live independently and are always identified with their identifier value. `Phone` and `Student` are examples of value type and entity type, respectively.

The following simple rules identify value types:

- They do not hold an identifier value.
- They do not live independently in the application.
- They are never shared between two or more persistent classes.

For example, if Phone is a property for both Student and another persistent class (for example, Teacher), then you should change Phone to an entity type by assigning an identifier property to it and storing its instances in a distinct database table. Note that sharing a persistent class among other persistent classes is different from sharing an instance among other instances.

The @Embedded annotation is used to annotate a property to map as a component. Therefore, in our case, we would have the Student class as follows:

```
@Entity
@Table(name = "STUDENT")
public class Student {
  @Embedded
  private Phone phone;
  //other fields and getter/setter methods
}
```

It is also possible to annotate the dependent class as a component with the @Embeddable annotation. Therefore, we don't need the @Embedded annotation anymore, and Hibernate always maps the object of the dependent class as a component. The following shows the Phone class annotated with @Embeddable:

```
@Embeddable
 public class Phone {

  private String comment;

  @Column(name="PHONE_NUMBER")
  private String number;
  //other fields and getter/setter methods
}
```

In the class above, we just marked the Phone class as a dependent component class. Therefore, all of Phone's properties will store to additional columns of the STUDENT table.

Mapping collections

When a class has a property of type java.util.Collection, or any of its subclasses, there is certainly a one-to-many or many-to-many relationship between the class and the collection's elements. The collection may contain objects of either value type or entity type. Regardless of the object type the collection maintains, we always need an extra table to store the collection elements. Obviously, this table must include a primary key column if the collection maintains entity types.

Java provides different collection types, all represented as subinterfaces or implementations of `java.util.Collection`. Additionally, Java provides maps, represented by implementations and subinterfaces of the `java.util.Map` interface. The `Map` interface does not extend `java.util.Collection`, so it is not called a collection. However, since both maps and collections represent nearly the same concepts, with methods of the same names and nearly the same behavior, maps can be viewed as collections. Collections and maps provide a hierarchy of interfaces with different implementation classes.

The `java.util.Collection` has two subinterfaces: `java.util.List` and `java.util.Set`. The `List` allows duplicate elements and maintains elements based on their positions in the list. In contrast, `Set` does not allow duplicate elements and does not preserve elements order. `Map` holds the objects as key/value pairs.

The following table shows the collection and map interfaces and the corresponding XML elements used for mapping them. This table also shows which collection and map is initialized with which concrete class:

Java Collection Type	**Description**	**Hibernate Mapping Definition**
`java.util.Collection`	(Initialized with `java.util.ArrayList`) The root interface for `java.util.List` and `java.util.Set` collections. It provides methods such as `add(),remove(),size()`, and `toArray()`.	Mapped with `<bag>` and `<idbag>` elements.
`java.util.List`	(Initialized with `java.util.ArrayList`) A collection that maintains its elements in a particular order unless it is modified. Each element can be accessed through its index in the list.	Mapped with the `<list>` element. This element uses an extra column in the target table to preserve the position of each element in the list. Any `java.util.List` object can also be mapped with `<bag>` and `<idbag>` if no real positional order is needed.
`java.util.Set`	(Initialized with `java.util.HashSet`) A collection that extends the `java.util.Collection` interface and does not allow duplicate elements. Its elements are not necessarily stored in any particular order.	Mapped with the `<set>` element.

Java Collection Type	**Description**	**Hibernate Mapping Definition**
`java.util.SortedSet`	(Initialized with `java.util.TreeSet`) A subinterface of `java.util.Set`, whose elements are sorted.	Mapped with the `<set>` element. This element can be used with an additional `sort` attribute, which dictates that the elements must be sorted based on property or object.
`java.util.Map`	(Initialized with `java.util.HashMap`) Holds its elements as key/value pairs. The keys are always unique, but values can be duplicate.	Mapped with the `<map>` element.
`java.util.SortedMap`	(Initialized with `java.util.TreeMap`) Extends `java.util.Map` and maintains the keys sorted.	Mapped with the `<map>` element. This element can be used with an additional `sort` attribute, which dictates that the elements must be sorted based on property or object.

Let's see how the `<set>`, `<bag>`, `<list>`, `<idbag>`, and `<map>` elements allow us to map collections and maps.

The <set> element

The `<set>` element maps an object of type `java.util.Set` which, unlike `java.util.List`, does not permit duplicates. This is the common way to use this element:

```
<set name="setName"
     table="SET_TABLE"
     inverse="true|false"
     lazy="true|false"
     order-by="aTableColumn"
     cascade="none|all|delete|persist|merge|
              save-update|evict|replicate|lock|refresh"
     sort="unsorted|natural|ComparatorClass"/>
```

The meanings of these attributes and their possible values are as follows:

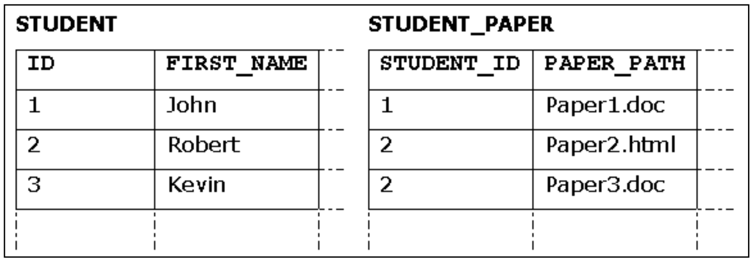
- `name` (mandatory) refers to the property name that represents an object of type `java.util.Set`.
- `table` (optional) specifies the table that stores the associated entities represented by the `Set` object. If `table` is not specified, the property name is used instead.

- `inverse` (optional) specifies whether the relationship can be navigated in the opposite direction. `inverse="true"` tells Hibernate the developer is responsible to manage the collection and Hibernate should not synchronize the collection with the database when the link between two instances is manipulated.
- `lazy` (optional) determines lazy loading mode in fetching the objects in the set.
- `order-by` specifies an arbitrary SQL `ORDER BY` clause with an optional `asc` or `desc`, which affects the populated `Set` object. Note that the string you use for this attribute does not include the `ORDER BY` keywords. (This is discussed separately in the next section of this chapter.)
- `cascade` (optional) determines how any persistent operation on the object affects its associated objects in the set.
- `sort` determines whether a sorted set is to be used. The valid values include `unsorted`, `natural`, or any `Comparator` class. (This is discussed separately in the next section of this chapter.)

Assume that the `Student` class has a property of type `java.util.Set`, called `papers`. This property represents all of the papers that a `Student` has written. The `papers` property maintains the file names of a student's papers in the system, but no extra information. To map this structure to the database, we use an extra table called `STUDENT_PAPER`. This table has two columns, `STUDENT_ID` and `PAPER_PATH`, which determine the identifier of the owner student and the path of the paper in the system, respectively. The following shows the `Student` class and its mapping file:

```
public class Student {
  private int id;
  private String firstName;
  private Set papers = new HashSet();
  //other fields and setter/gettter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <set name="papers" table="STUDENT_PAPER">
      <key column="STUDENT_ID"/>
      <element type="string" column="PAPER_PATH"/>
    </set>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, we have specified the `STUDENT_PAPER` table to store the elements of the `papers` property through the `<set>` element. The nested `<key>` element specifies the column in `STUDENT_PAPER` that is a foreign key to the primary key value of the associated `Student` object (in this case, `STUDENT_ID` is the foreign key to the `ID` primary key). The nested `<element>` element specifies the type and destination of each element in the `Set`. In our case, this is a string that represents the file path. The following figure shows the database table structure for this type of mapping:



STUDENT

ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_PAPER

STUDENT_ID	PAPER_PATH
1	Paper1.doc
2	Paper2.html
2	Paper3.doc

The <bag> element

The `<bag>` element is used to map an object of `java.util.Collection` and its subinterface, `java.util.List`. However, the `<bag>` element is not exactly a match for the `List` interface. While `List` keeps the items in order, the order of items in `<bag>` is ignored and is not kept in the database. The `<bag>` element offers the `order-by` attribute, which specifies an arbitrary SQL `ORDER BY` clause for the ordering of populated objects. The main shortcoming in `<bag>` is the lack of objects to be used as keys for the elements in the `<bag>`, which decreases performance when updating or deleting elements. When an element of the bag changes, Hibernate must update all of the elements since there is no way for Hibernate to find out which element has changed. Except for the `sort` attribute, all of the `<set>` element's attributes are supported by `<bag>` and have the same meanings.

To see how to use the `<bag>` element, suppose the `papers` property of the `Student` class, which is already presented as an instance of `java.util.Set`, now has changed to a property of type `java.util.Collection`. Although it is not realistic for the user to have written duplicate papers, we suppose that is the case in order to demonstrate how to use the `<bag>` element. Here is the `Student` class and its mapping definition with the `<bag>` element:

```
public class Student  {
  private int id;
  private String firstName;
  private Collection papers = new ArrayList();
  //other fields and setter/gettter methods
}
```

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <bag name="papers" table="STUDENT_PAPER">
      <key column="STUDENT_ID"/>
      <element type="string" column="PAPER_PATH"/>
    </bag>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As with `<set>`, we used `STUDENT_PAPER` to maintain the paper file paths. This table includes the `STUDENT_ID` column, which is a foreign key to the primary key of the `STUDENT` table. The paper file paths are stored in the `PAPER_PATH` column. The database structure for the `<bag>` element is exactly like `<set>`. The only difference is that with `<bag>` you are allowed to store duplicate elements.

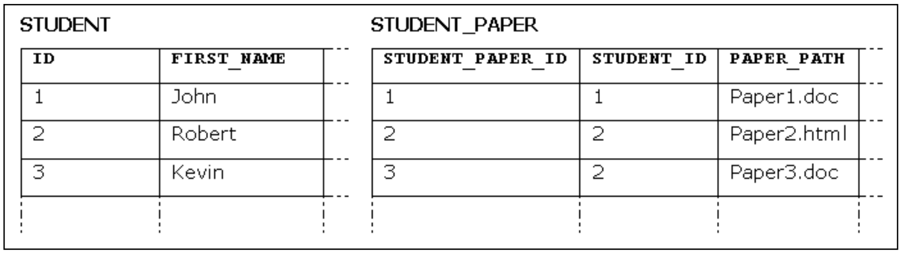
The <idbag> element

Like `<bag>`, the `<idbag>` element can be used to map an object of type `java.util.List`, but not necessarily `java.util.Collection`. However, you can use the `<idbag>` element for a `Collection` property if the property is initialized with an implementation of type `java.util.List`. Neither `<bag>` nor `<idbag>` cares about the order of its elements. However, `<idbag>` uses an additional key table column, which improves the performance of updating and deleting the collection's elements. Using this column, Hibernate can determine which element has changed or been removed. The following code shows the `Student` class and its mapping definition, which now uses the `<idbag>` element:

```
public class Student {
  private int id;
  private String firstName;
  private Collection papers = new ArrayList();
  //other fields and setter/gettter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
```

```
        </id>
        <property name="firstName" column="FIRST_NAME" type="string"/>
        <idbag name="papers" table="STUDENT_PAPER">
            <collection-id type="long" column="STUDENT_PAPER_ID">
                <generator class="sequence"/>
            </collection-id>
            <key column="STUDENT_ID"/>
            <element type="string" column="PAPER_PATH"/>
        </idbag>
        <!-- mapping of other fields -->
    </class>
</hibernate-mapping>
```

As you can see, we used a nested `<collection-id>` element inside `<idbag>`. The `<collection-id>` element specifies the collection table's primary key column, which holds the identifier value for each collection's element. The `<generator>` element determines the Hibernate strategy for generating the value for this identifier. In this case, we used sequence. Other elements are like those used for the `<bag>` element. The following figure shows the database table structure for this type of mapping:



STUDENT

ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_PAPER

STUDENT_PAPER_ID	STUDENT_ID	PAPER_PATH
1	1	Paper1.doc
2	2	Paper2.html
3	2	Paper3.doc

The `STUDENT_PAPER_ID` column never affects the Java code. Hibernate uses this column internally to manage the collection.

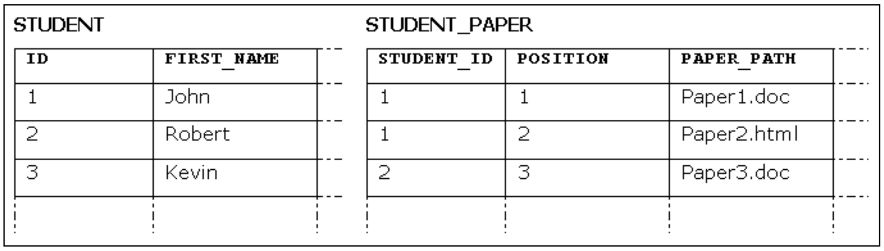
The <list> element

The `<list>` element, like `<bag>` and `<idbag>`, can map an object of type `java.util.List`. However, unlike those elements, `<list>` uses an extra column to maintain the order of the elements in the `List`. The nested `<list-index>` element is used for this purpose.

Using the `<list>` element is similar to `<set>` and `<bag>`, but without the `sort` and `order-by` attributes. The following code shows the `Student` class where its `papers` property has been defined as an instance of `java.util.List`:

```
public class Student {
  private int id;
  private String firstName;
  private List papers = new ArrayList();
  //other fields and setter/gettter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
   table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <list name="papers" table="STUDENT_PAPER">
      <key column="STUDENT_ID"/>
      <list-index column="POSITION"/>
      <element type="string" column="PAPER_PATH"/>
    </list>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, we have used a nested `<list-index>` element to specify the column which holds the order of each element in the `List`. Other elements, `<key>` and `<element>`, have the same meanings as the `<set>`, `<bag>`, and `<idbag>` elements. The following figure shows the database table structure for this type of mapping:



STUDENT

ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_PAPER

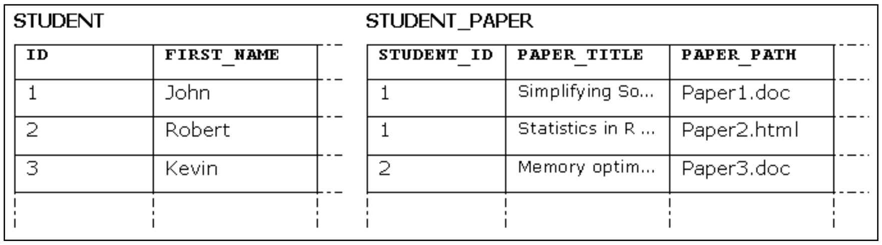
STUDENT_ID	POSITION	PAPER_PATH
1	1	Paper1.doc
1	2	Paper2.html
2	3	Paper3.doc

The <map> element

The `<map>` element maps an object of type `java.util.Map`. The `Map` object holds elements as key/value pairs. Using `<map>` is similar to using `<set>`. In the following code, our example has changed to show the mapping of a `Map` property:

```
public class Student {
  private int id;
  private String firstName;
  private Map papers = new HashMap();
  //other fields and setter/gettter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
   table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <map name="papers" table="STUDENT_PAPER">
      <key column="STUDENT_ID"/>
      <map-key column="PAPER_TITLE" type="string"/>
      <element type="string" column="PAPER_PATH"/>
    </map>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, in the mapping of a `Map`, we have used a `<map-key>` nested element inside the `<map>`. The `<map-key>` specifies the column name in the `STUDENT_PAPER` table, which holds the map's keys. We have named this column `PAPER_TITLE`, and it maintains paper titles as the map's keys. This approach is realistic since no student may write two or more papers with the same title. The following figure shows the database table structure for the mapping of a `Map`:



STUDENT

ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_PAPER

STUDENT_ID	PAPER_TITLE	PAPER_PATH
1	Simplifying So...	Paper1.doc
1	Statistics in R ...	Paper2.html
2	Memory optim...	Paper3.doc

Mapping collections with annotations

Hibernate provides a proprietary `org.hibernate.annotations.CollectionOfElements` annotation to map collections which contain value-typed elements. Let's get back to our first example, the `Student` class with a property of `java.util.Set`. We should use the `CollectionOfElements` annotation as follows to map `Set`:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "STUDENT_PAPER",
    joinColumns = @JoinColumn(name = "STUDENT_ID")
)
@Column(name = "PAPER_PATH")
private Set papers = new TreeSet();
```

The `CollectionOfElements` annotation has an optional `targetElement` property, which determines the type of objects held by the collection. If generics are used with the collection, then there is no need to use this property.

`CollectionOfElements` can be used with the `IndexColumn` annotation to map a `java.util.List`:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "STUDENT_PAPER",
    joinColumns = @JoinColumn(name = "STUDENT_ID")
)
@org.hibernate.annotations.IndexColumn(name="POSITION")
@Column(name = "PAPER_PATH")
private List papers = new ArrayList();
```

If the index of elements doesn't matter, you may omit the `IndexColumn` annotation, and then the list will be mapped as a bag collection.

You may also use `CollectionOfElements` with the `MapKey` annotation to map a `java.util.Map`. Here is an example:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "STUDENT_PAPER",
    joinColumns = @JoinColumn(name = "STUDENT_ID")
)
@org.hibernate.annotations.MapKey(
    columns = @Column(name="PAPER_TITLE")
)
@Column(name = "PAPER_PATH")
private Map papers = new HashMap();
```

Notice how other parts of this mapping have never changed.

Sorted sets and sorted maps

The `<set>` and `<map>` elements can come with an additional attribute called `sort`. Other collection elements, such as `<bag>`, `<idbag>`, and `<list>`, do not provide this attribute. The `sort` attribute allows us to specify whether Hibernate should sort the collection elements retrieved from the database. If so, it determines how they should be sorted. The possible values for this attribute are as follows:

- `unsorted` does not impose any sort for elements.
- `natural` indicates Hibernate should use a sorted `Set` or `Map` implementation, `SortedSet` and `SortedMap`, to sort the set or map elements. The elements inside a `SortedSet` instance are ordered according to the elements of the `compareTo()` method. The elements of a `SortedMap` instance are ordered according to the `compareTo()` method of the `SortedMap` keys. Note that the elements of a `SortedSet`, and the keys of a `SortedMap`, must implement the `java.lang.Comparable` interface, which causes them to have the `compareTo()` method.
- **The name of a comparator class**: you may use an implementation of the `java.util.Comparator` class to define a different strategy for sorting.

The following examples show how you can specify a sorting strategy.

This example indicates that no sort operation is performed on the map elements:

```
<map name="papers" table="STUDENT_PAPER" sort="unsorted">
  <key column="STUDENT_ID"/>
  <map-key column="PAPER_TITLE" type="string"/>
  <element type="string" column="PAPER_PATH"/>
</map>
```

This example indicates that the map elements must be sorted based on the map key, PAPER_TITLE. Since PAPER_TITLE entries are strings, use the String.compareTo() method to sort them:

```
<map name="papers" table="STUDENT_PAPER" sort="natural">
    <key column="STUDENT_ID"/>
    <map-key column="PAPER_TITLE" type="string"/>
    <element type="string" column="PAPER_PATH"/>
</map>
```

Finally, we may implement the java.util.Comparator interface to define a new strategy for sorting the Map or Set elements. The following code shows a comparator class which defines a different strategy for sorting PAPER_TITLE. This class changes the normal sort of strings, in which letters always appear before numeric characters:

```
package com.packtpub.springhibernate.ch06;
import java.util.Comparator;
public class CustomStringComparator implements Comparator<String> {
  public int compare(String s1, String s2){
    if(s1.length()==0){
      return -1;
    }
    if(s2.length()==0){
      return 1;
    }
    char ch1 = s1.charAt(0);
    char ch2 = s2.charAt(0);
    if(ch1 == ch2){
      s1 = s1.substring(1, s1.length());
      s2 = s2.substring(1, s2.12     length());
      return this.compare(s1, s2);
    }else {
      return this.compare(ch1, ch2);
    }
  }
  public int compare(char ch1, char ch2){
    if(Character.isDigit(ch1)&& Character.isLetter(ch2)){
      return 1;
    } else if(Character.isLetter(ch1)&& Character.isDigit(ch2)) {
      return -1;
    }else {
      return ch1 - ch2;
    }
  }
}
```

To use this comparator, we only need to specify the fully qualified name of the `CustomStringComparator` class as the value for the `sort` attribute as follows:

```
<map name="papers" table="STUDENT_PAPER"
     sort="com.packtpub.springhibernate.ch06.CustomStringComparator">
  <key column="STUDENT_ID"/>
  <map-key column="PAPER_TITLE" type="string"/>
  <element type="string" column="PAPER_PATH"/>
</map>
```

`<bag>`, `<idbag>`, and `<list>` cannot be sorted. The list index specifies the elements order in the list.

A collection can be sorted or ordered with the `Sort` annotation, provided by the Hibernate annotation API:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "STUDENT_PAPER",
    joinColumns = @JoinColumn(name = "STUDENT_ID")
)
@Column(name = "PAPER_PATH")
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private Set papers = new TreeSet();
```

The `type` attribute can have one of the `SortType`'s `NATURAL`, `UNSORTED`, and `COMPARATOR` values. All of the values have the same meaning as they are already used with the `<set>` and `<map>` elements. As before, if `COMPARATOR` is used, you need to introduce an implementation of the `java.util.Comparator` interface, as the customized strategy for sorting the `Map` or `Set` elements is through the `comparator` attribute. An example which uses `CustomStringComparator`, recently implemented, is as follows:

```
@org.hibernate.annotations.Sort(
 type = org.hibernate.annotations.SortType.COMPARATOR,
 comparator=com.packtpub.springhibernate.ch06.CustomStringComparator
)
```

Other annotations remain unchanged.

Using the order-by attribute to order collection elements

The `order-by` attribute can be used with `<bag>`, `<idbag>`, `<list>`, `<set>`, and `<map>` to order the collection elements through an `ORDER BY` clause when the elements are retrieved from the database.

For instance, we may use the `order-by` attribute, instead of `sort`, with the `<map>` element to order elements based on the map's keys:

```
<map name="papers" table="STUDENT_PAPER" order-by="PAPER_TITLE asc">
  <key column="STUDENT_ID"/>
  <map-key column="PAPER_TITLE" type="string"/>
  <element type="string" column="PAPER_PATH" not-null="true"/>
</map>
```

Note that you can use any column of the collection table to order the collection's elements.

The Hibernate annotation API provides the `OrderBy` annotation to determine, how a collection should be ordered on load by the database when the objects are fetched. The `OrderBy` annotation uses the clause attribute to determine the SQL order by clause in fetching the objects. The equivalent annotation mapping for the XML mapping definition above is as follows:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "STUDENT_PAPER",
    joinColumns = @JoinColumn(name = "STUDENT_ID")
)
@Column(name = "PAPER_PATH")
@org.hibernate.annotations.OrderBy(
    clause = "PAPER_TITLE asc"
)
private Set papers = new TreeSet();
```

The value of the `clause` attribute is appended to the SQL fragment generated by Hibernate to pass to the database.

Mapping object associations

So far in this chapter, you've learned about mapping components. A component is a property of a persistent class represented as another persistent class, which is stored with its owner class in one table. Instead of persisting associated classes in one table, each class may be persisted in its own table. This approach is the subject of this next section. We'll begin by looking at different types of object associations and how they are represented in the Java object model.

Suppose there are two persistent classes called `A` and `B`. The relationship of `A` and `B` is called **one-to-one** if any instance of `A` is associated with only a single instance of `B`, and no more. A one-to-one association is always presented as an instance of `B` defined as a property of `A`, or vice versa.

The association is called **one-to-many** from `A` to `B` if any instance of `A` can be associated with more than one instance of `B`. This relationship is established by a property of a collection type of `B` instances in class `A`. The one-to-many relationship from `A` to `B` is called **many-to-one** when going from `B` to `A`. In other words, when more than one instance of `B` can be associated with one instance of `A`.

The final type of relationship, which is rare, is **many-to-many:** more than one instance of `A` can be associated with more than one instance of `B`. In this relationship, each instance of `A` holds a collection of `B` instances, and vice versa.

Let's see how these object relationships can be persisted with Hibernate.

The <one-to-one> element

Two persistent classes may be associated with each other in a one-to-one relationship. A relationship is called one-to-one when each instance of a class is associated with a single instance of another class, and vice versa. If, when you have an instance of one class, and the other instance can be reached, then the one-to-one relationship is called **bidirectional**. On the other hand, if the objects cannot be reached from both sides, the relationship is **unidirectional**.

For example, consider the `Phone` and `Student` classes in the previous section. If each student has a unique phone number, and no phone number is shared between two or more students, the relationship is one-to-one. This is because each `Student` object is associated with only one `Phone` object, and each `Phone` object is owned by only one `Student` object. At the database level, a one-to-one relationship is represented using either the same primary keys or unique foreign keys.

When an instance of one class is stored, we expect its associated object to be stored as well. This scenario can be true for updating and removing. Note that we can configure the cascade operation for each class and disable or enable this behavior. Hibernate provides the `<one-to-one>` element to map a one-to-one relationship between two persistent classes. We will discuss each strategy in upcoming sections. Here is the common form of the `<one-to-one>` element:

```
<one-to-one name="propertyName"
            class="className" cascade="none|all|delete|persist|merge|
                    save-update|evict|replicate|lock|refresh"
            property-ref="propertyNameFromAssociatedClass"
                            constraint="true|false">
```

The meanings of these attributes and their possible values are as follows:

- `name` (mandatory) refers to the name of the associated object.
- `class` (optional) specifies the class name of the associated object. If `class` is not specified, Hibernate maps this property using the reflection API, to discover the associated object's class and find the mapping metadata for that class. This attribute is useful when the object's class and superclass(es) are mapped differently, and we want to map this property as one of its superclasses, instead of as its own class.
- `cascade` (optional) determines how changes to the parent object propagate to the associated object when the parent is created, updated, or removed.
- `property-ref` (optional) specifies a property of the associated object. The value of this property establishes the relationship between two objects.
- `constraint` specifies that a foreign key constraint links the primary key of the associated table to the primary key of the owner table. This guarantees that an associated row's primary key references a valid owner primary key.

The `<one-to-one>` element can come with other attributes, such as `access` and `lazy`, with the same meanings as with `<class>`.

Using identical primary keys

The first strategy for mapping a one-to-one relationship is to use identical primary key values for associated objects. This means each row of one table is associated with a row in another table through the same identifier value.

The following code shows `Student` and `Phone` classes with a one-to-one relationship. As you can see, this relationship is represented by a property of type `Phone` in the `Student` class. This relationship is unidirectional because the `Phone` class does not maintain any reference for `Student`:

```
public class Student {
  private int id;
  private Phone phone;
  //other fields and getter/setter methods
}

public class Phone {
  private int id;  private String comment;
  private String number;
  //other fields and getter/setter methods
}
```

You can use a `<one-to-one>` element in the mapping definition to map this relationship in `Student.hbm.xml`, as shown in the following code:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
   table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <one-to-one name="phone"
                class="com.packtpub.springhibernate.ch06.Phone"
                cascade="all"
                lazy="false"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

And here is `Phone.hbm.xml`:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Phone" table="PHONE">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="number" column="PHONE_NUMBER" type="string"/>
    <property name="comment" column="COMMENT" type="string"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

The relationship is unidirectional, each `Phone` object can be reached from its respective `Student`, but not the reverse. To change the relationship to bidirectional, first, `Phone` must maintain a reference to its respective `Student`, and second, the `Phone` mapping must include a one-to-one relationship. The following code illustrates the changes in the `Phone` class:

```
public class Phone {
  private int id;  private String comment;
  private String number;
  private Student student;
  //other fields and getter/setter methods
}
```

`Student.hbm.xml` remains unchanged, but `Phone.hbm.xml` changes as follows:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Phone" table="PHONE">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="number" column="PHONE_NUMBER" type="string"/>
    <property name="comment" column="COMMENT" type="string"/>
    <one-to-one name="student" property-ref="phone"
                class="com.packtpub.springhibernate.06.Student"
                cascade="all"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

JPA provides the `OneToOne` annotation for mapping a one-to-one relationship. To map a one-to-one relationship using the identical primary keys strategy, we also need the `PrimaryKeyJoinColumn` annotation. The following code shows how our recent example is mapped through JPA annotations:

```
@Entity
public class Student {
  @Id
  private int id;
  @OneToOne
  @PrimaryKeyJoinColumn
  private Phone phone;
  //other fields and getter/setter methods
}
```

If you want to make this relation bidirectional, you need to provide a property of type `Student` in the `Phone` class, and use the `OneToOne` annotation with the `mappedBy`, as follows:

```
public class Phone {
  @OneToOne(mappedBy="student")
  private Student student;
}
```

As you can see, the value of the `mappedBy` attribute is the property name, which holds a reference of `Student` in the `Phone` class.

Foreign key one-to-one

This strategy uses a foreign key column to establish the one-to-one relationship between two tables. One table has an extra column as a foreign key to the primary key of the other table. To map this relationship, use a `<many-to-one>` element instead of `<one-to-one>`, because when we say a table has a foreign key to another table, many rows in the source table can naturally refer to one single row in the target table. However, if the foreign key column is defined as unique, this strategy guarantees that only a single row in the source table can be associated with a row in the target table.

`Student.hbm.xml` should change as follows:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
   table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <many-to-one name="phone"
                 class="com.packtpub.springhibernate.ch06.Phone"
                 column="PHONE_ID" unique="true"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

And `Phone.hbm.xml` will change as follows:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Phone" table="PHONE">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="number" column="PHONE_NUMBER" type="string"/>
    <property name="comment" column="COMMENT" type="string"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

In this example, we have used the `<many-to-one>` element, which is discussed later in this chapter. For now, just note that this element specifies that the `STUDENT` table has a foreign key column, named `PHONE_ID`, to the primary key of the associated table, `PHONE`.

As before, to change the relationship to bidirectional, first, `Phone` must maintain a reference to its respective `Student`, and second, the `Phone` mapping must include a one-to-one relationship:

```
public class Phone {
  private int id;  private String comment;
  private String number;
  private Student student;
  //other fields and getter/setter methods
}
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Phone" table="PHONE">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="number" column="PHONE_NUMBER" type="string"/>
    <property name="comment" column="COMMENT" type="string"/>
    <one-to-one name="student"
         class="com.packtpub.springhibernate.ch06.Student"
                constrained="false" property-ref="phone" />
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, the `<one-to-one>` element uses two attributes: `constrained` and `property-ref`. The `property-ref` element indicates the name of a property in the `Student` object by which the associated `Phone` object is represented. `constrained="true"` adds a foreign key constraint which links the primary key of the `PHONE` table to the primary key of the `STUDENT` table. This approach guarantees that a `PHONE` row's primary key always references a valid `STUDENT` primary key.

The `OneToOne` annotation can also be used to map a one-to-one relationship using the foreign key strategy. For this purpose, you need to use `JoinColumn` annotation. The `JoinColumn` annotation uses a name attribute which specifies the foreign key constraint. The following shows how these annotations are used to map our simple one-to-one relationship:

```
public class Student {
@OneToOne
@JoinColumn(name="PHONE_ID")
private Phone phone;
}
```

As with identical primary keys, to make the relation bidirectional you need to provide a property of type `Student` in the `Phone` class, and use the `OneToOne` annotation with the `mappedBy`, as follows.

The <many-to-one> element

The `<many-to-one>` element maps a many-to-one relationship. A relationship is called many-to-one when multiple instances of a class are associated with a single instance of another class. For example, suppose that more than one student can have the same address because they live in the same household. Therefore, many `Student` objects can be associated with one and only one `Address` object. This relationship is usually carried out in the database by defining a foreign key in the many-class table which points into the primary key of the one-class table. The common form of this element is as follows:

```
<many-to-one
        name="propertyName"
        column="columnName"
        class="className"
        lazy="proxy|no-proxy|false"
        not-found="ignore|exception">
…
</many-to-one>
```

The meanings of these attributes and their possible values are as follows:

- `name` (mandatory) determines the name of the property inside the many-side class, representing the associated object (the one side).
- `column` (optional) specifies the foreign key column in the many-side table that points to the primary key of the one side. If this attribute is missed, the `name` attribute is used as the column name.

- `class` (optional) represents the class name of the associated object. The default value is the property type determined by reflection.
- `lazy` (optional) specifies whether the one-side object loads when the many-side object is loaded. Possible values are `proxy`, `no-proxy`, and `false`. The default is `proxy`, which tells Hibernate to load the associated object lazily with proxy generation. `no-proxy` indicates that Hibernate should use interception to load the associated object lazily. `false` means Hibernate should always load the associated object with the parent. Proxies and interceptions are discussed in Chapter 8.
- `not-found` (optional) describes Hibernate's behavior when the associated object (on the one side) does not exist. Hibernate either throws an exception or ignores it.

Other attributes for this element are `cascade`, `update`, `insert`, and `access`, which have the same meanings as with `<class>`.

As we saw earlier, the relationship between the `Student` and `Address` objects can be considered a many-to-one relationship. Therefore, we will have the Student and Address classes as follows:

```
public class Student {
  private int id;
  private Address address;
  //other fields and getter/setter methods
}
public class Address {
  private int id;
  private String street;
  private String city;
  private String zipCode;
  //other fields and getter/setter methods
}
```

The following code shows the mapping for this relation. Note that this example shows a unidirectional many-to-one relationship. We always reach an `Address` from its associated `Student` object, not vice versa. This relationship can be bidirectional if the `Address` class has a property of collections.

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <many-to-one name="address"
                 column="ADDRESS_ID"
```

```
                    cascade="all"/>
      <!-- mapping of other fields -->
    </class>
  </hibernate-mapping>
```

And this is `Phone.hbm.xml`:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Address"
   table="ADDRESS">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="street" column="STREET" type="string"/>
    <property name="city" column="CITY" type="string"/>
    <property name="zipCode" column="ZIP_CODE" type="string"/>
  <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

We used `cascade="all"` above to indicate that any operation (save, update, delete, and so on) on the many-side object (`Student` objects) should be propagated to the one-side object. Note that the new `ADDRESS_ID` column in the `STUDENT` table has been defined as a foreign key, pointing to the primary key of the `ADDRESS` table.

Any many-to-one relationship can be expressed with a one-to-many relationship on the opposite side. In other words, if object A has a relationship with more than one instance of B, then more than one instance of B may be associated with a single instance of A. The one-to-many relationship is always presented as a collection of many-side objects in the one-side object. Let's see how to map this type of relationship in Hibernate.

JPA provides the `ManyToOne` annotation to map a many-to-one relationship. The following code shows how this annotation is used to map our example:

```
public class Student {
  @ManyToOne
  @JoinColumn(name="ADDRESS_ID")
  private Address address;
}
```

The `JoinColumn` annotation specifies the foreign key column. This annotation is optional. If it is omitted, Hibernate automatically combines the name of the property, an underscore, and the database identifier name.

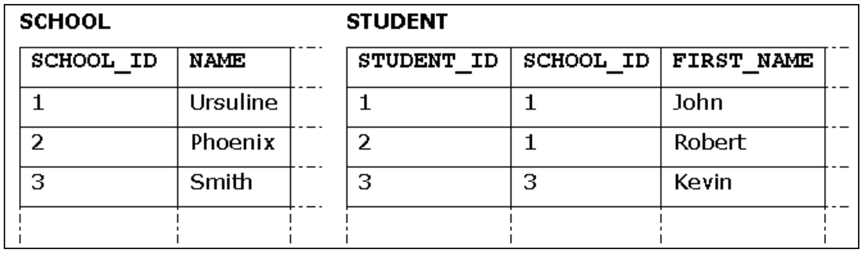
The <one-to-many> element

As with the `<many-to-one>` element, the `<one-to-many>` element maps a many-to-one relationship, but in the opposite direction. In other words, the `<one-to-many>` element maps a many-to-one relationship in which many objects can be reached through their single associated object.

For example, the relationship between `School` and `Student` objects can be considered one-to-many, because each `School` object is associated with more than one `Student` object.

The one-to-many relationship is formed when an object (the one side) has a reference to more than one other object (the many side). On the many side, each object has a reference to only one object. Although this relationship is bidirectional, the reference from the one side can be omitted to make it unidirectional.

To establish this relationship in the database, you need an extra column in the many-side table which is a foreign key to the primary key in the one-side table. The following figure shows the database view of persisting the `Student` and `School` relationship:



SCHOOL

SCHOOL_ID	NAME
1	Ursuline
2	Phoenix
3	Smith

STUDENT

STUDENT_ID	SCHOOL_ID	FIRST_NAME
1	1	John
2	1	Robert
3	3	Kevin

The `<one-to-many>` element is commonly used in this form:

```
<one-to-many class="className" not-found="exception|ignore"/>
```

The meanings of these attributes and their possible values are as follows:

- `class` (optional) specifies the class name of the many-side associated object.
- `not-found` (optional) describes Hibernate's behavior when the many-side associated object does not exist by loading the entity object. The possible values are `exception` and `ignore`, meaning Hibernate must either throw an exception or ignore it.

In the Java object model, any one-to-many relationship is represented by an array, or by collection instances of the many-side object in the one-side object if the object relationship is navigable from the one side to the many side.

Note that since each collection is mapped by using a `<list>`, `<bag>`, `<idbag>`, `<map>`, or `<set>` element, mapping a one-to-many relationship enforces using one of these elements.

Also, note again that each one-to-many relationship on the one side means a many-to-one relationship on the other side. For example, the navigation from `School` to `Student` involves a one-to-many relationship, and the opposite navigation involves a many-to-one relationship. The application designer may choose either a unidirectional or a bidirectional relationship for persistent objects, and does so based on the application's requirements.

These are the `Student` and `School` classes:

```
public class School {
  private int id;
  private String name;
  private List students = new ArrayList();
  //other fields and getter/setter methods
}

public class Student {
  private int id;
  private String firstName;
  private String lastName;
  //other fields and getter/setter methods
}
```

Hibernate provides `<list>`, `<bag>`, `<idbag>`, `<map>`, or `<set>` elements for mapping Java collections and maps, such as `java.util.List`, `java.util.Map`, and `java.util.Set`. Since any one-to-many relationship on the one side is represented by a Java collection or map, one of these elements is used for mapping the one-to-many relationship. The following code shows the mapping of the unidirectional relationship between the `Student` and `School` classes. The relationship is navigable only from `School` to `Student`. Here is the `School.hbm.xml`:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.School"
table="SCHOOL">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

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

    <bag name="students" cascade="all" >
      <key column="SCHOOL_ID"/>
```

```
        <one-to-many class="com.packtpub.springhibernate.ch06.Student"/>
      </bag>

    <!-- mapping of other fields -->
    </class>
  </hibernate-mapping>
```

And the following is the `Student.hbm.xml`:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

    <property name="firstName"  column="FIRST_NAME"  type="string"/>

    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

Here, the `<bag>` element maps the `java.util.List` object. The nested `<key>` element inside the `<bag>` maps the foreign key column in the many-side table to the primary key of the one-side table. The `<bag>` element uses the nested `<one-to-many>` element inside, which determines that the relationship is between the `School` and `Student` classes.

To map this relationship with annotations, you can simply use `OneToMany` annotation with the `mappedBy` attribute as follows:

```
public class School {

  @OneToMany(mappedBy = "school")
  private List students = new ArrayList();

}
```

To change this relationship to bidirectional, make these changes:

```
public class Student {
  private int id;
  private String firstName;
  private School school;
  //other fields and setter/getter methods
}

<hibernate-mapping>
```

```
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

    <property name="firstName" column="FIRST_NAME" type="string"/>

    <many-to-one name="school"
                 class="com.packtpub.springhibernate.ch06.School"
                 column="SCHOOL_ID"
                 cascade="all"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, the `Student` class maintains a reference to the associated `School` object. The mapping definition for the `Student` class now includes a `<many-to-one>` element which defines the mapping of the `school` property in the `Student` class.

The `ManyToOne` annotation is used as follows to map the reverse side of the one-to-many relationship:

```
public class Student {

  @ManyToOne
  @JoinColumn(name = "SCHOOL_ID")
  private School school;

}
```

Mapping a one-to-many relationship with other collections

In all of the examples we've seen so far, the one-to-many relationship is represented by an instance of `java.util.List` in the object model. This object is always mapped with the `<bag>` element. However, a one-to-many relationship can be represented by another collection, and may be mapped with another element as well. In this section, I have shown how you can map other collections, which maintain entity types, with Hibernate.

Mapping a set of entity types

Suppose the `School` class uses an instance of `java.util.Set` to maintain its associated `Student` objects:

```
public class School  {
  private int id;
  private String name;
  private Set students;
  //other fields and setter/gettter methods
}
```

You can map this with the following mapping definition:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.School"
table="SCHOOL">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

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

    <set name="students" cascade="all" >
      <key column="SCHOOL_ID"/>
      <one-to-many class="com.packtpub.springhibernate.ch06.Student"/>
    </set>

    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, we have a `<set>` element to map the `students` property of the `School` class. This `<set>` element nests with the `<key>` and `<one-to-many>` elements. The `<key>` element determines the table column that stores `Student` objects, which is a foreign key to the primary key value of the associated `School` object. In our example, there is a `SCHOOL_ID` column in the `STUDENT` table, which is a foreign key to the primary key of the `SCHOOL` table. The `<element>` element specifies the type and destination of each element in the `Set`, and here, it is the `Student` class. To make this relationship with JPA, you can easily use the `OneToMany` annotation to map the `students` property as the many side of the relationship, as shown in the following:

```
public class School {
  @OneToMany(mappedBy = "school")
  private Set students = new HashSet();
}
```

Note that you do not specify the foreign key column used to persist the relationship. This constraint is specified on the other side, where the many-to-one relationship is mapped.

Mapping a collection of entity types with <idbag>

Consider the `School` class that now uses an instance of `java.util.Collection` to refer to its associated `Student` objects:

```
public class  School {
  private int id;
  private String name;
  private Collection students = new ArrayList();
  //other fields and setter/gettter methods
}
```

The mapping definition is as follows:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.School"
table="SCHOOL">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

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

    <idbag name="students" cascade="all" order-by="firstName">
      <collection-id column="ID" type="int">
        <generator class="native"/>
      </collection-id>
      <key column="SCHOOL_ID"/>
      <one-to-many class="com.packtpub.springhibernate.ch06.Student"/>
    </idbag>

    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As you can see, the `<idbag>` nests with three subelements: `<collection-id>`, `<key>`, and `<one-to-many>`. The `<collection-id>` element specifies the primary key column of the collection table, which holds the identifier value for each collection's element and identifier generation strategy. `<one-to-many>` specifies the type and destination of each element in the `Collection`. In our case, it's the `Student` class.

Mapping a list of entity types

Assume that the `School` class uses an instance of `java.util.List` to maintain its associated `Student` objects:

```
public class School {
  private int id;
  private String name;
  private List students = new ArrayList();
  //other fields and setter/gettter methods
}

<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.School"
table="SCHOOL">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="name" column="NAME" type="string"/>
    <list name="students" cascade="all" >
      <key column="SCHOOL_ID"/>
      <index column="POSITION"/>
      <one-to-many class="com.packtpub.springhibernate.ch06.Student"/>
    </list>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

Using a `<list>` element to map a one-to-many relationship is similar to using it for value types, except that the `<list>` element uses a nested `<one-to-many>` element to specify the persistent instances are held by the `java.util.List` instance.

Mapping a java.util.Map of entity types

The following `School` class uses a `Map` to hold the `Student` associated objects:

```
public class School {
  private int id;
  private String name;
  private Map students = new HashMap();
  //other fields and setter/gettter methods
}
```

Here is the mapping definition for this class:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.School"
table="SCHOOL">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="name" column="NAME" type="string"/>
    <map name="students" cascade="all">
      <key column="SCHOOL_ID"/>
      <index column="SCHOOL_KEY" type="string"/>
      <one-to-many class="com.packtpub.springhibernate.ch06.Student"/>
    </map>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

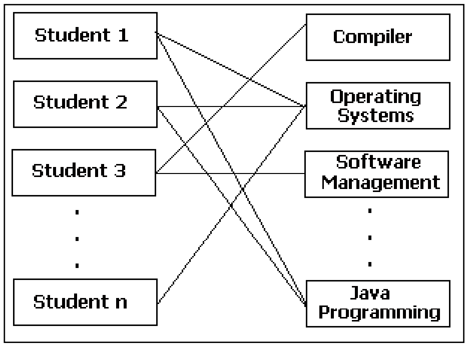
Using a `<map>` element here is similar to using it for value types, except that the `<map>` element uses a nested `<one-to-many>` element to specify the type of the `List` elements. In our case, it's the `Student` class. The `SCHOOL_KEY` column in the `STUDENT` table maintains the key of each map value. JPA provides the `MapKey` annotation to provide information about the key of the map in relationships which are established through `java.util.Map`. For our example, we would have the following to map the relationship:

```
@MapKey(name="ID")
@OneToMany
private Map students = new HashMap();
```

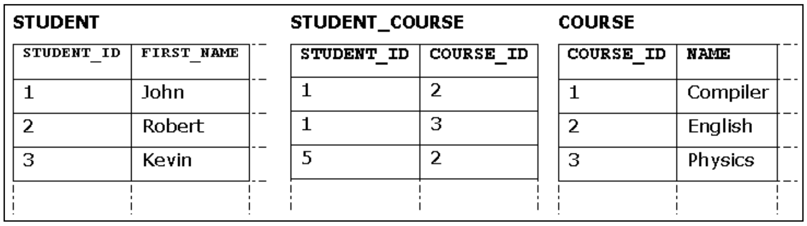
`MapKey` used a `name` attribute which maps a property of the target entity as the key of the map.

The <many-to-many> element

The `<many-to-many>` element maps a many-to-many relationship. A relationship is called many-to-many when each instance of one class is associated with many instances of another class, and vice versa. The following figure shows an example of a many-to-many relationship between `Student` and `Course` objects. As you can see in the figure, each `Student` object can be associated with more than one `Course` object, and vice versa:



A many-to-many relationship is usually mapped by using three tables: one for each class, and one for expressing their relationship, as shown in the following figure. The intermediate table has two columns: one column is a foreign key to the first table's primary key, and the other column is a foreign key to the second table's primary key:



STUDENT

STUDENT_ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_COURSE

STUDENT_ID	COURSE_ID
1	2
1	3
5	2

COURSE

COURSE_ID	NAME
1	Compiler
2	English
3	Physics

The `<many-to-many>` element is used like this:

```
<many-to-many column="columnName"
              class="className"
              not-found="ignore|exception"
              property-ref="propertyNameFromAssociatedClass"/>
```

The meanings of these attributes and their possible values are as follows:

- `column` (optional) refers to the column in the intermediate table which maintains the associated object's identifiers.
- `class` (mandatory) specifies the class name of the associated objects.
- `not-found` (optional) determines Hibernate's behavior when an associated object identifier exists in the intermediate table, but does not exist in the mapped table. Valid values are `exception` and `ignore`, meaning Hibernate should either throw an exception or ignore it.
- `property-ref` (optional) is used for nonprimary-key associations. This attribute specifies a property name of the associated object that must be used (instead of the identifier of the associated object) to establish association.

The following code shows the `Student` and `Course` classes and their mappings:

```
public class Student {
  private int id;
  private String firstName;
  private List courses = new ArrayList();
  //other fields and setter/getter methods
}
public class Course {
  privae int id;
  private String name;
  //other fields and setter/getter methods
}
<!-- Student.hbm.xml -->
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Student"
table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="FIRST_NAME" type="string"/>
    <bag name="courses"
         cascade="all"
         table="STUDENT_COURSE"
         lazy="false">
      <key column="STUDENT_ID"/>
      <many-to-many class="com.packtpub.springhibernate.ch06.Course"
                    column="COURSE_ID"/>
    </bag>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
<!-- Course.hbm.xml -->
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Course"
table="COURSE">
    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>
    <property name="name" column="NAME" type="string"/>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

JPA provides the `ManyToMany` annotation to map a many-to-many relationship. It's very simple and easy to use. The following code shows how this annotation can be used to map our example:

```
@ManyToMany
@JoinTable(
  name = "STUDENT_COURSE",
  joinColumns = {@JoinColumn(name = "STUDENT_ID")},
  inverseJoinColumns = {@JoinColumn(name = "COURSE_ID")}
)
private List courses = new ArrayList();
```

Note where the intermediate table's name and its columns names appear. This relationship is unidirectional. However, you could make it bidirectional with the following changes:

```
public class Course {
  privae int id;
  private String name;
  private List students = new ArrayList();
  //other fields and setter/getter methods
}

<!-- Course.hbm.xml -->
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch06.Course"
table="COURSE">

    <id name="id" type="int" column="ID">
      <generator class="increment"/>
    </id>

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

    <bag name="students"
         cascade="all"
         table="STUDENT_COURSE"
         lazy="false">
      <key column="COURSE_ID"/>
      <many-to-many class="com.packtpub.springhibernate.ch06.Student"
                    column="STUDENT_ID"/>
    </bag>

    <!-- mapping of other fields -->

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

To map bidirectional many-to-many relationship, you simply need to put another ManyToMany annotation on the opposite side of the relationship, as follows:

```
@ManyToMany(mappedBy = "courses")
private List students = new ArrayList();
```

The mappedBy attribute refers to the name of property, which holds a list of courses.

Summary

In this chapter, we discussed some of the more advanced concepts behind mapping. We started our discussion by introducing the <component> element to map a class property which is represented as an object of the other class in a single table.

We continued with different kinds of object associations and their mapping. In practice, there are four kinds of associations: one-to-one, one-to-many, many-to-one, and many-to-many. For each kind of association, a corresponding element exists in Hibernate mapping. A relationship is called bidirectional, if we can reach one side of the association from the other side, and vice versa. In contrast, a relationship is called unidirectional when we can navigate the relationship in one direction only.

Hibernate allows you to map collections using <list>, <bag>, <idbag>, <set>, and <map> elements. These elements always appear in a one-to-many or a many-to-many relationship. While <list>, <bag>, and <idbag> are used to map a collection object of type java.util.List, <map> and <set> are used to map objects of type java.util.Map and java.util.Set, respectively.

<list> lets you preserve the ordering of collection elements. <idbag> and <bag> act similarly, and neither cares about the elements order. However, <idbag> uses an extra column to enhance the performance of update and delete operations.

7
Hibernate Types

Hibernate allows *transparent persistence*, which means the application is absolutely isolated from the underlying database storage format. Three players in the Hibernate scene implement this feature: Hibernate dialect, Hibernate types, and HQL. The Hibernate dialect allows us to use a range of different databases, supporting different, proprietary variants of SQL and column types. In addition, HQL allows us to query persisted objects, regardless of their relational persisted form in the database.

Hibernate types are a representation of databases SQL types, provide an abstraction of the underlying database types, and prevent the application from getting involved with the actual database column types. They allow us to develop the application without worrying about the target database and the column types that the database supports. Instead, we get involved with mapping Java types to Hibernate types. The database dialect, as part of Hibernate, is responsible for transforming Java types to SQL types, based on the target database. This gives us the flexibility to change the database to one that may support different column types or SQL without changing the application code.

In this chapter, we will discuss the Hibernate types. We will see how Hibernate provides built-in types that map to common database types. We'll also see how Hibernate allows us to implement and use custom types when these built-in types do not satisfy the application's requirements, or when we want to change the default behavior of a built-in type. As you will see, you can easily implement a custom-type class and then use it in the same way as a built-in one.

Built-in types

Hibernate includes a rich and powerful range of built-in types. These types satisfy most needs of a typical application, providing a bridge between basic Java types and common SQL types. Java types mapped with these types range from basic, simple types, such as `long` and `int`, to large and complex types, such as `Blob` and `Clob`. The following table categorizes Hibernate built-in types with corresponding Java and SQL types:

Java Type	**Hibernate Type Name**	**SQL Type**
Primitives		
`Boolean` or `boolean`	`boolean`	`BIT`
	`true_false`	`CHAR(1)('T'or'F')`
	`yes_no`	`CHAR(1)('Y'or'N')`
`Byte` or `byte`	`byte`	`TINYINT`
`char` or `Character`	`character`	`CHAR`
`double` or `Double`	`double`	`DOUBLE`
`float` or `float`	`float`	`FLOAT`
`int` or `Integer`	`integer`	`INTEGER`
`long` or `Long`	`long`	`BIGINT`
`short` or `Short`	`short`	`SMALLINT`
String		
`java.lang.String`	`string`	`VARCHAR`
	`character`	`CHAR(1)`
	`text`	`CLOB`
Arbitrary Precision Numeric		
`java.math.BigDecimal`	`big_decimal`	`NUMERIC`
Byte Array		
`byte[]` or `Byte[]`	`binary`	`VARBINARY`
Time and Date		
`java.util.Date`	`date`	`DATE`
	`time`	`TIME`
	`timestamp`	`TIMESTAMP`
`java.util.Calendar`	`calendar`	`TIMESTAMP`
	`calendar_date`	`DATE`
`java.sql.Date`	`date`	`DATE`
`java.sql.Time`	`time`	`TIME`
`java.sql.Timestamp`	`timestamp`	`TIMESTAMP`

Java Type	**Hibernate Type Name**	**SQL Type**
Localization		
`java.util.Locale`	`locale`	`VARCHAR`
`java.util.TimeZone`	`timezone`	
`java.util.Currency`	`currency`	
Class Names		
`java.lang.Class`	`class`	`VARCHAR`
Any Serializable Object		
`java.io.Serializable`	`Serializable`	`VARBINARY`
JDBC Large Objects		
`java.sql.Blob`	`blob`	`BLOB`
`java.sql.Clob`	`clob`	`CLOB`

Although the SQL types specified in the table above are standard SQL types, your database may support somewhat different SQL types. Refer to your database documentation to find out which types you may use instead of the standard SQL types shown in the table above.

Don't worry about the SQL types that your database supports. The SQL dialect and JDBC driver are always responsible for transforming the Java type values to appropriate SQL type representations.

The `type` attribute specifies Hibernate types in mapping definitions. This helps Hibernate to create an appropriate SQL statement when the class property is stored, updated, or retrieved from its respective column.

The `type` attribute may appear in different places in a mapping file. You may use it with the `<id>`, `<property>`, `<discriminator>`, `<index>`, and `<element>` elements. Here is a sample mapping file with some `type` attributes in different locations:

```
<hibernate-mapping>
  <class name="Person" table="PERSON" discriminator-value="PE">

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

    <discriminator column="PERSON_TYPE" type="string"/>

    <property name="birthdate" column="BIRTHDATE" type="date"/>

    <list name="papers" table="STUDENT_PAPER">
```

```
            <key column="STUDENT_ID"/>
            <list-index column="POSITION"/>
            <element column="PAPER_PATH" type="string"/>
        </list>
        <!-- mapping of other fields -->
      </class>
    </hibernate-mapping>
```

If a property is mapped without the `type` attribute, Hibernate uses the reflection API to find the actual type of that property and uses the corresponding Hibernate type for it. However, you should specify the `type` attribute if that property can be mapped with more than one Hibernate type. For example, if a property is of type `java.lang.String`, and its mapping definition does not include the `type` attribute, Hibernate will use the reflection API and select the type `string` for it. This means you need to explicitly define the Hibernate type for a Java `String` if you want to map the `String` with a `character` or `text` Hibernate type.

Custom types

For most mappings, Hibernate's built-in types are enough. However, in some situations, you may need to define a custom type. These situations generally happen when we want Hibernate to treat basic Java types or persistent classes differently than it normally would. Here are some situations where you may need to define and use a custom type:

- **Storing a particular Java type in a column with a different SQL type than Hibernate normally uses**: For example, you might want to store a `java.util.Date` object in a column of type `VARCHAR`, or a `String` object in a `DATE` column.
- **Mapping a value type**: Value types, the dependent persistent classes that do not have their own identifiers, can be mapped with custom types. This means you can treat value types similarly to primitive types and map them with the `<property>` element, instead of `<component>`. For example, the `Phone` class in the previous chapter was mapped with `<component>`. You could implement custom type and use it to map `Phone` objects with `<property>`.
- **Splitting up a single property value and storing the result in more than one database column**: For example, assume that any phone number is split-up into four components—representing country code, area code, exchange, and line number, stored in four columns of the database. We may take this approach to provide a search facility for countries, areas, exchanges, and line numbers. If the phone numbers are represented as long numbers populated from four columns, we need to define a custom type and tell Hibernate how to assemble the number.

- **Storing more than one property in a single column**: For example, in Chapter 6, the `papers` property of the `Student` class was represented as an object of `java.util.List` and held the file paths of all of the papers the student has written. You can define a custom type to persist all of the papers file paths as a semicolon-separated string in a single column.
- **Using an application-specific class as an identifier for the persistent class**: For example, suppose you want to use the application-specific class `CustomIdentifier`, instead of the `int`, `long`, `String`, and so on, for persistent class identifiers. In this case, you also need to implement an `IdentifierGenerator` to tell Hibernate how to create new identifier values for non-persisted objects.

In practice, other use cases also need custom types for implementation and use. In all of these situations, you must tell Hibernate how to map a particular Java type to a database representation. You do this by implementing one of the interfaces which Hibernate provides for this purpose. The basic and most commonly used of these interfaces include `org.hibernate.usertype.UserType` and `org.hibernate.usertype.CompositeUserType`. Let's look at these in detail, discussing their differences, and how to use them.

UserType

`UserType` is the most commonly used Hibernate extension type. This interface exposes basic methods for defining a custom type. Here, we introduce a simple case and show how a custom type can provide a convenient mapping definition for it.

Suppose that the history of any school is represented by an individual class, `History`. Obviously, the `History` class is a value type, because no other persistent class uses the `History` class for its own use. This means that all `History` objects depend on `School` objects. Moreover, each school has its own history, and history is never shared between schools. Here is the `School` class:

```
package com.packtpub.springhibernate.ch07;

import java.io.Serializable;

public class School implements Serializable {

  private long id;
  private History history ;
  //other fields

  //setter and getter methods
  public long getId() {
    return id;
  }
```

```
    public void setId(long id) {
      this.id = id;
    }
    public History getHistory() {
      return history;
    }
    public void setHistory(History history) {
      this.history = history;
    }

    //other setters and getters
  }
```

And this is the `History` class:

```
  package com.packtpub.springhibernate.ch07;
  import java.io.Serializable;
  import java.util.Date;
  public class History implements Serializable {
    long initialCapacity;
    Date establishmentDate;
    public long getInitialCapacity() {
      return initialCapacity;
    }
    public void setInitialCapacity(long initialCapacity) {
      this.initialCapacity = initialCapacity;
    }
    public Date getEstablishmentDate() {
      return establishmentDate;
    }
    public void setEstablishmentDate(Date establishmentDate) {
      this.establishmentDate = establishmentDate;
    }
  }
```

Note that I have intentionally omitted all irrelevant fields of the two classes to keep the example simple.

Our strategy in mapping a value type so far is to use one table for persisting both the persistent class and its associated value types. Based on this strategy, we need to use a `SCHOOL` table, which stores all of the `School` and `History` properties, and then map both `School` and its `History` class into that table through the `<component>` element in the mapping file. The mapping definition for `School` and its associated `History` class is as follows:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch07.School"
table="SCHOOL">
    <id name="id" type="long" column="id">
      <generator class="increment"/>
    </id>
    <component name="history"
               class="com.packtpub.springhibernate.ch07.History">
      <property name="initialCapacity" column="INITIAL_CAPACITY"
                type="long"/>
      <property name="establishmentDate" column="ESTABLISHMENT_DATE"
                type="date"/>
    </component>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

As an alternative approach, you can map the `History` class with a custom type. You do this by implementing a custom type, `HistoryType`, which defines how to map `History` objects to the target table. Actually, Hibernate does not persist a custom type. Instead, the custom type gives Hibernate information about how to persist a value type in the database. Let's implement a basic custom type by implementing the `UserType` interface. In the next section of this chapter, we'll discuss how to map `History` with an implementation of another Hibernate custom type interface, `CompositeUserType`.

The following code shows the `HistoryType` class that implements the `UserType` interface, providing a custom type for the `History` class:

```
package com.packtpub.springhibernate.ch07;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import java.io.Serializable;
import java.util.Date;

import org.hibernate.Hibernate;
import org.hibernate.HibernateException;
import org.hibernate.usertype.UserType;
```

```
public class HistoryType implements UserType {
  private int[] types = { Types.BIGINT, Types.DATE};
  public int[] sqlTypes() {
    return types;
  }
  public Class returnedClass() {
    return History.class;
  }
  public boolean equals(Object a, Object b) throws HibernateException
{
    return (a == b) ||
      ((a != null) && (b != null) && (a.equals(b)));
  }
  public int hashCode(Object o) throws HibernateException {
    return o.hashCode();
  }
  public Object nullSafeGet(ResultSet rs, String[] names, Object owner)
          throws HibernateException, SQLException {
    Long initialCapacity = rs.getLong(names[0]);
    // check if the last column read is null
    if (rs.wasNull()) return null;
    Date establishmentDate = rs.getDate(names[1]);
    History history = new History() ;
    history.setInitialCapacity(initialCapacity.longValue());
    history.setEstablishmentDate(establishmentDate);
    return history;
  }
  public void nullSafeSet(PreparedStatement ps, Object value,
    int index) throws HibernateException, SQLException {
    if(value==null){
       ps.setNull(index, Hibernate.LONG.sqlType());
       ps.setNull(index+1, Hibernate.DATE.sqlType());
    }else{
       History history = (History) value;
       long initialCapacity = history.getInitialCapacity();
       Date establishmentDate = history.getEstablishmentDate();
       Hibernate.LONG.nullSafeSet(ps, new Long(initialCapacity),
       index);
```

```
        Hibernate.DATE.nullSafeSet(ps, establishmentDate, index + 1);
      }
    }
    public Object deepCopy(Object o) throws HibernateException {
      if (o == null) return null;
      History origHistory = (History) o;
      History newHistory = new History();
      newHistory.setInitialCapacity(origHistory.getInitialCapacity());
      newHistory.setEstablishmentDate(origHistory.
         getEstablishmentDate());
      return newHistory;
    }
    public boolean isMutable() {
      return true;
    }
    public Serializable disassemble(Object value) throws
      HibernateException {
      return (Serializable) value;
    }
    public Object assemble(Serializable cached, Object owner)
              throws HibernateException {
      return cached;
    }
    public Object replace(Object original, Object target, Object owner)
              throws HibernateException {
      return original;
    }
  }
```

The following table provides a short description for the methods in the `UserType` interface:

Method	Description
`public int[] sqlTypes()`	This method returns an array of `int`, telling Hibernate which SQL column types to use for persisting the entity properties. You may use the SQL types defined as constants in the `java.sql.Types` class directly, or you may call the `sqlType()` method of Hibernate types defined as constants in the `org.hibernate.Hibernate` class. For example, in `HistoryType`, recently discussed, you may alternatively define the types as follows: `private int[] types = {Hibernate.BIG_INTEGER.sqlType(),Hibernate.DATE.sqlType()};` Note that you should specify the SQL types in the order in which they appear in the subsequent methods.
`public Class returnedClass()`	This method specifies which Java value type is mapped by this custom type.
`public boolean isMutable()`	This method specifies whether the value type is mutable. Since immutable objects cannot be updated or deleted by the application, defining the value type as immutable allows Hibernate to do some minor performance optimization.
`public Object deepCopy(Object value)`	This method creates a copy of the value type if the value type is mutable. Otherwise, it returns the current instance. Note that when you create a copy of an object, you should also copy the object associations and collections.
`public Serializable disassemble(Object value)`	Hibernate may cache any value-type instance in its second-level cache. For this purpose, Hibernate calls this method to convert the value-type instance to the serialized binary form. Return the current instance if the value type implements the `java.io.Serializable` interface, otherwise, convert it to a `Serializable` object. Chapter 12 discusses Hibernate cache services and strategies.
`public Object assemble(Serializable cached, Object owner)`	Hibernate calls this method when the instance is fetched from the second-level cache and converted back from binary serialized to the object form.

Method	Description
`public Object replace(Object original, Object target, Object owner)`	Assume that the application maintains an instance of the value type that its associated session has already closed. As you will see in Chapter 8, such objects are not tracked and managed by Hibernate, so they are called **detached**. Hibernate lets you merge the detached object with a session-managed persistent object through the session's `merge()` method. Hibernate calls the `replace()` method when two instances, detached and session-managed, are merged. The first and second arguments of this method are value-type instances associated with a detached and session-managed persistent object, respectively. The third argument represents the owner object, the persistent object that owns the original value type: `School` in our case. This method replaces the existing (`target`) value in the persistent object we are merging, with a new (`original`) value from the detached persistent object we are merging. For immutable objects or null values, return the first argument. For mutable objects, return at least a copy of the first argument through the `deepCopy()` method. We will discuss the merge operation in Chapter 8.
`public Object nullSafeGet(ResultSet resultSet, String[] names, Object owner)`	This method constructs the value-type instance, when the instance is retrieved from the database. `resultset` is the JDBC `ResultSet` object containing the instance values, `names` is an array of the column names queried, and `owner` is the persistent object associated with the value-type instance. Note that you should handle the possibility of `null` values.
`public void nullSafeSet (PreparedStatement statement, Object value, int index)`	This method is called when the value-type instance is written to a prepared statement to be stored or updated in the database. Handle the possibility of `null` values. A multi-column type should be written to parameters starting from `index`.
`public boolean equals(Object x, Object y)`	This method compares two instances of the value type mapped by this custom type to check whether they are equal.
`public int hashCode( Object x)`	This method returns a hashcode for the instance, consistent with persistence equality.

In some of the methods shown in the table above, the owner object is passed as an argument to the method. You can use this object if you need the other properties of the value-type instance. For example, you can access a property of the owner if you need it to calculate the value for a value-type property.

Serializing and caching issue

If the value type, in our case `History`, does not implement the `java.io.Serializable` interface, then its respective custom type is responsible for properly serializing or deserializing the value type. Otherwise, the value-type instances cannot be cached by the Hibernate second-level cache service.

To use the defined custom type, you need to edit the mapping file as shown below:

```
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch07.School"
   table="SCHOOL">
    <id name="id" type="long" column="id">
      <generator class="increment"/>
    </id>
    <property name="history"
              type="com.packtpub.springhibernate.ch07.
     HistoryType">
      <column="INITIAL_CAPACITY" type="long"/>
      <column="ESTABLISHMENT_DATE" type="date"/>
    </property>
    <!-- mapping of other fields -->
  </class>
</hibernate-mapping>
```

Note that you should specify the columns in order, corresponding to the order of types returned by the `getTypes()` method and the index of the values the `nullSafeGet()` and `nullSafeSet()` handle.

So far, all we have done is implemented a custom type in the simplest form. The implemented custom type only transforms the value-type instances to the database columns and vice versa. A custom type may be more complicated than we have seen so far, and can do much more sophisticated things. The advantage of this implemented custom type is obvious: we can define our own strategy for mapping value types. For instance, a property of the value type can be stored in more than one column, or more than one property can be stored in a single column.

The main shortcoming of this approach is that, we have hidden the value-type properties from Hibernate. Therefore, Hibernate does not know anything about the properties inside the value type, or how to query persistent objects based on their associated value types as problem constraints are involved. Let's look at `CompositeUserType` and how it can solve this problem.

CompositeUserType

Another way to define a custom type is to use the `CompositeUserType` interface. This type is similar to `UserType`, but with more methods to expose the internals of your value-type class to Hibernate. `CompositeUserType` is useful when application query expressions include constraints on value-type properties. If you want to query the persistent objects with constraints on their associated value types, map the associated value types with `CompositeUserType`. The following code shows the `CompositeHistoryType` implementation for `History`:

```
package com.packtpub.springhibernate.ch07;

import org.hibernate.usertype.CompositeUserType;
import org.hibernate.type.Type;
import org.hibernate.HibernateException;
import org.hibernate.Hibernate;
import org.hibernate.engine.SessionImplementor;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.io.Serializable;
import java.util.Date;

public class CompositeHistoryType implements CompositeUserType {

  private String[] propertyNames = {"initialCapacity",
"establishmentDate"};

  private Type[] propertyTypes = {Hibernate.LONG, Hibernate.DATE};

  public String[] getPropertyNames() {
    return propertyNames;
  }

  public Type[] getPropertyTypes() {
    return propertyTypes;
  }
   public Object getPropertyValue(Object component, int property) {
    History history = (History) component;
    switch (property) {
      case 0:
        return new Long(history.getInitialCapacity());
      case 1:
        return history.getEstablishmentDate();
    }
    throw new IllegalArgumentException(property +
                " is an invalid property index for class type " +
                component.getClass().getName());
  }
   public void setPropertyValue(Object component, int property,
                                Object value) {
    History history = (History) component;
    switch (property) {
      case 0:
        history.setInitialCapacity(((Long) value).longValue());
      case 1:
        history.setEstablishmentDate((Date) value);
      default:
           throw new IllegalArgumentException(property +
               " is an invalid property index for class type " +
                 component.getClass().getName());
    }

  }
  public Class returnedClass() {
    return History.class;
  }
  public boolean equals(Object o1, Object o2) throws
HibernateException {
    if (o1 == o2) return true;
    if (o1 == null || o2 == null) return false;
    return o1.equals(o2);
  }
  public int hashCode(Object o) throws HibernateException {
    return o.hashCode();
  }
  public Object assemble(Serializable cached,
                         SessionImplementor session, Object owner)
          throws HibernateException {
    return deepCopy(cached);
  }
  public Object replace(Object original, Object target,
                  SessionImplementor sessionImplementor, Object owner)
          throws HibernateException {
```

```
      return original;
    }
    public Serializable disassemble(Object value,
                                    SessionImplementor session)
            throws HibernateException {
      return (Serializable) deepCopy(value);
    }
    public Object nullSafeGet(ResultSet rs, String[] names,
                              SessionImplementor session, Object o)
            throws HibernateException, SQLException {
      long initialCapacity = rs.getLong(names[0]);
      java.util.Date establishmentDate = rs.getDate(names[1]);
      return new History(initialCapacity, establishmentDate);
    }
    public void nullSafeSet(PreparedStatement ps,
                    Object value, int index, SessionImplementor session)
            throws HibernateException, SQLException {
      if (value == null) {
        ps.setNull(index, Hibernate.LONG.sqlType());
        ps.setNull(index + 1, Hibernate.DATE.sqlType());
      } else {
        History history = (History) value;
        long l = history.getEstablishmentDate().getTime();
        ps.setLong(index, history.getInitialCapacity());
        ps.setDate(index + 1, new java.sql.Date(l));
      }
    }
    public Object deepCopy(Object value) throws HibernateException {
      if (value == null) return null;
      History origHistory = (History) value;
      History newHistory = new History();
      newHistory.setInitialCapacity(origHistory.getInitialCapacity());
      newHistory.setEstablishmentDate(origHistory.
  getEstablishmentDate());
      return newHistory;
    }
    public boolean isMutable() {
      return true;
    }
  }
```

As you can see, this interface exposes some extra methods not seen in `UserType`. The following table shows the functionality of these methods:

Method	**Description**
`public String[] getPropertyNames()`	This method returns the names of the value type's properties that may appear in the query constraints. In the example shown in the code above, we have used both the `initialCapacity` and `establishmentDate` properties, meaning the application can query the persistent objects based on these property values.
`public Type[] getPropertyTypes()`	This method returns the corresponding types of the properties specified by the `getPropertyNames()` method. Each returned `Type` in the array corresponds to a property name with the same index in the array returned by `getPropertyNames()`. Each type is expressed as an instance of the `org.hibernate.type.Type` interface, defined as a static member in the `org.hibernate.Hibernate` class, or a custom type implemented by the developer.
`public Object getPropertyValue(Object component, int property)`	This method returns a property's value. It takes two arguments. The first argument is the value-type instance that holds the property value we want to fetch. The second argument specifies the index of the property, based on the property name returned by `getPropertyNames()`.
`public void setPropertyValue(Object component, int property, Object value)`	Hibernate uses this method to assign a value to any property of the value-type instance. This method takes three arguments. The first argument refers to the value-type instance, the second specifies the index of the property based on position of the property in the array returned by `getPropertyNames()`, and the third is the value assigned to the property.

Using this custom type is same as using `UserType`, except that you need to specify the `CompositeHistoryType` instead of `HistoryType` as follows:

```
<property name="history"
          type="com.packtpub.springhibernate.ch07.
CompositeHistoryType">
  <column="INITIAL_CAPACITY" type="long"/>
  <column="ESTABLISHMENT_DATE" type="date"/>
</property>
```

As mentioned earlier, this custom type provides an ability to query on properties of the `History` type. As you will see in Chapter 9, HQL is one approach provided by Hibernate to query the persistent object. For instance, suppose we are interested in schools established before 1980. The following code shows querying these objects with HQL, a Hibernate-specific query language that works with objects:

```
Calendar c = Calendar.getInstance();
c.set(Calendar.YEAR, 1980);
Query q = session.createQuery(
"select s from School s where s.history.establishmentDate < :edate"
).setParameter("edate", new Date(c.getTimeInMillis()));
```

All we have done in this snippet is created a `Query` object with an HQL expression indicating all `School` objects with establishment date before 1980. Note that `HistoryCompositeType` provides the ability to query the `School` object with criteria applied to `History` objects. (Don't worry about this for now since upcoming chapters cover it in detail.)

The only advantage of `CompositeUserType` over `UserType` is that `CompositeUserType` exposes the value-type properties for Hibernate. Therefore, it lets you query persistent instances based on values of their associated value-type instances.

Summary

In this chapter, we discussed Hibernate types, which define the mapping of each Java type to an SQL type. It is the responsibility of the Hibernate dialect and the JDBC driver to convert the Java types to the actual target SQL types. This means a Java type may be transformed to different SQL types when different databases are used.

Although Hibernate provides a rich set of data types, called built-in types, some situations require the definition of a new type. One such situation occurs when you want to change Hibernate's default behavior for mapping a Java type to an SQL type. Another situation is when you want to split up a class property to a set of table columns, or merge a set of properties to a table column.

Built-in types include primitive, string, byte array, time, localization, serializable, and JDBC large types.

Hibernate provides several interfaces for implementation by custom types. The most commonly used interfaces are `org.hibernate.usertype.UserType` and `org.hibernate.usertype.CompositeUserType`. The basic extension point is `UserType`. It allows us to map a value-type, but hides the value-type properties from Hibernate, so it does not provide the application with the ability to query value types. In contrast, `CompositeUserType` exposes the value-type properties to Hibernate, and allows Hibernate to query the value-types.

8
Hibernate Persistence Behavior

After you've configured Hibernate, it is ready to be used to persist application objects. An API, called the **Session API**, is responsible for persistence in Hibernate. This chapter discusses the life cycle of persistent objects in the application's lifetime: the states persistent objects go through with respect to persistence. Then, we'll delve into basic persistence operations, which trigger changes to the objects' state provided by the Session API. Finally, we'll look at some hot topics in object persistence, mainly, cascading operation and lazy loading.

The life cycle of persistent objects

Persistent classes follow the simple POJO rules without calling any Hibernate APIs. In other words, no persistent class is aware of its own persistence capability. This is why Hibernate is called a **transparent** persistence provider. Transparent persistence allows us to develop persistent classes regardless of the underlying persistence technology, whether it be Hibernate or another solution. Each POJO object can be stored by Hibernate if it meets these conditions:

- Its fields are storable in the database.
- There is a table in the database to store the object.
- There is a mapping definition that determines how to map each field of the object to a table column.

To persist objects, the application must call Hibernate through its interfaces, usually `Session` instances. The persistent objects that are passed and given back through sessions have these states in their life cycles:

- Transient
- Persistent
- Detached
- Removed

Let's look at each of these states.

Transient objects

Transient refers to persistent objects, which are not yet been persisted, and are not associated with any table's row in the database. These objects are not stored until they are explicitly stored by the issuance of a persistence request to Hibernate. Moreover, Hibernate considers transient objects in a nontransactional context: Hibernate does not monitor them for any modification, so there is no roll-back mechanism for them. Unless Hibernate is asked to persist these objects, Hibernate does not care about them, whether they are created, modified, or removed. When the application issues a persistence request for a transient object, Hibernate assigns an identifier value to the object and stores it in the database. The object is then moved to the persistent state until the session is closed, and to the detached state thereafter.

Persistent objects

Transient objects are moved to **Persistent** when they are stored in the database. Persistent refers to objects that have already been persisted. They have a database representation with appropriate primary-key values and are associated with valid `Session` objects. Until the associated sessions are closed by invoking the `close()` method, Hibernate caches these objects, tracks the changes, and keeps the database representation up-to-date as transactions are committed. Moving to the detached state is transparent, meaning no method is called for transition. Invoking the `close()` method of the associated session moves the persistent object to the detached state.

Detached objects

Detached objects, like persistent objects, have already been persisted, but their associated sessions have been closed. Hibernate does not care about them. It neither caches them nor detects their modifications to keep the database representation up-to-date. When reattached to a valid session object, these objects are moved to the persistent state.

Removed objects

Removed refers to persistent objects that are arranged by the `Session` object to be removed from the database. The removed objects are removed from the database and moved to the transient state as soon as the session's associated transaction commits. This may be done either explicitly, through invoking a delete operation for those objects, or implicitly, through invoking a delete operation for associated objects when the persistent objects have been mapped with cascading `delete`. Removed objects are erased from the database as soon as the session's associated transaction is complete, or during the explicit/implicit flushing of the session. They are then moved to the transient state.

Persistence with Hibernate

Hibernate's persistence service is exposed through different interfaces, including `Session`, `Query`, `Criteria`, and `Transaction`. Among these, the `Session` interface has a crucial role, since any Hibernate interaction involves at least a `Session` object.

`Session` objects perform basic persistence operations. These include storing a newly instantiated object and loading, updating, and deleting an already persisted object. `Session` objects provide a transaction handler to perform a set of persistence operations as a unit of work. Additionally, a `Session` object provides a simple cache for the objects loaded, stored, or updated through it.

The Session API is generally used for the following purposes:

- **Performing basic CRUD operations**: Performing the basic CRUD operations, such as save, load, update, and delete. These operations, which are exposed as the `save()`, `persist()`, `get()`, `load()`, `update()`, and `saveOrUpdate()` methods on the `Session` object, are the subject of this chapter.
- **Managing transactions**: Demarcating transactions through the `beginTransaction()`, `commit()`, and `rollback()` methods. These transactions are discussed in Chapter 12.
- **Working with filters**: Applying, enabling, and disabling filters on the data returned by executing queries. This is done by the `createFilter()`, `enableFilter()`, and `disableFilter()` methods, which are explained in the Appendix.
- **Creating queries**: Creating queries expressed in the Criteria API, or in HQL, through the `createQuery()`, `createSQLQuery()`, `createNamedQuery()`, and `createCriteria()` methods. Chapter 9 discusses these methods.

- **Working with Hibernate resources** (`Connection`, `SessionFactory`, `Transaction`): Accessing the resources that the session uses behind the scenes, including database `Connection` and `Transaction`, and the `SessionFactory` object to which the session belongs. This is done through the `connection()`, `isOpen()`, `isConnected()`, `close()`, `disconnect()`, `reconnect()`, `getTransaction()`, and `getSessionFactory()` methods, which are discussed in this chapter.

This chapter discusses the basic CRUD operations provided by the Session API. Other operations, such as querying objects, with other Hibernate interfaces, such as `Transaction`, `Query`, and `Criteria`, are covered in subsequent chapters.

The process of persistence

Any Hibernate interaction involves a resource factory, `org.hibernate.SessionFactory`, which is used to create a single-threaded resource, `org.hibernate.Session`, for each sequence of operations. The `Session` and `SessionFactory` objects are roughly analogous to `javax.sql.DataSource` and `java.sql.Connection` objects, respectively, in the case of JDBC.

The Hibernate interaction is started by configuring a `SessionFactory` object. This object wraps all connection information, and acts as a factory for `Session`. Since `SessionFactory` is a costly object, the application should use only one instance of `SessionFactory` for each database it uses. The application then obtains `Session` instances from the `SessionFactory` to perform operations. Regarding this, I introduced the `HibernateHelper` class in Chapter 4, which configures `SessionFactory` and provides `Session` instances. Using this class, any Hibernate interaction can be summarized as follows:

```
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
//perform persistence operation(s)
tx.commit();
session.close();
```

In the first line, a `Session` object is obtained. This object may be obtained through dealing with `SessionFactory` directly, or by using a utility class such as the `HibernateHelper` class. Additionally, as you will see in later chapters, Spring may be configured and set up to create a `SessionFactory` and provide `Session`.

The next line starts a Hibernate transaction. Each transaction is applied to a sequence of operations, which should be treated as an atomic operation. This means that the operations included in the transaction must all be performed successfully, or otherwise, all will fail. Since database operations are critical processes inside the application, they must be performed inside a transaction. When you are working with Hibernate, you may use the Hibernate API, or Spring, to demarcate transaction boundaries and manage transaction performance. As you will see, using Spring's declarative approach is highly recommended. For now, we will use the Hibernate API to demarcate transactions using the `beginTransaction()` and `commit()` methods of `Session`.

You are not obliged to manage transactions if your application does not have this requirement. In that case, you can set the `auto-commit` property of Hibernate to `true`, or always invoke the `flush()` method of `Session` to synchronize the database only for writing operations such as save, update, and delete. The `flush()` method flushes the session, and consequently commits all of the object modifications managed by the session.

Don't worry about transaction management with Hibernate here as Chapter 9 discusses it fully.

When the transaction starts, you are ready to perform your desired persistent operations. This is shown with the commented line in the preceding code.

The next line shows how the transaction is committed through the `Transaction` instance's `commit()` method. This flushes the session and synchronizes the database representation with the application's objects.

After everything is finished, the session instance may be closed by invoking its `close()` method. This closes the underlying database connection with which the session is associated, and releases other resources, such as the cache. Note that you cannot use the session instance after it has been closed. If you try, a `HibernateException` is thrown. Use the `isOpen()` of `Session` method to check whether the session is open or not.

To use Hibernate with JPA, similar objects to the native Hibernate API are involved. `javax.persistence.EntityManagerFactory`, `javax.persistence.EntityManager`, and `javax.persistence.EntityTransaction` are respectively similar to `SessionFactory`, `Session`, and `Transaction`, which were recently discussed. Therefore, to interact with Hibernate through JPA, we will use code similar to the following:

```
EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("instituteWebApp");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
//perform persistence operation(s)
tx.commit();
em.close();
```

Similar to `SessionFactory`, only one single `EntityManagerFactory` object is needed for each database used throughout the application. The first line of the code above does this. The next lines of the code should be done for each interaction with Hibernate. The unit of work is marked with `begin()`, `commit()`, and `rollback()` methods of the `EntityTransaction` object.

Using this style of code to manage transactions is only applicable for local transaction. As you will see in the next chapters, local transactions are a kind of transactions which are involved with only a single datasource. It is also possible to manage multiple-datasource transactions, but of course in a managed JEE environment. Discussing this type of transaction management is out of this book's scope.

Storing objects

When a persistent object is instantiated in memory, it is in the transient state. So until the application saves the object by explicitly invoking the `save()` method of a valid Hibernate session, the object is not persisted. A `Session` object is called valid, if it has not been closed by calling its `close()` method. The saving process can be as simple as follows:

```
Student std = … //std is a transient object
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
session.save(std);
tx.commit();
session.close();
```

The save() method associates the transient object with the current session and moves it to persistent state, so that the session tracks the object changes. The object is inserted into the database when the transaction is committed, or when the session is flushed. The object remains persistent while the session is open. As soon as the session is closed, the persistent object is moved to the detached state. After that point, the session does not track the object changes.

When you save an object, Hibernate uses an INSERT INTO SQL statement to synchronize that object with the database. If the persistent object changes after a save() or flush() method is invoked (and, of course, before the transaction is committed or the session is closed), Hibernate uses additional SQL UPDATE statement(s), besides the already prepared INSERT INTO statement, to synchronize the session with the database. Regarding performance optimization, call the save() or flush() method whenever the object is ready to be stored and no additional changes are applicable.

The Session API provides a variety set of save() methods for this purpose:

```
public Serializable save(Object obj) throws HibernateException
public void save(Object obj, Serializable id) throws
HibernateException
public Serializable save(String className,Object obj) throws
HibernateException
```

obj is the transient object we want to save. This object cannot be null. In the first save() method, Hibernate uses the reflection API to find the object's class, and then uses its already defined mapping to make the object persist. The obj is saved, and its assigned identifier value is returned as a Serializable object. The application can use the returned identifier value to look up the Session object for accessing the persistent object.

The second save() method takes an Object and a Serializable instance as arguments, representing the object being stored and the identifier value to be assigned to the object. This method is useful when the application, rather than Hibernate, generates identifier values.

The third save() method takes a String and an Object instance, which represent the classname and the object being stored, respectively. Hibernate uses the classname to find the object mapping. This method also returns the assigned identifier value as a Serializable instance.

If you don't know whether the object's state is transient or detached, you can use the following method as an alternative to `save()`:

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

When you use this method, Hibernate checks the object identifier to determine whether the object is transient, and therefore does not need to be saved or updated. Note that an object is in the transient state when its identifier has the unsaved value, defined in the mapping file through the `unsaved-value` attribute, similar to the following:

```
<id name="id" type="long" column="id" unsaved-value="0">
    <generator class="increment"/>
</id>
```

See Chapter 5 for more information about the `unsaved-value` attribute.

With JPA, you can use the `EntityManager`'s `persist()` method to make a transient object persistent. Therefore, we have the code like the following to save a transient object:

```
Student std = … //std is a transient object
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(std);
tx.commit();
em.close();
```

Unlike `Session`'s `save()`, the `EntityManager`'s `persist()` method here does not return the object identifier.

Object equality and identity

When an object stored in the database is detached, its corresponding table row is updated. If the object is transient, a new row is inserted. Now, if you load this object by the same session that was recently used, Hibernate provides the same reference to the object. As the result, you can use the == operator to compare persistent objects that are managed in the same session. However, if the persistent objects are associated with different sessions, the == operator will not work correctly. To compare persistent objects, you can implement the `equals()` method in persistent classes. You can find an example of the `equals()` implementation in Chapter 5.

Loading objects

Hibernate provides several `load()` and `get()` methods through `Session`, for loading persistent objects. The required data for loading any persistent object is the identifier, which must be `Serializable`. Here is an example:

```
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
Student std = (Student) session.load(Student.class, 1);
tx.commit();
session.close();
```

This example retrieves a `Student` object with identifier 1. Note that in this code, we have started and committed the transaction. This is not necessary for retrieving a persistent object. However, it is recommended because it allows us to insert or remove any other persistence operation before or after loading the object.

We can also use the `get()` method as an alternative to the `load()` method:

```
Student std = (Student) session.get(Student.class, 1);
```

However, the `get()` and `load()` methods work in different ways. If no object with the given identifier exists in the database, `get()` returns `null`, but the `load()` method throws `ObjectNotFoundException`. Moreover, the `get()` method always hits the database and fetches a fresh object, whereas `load()` returns an initialized object if the object has already been fetched and is cached by the current session, otherwise a proxy is returned. A **proxy** is an object that simulates the real instance without having real values. However, the real values may be loaded from the database if they are accessed while the associated session is open.

Although you can use the `load()` and `get()` methods interchangeably, each may be better suited to a particular case. Use the `get()` method when you need a fresh copy of an existing persistent object. You may then work with the returned object in the normal way. The `load()` method is useful when you merely need a reference to the object. For example, to remove the object, change its values, or establish an association with another object. In all of these cases, hitting the database is not required.

Here are the most common versions of the `load()` and `get()` methods:

```
public Object load(Class clazz, Serializable id) throws
HibernateException
public Object load(String className, Serializable id) throws
HibernateException
public void load(Object obj, Serializable id) throws
HibernateException
```

```
public Object get(Class clazz, Serializable id) throws
HibernateException
public Object get(String className, Serializable id) throws
HibernateException
```

In all of these methods, the `Serializable` object represents the identifier value of the object being loaded. If the object identifier is of a primitive type, and you are using JDK 1.4 or before, the identifier value should be wrapped in the corresponding wrapping object. This is not required in JDK 1.5 or later, which use autoboxing. Both `get()` and `load()` methods may use either a `Class` instance or a `String` instance as an argument. This argument represents the class of the persistent object that is being loaded from the database.

The `Object` returned by the `get()` and `load()` methods represents the persistent object being loaded in the application, either as a proxy or a real value. The only method that returns void takes the persistent object with the object identifier as arguments. The persistent object has two usages:

- It helps Hibernate find the persistent class of the object.
- Hibernate stores the loaded values in the persistent object.

Certainly, this object cannot be null.

The following snippet uses the `load()` method to load a `Student` instance with identifier 2:

```
Session session= HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
Student std1 = session.load(Student.class, 2);
Student std2 = session.load("com.packtpub.springhibernate.ch08.
Student", 2);
Student std3 = new Student();
session.load(std3, 2);
tx.commit();
//use the objects
session.close();
```

Since all of the `std1`, `std2`, and `std3` objects use the same session, they have the same reference, and comparing them with the `==` operator results in `true`.

In addition to this version of `load()` and `get()` methods, Hibernate also provides these others:

```
public Object load(Class clazz, Serializable id, LockMode lockMode)
throws HibernateException

public Object load(String className, Serializable id, LockMode
lockMode) throws HibernateException

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

public Object get(String className, Serializable id, LockMode
lockMode) throws HibernateException
```

All of these methods use an object of type `org.hibernate.LockMode` as their last argument. The `LockMode` object determines which locking strategy Hibernate should use for fetching the object. The `LockMode` class, and the locking strategies that Hibernate uses, are discussed in Chapter 12.

JPA provides the `find()` and `getReference()` methods through the `EntityManager` interface to retrieve the persistent objects:

```
<T> T find(Class<T> clazz, Object o);
<T> T getReference(Class<T> tClass, Object o);
```

Both methods receive the class and the identifier of the target object being loaded and return the loaded object. Note that the two methods use Java generics, meaning no type casting to the expected persistent class is needed. Here is an example:

```
Student std = entityManager.find(Student.class, 1);
```

The `find()` and `getReference()` methods are respectively equivalent to `Session`'s `get()` and `load()` methods. While the `find()` method returns null if there is no persistent object with the given identifier in the database, the `load()` method throws `EntityNotFoundException`. The `find()` method always hits the database and returns the fully initialized persistent object, if any exist. However, `getReference()` returns a proxy object that is initialized when a non-identifier field is accessed.

Refreshing objects

Sometimes, the persistent objects in memory need to be refreshed. **Refreshing** means reloading the in-memory object from the database to ensure that the in-memory instance represents the same value persisted in the database.

An advantage of Hibernate is that you usually do not need to worry about refreshing. Hibernate always manages the in-memory persistent objects if they are associated with valid `Session`. However, in some situations, including the following, persistent in-memory objects need to be refreshed:

- When the database is shared between your application and another application (or applications).
- When your application uses the database both directly and through Hibernate.
- When the persistent class uses a property that is set by the database triggers, so persistent objects need to be reloaded to maintain the correct value.

Hibernate provides the `refresh()` method exposed by the `Session` interface to refresh persistent objects. This method is provided in two versions:

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

For example, suppose the educational system application's `Student` class uses a property of type `Date`, which represents the time that the student registered. We use an extra column in the `STUDENT` table to hold that information. When `Student` objects are created in the application, we do not want the application to assign the registration time, because the registration process may take a long time, and the application does not know exactly when it should assign a registration time to the student. As the result, we need to use a database trigger to set the time when the `Student` instance is inserted into the database. The following snippet shows how the `Student` instance is reloaded from the database when it is stored:

```
Student std = new Student();
//set values...
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
session.save(std);
session.flush(); // Force the INSERT to occur in the database
session.refresh(std); // Reload the Student object in memory
tx.commit();
session.close();
System.out.println(std.getRegisterTime());
```

In the example, Hibernate never inserts or updates the `registerTime` property. Therefore, its mapping should use `insert="false"` and `update="false"` to specify that this property is not included in the SQL `INSERT/UPDATE` statements when a `Student` instance is inserted or updated.

JPA provide the `refresh()` method through the `EntityManager` interface to refresh an in-memory object with its data representation in the database. This method's signature is as follows:

```
void refresh(Object obj);
```

The only parameter for this method is the object to refresh.

Updating objects

As with refreshing, you do not need to worry about updating persistent objects while the `Session` object is open. Hibernate automatically manages all of the objects modifications by putting them in a queue and stores the modifications when the transaction is committed. This means that you do not need to call any Hibernate API regarding the update, as the following snippet shows:

```
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
Student std = (Student) session.get(Student.class, 1);
std.setLastName("Johnston");
tx.commit(); //update the Student instance
session.close();
```

As soon as the session is closed, the object becomes detached, meaning Hibernate does not track changes, and any modification to the object has no effect on the persistent representation of the object in the database.

The `Session` interface provides the `update()` method for updating detached objects. The `update()` method both reattaches the detached object to a valid session and flushes the changes. Here is an example:

```
Student std = … // Loaded in the previous Session
Session newSession = HibernateHelper.getSession();
Transaction tx = newSession.beginTransaction();
newSession.update(std);
std.setLastName("Johnston");
tx.commit(); //update the database with the new value
newSession.close();
```

This example reattaches the `std` object to a new session by invoking the `update()` method of `Session`. The object is synchronized with the database by issuing an SQL `UPDATE` statement when the transaction is committed. Note that in this example, we have changed the `lastName` value after invoking the `update()` method. The `update()` method reattaches the `Student` object to a new `Session`. Therefore, the object is moved to the persistent state. After all, when the transaction is committed, the session is synchronized with the database, and database representation is updated.

Note that Hibernate updates all of the object's properties, whether they have been modified or not. This is because the session does not use any mechanism for identifying modified properties, meaning which properties have changed and which have not. Instead, it assumes that all have changed. You may avoid this issue by using the `select-before-update="true"` attribute in the object's mapping definition, which forces Hibernate to select the object from the database before updating to find the modified properties.

Whether the object is modified before or after invoking the `update()` method, the session is synchronized with the database representation when the session is flushed.

Alternatively, you can use the `lock()` method to associate a detached object with a `Session` object. However, the `lock()` method does not force Hibernate to update all changes before reattaching. This means the `Session` only tracks those changes to the object that have been performed after invoking the `lock()` method. Here is an example:

```
Student std = … // Loaded in previous Session
Session newSession = HibernateHelper.getSession();
Transaction tx = newSession.beginTransaction();
std.setFirstName("Thomas"); //will be not be updated
newSession.lock(std, LockMode.NONE);
std.setLastName("Johnston");
tx.commit();
newSession.close();
```

The `lock()` method takes an `org.hibernate.LockMode` instance as the second argument. The `LockMode` instance tells Hibernate whether Hibernate should obtain a database lock, or perform a version check, to verify that the object in memory and data in the database are the same. For example, `LockMode.NONE` in the preceding example tells Hibernate there is no need to obtain any database locks or to perform a version check. This and other lock modes are discussed in Chapter 10.

JPA does not provide a specific method for updating. If the object being updated is still associated with a valid `EntityManager`, the object that changes is monitored by the `EntityManager` and it will be updated as soon as the transaction is committed. If the object is a detached object, you will need to reattach it with a valid `EntityManager`, through the `merge()` method, so the object will be monitored by the `EntityManager`. The `merge()` method will be discussed in the sequent sections.

Checking for dirty Sessions

To work more effectively with updating objects, the Session API provides other methods to check whether the session is dirty and needs to be flushed. The session is called **dirty** if it includes a modified object that needs to be synchronized with the database.

The `isDirty()`method determines whether session-managed objects have changed, and the database representation needs to be updated. Hibernate also offers a flush mode, by which you can indicate when Hibernate should flush the session. This is represented by an object of type `org.hibernate.FlushMode`, accessed by the following setter and getter methods through the `Session` object:

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

The `FlushMode` class is a type-safe enumeration with the following values:

- `ALWAYS`: After any query has been executed, the session is flushed.
- `AUTO`: After a query has been successfully executed, the session is flushed.
- `COMMIT`: The session is flushed when a transaction is committed.
- `NEVER`: The application manages session flushing by manually invoking the `flush()` method.

`AUTO` is the default flush mode. In most situations, you do not need to change it, because letting Hibernate handle this automatically usually works best.

JPA provides two values, `COMMIT` and `AUTO` through the `javax.persistence.FlushModeType` enumeration. The default flush type is `AUTO`, meaning the database representation is synchronized with the object when the `EntityTransaction` is committed, before a query is executed, or when the `EntityManager's flush()` is invoked. Making it `COMMIT` just flushes the `EntityManager` when the `TransactionManager` is committed, or when `flush()` is invoked.

Using the merge() method

The `merge()` method merges two objects which have different references in memory, but represent the same row in the database. If you were to use the `update()` method to associate a detached object with a `Session` object that already manages another instance representing the same row in the database, a `NonUniqueObjectException` is thrown. This indicates that two instances with the same identifier value will be managed with the `Session` object. The following snippet shows how the `merge()` method can be used in such cases:

```
Student std1 = …//a detached object
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
Student std2 = (Student) session.get(Student.class, 1);
//session.update(std1) Not allowed!, throws exception!
Std2 = session.merge(std1);
tx.commit();
session.close();
```

With the `merge()` method, if the session already manages the persistent instance, the state of the detached instance is copied onto the persistent instance. Otherwise, Hibernate tries to load the persistent instance from the database to merge it with the detached instance. If no corresponding object is found, a new object is instantiated to be merged with the object and inserted into the database.

Multiple invocation of the `merge()` method with arguments of different in-memory instances of the same database row always returns the same object reference.

The `EntityManager` interface provided by JPA provides a similar `merge()` method with the same meaning, signature, and usage.

Deleting objects

The session's `delete()` method removes a persistent object from the database. Its syntax is as follows:

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

obj is the persistent object that is removed. This object's identifier is all that Hibernate uses, so this object can be a dirty object not synchronized with the database. However, there should be a corresponding object with its identifier in the database.

Here is an example:

```
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
Student std = (Student) session.load(Student.class, new Long(1));
session.delete(std);
tx.commit();
session.close();
```

The delete() method moves a persistent object from the persistent to the removed state. The object is moved to the transient state as soon as the associated transaction is committed.

> If your application uses Hibernate interceptors, and the object must be passed through these interceptors to complete its life cycle, the object must first be loaded as either a proxy or a real object. Refer to the Appendix for information about interceptors.

This method will throw a HibernateException, if the object identifier doesn't have the value, or no objects in the database correspond to the object identifier value.

In some situations, you may want some operations to be performed when a persistent object is removed, meaning that you may want some tasks to be done when the delete() method is successfully invoked. The best way to implement this mechanism is by using Spring: implementing an advice and then applying it to the delete() method. This method is discussed in Chapter 11. In addition, Hibernate allows the implementation of an event/listener model, or an interceptor, to handle these cases. I have explained these techniques in the Appendix.

In addition to the delete() method provided through the Session interface, you may use HQL to remove persistent objects. However, HQL is usually used for removing objects when you need to deal with a collection of particular objects, rather than a single object. The following snippet shows an example of using HQL to remove a Student object with identifier value 1:

```
String hql = "delete from Student where id = :studentId";
Query query = session.createQuery(hql);
query.setInt("studentId", 1);
query.executeUpdate();
```

Using HQL is often preferred because it does not need to reload the objects. Note that the `delete()` method takes the persistent object as an argument. When using this method, you must have already loaded the object. As a result, HQL uses less memory, is executed faster, and reduces the network traffic communicating with the database.

In addition to using the `delete()` method and HQL, sometimes an update operation may cause Hibernate to remove another object or objects. This happens when there is a one-to-many relationship between an object and its associated objects, and their relationship is mapped with the `delete-orphan` cascading operation. Removing any associated object from its parent, means the object is removed from the database when the parent is updated. In other words, if you remove one or more associated objects from the parent, when the parent is saved, the removed objects are erased from the database. HQL is discussed in Chapter 9.

JPA provides the `remove()` method through the `EntityManager` interface. This method is used in the same way as `Session`'s `delete()` method to remove a persistent object from the database. To remove an object, that object must be associated with the `EntityManager`, meaning you must find the persistent object through its identifier, through `EntityManager`'s `getReference()`, and then remove the object.

Replicating objects

Hibernate provides a replication feature that may be useful when the application works with two or more databases. This feature allows objects to be retrieved from one database and then stored in another one. You may need replication when you need to synchronize databases with different data, when you need to roll back all changes after a particular time or all changes made under particular circumstances (for example, during non-ACID transactions), or when you are switching to a new database instance and need to move the previous data to the new schema.

The `replicate()` method of `session` is used for this purpose. This method is exposed through the `Session` interface with the following signatures:

```
public void replicate(Object persistentObject, ReplicationMode
replicationMode) throws HibernateException;
public void replicate(String className, Object persistentObject,
         ReplicationMode replicationMode) throws HibernateException;
```

These methods write a persistent object to the target database with which the session is associated. These methods take a `persistentObject` as an argument, which represents the object being replicated. These methods also take an `org.hibernate.ReplicationMode` object as an argument, which defines Hibernate's behavior when the object already exists in the target database. (Note that we say an object already exists in the target database if there is an existing database row with the same identifier as the object identifier in the target database.)

The `ReplicationMode` class is a type-self enumeration with the following values:

- `ReplicationMode.EXCEPTION`: Hibernate throws an exception, if the object already exists in the target database.
- `ReplicationMode.IGNORE`: Hibernate ignores replication, if the same object is in the target database.
- `ReplicationMode.OVERWRITE`: Hibernate overwrites the object, if it already exists in the target database.
- `ReplicationMode.LATEST_VERSION`: If the source object does not exist in the target database, Hibernate copies the object. If the object already exists in the target database, Hibernate overwrites the object if and only if the source's version is newer than the target's version.

To replicate objects, we first need two `Session` objects opened from different `SessionFactorys`, in which each `SessionFactory` works with one distinct database, but has been configured with a mapping for the same persistent class. We can then retrieve objects from one session and store them in another session. Here is an example:

```
//opening two distinct session objects, referring to two different
databases
Session session1 = HibernateHelper.getSession1();
Session session2 = HibernateHelper.getSession2();

//retrieving the object from the source database
Transaction tx1 = session1.beginTransaction();
Student std = (Student) session1.get(Student.class, new Long(1));
tx1.commit();
session1.close();

//storing the object to the target database
Transaction tx2 = session2.beginTransaction();
session2.replicate(std, ReplicationMode.OVERWRITE);
tx2.commit();
session2.close();
```

Note that in this example, the `HibernateHelper` class is different from the `HibernateHelper` class in Chapter 4, which only works with a single database. Of course, to use the `HibernateHelper` class in this case, you should refactor that class so that it can provide `Session` objects associated with two different `SessionFactory`. The following code snippet shows the refactored `HibernateHelper` class:

```
package com.packtpub.springhibernate.ch08;

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

public class HibernateHelper {

  private static final ThreadLocal session1 = new ThreadLocal();
  private static final ThreadLocal session2 = new ThreadLocal();
  private static final SessionFactory sessionFactory1 =
         new Configuration().configure("/hibernate1.cfg.xml").
buildSessionFactory();
  private static final SessionFactory sessionFactory2 =
         new Configuration().configure("/hibernate2.cfg.xml").
buildSessionFactory();

  //inaccessible constructor
  private HibernateHelper() {
  }

  public static Session getSession1() {
    Session session = (Session) HibernateHelper.session1.get();
    if (session == null) {
      session = sessionFactory1.openSession();
      HibernateHelper.session1.set(session);
    }
    return session;
  }
  public static Session getSession2() {
    Session session = (Session) HibernateHelper.session2.get();
    if (session == null) {
      session = sessionFactory2.openSession();
      HibernateHelper.session2.set(session);
    }
    return session;
  }
}
```

Cascading operations

A **cascading operation** indicates how changes to the persistent object affect associated objects when an object is persisted. For instance, if the removed object is associated with other objects, what should happen to the others when the removed object is erased from the database?

I'll explain with an example. Consider the `Teacher` class in the educational system application. Suppose that each teacher is associated with a single course, meaning the `Teacher` class has a property of type `Course`. If we use the `TEACHER` and `COURSE` tables, respectively, to store the `Teacher` and `Course` objects in the database, then corresponding to each `Teacher` object is a single `Course` object, and corresponding to each row in the `TEACHER` table is a single row in the `COURSE` table. In this situation, when a `Teacher` object is removed, what do you expect to happen to its related `Course` object?

The application's requirements may force you to either delete the associated object or to ignore it. In Hibernate, the result of this decision is expressed through a configuration option in the mapping definition called `cascade`. This property specifies how changing an object affects its associated objects. In our example, the `cascade` attribute in the mapping file can specify whether or not the `Course` object should be removed when its associated `Teacher` object is deleted.

Hibernate offers several different values for cascading, including the following:

- `none`: This is the default value and specifies that Hibernate must ignore the associated object(s) when it saves, updates, or removes the persistent object.
- `save-update`: This specifies that Hibernate must save or update the associated object(s) only when it saves or updates a persistent object.
- `delete`: This determines that Hibernate must delete the associated object(s) when it removes the persistent object.
- `all`: This indicates that Hibernate must save, update, or remove the associated object(s) when it respectively saves, updates, or removes the persistent object.
- `delete-orphan`: This is used in mapping the one-to-many relationship to tell Hibernate if a many-side object is removed from its associated one-side object, Hibernate must remove the many-side object from the database when the one-side is updated.

You can specify cascading operations in the mapping file, where the mapping of object associations is defined, through the `cascade` attribute. Note that the `cascade` attribute can have more than one value. If it does, the values are presented as a comma-separated list.

JPA defines different cascading values through the `javax.persistence.CascadeType` enumeration. These values, with their meanings, are as follows:

- `PERSIST`: Hibernate persists all associated transient instances of an object being passed to the `EntityManager`'s `persist()` when the `EntityManager` is flushed.
- `MERGE`: Hibernate merges all associated detached instances of an object being passed to the `EntityManager`'s `merge()` method with their corresponding persistent values in the database.
- `REMOVE`: Hibernate deletes the associated instances when the object is removed by the `EntityManager`'s `remove()` method.
- `REFRESH`: Hibernate refreshes the associated objects for an object being refreshed by the `EntityManager`'s `refresh()`.
- `ALL`: This option allows all of the above cascading operations to be used together.

The cascading operation can be defined through the `cascade` attribute for all object associations. Here is an example:

```
public class Student {

  @ManyToOne(cascade = CascadeType.PERSIST)
  @JoinColumn(name = "SCHOOL_ID")
  private School school;

}
```

You can also use a comma-separated list of cascading operations, if more than one is required. For example you can use:

```
  @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

That means both merge and persist cascading should be applied on the `Student` to `School` relationship.

An example cascading operation

Here, I'll explain different cascading values by referring to our example of an educational system application. Consider the `Student` class as shown below:

```
package com.packtpub.springhibernate.ch08;

import java.util.List;

public class Student {

  private int id;
  private String firstName;
```

```
    private String lastName;
    private Address address;
    private List courses;

    //default constructor
    public Student(){
    }

    public Student(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName= lastName ;
    }

    //setter and getter methods
    //equals() and hashCode()
  }
```

This class is associated with another persistent class, `Address`. The `Student` class also holds a `courses` property of type `java.util.List`, representing all courses that a student has passed. Each course is represented by another persistent class called `Course`.

The following code shows the `Address`:

```
  package com.packtpub.springhibernate.ch08;

  public class Address {
    private int id;
    private String street;
    private String city;
    private String zipCode;

    //default constructor
    public Address() {
    }

    public Address(String street, String city, String zipCode) {
      this.street = street;
      this.city = city;
      this.zipCode = zipCode;
    }
    //setter and getter methods
  }
```

And this is the `Course` class:

```
package com.packtpub.springhibernate.ch08;

public class Course {
  private int id;
  private String name;
  private int unit;

  //default constructor
  public Course(){
  }

  public Course(String name, int unit) {
    this.name = name;
    this.unit = unit;
  }

  //setter and getter methods
}
```

The following code shows the mapping definition for the `Address` class:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch08.Address"
table="ADDRESS">

    <id name="id" type="int" column="id">
      <generator class="increment"/>
    </id>

    <property name="street" column="STREET" type="string"/>
    <property name="city" column="CITY" type="string"/>
    <property name="zipCode" column="ZIPCODE" type="string"/>
  </class>
</hibernate-mapping>
```

Here is mapping metadata for the `Course` class:

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

<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch08.Course"
table="COURSE">
```

```
      <id name="id" type="int" column="id">
        <generator class="increment"/>
      </id>

      <property name="name" column="NAME" type="string"/>
      <property name="unit" column="UNIT" type="integer"/>
    </class>
  </hibernate-mapping>
```

And this is the `Student`'s mapping definition:

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

<hibernate-mapping>
  <class name="com.packtpub.springhibernate.ch08.Student"
table="STUDENT">

    <id name="id" type="int" column="id">
      <generator class="increment"/>
    </id>

    <property name="firstName" column="FIRST_NAME" type="string"/>
    <property name="lastName" column="LAST_NAME" type="string"/>

    <many-to-one name="address"
                 class="com.packtpub.springhibernate.ch08.Address"
                 column="ADDRESS_ID"
                 cascade="save-update"/>

    <bag name="courses" inverse="true" cascade="save-update">
      <key column="STUDENT_ID"/>
      <one-to-many class="com.packtpub.hibernate.Course"/>
    </bag>

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

Note that, the `ADDRESS_ID` and `STUDENT_ID` are extra columns in the `STUDENT` and `COURSE` tables, respectively. They are the foreign keys to the `STUDENT` table's primary key, and refer to the associated `Student` object.

The `Course` and `Address` mapping files do not have considerable information, since they consist merely of plain properties. The only thing we are interested in here is the `cascade` attribute in the mapping definition of `Student`, which currently has a `save-update` value.

When a `Student` object is persisted in the `STUDENT` table, its associated `address` is persisted in another table called `ADDRESS`. The many-to-one relation in the mapping file expresses that every `Student` object is associated with an `Address` object. For each `Student` object persisted in the `STUDENT` table, there should be at least one corresponding address object in the `ADDRESS` table.

The `<bag>` element specifies that the `Student` object is associated with a collection of the courses the student has passed.

Note that the corresponding mapping file for

```
private List courses;
```

has been represented with the `<bag>` element as follows:

```
<bag name="courses" inverse="true" cascade="save-update">
    <key column="STUDENT_ID"/>
    <one-to-many class="com.packtpub.springhibernate.ch08.Course"/>
</bag>
```

The `cascade` attribute states how changing a `Student` object affects its associated courses and address. In other words, if a `Student` object is saved, updated, or removed, this attribute states whether the object's associated `address` and `courses` should be saved, updated, or removed, as well. Let's see what each value for the `cascade` attribute means.

Using cascade="save-update"

In the previous section, relations of the `Student` with the `Course` and `Address` objects were mapped by using the `save-update` cascading operation. We expect that when a `Student` object is saved or updated, its associated objects, address, and courses are saved or updated, as well. However, when a student object is removed, we do not expect its associated objects to be erased from the database, since the cascading operation does not state delete cascading with the value `delete`. Look at the following snippet:

```
Student std = new Student("Arash", "Hoseini");
Address addr = new Address("Pastor", "Tehran", "1415833243");
Course course1 = new Course("Core Java", 2);
Course course2 = new Course("J2EE Programming", 3);
List courses = new ArrayList();
courses.add(course1);
courses.add(course2);
std.setAddress(addr);
std.setCourses(courses);
Session session = HibernateHelper.getSession();
```

```
Transaction tx = session.beginTransaction();
session.save(std);
tx.commit();
session.close();
```

After this, we expect that when `std` is saved, `course1`, `course2`, and `phone` are all saved, as well.

Using cascade="none"

If we use `cascade="none"` for each relationship in the mapping, no persistent operation on a `Student` object may be propagated to its associated objects mapped with that relationship. In other words, saving, updating, or removing a `Student` object does not lead to saving, updating, or deleting its associated object(s) that are mapped with `cascade="none"`.

Note that `none` is the default value of the `cascade` attribute.

Using cascade="delete"

`cascade="delete"` determines that, when a `Student` object is removed, all of its associated objects are erased from the database. However, saving or updating the `Student` object does not affect the associated objects in the database, even if they have changed.

Using cascade="delete-orphan"

Another cascade value is `delete-orphan`, used for objects that have a one-to-many relationship with others. This cascade specifies that, if one or more associated objects are removed from the parent, the removed object(s) are removed from the database when the parent is updated.

For example, suppose that the `Teacher` class has a property called `courses`. This property represents the list of courses that a teacher teaches. When a `Course` object is removed from the `courses` property, we want the course to be removed from the database when the `Teacher` object is updated. To do this, we can define the `Teacher` and `Course` relationship as follows:

```
<bag name="courses" inverse="true" cascade="all,delete-orphan">
  <key column="TEACHER_ID" />
  <one-to-many  class="com.packtpub.springhibernate.ch08.Course" />
</bag>
```

The `default-cascade` attribute of the `<hibernate-mapping>` element can be used to specify a general cascading operation for all of the object relationships defined inside the `<hibernate-mapping>` element. Note that each object relationship can use its own `cascade` attribute to overwrite the default value specified generally by the `default-cascade` attribute.

Lazy loading

Loading persistent objects from the database is an expensive operation, particularly when the objects are associated with many other objects. To enhance loading performance, Hibernate provides a great feature called lazy loading, which we saw briefly in Chapter 3. Let's look at it again, now that we know more about Hibernate's persistence functionality. Using lazy loading, Hibernate provides proxy objects, instead of actual persistent objects, whenever they are queried. The proxy objects act like the real object, but without the real values. This prevents Hibernate from unnecessary hitting the database, which would reduce the application's performance. The real values of the properties, object associations, and collections are loaded from the database when their accessors explicitly access them.

In the *Loading Objects* section of this chapter, you learned how to load persistent objects as proxies. Hibernate provides `get()` and `load()` methods to retrieve objects. `get()` always hits the database and fully initializes the retrieved object, but `load()` just initializes a proxy object which acts as the real object without real values. When each non-identifier property is accessed for an object that is retrieved through `load()`, the database is queried for that property. To provide this functionality, Hibernate creates a subclass of the persistent class at runtime, and overrides the accessor methods, so that when the accessor methods are called, the real values are fetched from the database.

Since proxy objects are instances of Hibernate runtime-generated classes, you can not use the loaded instance's `getClass()` method to find the persistent object's class. Instead, you can pass the object to the `getClassWithoutInitializingProxy()` method of the `org.hibernate.proxy.HibernateProxyHelper` class to get the actual persistent class.

Proxy generation for a particular persistent class can be disabled with `lazy="false"`, in the following mapping definition:

```
<class name="com.packtpub.springhibernate.ch08.Student"
       table="STUDENT"
       lazy="false">
</class>
```

In this situation, the `load()` and `get()` methods for loading the `Student` instances work in the same way.

It is also possible to change the default lazy initialization for a particular associated object or collection. By default, Hibernate loads the associated objects and collections as a proxy, even if you are loading the persistent object through the `get()` method of `Session`. To disable lazy initialization, you need to use `lazy="false"` for mapping elements inside the mapping definitions. Here is an example:

```
<class name="com.packtpub.springhibernate.ch08.Student"
table="STUDENT">
    <bag name="courses"  cascade="all" table="STUDNET_COURSE"
         lazy="false">
      <key column="STUDENT_ID"/>
      <many-to-many
                   class="com.packtpub.springhibernate.ch08.Course"
                   column="COURSE_ID"/>
    </bag>
</class>
```

JPA provides `find()` and `getReference()` methods through the `EntityManager` interface, which are somewhat equivalent to `Session`'s `get()` and `load()` methods, but not exactly the same! While `find()` always hits the database, `getReference()` always uses a proxy with just initialized identifier. The difference in JPA and Hibernate native API is in loading associated objects and collections. While in Hibernate, `get()` always hits the database and `load()` always loads the object lazily. In JPA, `find()` loads the `ManyToOne` and `OneToOne` associations not lazily, but by default. The `getReference()` method always retrieves the associated objects and collections lazily.

JPA allows the default behavior of association loading to be changed through the mapping metadata. This is done by using the `javax.persistence.FetchType` enumeration with two values: `LAZY` and `EAGER` with the same meaning as `lazy="true"` and `lazy="false"`, respectively. Here is an example:

```
public class Student {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "SCHOOL_ID")
  private School school;
}
```

This means the associate `School` object for each `Student` must be loaded lazily.

Some useful Session methods

At the beginning of this chapter, we introduced different methods exposed through the `Session` interface. Some of these methods have been discussed in this chapter, while others are fully explained in other chapters. However, there are some useful methods that fit into this chapter's scope. These methods may be used any time when working with a `Session` object or objects managed by the `Session` instance. The following table summarizes the `Session`'s methods:

Method	Description
`void persist(Object transientObject)`	Takes a transient object as an argument and associates it with the current session. The session tracks the object's changes and synchronizes the object with the database when Hibernate commits the transaction or flushes the session.
`Serializable getIdentifier(Object persistentObject)`	Returns the identifier of the persistent instance cached by the session, or throws an exception if the object is transient or is not associated with the current session. Note that the object identifier's name is determined through the mapping definition. This method allows us to access the object identifier when we do not know which of an object's properties is its identifier.
`boolean contains(Object persistentObject)`	Determines whether the persistent instance is associated with the current session.
`void evict(Object object)`	Disassociates the persistent instance with the current session. The session does not track the changes to be synchronized with the database.
`void clear()`	Clears the session of all cached instances, then cancels all pending saves, updates, and deletions that are managed by the session to be synchronized with the database.
`void flush()`	Forces the pending saves, updates, and deletions, currently managed by the session, to be synchronized with the database. This method must be called before committing the transaction and closing the session. If you do not call this method, the `Transaction`'s `commit()` method calls it internally.
`boolean isOpen()`	Determines whether the session is open or closed.
`boolean isDirty()`	Determines whether the session contains any pending saves, updates, or deletions that must be synchronized with the database.

Summary

All persisting operations in Hibernate are performed through the Session API. In this chapter, we discussed the life cycle of persistent objects inside Hibernate. The states an object goes through with respect to persistence are transient, persistent, detached, and removed. Transient objects are objects that have not been stored yet. Detached objects have already been stored but currently are not associated with a valid session instance. Persistent objects have already been stored and are currently associated with a valid session object. Removed objects are arranged to be removed from the database, but have not yet been removed. These objects are removed as soon as the session's associated transaction is committed. A session object is called valid when it has not been closed by invoking its `close()` method.

We also discussed the basic persistent operations which are exposed through some essential methods of the `Session` interface. These methods include the `load()`, `save()`, `update()`, `refresh()`, `replicate()`, and `delete()` methods.

Cascading operations was another topic introduced in this chapter. Cascading expresses what happens to associated object(s) when a persistent object is stored, updated, or removed from the database. Some valid values include `save`, `save-update`, `delete`, and `delete-orphan`.

Hibernate, like many other O/R mapping technologies, provides a lazy initialization mechanism, called lazy loading. This allows a proxy object to be used, instead of loading the real object from the database. Lazy loading enhances persistence performance, particularly when huge persistent objects exist in the database. Proxy generation can be disabled through the `lazy` attribute, used with the `class` and its nested mapping elements.

9
Querying In Hibernate

Queries are the vital part of any data-access code. Almost all applications need to investigate persistent objects and explore objects that satisfy specific criteria. SQL is a powerful language for querying persistent data and determining query criteria with `WHERE` and `JOIN` clauses. Using SQL, you can express any query restriction for the objects your application needs. As the number of query restrictions grows, the query expressions become more complex, making those query expressions hard to develop, test, and maintain. Moreover, the created query expressions may not be effective and optimized. Querying persistent objects is another area of the persistence layer where Hibernate simplifies data access development and produces effective queries.

Hibernate comes with a rich set of approaches to query persisted objects. It provides a wide range of querying methods, from native SQL to advanced Hibernate-specific approaches. In this chapter, we will discuss the querying approaches supported by Hibernate, which include the following:

- **Hibernate Query Language (HQL)**: Hibernate's HQL is a flexible, powerful SQL-like query approach. However, HQL and SQL have been designed to work with different data. While SQL queries deal directly with raw data in the database, HQL queries work with persisted objects and their properties through Hibernate. In other words, SQL queries are expressed in terms of tables, columns, views, and so on, whereas HQL expressions are states in object-oriented terms, using classes and properties of classes. Both SQL and HQL queries are specified as strings, composed of `select`, `from`, or `where` clauses.
- **Native SQL**: Hibernate lets us use native SQL. As we will see later, if you use native SQL, you will lose some Hibernate benefits, including caching, so using native SQL is not recommended except in special cases.

- **Criteria API**: Criteria API, an alternative to HQL, allows us to express queries and their restrictions programmatically, as opposed to HQL queries, which are specified as strings. Each query in the Criteria API is represented by an object of type `Criteria`. The `Criteria` objects use nested `Criterion` objects as their restrictions, which act as filters to pick up the persisted objects. The Criteria approach is also enriched by a wonderful capability called **Query By Example (QBE)**, which lets you supply example objects to indicate the objects you are looking for.

HQL queries are great in most situations, particularly when object inheritance or associations come into query restrictions. In contrast, Criteria-based queries are useful when the query criteria are changing continuously, perhaps due to user input.

> The querying language idea is not unique to Hibernate. Other ORM frameworks also provide their own query languages.

In this chapter, we will look at the different approaches provided by Hibernate, which may be used either by native Hibernate API or JPA.

The Session API and querying

In all of the approaches mentioned so far, the Session API is used to create an appropriate query object. This object is a handler for the query expression to specify restrictions or projections of the objects you want to retrieve, to pass runtime query parameters to query expressions, and to execute the query and retrieve the results. The following table provides a quick overview of the query interfaces involved in each approach, and the corresponding methods in the `org.hibernate.Session` and `javax.persistence.EntityManager` interfaces for supported query approaches:

Query Approach	Hibernate Query Interface	Example
HQL	`org.hibernate.Query`	`Query query = session.createQuery("from Student"); List results = query.list();`
Native SQL	`org.hibernate.SQLQuery`	`SQLQuery query = session.createSQLQuery( "SELECT * FROM STUDENT"); List results = query.list();`

Query Approach	Hibernate Query Interface	Example
Criteria API	`org.hibernate.Criteria`	`Criteria criteria = session.createCriteria(Student.class); List results = criteria.list();`
JPA	`javax.persistence.Query`	`Query jpaQuery =` `entityManager.createQuery("from Student");` `List results = jpaQuery.getResultList();`

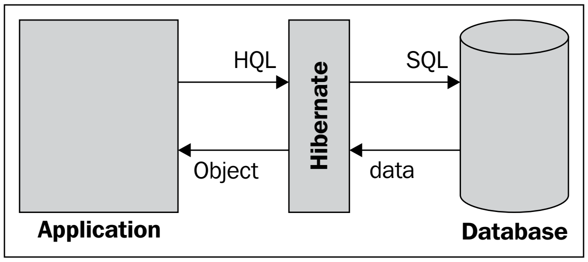
Note that in all cases, a `list()` method is always called to retrieve the results.

Let's look at these approaches in detail and see how to use each approach in practice.

HQL

As mentioned earlier, HQL is an SQL-like, Hibernate-specific, and database independent language used to query persisted objects from the back-end rational database. Since Hibernate handles many relational storage issues, and solves the relevant problems when mapping the object-oriented world to the relational world, HQL is a great substitute for SQL when you're developing the object-oriented application which interacts with a relational database through Hibernate.

HQL is not a substitute for SQL. Rather, it lets us express our queries in an object-oriented form. When an HQL expression is executed, it is first transformed into an SQL statement. The generated SQL is then executed against the database, the result is placed in persistent objects, and the objects are returned to the application. The following figure depicts this scenario:



The from clause

In its simplest form, an HQL expression consists of a `from` clause that ends with the name of a particular persistent class. Here is an example:

```
from Teacher
```

This HQL expression is used to query all instances of the `Teacher` class. Since `auto-import` is the default in the mapping files (see Chapter 5, the `<hibernate-mapping>` element), the `Teacher` class does not need to be expressed in the qualified form. In situations where the classnames are duplicated in the application, the duplicated classnames must be specified in the qualified form.

To execute an HQL expression, an `org.hibernate.Query` instance is first created by invoking the `createQuery()` method of a `Session` object. The method signature for the `createQuery()` method of `Session` is as follows:

```
public Query createQuery(String hqlQuery) throws HibernateException
```

This method's only `String` argument is the HQL expression you are executing. The returned `Query` instance provides many methods to specify restrictions on the results, execute the query, and obtain the query results. It also has other methods for setting named parameter values, adding a comment to Hibernate-generated SQL, setting a JDBC timeout and JDBC fetch sizes, and scrolling the result. We'll see more about this later in this section.

After you've created and prepared the `Query` instance, you can invoke its `list()` method to execute the query against the database and retrieve the results as a list of persistent objects. Here is an example:

```
Query query = session.createQuery("from Teacher");
java.util.List<Teacher> teachers = query.list();
```

The returned `teachers` object is the result of query execution, which contains persistent `Teacher` objects satisfying the query restrictions (there is no restriction in this query).

Like SQL, HQL is case-insensitive. Because Java is a case-sensitive language, you should express all Java-related words (such as classnames, property names, and so on) in the correct case. For example, the proper form of the HQL expression to query objects of the `Teacher` class is `from Teacher`, not `from teacher`, or any other variation of the word `Teacher`.

The as clause

The simplest form of an HQL expression, which includes only a `from` clause followed by a classname, is sufficient to query all persistent instances of a particular class. However, in many situations, the application needs a subset of objects that match a set of criteria, instead of the entire set. In such situations, you need to assign an alias to the superset of objects, so you can refer back to them in other parts of the query, such as the criteria expression. For this purpose, you can use the `as` clause in the HQL expression, as in SQL.

For instance, to load all `Teacher` objects with an age of 30, you can use the `as` clause in the HQL expression (as follows) to assign the `teacher` alias to the persistent `Teachers`:

```
from Teacher as teacher where teacher.age=30
```

As you can see, the `teacher` alias lets us refer back to the objects in the `where` clause to restrict the result (I'll cover the `where` clause later in this section as this example is fairly self-explanatory). The `as` keyword is optional. It just provides a sense of familiarity for developers who have experience with SQL. Therefore, this HQL expression would be expressed as follows:

```
from Teacher teacher where teacher.age=30
```

The query expression may include more than one class. Each can have its own alias:

```
from Teacher as teacher, Student as student
```

[It is good practice to name query aliases using an initial lowercase letter, consistent with the Java naming convention for variables.]

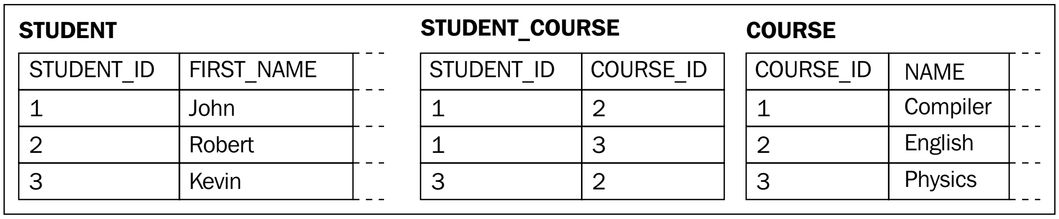
Query an object's associations

SQL uses the `JOIN` clause to join two or more tables and query them together. Its cousin, HQL, also supports the `join` clause to query instances of two or more persistent classes in a single query. You can also use the `join` clause when the object's association criteria are part of the restrictions.

Hibernate presents five different joins borrowed from ANSI SQL, as shown in the following table:

Join Type	Description
Inner Join	Used to select persistent instances from one table, with their associated instances in another table. For instance, `from Student s inner join s.courses as c` selects all students with their associated courses. If a student instance is not associated with any course, it will not be selected.
Cross Join	Used to select persistent instances stored in different tables, whether or not they are associated. For instance, to select all Students, Teachers, and Courses, you can use HQL `from Student s, Teacher t, Course c`.
Left Outer Join	Is similar to inner join, but additionally selects all instances of the left persistent class that are not associated with any instance of the right persistent class. For instance, `from Student s left outer join s.courses as c` selects all students with their associated courses, and all students that are not associated with any `Course` instance.
Right Outer Join	Selects all instances of the right persistent class with their associated instances of the left persistent class. For instance, `from Student s right outer join s.phone as p` selects all students with their associated phones, and all students that are not associated with any `phone` instance.
Full Join	Selects all instances of two associated persistent classes, whether or not the instance in the database on either side is associated with another instance on the other side. For instance, `from Student s full join s.phone as p` selects all `Student` and `Phone` instances, that is, both the associated and the unassociated instances.

To understand retrieving persistent objects using join clauses, we assume that the `STUDENT` and `COURSE` tables contain the data shown in the figure below:



STUDENT

STUDENT_ID	FIRST_NAME
1	John
2	Robert
3	Kevin

STUDENT_COURSE

STUDENT_ID	COURSE_ID
1	2
1	3
3	2

COURSE

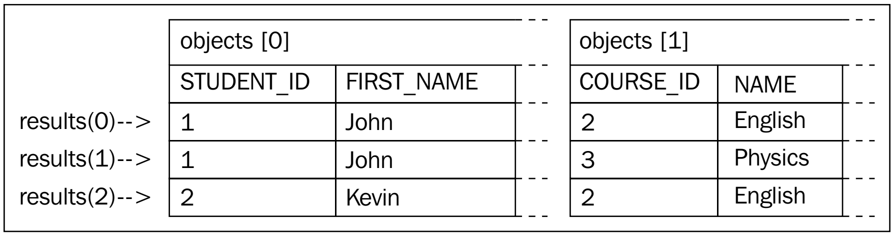
COURSE_ID	NAME
1	Compiler
2	English
3	Physics

Next, assume that we use the inner join to load the `Student` and `Course` instances as follows:

```
String joinHQL = "from Student s inner join s.courses as c";
Query joinQuery = session.createQuery(joinHQL);
```

```
List results = joinQuery.list();
for(int i=0; i<results.size(); i++){
  Object[] objects = (Object[]) results.get(i);
  Student student = (Student) objects[0];
  Course course = (Course) objects[1];
}
```

The result is a list of arrays of Objects (`Object[]`). The first element of each array is a `Student`, and the second element is a `Course` instance. The following figure shows the structure of the results:



	objects [0]		objects [1]	
	STUDENT_ID	FIRST_NAME	COURSE_ID	NAME
results(0)-->	1	John	2	English
results(1)-->	1	John	3	Physics
results(2)-->	2	Kevin	2	English

The select clause

Like SQL, HQL supports the `select` clause. This provides an efficient approach, letting you select only the desired properties of queried objects. In other words, while the `from` clause works with objects, the `select` clause works with the object's properties.

For example, to query only the first name of teachers, instead of loading the entire set of `Teacher` objects, you can use the following query expression with the `select` clause:

```
select teacher.firstName from Teacher teacher
```

If you want to select more than one property, specify the properties names as a comma-separated list. For example, to load `firstName` and `lastName` of `Teacher` objects, change the HQL query as follows:

```
select teacher.firstName, teacher.lastName from Teacher as teacher
```

The result of executing this HQL is a list of arrays. Each array has two members, maintaining, respectively, the `firstName` and the `lastName` of a selected object.

The generated SQL will only query the `FIRST_NAME` and `LAST_NAME` columns, instead of the entire `TEACHER` table. Retrieving the required properties, instead of entire objects, has a performance effect (reducing network traffic, decreasing memory usage, and querying fewer columns in the database) of speeding up execution.

HQL's aggregate functions

HQL provides aggregate functions, so we can query information related to a group of objects. The syntax of these functions is the same as the syntax in SQL, and they also work in the same way. The only difference is that HQL's aggregate functions work with objects and the object's properties, instead of tables and the table's columns.

Aggregate functions supported in HQL are `avg()`, `min()`, `max()`, `sum()`, and `count()`. Each of these methods takes a string as an argument, which is the property name on which the function works. For example, the following HQL query returns the average age of the students:

```
select avg(student.age) from Student student
```

When you execute this query, you get a `List` object which contains a single `Integer` value. This is the case with all aggregate functions.

The `count()` function takes a property name as argument and returns the number of times that property appears in the result set. This function has two versions:

```
count(*)
count(aliasName.propertyName)
```

These count the number of all objects and the number of objects with the property `propertyName` in the result set, respectively.

You can use the `distinct` and `all` keywords with the `count()` function with the same semantics as in the `count()` method in SQL. The `distinct` keyword tells `count()` to only count the distinct and different values in the result set. For example, the following HQL returns only the number of different ages for students:

```
select count(distinct student.age) from Student student
```

If your query expression includes more than one aggregate method, the returned `List` object contains a number of objects. Each object corresponds to an aggregate method. For instance, the following snippet retrieves the minimum and maximum ages of students:

```
String queryHQL = "select min(std.age), max(std.age) from Student
std";
Query query = session.createQuery(queryHQL);
List bounds = query.list();
Double min = (Double)bounds.get(0);
Double max = (Double)bounds.get(1);
```

The where clause

HQL is as flexible and efficient as SQL, because it provides the `where` clause to restrict a query's result. In HQL expressions, the `where` clause lets us express constraints on the property values of objects returned by the query, and restrict the query's result. Additionally, as you will see in this chapter, you may use the `where` clause in bulk-update or bulk-delete query expressions to restrict the objects being updated or deleted, respectively. Usually, you refer to the selected objects by an alias. If no alias exists, then you may refer to properties by name.

For example, the following query expressions are the same, returning all instances of `Student` with an age greater than 25:

```
from Student student where student.age>25
from Student where age>25
```

As you can see in these HQL statements, the `where` clause may include primitive values, such as numbers and Booleans, when a condition is specified. You can also use string values for comparisons in the `where` clause, but these must be inside single quotes. Additionally, it is possible to compare properties of objects to other properties.

To express restrictions in HQL and JPA-QL expressions, and define comparison restriction between properties, Hibernate and JPA support a wide range of rich and powerful comparison operators and functions, including these:

Operators/Functions	**Names**
Mathematical operators	`+, -, *, /`
Comparison operators	`=, >=, <=, <>, !=, like, not like`
Logical operators	`and, or, not`
Grouping operators	`in, not in, between, is null, is not null, is empty, is not empty, member of, not member of`
Scalar database-supported functions	`sign(), trunc(), rtrim(), sin()`
Collection-valued taken functions (HQL only)	`minelement(), maxelement(), minindex(), maxindex()`
Time and date functions (HQL only)	`current_date(), current_time(), current_timestamp(), second(), minute(), hour(), day(), month(), year()`
JPA standardized functions	`substring(),trim(),lower(),upper(), length(),locate(),abs(),sqrt(),mod(), sort(), concat(), locate(), size()`

Here is an example of using the `not like` and `>` operators to query all students who are older than 18, and whose first names do not start with `"Jo"`:

```
from Student where age>18 and firstName not like 'Jo%'
```

The `like` and `not like` operators come with the wildcard symbols `%` and _. The `like` and `not like` operators allow us to get matches that are, respectively, similar to or different from a phrase we supply before or after a symbol.

Refer to the Hibernate documentation at `http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/#queryhql-expressions` for more details about these functions.

Positional versus named parameters

Like JDBC, Hibernate lets us use positional parameters to construct queries dynamically at runtime. These parameters are specified with a `?` in query expressions, and are addressed with their position indexes to be filled with the desired values at runtime. The `Query` interface provides methods similar to the `java.sql.PreparedStatement` methods for binding arguments of Java types. These methods include `setInteger()`, `setString()`, `setDate()`, `setLocale()`, and `setTimestamp()`. Additionally, the `Query` interface provides the general `setParameter()` method for binding parameters with any type. When you use `setParameter()`, Hibernate automatically detects the parameter type and binds the parameter with the proper type. The following snippet uses parameters to construct an HQL expression:

```
String hql = "from Student where firstName = ? and lastName=?";
Org.hibernate.Query query = session.createQuery(hql);
query.setString(0, "Arash");
query.setString(1, "Moradi");
List list = query.list();
```

We may use positional parameters with JPA as follows:

```
String jpaQL = "from Student std"+
               " where std.firstName=?1 and std.lastName=?2";
javax.persistence.Query query = entityManager.createQuery(jpaQL);
query.setParameter(1, "Arash");
query.setParameter(2, "Moradi");
List list = query.getResultList();
```

Hibernate index parameters start at `0` (`query.setString( 0, "Arash");`), whereas JPA's start at `1` (`query.setParameter( 1, "Arash");`).

Although this method of constructing a query dynamically is flexible enough, the following issue still needs to be addressed. If the position of parameter changes, the code that next assigns a value must change, as well. For example, suppose we swap `firstName` and `lastName` in the preceding HQL. In that case, we must change the `query.setString()` or `query.setParameter()` statements to assign the parameter's values properly.

To deal with this problem, Hibernate offers a more flexible approach, which is **named parameters**. Named parameters let you refer to each parameter with a name, instead of by index. Each parameter name is specified with an arbitrary name that begins with the prefix in query expressions. Then, parameters are referred to by their names, instead of their positions.

The following code shows changes to the previous snippet, which now uses named parameters instead of JDBC-style positional parameters:

```
String hql = "from Student where firstName=:firstName and
lastName=:lastName";
Query query = session.createQuery(hql);
query.setString("firstName", "Arash");
query.setString("lastName", "Moradi");
List list = query.list();
```

As you can see, named parameters are assigned on the basis of their names, and not on their positions in a query. Using named parameters improves application maintenance, because adding or removing named parameters may effect fewer changes to the code than JDBC-like positional parameters.

HQL also permits us to use object parameters that are filled with persistent objects, rather than basic Java types. You accomplish this by using the `setEntity()` method of a `Query` object, which sets the persistent object as a parameter value. The following code queries `Student` objects that have a particular phone number, represented by a `Phone` object:

```
String phoneHQL = "from Phone where number='09723455667'";
Query  phoneQuery = session.createQuery(phoneHQL);
Phone phone = (Phone)phoneQuery.list().get(0);
…
String hql = "from Student as student where student.phone=:phone";
Query query = session.createQuery(hql);
query.setEntity("phone", phone);
List students = query.list();
```

Note that `setEntity()` implicitly expands to the identifier of the phone object. This means we are looking for all students associated with a phone object that has a particular identifier value.

[You can still use `setParameter()` instead of the `setEntity()` method to set a persistent object. In the snippet shown here, simply replace `query.setEntity("phone", phone);` with `query.setParameter("phone", phone);`.]

The order by and group by clauses

The returned objects of an HQL query may be ordered by any property of the selected objects. To do this, use the `order by` clause in the HQL expressions, either ascending (`asc`) or descending (`desc`). For example, the following HQL queries `Student` objects older than 20, and sorts them in ascending order by the `firstName` property:

```
from Student s order by s.firstName asc
```

The result can be ordered on more than one property. In this case, you need to add additional properties to the end of the `order by` clause as a comma-separated list. For instance, you can do this:

```
from Student s order by s.firstName asc, s.age desc
```

This selects all `Student` objects and first orders them in ascending order by `firstName`, and then in descending order by the `age` property.

HQL also lets us use the `group by` clause to group the returned aggregate values by any property of the selected objects:

```
select student.age, avg(student.age), count(student) from Student
student
group by student.age
```

If the underlying database supports it, you can use aggregate functions with `having` or `order by` clauses.

[Neither the `group by` clause nor the `order by` clause may contain arithmetic expressions.]

Bulk updates and bulk deletes with HQL

Using HQL, you can also update or delete a set of persisted objects. The syntax of HQL for update and delete operations is similar to SQL. HQL uses the `update` and `delete` keywords with the `set` and `where` clause to update or delete rows that satisfy a set of criteria. For instance, to modify the name of a course, you can use the following HQL expression:

```
String updateHql = "update Course set name = :newName where name = :
name";
```

The usage of the `set` keyword in HQL is similar to the usage of `set` in SQL. It determines new values for object's properties. You can optionally use the `where` clause to restrict the object being updated.

Similarly, you can use the `delete` query with an optional `where` clause to remove a set of objects from the database. The following HQL expression is an example of a `delete` query to remove a course with a particular name from the database:

```
String deleteHql = "delete from Course where name = :name";
```

Note that both of these examples use named parameters to be set with runtime values before query execution.

Other steps of query execution are similar to the execution of an ordinary HQL query. You need to obtain a `Query` instance associated with the query expression, and then set the parameter values to the `Query` object before execution. Finally, you must invoke the `executeUpdate()` method of the `Query` instance, instead of the `list()` method that is used for normal HQL expressions. Here is an example:

```
String updateHql = "update Course set name = :newName where name = :
name";
Query query = session.createQuery(updateHql);
query.setString("name","Computer Programming");
query.setString("newName","Java Language Programming");
int rowsAffected = query.executeUpdate();
```

The `executeUpdate()` method returns an `int` value, representing the number of rows affected by execution of the `update` or `delete` query.

> Since bulk-update and bulk-delete queries do not need to reload the object before `update` or `delete` execution, they are preferred to the Session's `update()` or `delete()` methods, particularly when there are many objects to update or delete.

Queries in native SQL

In addition to HQL, Hibernate lets us load persistent objects with native SQL queries. Although using native SQL doesn't seem useful, you may prefer to use native SQL in some situations. For example, you may need to utilize database-specific features which are not supported by HQL. Another situation is when you want to call a stored procedure. Moreover, you may use native SQL when you are migrating from a legacy JDBC-based data tier to Hibernate and need to use SQL in the application. In these cases, either you must use native SQL, or you may prefer to continue working with SQL.

Hibernate provides the `org.hibernate.SQLQuery` interface to work with SQL queries. To use it, pass the SQL query expression as a `String` argument to the `createSQLQuery()` method of the `Session` instance and obtain an `SQLQuery` instance. After all, the `list()` method of `SQLQuery` executes the query and fetches the result. The following snippet shows how to create an `SQLQuery` instance to execute an SQL query and execute it:

```
String sql = "SELECT * FROM STUDENT";
SQLQuery query = session.createSQLQuery(sql);
List results = query.list();
```

In its basic form, an SQL query returns a list of scalars. In this example, the `results` are a list of `Object` arrays (`Object[]`) that is, each array holds a row of the `STUDENT` table. Hibernate uses `ResultSetMetadata` to deduce the actual order and types of the returned scalar values. You can discard undesired columns, by invoking the `addScalar()` method as follows:

```
String sql = "SELECT * FROM STUDENT";
SQLQuery query = session.createSQLQuery(sql);
query.addScalar("ID", Hibernate.LONG);
query.addScalar("FIRST_NAME", Hibernate.STRING);
query.addScalar("BIRTHDAY", Hibernate.DATE);
List results = query.list();
Iterator itr = results.iterator();
while (itr.hasNext()) {
  Object[] row = (Object[]) itr.next();
  for (int i = 0; i < row.length; i++) {
    Object column = row[i];
    System.out.println(column);
  }
}
```

Although the query uses `*`, and could return more than the three listed columns, Hibernate will not use `ResultSetMetdata`. Instead, it explicitly gets the `ID`, `FIRST_NAME`, and `BIRTHDAY` columns as a `Long`, a `String`, and a `DATE` from the returned result set. Note that Hibernate still uses `"SELECT * FROM STUDENT"` to query the database. Hibernate then selects the specified columns from the result of the query execution and puts each selected row in the returned list as an `Object` array.

In addition to the `addScalar()` method, `SQLQuery` provides another method, which is `addEntity()`. With this method, you can tell Hibernate to select only those columns from the returned `resultset` that are required to populate a persistent class. Look at the following example:

```
String sql = "SELECT {student.*} FROM Student student";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity("student", Student.class);
List results = query.list();
```

`{student.*}` determines all columns associated with the alias `student`, that is, the `STUDENT` table, to be selected from the database. After the query executes, the `results` object contains instances of the `Student` class. Notice that this query is not executed in the target database as is. Hibernate transforms it to the appropriate SQL of the target database.

Look at another example:

```
String query = "select {student.*} from STUDENT student" +
     " join ADDRESS address on student.ADDRESS_ID = address.ID" +
               " where address.STREET = :street";
SQLQuery sqlQuery = session.createSQLQuery(query);
sqlQuery.addEntity("student", Student.class);
sqlQuery.setParameter("street", "rockway");
```

This example queries students who live on the same street. As you can see, the query selects the `STUDENT` table with the alias `student`, and joins it to the `ADDRESS` table with the alias `address`. The query uses these aliases to refer to the tables later, in the query restrictions. The `addEntity()` method here indicates that Hibernate should only return the columns required to populate the `Student` class.

Note that this example also shows how to use a named parameter with an SQL query. With `SQLQuery`, you still have the option to use the general `setParameter()` method for setting parameters with any type, or proprietary methods for each Java type such as `setString()`, `setDate()`, `setTimestamp()`, and so on.

In addition to the `addScalar()` and `addEntity()` methods used for selecting the required columns, `SQLQuery` provides the `addJoin()` method to select columns of tables joined to a table by the query. Look at the following example:

```
String sql = "select {student.*}, {phone.*} from STUDENT student" +
             " join PHONE phone on student.ID = phone.ID" +
             " where LAST_NAME=:lastName";
SQLQuery sqlQuery = session.createSQLQuery(sql);
sqlQuery.addEntity("student", Student.class);
sqlQuery.addJoin("phone", "student.phone");
sqlQuery.setParameter("lastName", "Romeny");
List results = sqlQuery.list();
```

This example selects a row of `STUDENT`, and a row of `PHONE`, which are associated with each other through their `ID` column for a particular student with last name `Romeny`.

This example selects columns from two tables, and we can select the columns obviously with two persistent classes. The `addEntity()` method selects columns of the first table that populate the `Student` class. The `addJoin()` method tells Hibernate to select columns that populate the associated `phone` of the `Student` instance.

Named SQL and HQL queries

Named queries refer to SQL or HQL queries that are defined declaratively in the mapping files, but are called programmatically in the Java code. The syntax of named SQL and HQL queries is the same as the syntax of ordinary SQL and HQL queries, which are defined in the Java code. They could include either JDBC-like positional parameters (`?`) or named parameters.

Named queries have many benefits. At first, their syntax is checked at deploy time, and not during program execution. You can have several different named queries and easily switch from one to another during development. Additionally, because they are kept separated from Java code, they are easy to maintain. During application, development or deployment can easily change, without requiring any change to the application code.

The `<query>` and `<sql-query>` elements are used, respectively, to declare HQL and SQL queries in the mapping files. Both use a `name` attribute, which assigns a name to the query to use for calling the query. Note that the name used for each query must be globally unique in the application.

The following code shows a definition of both a named HQL query, and an SQL query, in the mapping file for the `Student` class:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.packtpub.springhibernate.Student">
    <!—properties mappings omitted -->

    <query name="hqlOlderQuery">
      <![CDATA[select std.age from Student std where std.age > :age]]>
    </query>

    <sql-query name="sqlOlderQuery">
      <return-scalar column="age" type="double"/>
      <![CDATA[select std.age from Student as std where std.age>:
       age]]>
    </sql-query>
  </class>
</hibernate-mapping>
```

As you can see, the `<sql-query>` element comes with a nested element, called `<return-scalar>`. This element tells Hibernate the type of data it expects to receive when the SQL query executes. This element is not used with HQL query definitions, since Hibernate parses the HQL queries and discovers the query execution's result type.

The `<return-scalar>` element comes with two attributes, `column` and `type`, which specify the name of the selected column and its SQL type, respectively.

[You should define the SQL or HQL queries inside a `CDATA` section if the query contains XML markup. Doing so maintains the integrity of the XML, and ensures that the XML parser can always process the query during development and maintenance.]

It is also possible to define named queries through JPA annotations. For this purpose, JPA provides `@NamedQuery` and `@NamedNativeQuery` annotations, located in the `javax.persistence` package, to define named queries inside the Java code. The following code snippet shows an example of using the `@NameQuery` annotation to define a named query:

```
package com.packtpub.springhibernate.ch09;

import ...;

@NamedQueries({
    @NamedQuery(
        name = "hqlOlderQuery",
        query = "select std.age from Student std where std.age>:age"
    ),
    ...
})
@Entity
public class Student { ... }
```

Every named query is called by its name, defined by the `name` attribute for that query in the mapping file. To execute a named query, first call the `getNamedQuery()` of the `Session` instance with the specified named query and obtain a `Query` object. The `Query` instance can then be used as usual to execute the query and fetch the result. For example, to execute the previous queries, use the following snippets:

```
//executing the name HQL query
Query hqlQuery = session.getNamedQuery("hqlOlderQuery");
hqlQuery.setDouble("age", 20);
List r1 = hqlQuery.list();

//executing the name SQL query
Query sqlQuery = session.getNamedQuery("sqlOlderQuery");
sqlQuery.setDouble("age", 20);
List r2 = sqlQuery.list();
```

To execute a JPA named query, we need to use code similar to the above, but with the `java.persistence.EntityManager` and `javax.persistence.Query` interfaces, as follows:

```
Query query = entityManager.createNamedQuery("hqlOlderQuery");
query.setParameter("age", 20);
List result = query.getResultList();
```

Using the Criteria API

The Criteria API is another approach to querying persistent objects. This API lets us build query expressions programmatically through Java objects. It provides compile-time syntax checking, rather than runtime processing (which is provided by HQL).

The Criteria API provides a unique **Query By Example (QBE)** functionality that is not provided by other query approaches. Using QBE, you can create an instance of the queried class, and set the instance's properties with the values that you want the result to have. The result of the query is all persistent objects that match the instance.

In general, the Criteria approach is less flexible and less powerful than HQL. Criteria queries are less readable and harder to understand than HQL queries. Typically, creating the right query with Criteria requires more effort. More importantly, Criteria queries cannot be declared in the mapping metadata, as HQL queries can. This may increase maintenance problems.

Although HQL and Criteria queries are interchangeable, in some situations one is more fitting than the other. For instance, when query restrictions are constructed from user input, or when they change during application execution, it's better to use the Criteria API to create a query expression. HQL expressions are simple string literals whose syntax is checked at runtime, whereas the Criteria queries are Java objects that naturally are checked at compile time. Therefore, manipulating the string query expressions which are constructed from the user input, which might fail in some conditions, would be painful. Using Criteria, you can create nested, structured query expressions based on the data input by a user, which is a more convenient approach.

Using a simple Criteria

Using the Criteria API involves the `org.hibernate.Criteria` and `org.hibernate.criterion.Criterion` interfaces and the `org.hibernate.criterion.Restrictions` class. Each query is presented with a `Criteria` object, and any restriction on the query is indicated with an object of `Criterion`. Because `Criteria` and `Criterion` are interfaces and cannot be instantiated, session objects and the `Restrictions` class serve as factories for them.

To create a `Criteria` query, call the `createCriteria()` method of the `Session` instance. This method takes a `java.lang.Class` as an argument, which determines the persistent class you are going to query, and returns an object of type `Criteria`. Here is an example of querying `Teacher` instances persistent in the database:

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

The `Criteria` interface is analogous to the `Query` interface in HQL. Both interfaces have a `list()` method which returns the query result as an object of type `java.util.List`. You can then iterate over the result to retrieve each restrictions-matched object:

```
List results = criteria.list();
Iterator itr = results.iterator();
while (itr.hasNext()) {
  Teacher teacher = (Teacher) itr.next();
}
```

However, in most situations, you need to retrieve a subset of objects that satisfy a set of restrictions, instead of all persistent objects of one type. The Criteria API lets you express restrictions with nested, structured `Criterion` objects. The `Criterion` instances are added in a tree-like structure to express query restrictions.

To create a `Criterion` instance, call the appropriate factory method of the `Restrictions` class. The `Restrictions` class provides many static methods to create different `Criterion` objects with different restrictions. Each `Criterion` object represents a specific restriction that corresponds to the factory method that is called for creating that object. For instance, the `eq()` method is used to create an equal restriction. Here is an example:

```
Criterion criterion = Restrictions.eq("firstName","Edward");
```

In this example, the `criterion` object indicates that the `firstName` property must have `"Edward"` as its value.

Next, associate the `Criterion` object with the `Criteria` instance through the `Criteria` object's `add()` method:

```
Criteria criteria = session.createCriteria(Teacher.class);
Criterion criterion = Restrictions.eq("firstName","Edward");
criteria.add(criterion);
List results = criteria.list();
```

Looking at the Restrictions class's factory methods

As mentioned earlier, the `Restrictions` class provides many static methods which express a wide range of restrictions for queries. Let's explore these methods and look at examples to see how to use them.

Equality restrictions

We use the `eq()` and `ne()` methods to restrict a particular property of objects to have or not to have a particular value. Here is the syntax of each:

```
public static SimpleExpression eq(String propertyName, Object
propertyValue)
public static SimpleExpression ne(String propertyName, Object
propertyValue)
```

Both methods return an object of type `SimpleExpression`, which is an implemented class of `Criterion`. The first method constructs an equality restriction, which determines that the `propertyName` of the desired objects must have the value `propertyValue`.

For example, the following snippet obtains all `Student` objects whose `firstName` property has the value `"Edward"`:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.eq("firstName","Edward"));
List results = criteria.list();
```

In contrast to `eq()`, the `ne()` method specifies inequality. It determines that the `propertyName` of the desired objects must *not* have the `propertyValue` value. The following snippet retrieves all `Student` objects whose `firstName` property value differs from `"Edward"`:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.ne("firstName","Edward"));
List results = criteria.list();
```

Note that these methods cannot be used to retrieve objects with a property of null or not-null values. As you will see, the `isNull()`or `isNotNull()` methods are used instead. If the value to which the restrictions apply is calculated dynamically by a process in the application, or is entered by the user, you must control the values and prevent nulls in advance.

In addition to these methods, the `Restrictions` class provides the `allEq()` method to select objects with several properties that have particular values, so it's like an AND operation (discussed later in this chapter). This method's syntax is as follows:

```
public static Criterion allEq(Map properties)
```

This method takes a parameter of type `java.util.Map`. The keys in the map specify the property names we want to constrain, and the values in the map indicate the corresponding values for those properties. Here is an example of this method:

```
Map map = new HashMap();
map.put("firstName", "John");
map.put("lastName", "Robinston");
map.put("age", new Double(19));
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.allEq(map));
List results = criteria.list();
```

The `results` object contains all `Student` objects with a `firstName` of `"John"`, `lastName` of `"Robinston"`, and `age` of `19`.

Null and empty restrictions

The `isNull()` and `isNotNull()` methods select persistent objects that have a particular property that is null or is not null, respectively:

```
public static Criterion isNull(String property)
public static Criterion isNotNull(String property)
```

For instance, the following snippet selects `Student` objects whose middle names are null:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.isNull("middleName"));
List results = criteria.list();
```

Besides the null restrictions, two other methods provided by the `Restrictions` class constrain an object property to be empty or not empty:

```
public static Criterion isEmpty(String property)
public static Criterion isNotEmpty(String property)
```

Likeness restrictions

We can retrieve all objects with a property that matches a given pattern. The `Restrictions` class provides two methods for this purpose: `like()` and `ilike()`. The simplest forms of these methods are as follows:

```
public static SimpleExpression like(String propertyName, Object value)
public static SimpleExpression ilike(String propertyName, Object
value)
```

Both `like()` and `ilike()` methods construct a likeness restriction that is transformed into an SQL `LIKE` clause. The `like()` method cares about case, while `ilike()` ignores case. The following snippet fetches all `Student` objects that have a `firstName` starting with `"Jo"` (case doesn't matter):

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.ilike("firstName","Jo%"));
List results = criteria.list();
```

Note that we used the `%` character, which has the same meaning as in the SQL `LIKE` clause, to match patterns in the string.

Two other forms of these methods are as follows:

```
public static SimpleExpression like(String propertyName,
                                    String value,
                                    MatchMode matchMode)
public static SimpleExpression ilike(String propertyName,
                                     String value,
                                     MatchMode matchMode)
```

These take an extra parameter of type `org.hibernate.criterion.MatchMode`, which lets us specify how the persisted data must match the given pattern. The `MatchMode` class is a type-safe enumeration with four static members, allowing specification of four different matches:

- `ANYWHERE`: Any part of the selected data may match the given pattern.
- `END`: The end of the selected data must match the given pattern.
- `EXACT`: The selected data must exactly match the given pattern.
- `START`: The start of the selected data must match the given pattern.

The following snippet gets every `Student` object that has a `firstName` which includes `"bob"`, in which the matching operation does not care about case:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.ilike("firstName","bob",
                                MatchMode.ANYWHERE));
List results = criteria.list();
```

Comparison restrictions

The `Restrictions` class provides several methods for constructing comparison expressions. These methods select all objects that have a particular property that is greater than, greater than or equal to, less than, less than or equal to, or between the specified values, respectively:

```
public static SimpleExpression gt(String propertyName, Object
propveryValue)
public static SimpleExpression ge(String propertyName, Object
propertyValue)
public static SimpleExpression lt(String propertyName, Object
propertyValue)
public static SimpleExpression le(String propertyName, Object
propertyValue)
public static Criterion between(String propertyName, Object lowValue,
ObjecthighValue)
```

The following snippet shows how `ge()` can select all students whose ages are greater than or equal to 25:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.ge("age",new Integer(25)));
List results = criteria.list();
```

In addition to these methods, selecting objects by comparing their property values with a specified value, the `Restrictions` class provides other methods that let us select objects based on comparing of two properties values. The following code shows these methods:

```
public static PropertyExpression eqProperty(String property1, Object
property2)
public static PropertyExpression leProperty(String property1, Object
property2)
public static PropertyExpression ltProperty(String property1, Object
property2)
```

These methods indicate, respectively, that two named properties must have the same value, that the first named property is less than or equal to the second, and that the first named property is less than the second.

For instance, the following snippet selects all customers whose deposits are less than their credit:

```
Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Restrictions.lt("deposit","credit"));
List results = criteria.list();
```

Logical restrictions

You can combine `Criterion` objects to build a nested, structured query expression. This means that a `Criteria` object may be constructed of more than one `Criterion`. This is easy to demonstrate:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.eq("firstName","John"));
criteria.add(Restrictions.ge("age",new Double(19)));
List results = criteria.list();
```

The `results` object contains all `Student` objects with the `firstName "John"`, and an `age` greater than or equal to `19`.

When `Criterion` objects are added to the `Criteria` by the `add()` method, the operation is interpreted as an AND.

The `Restrictions` class also provides the `or()` and `and()` methods. The `or()` method builds a compound `Criterion` that satisfies one or both `Criterion` restrictions. In contrast, the `and()` method builds a compound `Criterion` that satisfies both `Criterion` restrictions. These methods are as follows:

```
public static LogicalExpression and(Criterion criterion1, Criterion
criterion2)
public static LogicalExpression or(Criterion criterion1, Criterion
criterion2)
```

The following snippet shows how to use `or()`, in which the selected objects must either have `"John"` as their `firstName`, or be equal to or older than `19`:

```
Criteria criteria = session.createCriteria(Student.class);
Criterion criterion1 = Restrictions.eq("firstName","John");
Criterion criterion2 = Restrictions.ge("age",new Double(19));
criteria.add(Restrictions.or(criterion1, criterion2));
List results = criteria.list();
```

In addition to AND and OR restrictions, the `Restrictions` class allows us to build a negated `Criterion` through the `not()` method:

```
public static Criterion not(Criterion criterion)
```

For example, to load all students whose ages are not greater than and equal to 19 (that is, less than 19), the `not()` method can be used as follows:

```
Criteria criteria = session.createCriteria(Student.class);
Criterion criterion = Restrictions.ge("age",new Double(19));
criteria.add(Restrictions.not(criterion));
List results = criteria.list();
```

Size restrictions

You may also use the `sizeXx()` methods to construct constraints using size comparisons. These methods include the following:

```
public static Criterion sizeEq(String s, int i)
public static Criterion sizeNe(String s, int i)
public static Criterion sizeGt(String s, int i)
public static Criterion sizeLt(String s, int i)
public static Criterion sizeGe(String s, int i)
public static Criterion sizeLe(String s, int i)
```

The description of each size restrictions is as follows:

- `sizeEq`: size equality
- `sizeNe`: size not-equality
- `sizeGt`: size greater than
- `sizeLt`: size less than
- `sizeGe`: size greater than or equal to
- `sizeLe`: size less than or equal to

Disjunctions and conjunctions

If you want to create an OR expression with many `Criterion` objects, you can use an instance of `org.hibernate.criterion.Disjunction`. Using this object is equivalent to, but more convenient than, using several OR restrictions. To obtain a `Disjunction` object, call the `disjunction()` method:

```
public static Disjunction disjunction()
```

If you want to create an AND expression with many criterion objects, you can use an object of `org.hibernate.criterion.Conjunction`. The `conjunction()` method returns a `Conjunction` object as follows:

```
public static Conjunction conjunction()
```

The `Disjunction` class and the `Conjunction` class provide `add()` methods to apply an OR or an AND, respectively, between the criteria.

The following code uses a conjunction object to construct an AND expression:

```
Criteria criteria = session.createCriteria(Student.class);
Criterion age = Restrictions.gt("age",new Double(25.0));
Criterion firstName = Restrictions.like("firstName","John%");
Conjunction conjunction = Restrictions.conjunction();
conjunction.add(age);
conjunction.add(firstName);
criteria.add(conjunction);
List results = criteria.list();
```

SQL restrictions

SQL restrictions are useful when you need to use an SQL clause that is not supported by Hibernate through the Criteria query API. Different methods in the `Restrictions` class construct this type of restriction:

```
public static Criterion sqlRestriction(String)
public static Criterion sqlRestriction(String, Object, Type)
public static Criterion sqlRestriction(String, Object[], Type[])
```

The first form of the method only takes a `String` parameter, indicating the SQL restriction. The other forms are useful when the SQL restriction includes JDBC parameters (`?`). The extra arguments determine the value(s) and type(s) of the parameters.

The following snippet shows how to use this method to create an SQL restriction:

```
Criteria criteria = session.createCriteria(Student.class);
criteria.add(Restrictions.sqlRestriction("{alias}.FIRST_NAME like
'John%'"));
List results = criteria.list();
```

`{alias}` and `FIRST_NAME` refer, respectively, to the table and a column in that table that store the `Student` objects, and the `firstName` property of the `Student` objects.

Query By Example (QBE)

Hibernate lets us create an example instance of a persistent class and set properties of the instance with the desired values, and then use the sample instance to retrieve all persistent instances in the database that match the properties of the example instance. This functionality, called *Query By Example (QBE)*, provides much flexibility in development, and produces cleaner, neater, and more testable code.

QBE is provided by `org.hibernate.criterion.Example` as a subclass of `Criterion`. To use QBE, create an example instance of the persistent class and set the desired values. Then, use the `create()` method of `Example` to create an `Example` instance. Use the `Example` instance in your `Criteria` construction to find the matches. Look at this example:

```
Criteria criteria = session.createCriteria(Student.class);
Student student= new Student();
student.setFirstName("John");
Example example = Example.create(student);
criteria.add(example);
List results = criteria.list();
```

The `results` list contains all persistent Students whose `firstName` is John. Note that Hibernate only takes care of the `firstName` field, ignoring others that were not set. However, it is possible to configure the `Example` instance to indicate how Hibernate should examine the property values to create a query. The methods of the `Example` instance are used for this purpose. The following table summarizes some of these methods:

Method	**Description**
`static Example create(java.lang.Object persistentObject)`	Creates a new instance of the `Example` class, which identifies all persistent instances in the database that match non-null properties of the `persistentObject`.
`Example enableLike()`	Enables Hibernate to use the `like` operator when it finds matches for `String` properties.
`Example enableLike(MatchMode matchMode)`	Indicates that Hibernate must use the `like` operator to find matches that occur in particular area: `MatchMode.START`, `MatchMode.END`, `MatchMode.ANYWHERE`, and `MatchMode.EXACT`. (See the *Likeness Restrictions* section for more information.)
`Example excludeNone()`	Determines that properties with zero or null values should also be used when matches are found.
`Example excludeProperty(String propertyName)`	Excludes a particular named property when finding matches.
`Example excludeZeros()`	Excludes properties with zero values when finding matches.
`Example ignoreCase()`	Determines that Hibernate must ignore the case of `String` properties when finding matches.

The following code illustrates these methods:

```
Criteria criteria = session.createCriteria(Student.class);
Student student = new Student();
student.setFirstName("Bob");
Example example = Example.create(student);
example.ignoreCase();
example.enableLike(MatchMode.ANYWHERE);
criteria.add(example);
List results = criteria.list();
```

This example retrieves all `Student` instances whose `firstName` includes the string `"bob"`, regardless of case.

Paging the query result

Often, a huge number of persistent objects satisfies your query restrictions. This means that many rows of the database are selected, and a large number of objects are loaded in memory when the query is executed. Two problems arise:

- Loading a large set of data in memory affects the application's performance.
- It may not be possible to show the entire set of data in the user interface. Even if this is possible, application users do not want to view the entire result in a single page.

In these situations, you can use a pagination technique to load and show only a subset of data, rather than all of it. The user interface that shows the data allows the user to navigate to the next and previous subsets of data.

In Hibernate, the `Query` and `Criteria` interfaces both provide the `setFirstResult()` and `setMaxResults()` methods to implement the pagination technique. The `setFirstResult()` method allows us to specify the starting point of the bounds (that is, the first row in the result set to load) using a zero-based index. The `setMaxResults()` method determines the maximum number of objects that you expect to load. Note that the pagination technique applies to `Query` and `Criteria` objects, and does not affect HQL query expressions or the `Criteria` construction process.

Here is an example of using Hibernate pagination when loading objects of type `Teacher`:

```
Query query = session.createQuery("from Teacher");
query.setFirstResult(10);
query.setMaxResults(20);
List teachers = query.list();
```

This snippet retrieves 20 `Teacher` instances, starting from row 11 and finishing with row 30. You can use the returned value of the `list()` method normally, as when the pagination technique is not used. To load the next page of the query result, set up the `Query` instance with the first result 30, and the max result 20, and then execute query again.

In addition to the `setFirstResult()` and `setMaxResults()` methods, the `Query` and `Criteria` interfaces both provide the `uniqueResult()` method, which is useful for retrieving the query result when the result set contains only one object:

```
String hql = "from Teacher where phoneNumber = '+98-912-3456789'";
Query query = session.createQuery(hql);
query.setMaxResults(1);
Teacher teacher = (Teacher)query.uniqueResult();
```

If the result set contains more than one object, calling this method will throw `org.hibernate.NonUniqueResultException`. Note that we used `query.setMaxResults(1)` to retrieve only one object from the database, even if more than one object exists, so we prevented a possible exception throwing.

The following code shows how pagination is applied to a `Criteria` object:

```
Criteria criteria = session.createCriteria(Teacher.class);
Criterion criterion = Restrictions.eq("phoneNumber","+98-912-
3456789");
criteria.add(criterion);
criteria.setFirstResult(1);
criteria.setMaxResults(10);
List results = criteria.list();
```

The pagination is applicable in JPA through the `setFirstResult()`, `setMaxResults()`, and `getSingleResult()` methods provided by the `javax.persistence.Query` interface. `setFirstResult()` and `setMaxResults()` specify the bounds of the retrieved objects. The `getSingleResult()` method works similarly to `uniqueResult()` in native Hibernate API, but throws an exception whether the result is empty, or there is more than one result. The following shows an example of JPA pagination support:

```
String jpaQL = "from Teacher where phoneNumber = '+98-912-3456789'";
javax.persistence.Query query =
              entityManager.createQuery(jpaQL);
query.setMaxResults(1);
Teacher teacher = query.getUniqueResult();
```

You can find much similarity between JPA and Hibernate native API in the code snippet above.

Logging the Hibernate-Generated SQL

In some situations, using an HQL or Criteria query may not provide what you want, or you may not know how changing the HQL or Criteria query will affect results. In such cases, you can configure Hibernate to show the generated SQL in a log file, or in the application console. A query analyzer can then trace or analyze the generated SQL.

The simplest way to show the generated SQL is by setting `hibernate.show_sql` in the configuration properties file, or `show_sql` in the configuration XML file, with `true`. You can also forward the generated SQL in the log files by setting up Log4j to debug. Besides, there are specific categories for the SQL log output, providing full control on the SQL logging. To learn more about logging in Hibernate, look at this book's Appendix, or at the Hibernate website:

```
http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/
#configuration-logging
```

Summary

In this chapter, we discussed querying in Hibernate. Hibernate provides three distinct ways to query the persistent objects: HQL, native SQL, and the Criteria API.

HQL is an SQL-like language that, unlike SQL (which works on raw data in the database), lets us express queries in an object-oriented form. Like SQL, HQL is not case-sensitive, but Java-related words, such as class and property names, should appear in the correct case.

The simplest form of HQL starts with a `from` clause. A `where` clause can be added to the end of a `from` clause to restrict the query with constraints.

Hibernate lets us query persistent objects with native SQL. To execute a native SQL query, you can pass the query expression to the `createSQLQuery()` method of `Session` and obtain a `org.hibernate.SQLQuery` object. You can then treat the `SQLQuery` instance like the `Query` instance in HQL. Using native SQL bypasses some Hibernate benefits, such as caching, so using native SQL is not recommended.

The Criteria API offers another approach to querying objects. This API lets you express the desired query programmatically. The core interfaces inside this API are `Criteria` and `Criterion`. Each `Criteria` object specifies a query expression, which may consist of an arbitrary number of `Criterion` objects, each one referring to a restriction. The `Criteria` object is obtained through a `Session` object, and each `Criterion` is constructed by the `Restrictions` class, which acts as a factory for all `Criterion`.

10 Inversion of Control with Spring

Welcome to Chapter 10, where you will start developing with Spring. This chapter introduces the **Inversion of Control (IoC)** pattern and discusses how Spring provides IoC at the heart of its framework. Chapter 1 had a quick discussion about IoC and its motivation. In that chapter, you learned why we are interested in IoC and how it allows us to create more manageable and maintainable code. Also, Chapter 3 gave you a simple example of using IoC.

As those chapters discussed, IoC is about application objects. It aims to provide a simple mechanism for managing application objects and their dependencies. With IoC, the application itself is not responsible for obtaining required objects. Instead, Spring (or any other IoC framework) is configured to provide them for the application. Using IoC involves two activities:

- Defining object relations in terms of Java interfaces or abstract classes.
- Calling an outside object, called an *IoC container* (in our case, Spring), to instantiate objects, provide them where they are needed, and manage their life cycles.

Objects configured with IoC are not required to have dependencies on the container itself. They are also not required to be aware of the concrete classes of their relationships or how to locate them.

In contrast to many other features of Spring, which are dedicated to a particular layer of application architecture, IoC is a general and extensive concept that may appear in any area of application architecture.

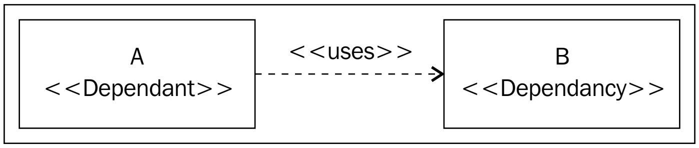
In this chapter, you will learn more about IoC and its flavor of dependency injection. You will learn how Spring is configured and used as an IoC container for application objects. You will also learn about *beans*, which are the objects managed by the Spring IoC container, as well as bean factories and application context (the Spring IoC functionality for managing these beans).

IoC is not a Spring-specific concept. Instead, it's based on Java language structures. Spring (or any other IoC container) merely provides a framework to develop IoC-style code, and allows us to configure application objects and their relations in an IoC fashion through XML, instantiate objects, and construct object graphs at runtime.

First, let's discuss what the term *Inversion of Control* means.

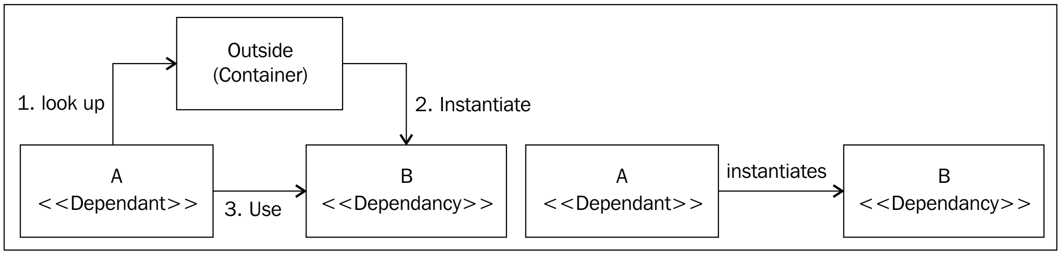
Inversion of Control and dependency injection

At the lowest level, Java applications consist of Java interfaces and classes. These classes and interfaces make up the application components, which interact with each other to provide services and accomplish the application's job. These objects are dependent on each other, and we call an object *dependent* if it uses other objects to do its job. In this case, all of the other objects that are used by the object are called the object's *dependencies*. The following figure shows a dependency relationship between objects A and B:



Dependency push versus dependency pull

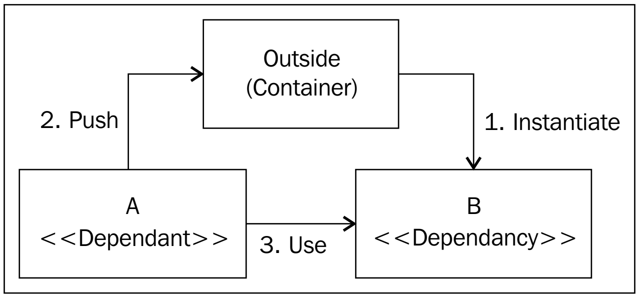
It is said that an object *pulls* its dependencies if the object itself is responsible for providing its dependencies from its environment. The object may do this by instantiating dependencies, or by looking up an outside object for them, as the following figure illustrates:



Pulling dependencies is the traditional object configuration normally used in non-IoC-style code.

> Most of the time, a common design pattern called *Service Locator*, is used with the pulling approach. A service locator object intermediates the dependant and the container, hides the complexities of interacting with the container, and provides a single point of control. Sometimes it improves performance by providing a caching mechanism.

In contrast, the object may be free from providing its dependencies. Instead, another object is responsible for providing dependencies and pushes them into the object. This type of object provision, illustrated in the following figure, is called *dependency pushing*:



In this push strategy, object A no longer takes care of dependency creation, and the container injects the dependency into the dependent.

> As its name implies, Inversion of Control indicates inverting the control of object instantiations. IoC lets us move control from dependents to the container and provide dependents with the dependencies they need.

Dependency injection

Generally, externalizing object creation and management is called Inversion of Control, but in practice, different approaches can implement IoC. The type shown in the previous figure is a particular type of IoC, in which dependencies are injected by the container to the dependent object. This is formally known as *dependency injection*.

> Martin Fowler calls IoC dependency injection when the dependencies are provided by a container and injected into dependents. Therefore, dependency injection is a specific form of IoC, in which the dependencies are provided by a container.

Because it decouples the application code meaningfully, IoC allows the creation of more manageable and testable code.

Inversion of Control in Spring

Spring supports IoC at its core. All other features of Spring, such as AOP, the web framework, transaction management, and so on, run on its lightweight IoC container. This means that you are always involved with the Spring IoC, even if you are using another feature of the Spring framework.

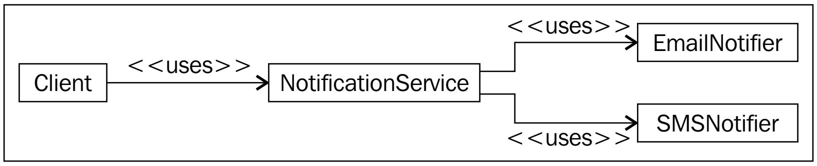
Let's continue our discussion with an example application and see how it can be implemented with non-IoC-style code. We'll then apply IoC to the application code to see how Spring's IoC capabilities decouple application classes and solve the dependency problem.

Application definition

Suppose the application uses a class as a notification service to notify the system administrator when a fatal problem occurs inside the system. The notification service may use either an email notifier or an SMS notifier to report the problem.

Implementing non-IoC-style code

The notification system's class diagram is shown in the following diagram. In this diagram, note that `Client` is a typical class in the application, using the notification service:



As the previous figure shows, any client can call the `NotificationService` class. Consequently, `NotificationService` may choose either `EmailNotifier` or `SMSNotifier`, based on the administrator's preference, to report the problem. In this example, the `Client` class is dependent on `NotificationService`, and `NotificationService` is dependent on `EmailNotifier` or `SMSNotifier`. The following code shows the `EmailNotifier` class:

```
package com.packtpub.springhibernate.ch10;

public class EmailNotifier {

  public void notify(String text) throws NotifyException {
    //sending text as an email to the administrator
  }
}
```

And this is the `SMSNotifier` class:

```
package com.packtpub.springhibernate.ch10;

public class SMSNotifier {

  public void notify(String text) throws NotifyException {
    //sending text as a SMS
  }
}
```

And here is the `NotificationService` class:

```
package com.packtpub.springhibernate.ch10;

public class NotificationService {
  EmailNotifier notifier = new EmailNotifier();
  boolean retryOnFail = true;

  public void notify(String message) throws ServiceException {
    try {
      notifier.notify(message);
    } catch(NotifyException e) {
      if(retryOnFail) {
        //notifying again
```

```
          notify(message);
        } else {
          throw new ServiceException(e);
        }
      }
    }
  }
```

These classes use the specialized exception classes; `ServiceException` and `NotifyException` extend `RuntimeException`.

Now, any client may use code such as the following to notify the administrator:

```
  try {
    NotificationService ns = new NotificationService();
    String msg = "A message indicating "+
                 " a fatal error inside the system...";
    ns.notify(msg);
  } catch(ServiceException e) {
    //handling exception
  }
```

`NotificationService` uses an instance of `Notifier` to notify the administrator. The `NotificationService` class directly instantiates a known types of `Notifier`, in this case, it is the `EmailNotifer`. As a result, `NotificationService` has a dependency on both the `Notifier` interface and a specific implementation class: if we want to change the notification strategy, we need to change the `NotificationService` class.

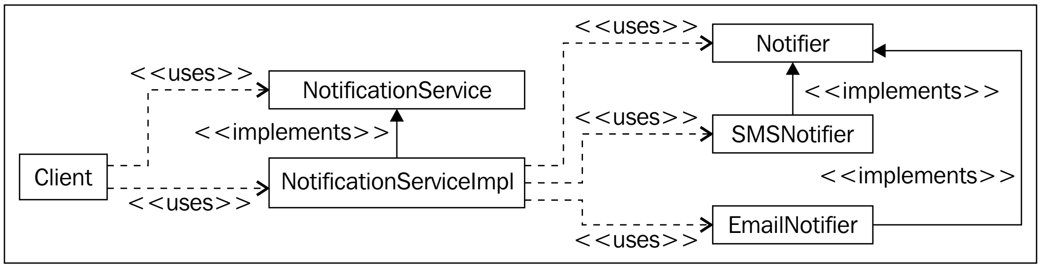
The same problem exists for the client, since this class directly uses a specific type of `NotificationService`, with no possibility of specialization.

Let's discuss how these dependencies can be eliminated when the IoC pattern is applied to the code.

Applying IoC

To apply IoC, and see how the object relationships change when they use IoC, we split up `NotificationService` into the `NotificationServiceImpl` class and the `NotificationService` interface. This interface exposes the functionality of the concrete notification class, `NotificationServiceImpl`. We also define the functionality of the two other classes, `EmailNotifer` and `SMSNotifier`, through another interface, `Notifier`.

We now have a class diagram, shown in the following figure, in which both `EmailNotifer` and `SMSNotifier` implement `Notifier` and `NotificationServiceImpl` implements `NotificationService`. The `Client` and `NotificationServiceImpl` classes use instances of interfaces, instead of the concrete implementations. Both the `Client` and the `NotificationServiceImpl` classes use these instances to interact to the target classes without instantiating them:



At this time, we still have the option of using either the pull strategy, allowing each object to provide its dependencies itself by instantiating them or by looking up the container to obtain a prepared one, or the push strategy, letting the container create the dependencies and then push them to the dependents. The following code implements the traditional pull strategy.

The `Notifier` interface remains unchanged as follows:

```
package com.packtpub.springhibernate.ch10;

public interface Notifier {
  public void notify(String text) throws NotifyException;
}
```

The `EmailNotifier` implements `Notifier` as follows:

```
package com.packtpub.springhibernate.ch10;

public class EmailNotifier implements Notifier {

  public void notify(String text) throws NotifyException {
    //sending text as an email to the administrator
  }
}
```

The `SMSNotifier` class is similar to `EmailNotifier`, as follows:

```
package com.packtpub.springhibernate.ch10;

public class SMSNotifier implements Notifier{

  public void notify(String text) throws NotifyException {
    //sending text as a SMS
  }
}
```

The `NotificationService` outlines the service class, as shown in the following code:

```
package com.packtpub.springhibernate.ch10;

public interface NotificationService {

  public void notify(String message) throws ServiceException;
}
```

And `NotificationServiceImpl` implements `NotificationService` as follows:

```
package com.packtpub.springhibernate.ch10;

public class NotificationServiceImpl implements NotificationService {
  Notifier notifier = new EmailNotifier();
  boolean retryOnFail = true;

  public void notify(String message) throws ServiceException {
    try {
      notifier.notify(message);
    } catch(NotifyException e) {
      if(retryOnFail) {
        //notifying again
        notify(message);
      } else {
        throw new ServiceException(e);
      }
    }
  }
}
```

We can now apply IoC to our class diagram. In practice, there are different types of IoC:

- **Setter injection**: The IoC container uses the dependents' JavaBean setter methods to provide dependencies for them.
- **Constructor injection**: The IoC container uses constructor parameters to provide dependencies.
- **Method injection**: The IoC container implements an abstract method in the dependents at runtime to provide dependencies for them.

The next sections discuss the different types of IoC and how they are implemented in Spring.

Setter injection

Setter injection is the most common type of IoC, in which the container injects dependencies through the dependent objects' JavaBean setter methods. In this approach, the implementation class only uses instances of interfaces, without directly instantiating the concrete classes. The dependents must contain JavaBean setter methods for their dependencies, by which the container can inject the appropriate implementation object. Note that no interfaces change. The following code shows the setter-based, IoC-style code for the `NotificationServiceImpl` class:

```
package com.packtpub.springhibernate.ch10;

public class NotificationServiceImpl implements NotificationService {
  Notifier notifier;
  boolean retryOnFail;

  public void notify(String message) throws ServiceException {
    try{
      notifier.notify(message);
    } catch(NotifyException e) {
      if(retryOnFail) {
        //notifying again
        notify(message);
      } else {
        throw new ServiceException(e);
      }
    }
  }

  public void setNotifier(Notifier notifier) {
    this.notifier = notifier;
  }

  public void setRetryOnFail(boolean retryOnFail) {
    this.retryOnFail = retryOnFail;
  }
}
```

Next, we need to configure object dependencies through the IoC configuration file. This file can be either an XML or a properties file.

The following code shows the XML configuration for our example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
        NotificationServiceImpl">
    <property name="notifier">
      <ref local="notifier"/>
    </property>
    <property name="retryOnFail">
      <value>true</value>
    </property>
  </bean>

  <bean id="notifier"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

The `<bean>` elements describe objects managed by the IoC container. Each `<bean>` element takes two attributes, `id` and `class`, referring to the name by which that object is identified and the concrete class of the object, respectively. In our example, we have used two `<bean>` elements: one for the notification service and another one for the notifier. For `notificationServiceImpl`, we used a nested `<property>` element to determine that the `notifier` property of this object should be set to an instance of the `notifier` bean, which is defined through another `<bean>` element. Note that we used the `<ref>` element to refer to another object, `notifier`, defined elsewhere in the XML configuration file. This definition expresses that Spring should instantiate the `NotificationServiceImpl` class when necessary, and wire it together with an instance of the `EmailNotifier`.

At this point, we need to start up the container to obtain the desired objects. The IoC container in Spring is called a *bean factory*. Spring provides different bean factories, all as an implementation of `org.springframework.beans.factory.BeanFactory`, or any extensions of this interface. Generally, a bean factory creates objects, wires the objects together, and manages the objects' life cycles. To start up the container, you need to create and configure an object of an appropriate bean factory. In our case, we'll use `org.springframework.context.support.ClassPathXmlApplicationContext`, a subinterface of `BeanFactory`, which reads object configurations in XML format from the classpath.

Now, to use an IoC-managed object, you only need to start up the container and obtain the object as follows:

```
try {
  ApplicationContext ctx = new ClassPathXmlApplicationContext(
                     "com/packtpub/springhibernate/ch10/beans.xml");
  NotificationService ns =
      (NotificationService)ctx.getBean("notificationService");
  ns.notify("A message indicating a fatal error inside the system...");
} catch(ServiceException e) {
  //handling exception
}
```

The `ClassPathXmlApplicationContext` class takes the path of the `beans.xml` file in the application classpath. To obtain a configured object from the IoC container, you need to call the `getBean()` method of `ClassPathXmlApplicationContext` with the object identifier, which we have already defined through the `<bean>` element, as this method's argument.

Now, the object of `NotificationService` is created, configured, and managed by the IoC container. All object dependencies have been defined in terms of interfaces. No object is aware of the actual implementation of its dependencies. The client code does not care about the implementation details of `NotificationService`, nor does `NotificationServiceImpl` know about the implementation details of `Notifier`. This isolates the client code from the implementation details of `NotificationService`, and `NotificationServiceImpl` from the implementation details of `Notifier`. It consequently lets us switch the implementation of each object to any other implementation, without making any changes in the application code, merely by changing the XML configuration file.

Constructor injection

Constructor injection, another type of IoC, is used less frequently than setter injection. In this technique, dependencies are supplied to an object through that object's constructor. The following code shows the `NotificationServiceImpl` class, which has changed to use constructor injection instead of setter injection:

```
package com.packtpub.springhibernate.ch10;
public class NotificationServiceImpl implements NotificationService {
  Notifier notifier ;
  boolean retryOnFail;
  public NotificationService(Notifier notifier,
                             boolean retryOnFail) {
    this.notifier= notifier;
```

```
        this.retryOnFail = retryOnFail;
      }
      public void notify(String message) throws ServiceException {
        try {
          notifier.notify(message);
        } catch(NotifyException e) {
           if(retryOnFail) {
             //notifying again
             notify(message);
           } else {
             throw new ServiceException(e);
           }
        }
      }
    }
```

As with setter injection, no interface has changed.

As you can see, we have replaced the setter method with a constructor in the `NotificationServiceImpl` class, which takes two arguments of `Notifier` and `boolean`, respectively. After that, the application context configuration needs to be modified so that an object of `Notifier` and a `boolean` value for `retryOnFail` are used as an argument of the `NotificationServiceImpl` constructor, as shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
NotificationServiceImpl">
    <constructor-arg>
      <ref local="notifier"/>
    </constructor-arg>
    <constructor-arg>
      <value>true</value>
    </constructor-arg>
  </bean>
  <bean id="notifier"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

The `Client` class that uses `notificationServiceImpl` remains unchanged:

```
try {
  ApplicationContext ctx = new ClassPathXmlApplicationContext(
                      "com/packtpub/springhibernate/ch10/beans.xml");
  NotificationService ns =
      (NotificationService)ctx.getBean("notificationService");
  ns.notify("A message indicating a fatal error inside the system...");
} catch(ServiceException e) {
  //handle exception
}
```

It is highly recommended that you use the `<constructor-arg>` element with either the `index` or the `type` attribute. When you don't use these attributes, Spring finds a matching bean for each argument based on the type of that argument. Therefore, when there are some arguments with the same type, or when there are arguments with values specified with string literals that are converted automatically by Spring, failure to use these attributes results in ambiguity about which argument you intend. Using these attributes can avoid mistakes in the development and maintenance phases.

For example, suppose that the `NotificationServiceImpl` class uses an extra property called `prefixMessage`. As with `notifier` and `retryOnFail`, the `prefixMessage` property is set through the IoC container. The following code shows the modified `NotificationServiceImpl` class:

```
package com.packtpub.springhibernate.ch10;
public class NotificationServiceImpl implements NotificationService {
  Notifier notifier ;
  String prefixMessage;
  boolean retryOnFail;
  public NotificationServiceImpl(Notifier notifier,
                             String prefixMessage,
                             boolean retryOnFail) {
    this.notifier= notifier;
    this.prefixMessage = prefixMessage;
    this.retryOnFail = retryOnFail;
  }
  public void notify(String message) throws ServiceException {
    String message2Send = prefixMessage +" "+ message;
    try {
      notifier.notify(message2Send);
    } catch(NotifyException e) {
      if(retryOnFail) {
        //notifying again
        notify(message2Send);
      } else {
```

```
                throw new ServiceException(e);
            }
        }
    }
}
```

If we use constructor injection for this property, and the `<value>` tag to supply the value for its argument, Spring may not distinguish the `prefixMessage` value from the `retryOnFail` value because both `prefixMessage` and `retryOnFail` values are expressed through string literals. You must use either the `type` or the `index` attribute to give Spring accurate information about the argument type or order, respectively. For our example, the bean definitions are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
               NotificationServiceImpl">
    <constructor-arg index="0">
      <ref local="notifier"/>
    </constructor-arg>
    <constructor-arg index="1">
      <value>under-test</value>
    </constructor-arg>
    <constructor-arg index="2">
      <value>true</value>
    </constructor-arg>
  </bean>
  <bean id="notifier"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

Here, the `index` attribute determines that the `notifier`, `prefixMessage`, and `retryOnFail` are the first, second, and third arguments, respectively, of the class constructor. As you can see, the `index` starts from 0. Therefore, the first argument in the argument list, `notifier` in this case, is specified with `index="0"`. Alternatively, we can use the `type` attribute to specify the type of each constructor argument, and let Spring pass the value to the proper constructor argument based on its type:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
               NotificationServiceImpl">
    <constructor-arg type="com.packtpub.springhibernate.ch10.
     Notifier">
      <ref local="notifier"/>
    </constructor-arg>
    <constructor-arg type="java.lang.String">
      <value>under-test</value>
    </constructor-arg>
    <constructor-arg type="boolean">
      <value>true</value>
    </constructor-arg>
  </bean>
  <bean id="notifier"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

If you are developing from scratch, there is actually no difference between setter injection and constructor injection. However, when you are using existing classes, your choice depends on the classes that IoC is applied to. In simple words, you need to use setter injection for classes that have no-argument constructors with simple javabean properties, and you need to use constructor injection for classes that are completely initialized through their constructors. For classes that are initialized with the combination of constructors and setter methods, you need to use mixed approaches.

Method injection

Method injection is another type of dependency injection. With method injection, instead of defining a property dependency to be set by the container, we define an abstract method which the container implements at runtime. This abstract method returns an implementation of the dependency object. Therefore, the dependent class can use the abstract method to access the appropriate implementation object on the assumption that the container will implement the method at runtime.

When our example uses method injection to provide a `Notifier` implementation for the `NotificationServiceImpl`, the injection is done through an abstract `getNotifier()` method and allows the `NotificationServiceImpl` use this method to access the `Notifier` implementation. The configuration file changes accordingly to indicate that the container should override the method and provide an appropriate implementation object as the method return value. The following code shows the `NotificationServiceImpl` class:

```
package com.packtpub.springhibernate.ch10;
public abstract class NotificationServiceImpl implements
NotificationService {
  public abstract Notifier getNotifier();
  boolean retryOnFail;
  public void notify(String message) throws ServiceException {
    try {
      getNotifier().notify(message);
    } catch(NotifyException e) {
      if(retryOnFail) {
        //notifying again
        notify(message);
      } else {
        throw new ServiceException(e);
      }
    }
  }
  public void setRetryOnFail(boolean retryOnFail) {
    this.retryOnFail = retryOnFail;
  }
}
```

The `getNotifier()` method can be moved in the `NotificationService` interface. However, I intentionally put it in this class to teach you how you can let Spring to override an abstract method at runtime, regardless of whether the class implements an interface or not.

Here is the `beans.xml` configuration file which uses the injection approach:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
               NotificationServiceImpl">
    <lookup-method name="getNotifier" bean="notifier"/>
    <property name="retryOnFail">
```

```
        <value>true</value>
      </property>
    </bean>
    <bean id="notifier"
          class="com.packtpub.springhibernate.ch10.EmailNotifier">
    </bean>
  </beans>
```

As you can see, a `<lookup-method>` element configures the `notifier` property to set the method injection. This element uses two attributes, `name` and `bean`, which refer, respectively, to the name of the abstract method and to the name of the property that is initialized with method injection. In this example, the `notifier` property is initialized through method injection, while `retryOnFail` is still set through setter injection. Although it is possible to use method injection for `retryOnFail`, I have done it this way because the `retryOnFail` property has a simple type and the application only needs its value. No implementation is defined for this property. The client remains unchanged.

Now that we've seen the different approaches to dependency injection, let's look at object configuration details, exploring how Spring provides us flexibility in configuring and initializing the objects, as well as in wiring them together.

Bean configuration

Bean configuration is at the heart of any Spring-based application. You may configure Spring either programmatically (through application code), or declaratively (through XML or properties files), as you will see later in this chapter. We will focus on the XML approach, because other approaches are neither as flexible, nor as powerful as using XML. The XML and properties files used as bean definitions in the Spring IoC container are called the *Spring application context*, or just the *Spring context*.

You have already seen some XML configuration files in the examples. Here, we'll dig a bit deeper into XML bean definitions.

Spring XML configuration is started with the `<beans>` element as the root:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd">
</beans>
```

The `<beans>` element introduces `spring-beans-2.5.xsd`, as the XML schema for validation of the Spring bean configuration file.

Support for DTD-based configuration

The Spring framework version 1.x uses DTD for validation of the Spring application context. Therefore, if you are working in Spring 1.x, you need to configure the Spring context with the `DOCTYPE` declaration and the `<beans>` element as follows:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
       "http://www.springframework.org/dtd/
             spring-beans-2.0.dtd">
<beans>
</beans>
```

Spring 2.x supports schema-based configuration, along with the classic DTD-configuration approach. This means you can still use DTD with Spring 2.x. However, you should use the schema variant to simplify the configuration file and to use new features provided by Spring.

Any bean is defined with a `<bean>` element inside `<beans>`. Each `<bean>` element includes all of the information the container needs to configure an object in an IoC style, manage the object life cycle, and provide the object dependencies.

The `<bean>` element almost always comes with two attributes: `id` and `class`. The `id` attribute assigns an arbitrary, unique name to the bean, by which the container and other beans can refer to it. The `class` attribute determines the type of the bean that is instantiated. Here is a sample bean definition, which you have already seen in previous examples:

```
<bean id="notifier" class="com.packtpub.springhibernate.ch10.
EmailNotifier">
</bean>
```

In the simplest form, the `<bean>` element can come with just one `class` attribute. In that form, the bean can only be initialized and retrieved by its type.

Alternatively, you may use the `name` attribute instead of `id`. The main advantage of using `name` is that you can use more than one ID for the bean, specified as a comma-separated list. Moreover, the `name` attribute does not have `id` naming limitations, such as starting with a letter followed by alphanumeric characters with no white space. Here is an example:

```
<bean name="notifier, emailNotifier"
      class="com.packtpub.springhibernate.ch10.EmailNotifier">
</bean>
```

In this case, the `EmailNotifier` object can be referred to as either `notifier` or `emailNotifer`.

Multiple IDs are useful when the application includes multiple bean definitions. This commonly happens when the application consists of multiple modules and each module needs its own bean definitions. To make bean definitions more expressive, each module may name its beans with a specific prefix. By using multiple IDs for shared beans, you can name these beans with more than one name. Each name begins with a particular prefix associated with a particular bean definition.

Commonly, the container instantiates beans through their constructors. If the constructor takes one or more arguments, use a nested `<constructor-arg>` element to provide the required values for each argument. Here is an example:

```
<bean name="notificationService"
      class="com.packtpub.springhibernate.ch10.
            NotificationServiceImpl">
  <constructor-arg>
    <ref local="notifier"/>
  </constructor-arg>
</bean>
```

Optionally, you can tell Hibernate to use either a static factory method or a nonstatic factory method as an instantiation approach.

You may use a static factory method of a class, instead of the class constructor, to instantiate an object. To do so, use the `factory-method` and `class` attributes to specify the factory method's name and class, respectively:

```
<bean name="notifier"
      class="com.packtpub.springhibernate.ch10.NotifierFactory"
      factory-method="getNotifierInstance">
</bean>
```

Alternatively, you may use a nonstatic factory method for creating a bean. Obviously, the factory method is defined in another bean instance in the container. Here is an example:

```
<bean name="nonStaticNotifierFactory"...>
</bean>
<bean name="notifier, emailNotifier"
      factory-bean="nonStaticNotifierFactory"
      factory-method="getNotifierInstance">
</bean>
```

As you can see, we have used the `factory-bean` attribute to refer to another bean instance, including the factory method. Note that we have not specified the type of the bean being instantiated in any of the cases so far.

Singleton versus prototype beans

Spring lets objects be configured as either singleton or nonsingleton. An object is called a *singleton* if there is only a single instance of the class throughout the application. In Spring, this means multiple invocations of `ApplicationContext.getBean()` return the same reference. In contrast, an object can be a *nonsingleton* (also called a *prototype*) if more than one instance of the class exists in the application. This means that multiple invocations of `ApplicationContext.getBean()` return different instances.

By default, all beans in Spring are defined as singletons. To define a bean as a prototype, set the `singleton` attribute to `false`, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd">
  <bean name="notifier, emailNotifier"
        singleton="false"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

Using a singleton bean is helpful when the creation of more than one instance is expensive in terms of time, memory, network bandwidth, or CPU usage. Note that if you use the instance in a multithreaded environment, the instance should be thread-safe.

Wiring beans

As we've seen, developing an IoC-based application involves defining the objects' dependencies in terms of interfaces or abstract classes. The dependencies are then set up in the configuration file(s) by specifying the actual implementations. Expressing bean dependencies with concrete implementation in the configuration files is called *bean wiring*.

In the previous sections, you learned basic bean configuration using the `<bean>` element. In this section, we will discuss configuration details in more depth. As you have seen, each `<bean>` element assigns a name to an object, indicates which method instantiates the object, and determines how to provide object properties.

The `<bean>` element can contain zero or more `<constructor-arg>` subelements, which define the constructor argument values. The bean may use setter injection, and therefore use nested `<property>` elements. The bean may use a combination of setter injection for some properties, and constructor injection for others.

The following example uses constructor injection for the `notifier` constructor argument, and setter injection for the `retryOnFail` property:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
               NotificationServiceImpl">
    <constructor-arg index="0">
      <ref local="notifier"/>
    </constructor-arg>
    <property name="retryOnFail"><value>true</value></property>
  </bean>
  <bean id="notifier" class="com.packtpub.springhibernate.ch10.
   EmailNotifier">
  </bean>
</beans>
```

The `<property>` and `<constructor-arg>` elements can contain other elements. These elements, including `<bean>`, `<ref>`, `<idref>`, `<value>`, `<null>`, `<list>`, `<set>`, `<map>`, and `<props>`, let us provide values for `<property>` and `<constructor>` in different ways. Note that each `<property>` or `<constructor-arg>` element can contain a `<bean>` element that is the same as the usual `<bean>` element. For instance, the recent bean configuration for `NotificationServiceImpl` can be expressed as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        class="com.packtpub.springhibernate.ch10.
               NotificationServiceImpl">
    <constructor-arg index="0">
      <bean id="notifier"
             class="com.packtpub.springhibernate.ch10.EmailNotifier">
      </bean>
    </constructor-arg>
    <property name="retryOnFail"><value>true</value></property>
  </bean>
</beans>
```

This type of bean configuration is useful when only the outer bean uses the inner bean. No use of the inner bean outside the scope of the outer bean is applicable.

The following sections discuss these elements.

The <ref> element

The `<ref>` element applies another bean as the value for a property or constructor argument. This element has the `local`, `bean`, and `parent` attributes. Here are some examples:

```
<ref local="notifier"/>
<ref bean="notifier"/>
<ref parent="notifier"/>
```

The `local`, `bean`, and `parent` attributes have these meanings:

- `local` refers to another bean that is defined in the same XML file. Note that this attribute can only use the ID (never the name) of the bean to which it refers.
- `bean` is similar to `local`, the only difference being that the bean to which it refers can be defined either in another bean definition file or in the same definition file.
- `parent` refers to a bean defined in the parent factory. This attribute is useful when beans with the same name exist in both the current and the parent factory.

The <idref> element

The `<idref>` element specifies another bean as the value. With this element, an exception will be thrown if the container cannot find the specified bean. Here is an example:

```
<property name="beanName"><idref local="notificationService"/>
</property>
```

The `<idref>` element can come with `local`, `bean`, or `parent`, with the same meanings as for the `<ref>` element.

The <value> element

The `<value>` element determines a value for a property or a constructor argument. The value is always expressed as a string and is transformed by Spring to the appropriate type. For simple values, such as primitive types or their corresponding wrapper types, Spring can convert the string literal to the appropriate representation. However, for other types, or custom value types, you can implement `java.beans.PropertyEditor` and register the implementation with Spring. The implementation of `PropertyEditor` defines how to convert a string literal to the type of target property or constructor argument. You can find a number of `PropertyEditor` implementations in the `org.springframework.beans.propertyeditors` package of the Spring distribution.

Here is an example of the `<value>` element using a primitive `boolean` type:

```
<property name="debug">
  <value>true</value>
</property>

<property name="debug" value="true">
```

Both of these snippets do the same thing and will set the `debug` property to `true`. If `debug` is defined as `Boolean`, the container automatically converts the string literal value `"true"` to the `Boolean` or `boolean` value `true`.

The <null> element

You can use the `<null>` element to set a property or constructor argument with `null`, as in this example:

```
<property name="comment"><null/></property>
```

Note that if you omit the `<null/>` element, an empty string will be set instead of `null`.

The <list>, <set>, <map>, and <props> elements

The `<list>` element sets a value for a property or constructor argument of type `java.util.List`, `java.util.Set`, or array. The `<set>` element is similar to `<list>`, but does not accept duplicate values. The `<map>` and `<props>` elements set values of type `java.util.Map` and `java.util.Properties`, respectively. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="complexBean" class="com.packtpub.springhibernate.ch10.
   ComplexBean">
    <property name="adminIDList">
      <list>
        <value>221</value>
        <value>323</value>
        <value>212</value>
        <value>412</value>
        <value>511</value>
     </list>
    </property>
    <property name="adminNameSet">
      <set>
        <value>John</value>
        <value>David</value>
        <value>Andrew</value>
        <value>David</value>
        <value>Bobby</value>
     </set>
    </property>
    <property name="adminEmailMap">
      <map>
```

```
          <entry key="221">
            <value>John@domain.com</value>
          </entry>
          <entry key="323">
            <value>David@domain.com</value>
          </entry>
          <entry key="212">
            <value>Andrew@domain.com</value>
          </entry>
          <entry key="412">
            <value>David@domain.com</value>
          </entry>
          <entry key="511">
            <value>Bobby@domain.com</value>
          </entry>
        </map>
      </property>
      <property name="adminAgesProps">
        <props>
          <prop key="John">32</prop>
          <prop key="David">41</prop>
          <prop key="Andrew">28</prop>
          <prop key="David">34</prop>
          <prop key="Bobby">45</prop>
        </props>
      </property>
    </bean>
  </beans>
```

Here, the values of `<list>`, `<map>`, and `<set>` are simple string or numeric values. However, any of these elements can be nested with `<bean>`, `<ref>`, `<idref>`, `<list>`, `<set>`, `<map>`, `<props>`, `<value>`, and `<null>` elements to hold objects.

Automatic wiring

In the previous section, you learned how to configure objects with their dependencies inside the bean configuration file. This way of declaring dependencies is called *explicit declaration* since you manually specify how to configure each object using other beans. However, Spring also allows automatic wiring of beans. This means you can leave dependencies undeclared in the XML file and let Spring find appropriate values for each property or constructor argument.

Autowiring is off by default, so you must enable autowiring before you can use it. To do so, set the `autowire` attribute of the `<bean>` element with one the following values:

- `no`: No autowiring is used for the bean. All dependencies must be declared explicitly if the default autowiring is not changed at the bean factory level.
- `byName`: The bean is autowired by property name. Spring uses property names to find matching beans in the factory. For instance, if we use `byName` autowiring for the `notifier` property in the `NotificationServiceImpl` class, Spring sets this property with a bean named `notifier`. If no bean with that name exists, the property remains unset.
- `byType`: The bean is autowired by property type. Spring finds a matching bean for each property based on the type of the property. If Spring does not find a matching bean, the property remains unset. If more than one matching bean exists, an exception is thrown.
- `constructor`: The bean is autowired by type in its constructor. This means that Spring finds a matching bean for each constructor argument. If the bean has more than one constructor, the bean is autowired with the constructor that has the most matching arguments.
- `autodetect`: The bean is autowired by `constructor` if it does not have a default no-argument constructor. Otherwise, it is autowired by `bytype`.

> You can use the `dependency-check` attribute of the bean definition to specify whether Spring should treat unmatched properties as an error case. `dependency-check="none"` indicates no dependency checking. `dependency-check="simple"` determines that dependency checking is only performed for primitive types and collections. `dependency-check="objects"` specifies that dependency checking is only performed only for properties that are objects, neither primitives nor collections. Finally, `dependency-check ="all"` means that dependency checking is done for all dependencies, including primitive types, collections, and associated objects.

You can mix autowiring and explicit wiring. Therefore, you can declare a bean as autowired while some properties or constructor arguments are wired explicitly. In this case, Spring first wires the properties or constructor arguments, declared explicitly and then autowires others.

> If you don't use it carefully, autowiring may have unexpected results. With autowiring, you must always check dependencies in your mind, instead of hard-coding them, and track the dependency relationships. Do not use autowiring for large deployments.

The following code shows an example of autowiring. I have changed our notification service example to use autowiring instead of explicit wiring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
             spring-beans-2.5.xsd">
  <bean id="notificationService"
        autowire="byType"
        class="com.packtpub.springhibernate.ch10.NotificationService">
    <!-- notifier is not wired explicitly -->
    <property name="retryOnFail">
      <value>true</value>
    </property>
  </bean>
  <bean id="notifier"
        class="com.packtpub.springhibernate.ch10.EmailNotifier">
  </bean>
</beans>
```

This example removes explicit wiring of the `notifier` property, and instead inserts `autowire="byType"` in the bean definition.

Annotation-based container configuration

Spring allows bean configuration with annotations. Spring 2.0 enabled bean configuration with the `@Required` annotation. Spring 2.5 provides some additional annotations, including `@Autowired`, `@Resource`, `@PostConstruct`, `@PreDestroy`. To enable annotation-based configuration, you need to add the `<context:annotation-config>` element in your bean configuration file, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/
          spring-context-3.0.xsd">

     <context:annotation-config/>

</beans>
```

Let's look at the annotations used for bean configuration.

@Required

The `@Required` annotation is applied to a bean property setter method to indicate that the related property must be populated at configuration time. Otherwise, it should be treated as an error. The property can then be valued explicitly through the bean definition or by autowiring.

Here is an example:

```
public class NotificationServiceImpl implements NotificationService {
  private Notifier notifier;

  @Required
  public void setNotifier(Notifier notifier) {
    this.notifier = notifier;
  }

}
```

As you can see, the `@Required` annotation is applied to the `setNotifier()` method. Therefore, the `notifier` property is a mandatory property that must be initialized for the `NotificationService` object.

@Autowired

The `@Autowired` annotation can be put on any setter method, class property method, or even constructor with any arguments. This annotation can be used with the `required` property to indicate whether that property is required for autowiring purposes or not. This is an example:

```
public class NotificationServiceImpl implements NotificationService{

  private Notifier notifier;

  @Autowired(required=false)
  public void setNotifier(Notifier notifier) {
    this.notifier = notifier;
  }

}
```

Although there is no limitation on the number of annotated constructors in each class, only one annotated constructor can be marked as required.

@Resource

It is also possible to use JSR-250 `@Resource` annotation to mark a field or a setter method. The `@Resource` annotation uses an optional `name` attribute which is the name of the object to inject. If this attribute is not specified, Spring uses the name of the bean property as the value for this attribute. Here is an example:

```
public class NotificationServiceImpl implements NotificationService {
  private Notifier notifier;
  @Resource(name="myNotifier")
  public void setNotifier(Notifier notifier) {
    this.notifier = notifier;
  }
}
```

By this code, Spring injects the `Notifier` object whose name in the Spring context is `myNotifier` to the `NotificationService` object. If the `name` attribute is not specified, Spring injects the bean named `notifier` to the `NotificationService` object.

Classpath scanning for annotated classes

Spring provides stereotype annotations to mark classes as a specific managed bean in its context. These annotations are as follows:

- `@Component`: It is the basic annotation type to mark a class as a Spring bean.
- `@Controller`: It is used to mark a class as a controller in the Spring MVC.
- `@Repository`: It marks a class as a repository, such as a Data Access Object.
- `@Service`: This annotation indicates that the annotated class is a part of the business logic of the application.

All of these annotations make the annotated class a managed bean in the Spring context. The following shows the `NotificationService` class which is now annotated with `@Component`:

```
@Component
public class NotificationServiceImpl implements NotificationService {
  private Notifier notifier;
  @Autowired
  public void setNotifier(Notifier notifier) {
    this.notifier = notifier;
  }
}
```

When a class is annotated with a stereotype annotation, Spring uses the uncapitalized non-qualified class name as the bean name for that object. Therefore, for the annotated class above, Spring uses the name `notificationService`. It is also possible to override this default behavior with a name through the stereotype annotation like as this:

```
@Component("myNotificationService")
public class NotificationServiceImpl implements NotificationService{
…
}
```

Therefore, Spring uses `myNotificationService` as the name for the `NotificationService` bean.

To Enable Spring to auto-detect the annotated classes, you need to use the `<context:component-scan>` element inside the configuration file to introduce the base package in which Spring should look for annotated classes. The following shows how it should be done:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
2.5.xsd">

     <context:component-scan base-package="com.packtpub"/>
</beans>
```

You can use a comma separated list of packages as the value of the `base-package` attribute, if the annotated classes are located in different packages.

Other format forms for bean definition

You can use the properties file format as an alternative to the XML format for bean configuration. However, the properties file format doesn't support a number of container capabilities that are expressed easily through the XML format, such as constructor injection, method injection, nested beans, and so on.

You can also configure the IoC container programmatically. In this book, we are not interested in the programmatic approach, because it removes the flexibility from the code, and increases maintenance costs.

BeanFactory and ApplicationContext

Spring provides a variety of bean factory implementations to support a range of different situations, each having different IoC requirements. All bean factories are implementations of `org.springframework.beans.factory.BeanFactory`, or any of its subinterfaces. Some of the most significant of these are `BeanFactory`, `HierarchicalBeanFactory`, `ListableBeanFactory`, `AutowireCapableBeanFactory`, and `ConfigurableBeanFactory`.

Additionally, Spring provides `ApplicationContext` as a specific type of bean factory with some advanced functionality. The `ApplicationContext` interface extends the `BeanFactory` interface, so it does everything a bean factory can do. However, application context has additional functionalities such as these:

- Application context lets us work with the IoC container in a completely declarative fashion. (We refer to both the `ApplicationContext` object, and the bean definition file, as the application context.) Furthermore, Spring provides some utility classes which let us add IoC capability to the web layer, and allows automatic loading of application contexts in web frameworks. Of course, the aim of this is to add IoC functionality to the framework, instead of the user's code.
- Application context extends the `MessageResource` interface, so it provides messaging functionality.
- Application context supports an event-handling model. It can notify beans that implement the `ApplicationListener` interface when an `ApplicationEvent` gets published to the `ApplicationContext`.
- Application context is a resource loader because it extends the `ResourcePatternResolver` interface. This means that `ApplicationContext` can load any resources from almost any location in a transparent fashion, including from the classpath, a file system location, or anywhere that can be described with a standard URL, and other variations.

In practical applications, `ApplicationContext` is always used instead of `BeanFactory`.

Let's look at the different bean factories available to us:

- `BeanFactory`: This is the basic interface that defines the general behaviors of all bean factories. This interface defines various `getBean()` methods for obtaining beans from the container. It also includes some extra methods, such as `containsBean()`, `isSingleton()`, and `getType()`, which allow us to query the bean definition in application code.
- `HierarchicalBeanFactory`: This bean factory provides a factory for hierarchical bean definitions. When a factory is queried for a bean, and does not have the requested bean, its parent factory is asked for that bean. The parent may also ask its own parent when it does not have the bean, and so on. This factory is useful when the application has a number of bean definitions. If each bean definition has its own beans, and uses some general beans of others, you must create a hierarchical bean factory, in which the general beans are defined in the parent factory or factories. Note that the entire process of bean exploring is transparent for all clients, so you can treat this factory the same as other bean factories. The `HierarchicalBeanFactory` interface defines only two methods: `getParentBeanFactory()` returns the current factory's parent factory, and `containsLocalBean()` determines whether or not the current factory contains a particular bean.
- `ListableBeanFactory`: This bean factory allows us to query the beans. This factory provides `getBeanDefinitionNames()` to get the names of all beans, `getBeanNamesForType(Class class)` to get the names of all beans of a certain type, `getBeanDefinitionCount()` to get the number of all beans, `containsBeanDefinition(String beanName)` to check whether any beans with a particular name exist, and `getBeansOfType(Class aClass)` to obtain all beans of a certain type in a `java.util.Map` object.
- `AutowireCapableBeanFactory`: This bean factory is capable of autowiring for existing bean instances. When you are extending an existing application that does not rely on Spring for object instantiations, or when you are using third-party code that uses its own mechanism for instantiating objects, you can use this factory if you want Spring to provide dependencies and wire the objects together. The main method of this factory is `autowire()`, which lets you specify a class name to the factory and get a fully configured object. Two other significant methods are `autowireBeanProperties()` and `applyBeanPropertyValues()`, which let you configure a preexisting external object and supply its dependencies using Spring 3.

Although Spring allows us to configure the IoC container either programmatically, through its rich API, or declaratively, through different configuration formats, this book discusses only declarative bean factories and application contexts with XML as the configuration format.

Now that we've seen the various bean factories, let's look at the different application contexts available in Spring:

- `ApplicationContext`: This interface is the basic application context. It extends `BeanFactory` and defines the basic functionality for all application contexts.
- `WebApplicationContext`: This subinterface of `ApplicationContext` defines an extra `getServletContext()` method, and provides IoC ability in web applications. I will discuss this application context in Chapter 14.
- `FileSystemXmlApplicationContext`: This implementation of the `ConfigurableApplicationContext` interface allows us to work with XML configuration files represented as plain string paths to the file system or URLs. This can be used in stand-alone applications or for testing purposes.
- `ClassPathXmlApplicationContext`: This general-purpose application context may be used in any environment and condition. It allows us to configure the container through an XML file in the classpath. The path is expressed as a plain string relative to the root of the classpath and is separated by a forward slash (/) instead of a dot (`.`).

Applying IoC to Hibernate resources

Hibernate applications may use either container-managed or application-managed connections behind the scenes. We can use IoC in a Hibernate application and configure a data source as a bean. IoC lets us manage the data source transparently and declaratively. Therefore, we can look up the container to obtain the `SessionFactory` object initialized with a configured data-source object.

We can assume that `SessionFactory` is a singleton bean in the Spring IoC container. However, configuring a `SessionFactory` as a bean is not effortless as it has many properties and a complex structure. For this and similar cases, Spring provides a specific type of bean called a *factory bean*.

A factory bean is a bean inside the IoC container that produces other objects. Note that a factory bean is different from a bean factory. Bean factories are the Spring containers that instantiate and manage beans, whereas factory beans are normal beans managed by a bean factory. However, when a factory bean is referred to by another bean in the container, or by application code by using the `getBean()` method, the container does not return an instance of the factory bean. Instead, it returns an object that the factory bean produces.

You can easily define a factory bean by implementing the `org.springframework.beans.factory.FactoryBean` interface:

```
public interface FactoryBean {
  Object getObject() throws Exception;
  Class getObjectType();
  boolean isSingleton();
}
```

The `getObject()` method returns the output object of the factory. The container automatically calls it when the factory bean is accessed. The `isSingleton()` flag determines whether the returned object is a singleton or not. Finally, the `getObjectType()` method determines the type of the output object (`null` if this is not known).

Let's look at an example to see how a `FactoryBean` is implemented and used. Suppose that we need to obtain a distinct `java.sql.Connection` object whenever the container is looked up by the `connection` name, as follows:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
                      "com/packtpub/springhibernate/ch10/beans.xml");
Connection connection = (Connection)ctx.getBean("connection");
```

Because `java.sql.Connection` cannot easily be configured in the bean definition, we may decide to implement a `BeanFactory` class and use it as a factory for `Connection` objects in the Spring context. The following code shows this implementation:

```
package com.packtpub.springhibernate.ch10;

import org.springframework.beans.factory.FactoryBean;

import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class ConnectionFactoryBean implements FactoryBean {
  String url;
  String driver;
  String username;
  String password;
  Properties props;

  public Object getObject() throws Exception {
    if(props == null) {
      props = new Properties();
    }
    if(username != null) {
      props.setProperty("username", username);
    }
    if(password != null) {
      props.setProperty("password", password);
    }
    validateUrl();
    validateDriver();

    Class.forName(driver);
    return DriverManager.getConnection(url, props);
  }

  public Class getObjectType() {
    return Connection.class;
  }

  public boolean isSingleton() {
    return false;
  }

  private void validateUrl() throws Exception {
    if ((url == null) || (url.equals(""))) {
      url= props.getProperty("url");
    }
    if(url == null) {
      throw new Exception("Database URL is not configured or is
invalid");
    }
  }

  private void validateDriver() throws Exception {
    if ((driver == null) || (driver.equals(""))) {
      driver= props.getProperty("driver");
    }
    if(driver == null) {
      throw new Exception("Database Driver is not configured or is
invalid");
    }
  }

  public void setUrl(String url) {
    this.url = url;
  }
  public void setDriver(String driver) {
    this.driver = driver;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public void setProps(Properties props) {
    this.props = props;
  }
}
```

This class has five properties that are set through setter injections. These properties determine the database URL, the driver's class name, username, password, and optional JDBC properties for the database connection. To use `ConnectionFactoryBean`, you need to configure it in the Spring context as follows:

```
<bean id="connection"
      class="com.packtpub.springhibernate.ch10.ConnectionFactoryBean">
  <property name="username" value="sa"/>
  <property name="password" value=""/>
  <property name="driver" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:hsql://localhost/hiberdb"/>
  <property name="props">
  <value>
    defaultRowPrefetch=15
  </value>
  </property>
</bean>
```

Note that this example would not be used in a real application, because in practice, there are better approaches to obtain the `Connection` object, such as looking up an application server.

Although you can implement a factory bean whenever you need to, Spring includes some useful factory beans for common resources and services. The following table shows some of these factories:

Factory Bean	Description
`JndiObjectFactoryBean`	Returns the result object of a JNDI lookup.
`ProxyFactoryBean`	Returns a proxy object. A *proxy* is an object that wraps an existing object and provides added functionality for the wrapped object. Chapter 11 discusses proxies in detail.
`TransactionProxyFactoryBean`	Returns a transactional proxy for persisting objects, such as Data Access Object (DAO) classes. I have discussed `TransactionProxyFactoryBean` in Chapter 11 and Chapter 12.
`LocalSessionFactoryBean`	Returns a configured Hibernate `SessionFactory` object that may be used in DAO classes.

PropertyEditors

`PropertyEditor` defines how to convert a string literal that is defined as a value for a property, or constructor argument in the bean definition files, to the target type. Spring can convert simple primitive values (such as `int`, `long`, and `boolean`) and their corresponding wrapper types (such as `Integer`, `Long`, and `Boolean`) from the string form to the target primitive type. Additionally, Spring uses a number of built-in `PropertyEditor`, allowing some extra types to be converted. The following table shows these built-in `PropertyEditor` and the target type to which each converts the string literal. All of these `PropertyEditor` are located in the `org.springframework.beans.propertyeditor` package:

Built-in PropertyEditor	Description
`ClassEditor`	Converts a string literal that represents a class to an object of `java.lang.Class`. If the class is not found, an `IllegalArgumentException` is thrown.
`FileEditor`	Converts a string literal that represents a file to an object of `java.io.File`.
`LocaleEditor`	Generates a `java.util.Locale` from a string literal expressed as `[language]_[country]_[variant]`.
`PropertiesEditor`	Converts strings expressed as the key and value pairs (`key=value`) to an object of `java.util.Properties`.
`StringArrayPropertyEditor`	Converts a comma-delimited list of strings to an array of `String`.
`URLEditor`	Resolves a string representation of a URL to an object of `java.net.URL`.

Let's look at a simple example to see how Spring uses these `PropertyEditor`. Suppose the application uses a `ConfigurationBean` class, shown in the following code:

```
package com.packtpub.springhibernate.ch10;
import java.io.File;
import java.net.URL;
import java.util.Locale;
import java.util.Properties;
public class ConfigurationBean{
  private int intValue;
  private boolean booleanValue;
  private String[] stringArray;
  private Class clazz;
  private File file;
  private Properties props;
  private Locale locale;
  private URL url;
  public int getIntValue() {
    return intValue;
  }
  public void setIntValue(int intValue) {
    this.intValue = intValue;
  }
  public boolean isBooleanValue() {
    return booleanValue;
  }
  public void setBooleanValue(boolean booleanValue) {
    this.booleanValue = booleanValue;
  }
  public String[] getStringArray() {
    return stringArray;
  }
  public void setStringArray(String[] stringArray) {
    this.stringArray = stringArray;
  }
  public Class getClazz() {
    return clazz;
  }
  public void setClazz(Class clazz) {
    this.clazz = clazz;
  }
  public File getFile() {
    return file;
  }
  public void setFile(File file) {
    this.file = file;
  }
  public Properties getProps() {
    return props;
  }
  public void setProps(Properties props) {
    this.props = props;
  }
  public Locale getLocale() {
    return locale;
  }
  public void setLocale(Locale locale) {
    this.locale = locale;
  }
  public URL getUrl() {
    return url;
  }
  public void setUrl(URL url) {
    this.url = url;
  }
}
```

The following code shows how an object of this class can be configured as a bean in the Spring bean definition file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd">
  <bean id="configurationBean" class="com.packtpub.springhibernate.
ch10.ConfigurationBean">
    <property name="intValue" value="10"/>
    <property name="booleanValue" value="true"/>
    <property name="stringArray" value="Spring,Hibernate,Ant,Log4j,HS
QLDB"/>
    <property name="clazz" value="java.util.Stack"/>
    <property name="file" value="/images/sample.gif"/>
    <property name="url" value="http://www.packtpub.com"/>
    <property name="locale" value="en_US"/>
    <property name="props">
      <value>
        username=administrator
        password=123
      </value>
    </property>
  </bean>
</beans>
```

When the client obtains the bean, the container automatically converts literal strings to the proper types.

Summary

In this chapter, you learned about Inversion of Control (IoC) and dependency injection. Inversion of Control, a concept based on Java language constructs, allows us to define object dependencies in terms of interfaces or abstract classes, and lets an outside object, the IoC container, instantiate them from concrete classes.

We saw how the Spring IoC container can be used to develop IoC-style code. First, you must define all object dependencies of interfaces or abstract classes. Then, you must configure bean definitions to tell the container which dependency of which concrete class should be instantiated. Finally, you must start up the IoC container and obtain configured objects by their names.

In Spring, bean definitions can be in XML or properties files. In this chapter, we discussed the XML variant only, because it's the most widely used format for bean definitions. Each XML bean definition starts with an XML declaration, followed by a `<beans>` element as its root. The `<beans>` element introduces the schema version that Spring uses for validation of the bean definition file. All beans are declared through `<bean>` elements inside `<beans>`. Any `<bean>` can come with two attributes: `id` and `class`. The `id` assigns an identifier to the bean, by which it is referred to along the bean definition or in the application code. The `class` specifies the class of which the object is instantiated. The `<bean>` element can nest with `<property>` and `<constructor-arg>` elements, which determine IoC injection values for the object properties or constructor arguments, respectively.

We also learned how to create the `SessionFactory` object in IoC-style code. To do this, we used `LocalSessionFactoryBean` as the factory bean, which is configured in the same way as any normal bean in the Spring IoC container. Accessing it always returns an object of `SessionFactory`.

Finally, we looked at `PropertyEditors`, which define how Spring converts a property value expressed as a string literal to a Java object.

11
Spring AOP

Today, **Object-Oriented Programming (OOP)** is a common development methodology for almost all software applications. With OOP, the application comprises a collection of classes and interfaces. OOP indicates that classes have clear, simple, and distinct definitions and responsibilities. Minimizing the interdependency of classes makes it easier to create, test, and maintain the application. To achieve this, you should break the application into smaller and smaller meaningful, simple, and well encapsulated classes. However, the nature of OOP means that this is not always possible. Sometimes, it is difficult, or even impossible, to express logic in an encapsulated class. In such situations, developers must mix the logic with functions and responsibilities of other classes.

Aspect-Oriented Programming (AOP) is a new methodology which offers software developers clean separation of concerns. When you use AOP, the application is organized by clean, simple, well encapsulated classes without interference from other classes' functions.

AOP does not compete with OOP. Instead, it complements OOP where OOP cannot perform its role perfectly.

This chapter has four sections:

- **Introduction to AOP**: This section provides an overview of what AOP is, what it offers us, and how it is organized. This section discusses how AOP solves the problems associated with OOP.
- **Using Spring AOP with Spring IoC: An example**: This section uses a simple example to demonstrate how, in practice, AOP concepts are implemented with Spring AOP.

- **Spring's AOP Framework**: This section delves into the implementation details of Spring AOP. This section explains different aspects of Spring AOP that you can use to implement simple, flexible, and powerful AOP.
- **Moving to Spring 2.x's AOP**: This section offers insight into new AOP features introduced by Spring 2.x. This section explains how AOP components are easily declared in the Spring context through new XML elements defined by the AOP schema.

Introduction to AOP

Before discussing AOP, I want to introduce a new term: **concern**. A concern is a function or behavior that the application performs. For example, persistence, validation, security checking, and so on are typical concerns. Application development is the process of implementing all application concerns.

With OOP, developing a concern means creating a class which fully encapsulates the concern's functionality. Each concern that is implemented as a class is instantiated and is used by other concerns, which in turn are implemented by other classes. Using this approach, the concerns are individual classes, which are easy to debug, refactor, document, and support. Unfortunately, this approach is not always appropriate. In practice, concerns may be scattered over many other concerns. For example, consider the logging concern that is always implemented as glue code in other concerns. To implement the logging concern, you need to mix logging code with other application code. Relying on OOP, the logging concern cannot be encapsulated in an individual class. Logging and similar concerns are called *cross-cutting* concerns since they are logic scattered over other concerns' logic.

Here are some other examples of cross-cutting concerns:

- All of the methods with certain names are logged by the application's logging mechanism.
- All of the persistent methods are invoked inside a transaction.
- Exceptions thrown by certain methods are handled similarly.
- The application provides customers' statistical information, reporting information about different parts of the application.
- Only authorized clients can invoke certain methods.

To implement these concerns with OOP, you must change the code to insert the concern code. If any of these concerns needs to be updated or removed, you must change the other concern's code to update or remove the concern. More importantly, a concern is logic that may be applied to many objects, which means you must change many methods just to insert or update the same logic. Obviously, this is not an effective approach because it does not follow the encapsulation model provided by OOP, or provide reusable functionality. As a result, the code is difficult to test and maintain.

Although the application concerns can be implemented by the developer to be managed by the AOP container, they are almost always ready implemented concerns that are only configured to be managed by the container. Security and transaction are common cases for concerns.

As you can see, OOP cannot provide a neat, clean solution in such cases. This is where AOP comes in. AOP provides a mechanism to fully separate concerns, and to describe cross-cutting concerns with individual classes in a way that's similar to other ordinary concerns.

To see how AOP helps us to implement cross-cutting concerns, let's refer back to an example from Chapter 3 and see how OOP and AOP handle these concerns.

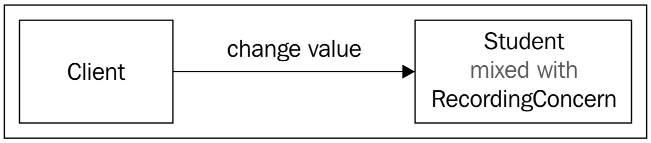
Implementing cross-cutting concerns with OOP

Chapter 3 introduced a simple case to demonstrate Spring IoC. According to that case, a nonfunctional requirement indicates that the application records any change of an object's properties for auditing purposes. In that example, three different recorders were introduced: `SetterInfoConsolePrinter` for recording object modifications in the console, `SetterInfoDBPrinter` for recording object modifications in the database, and `SetterInfoLogPrinter` for recording object modifications in log files. Each of these classes implements a common interface, called `SetterInfoPrinter`, which outlines the recorder behavior.

You change object properties through their setter methods. This means we need to track the invocation of objects' setter methods to record object modifications. For this purpose, we changed the `Student` class, as shown in code below, to record object modifications in the console, database, or log files:

```
package com.packtpub.springhibernate.ch11;
public class Student {
    private int id;
    private String firstName;
    private String lastName;
    private SetterInfoPrinter printer; //ideally initialized through
Spring IoC
    //zero-argument
    public Student() {
    }
    public Student(String firstName, String lastName) {
        this();
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public void setId(int id) {
        printer.print("setId", this.id, id);
        this.id = id;
    }
    public void setFirstName(String firstName) {
        printer.print("setFirstName", this.firstName, firstName);
        this.firstName = firstName;
    }
    public void setLastName(String lastName) {
        printer.print("setLastName", this.lastName, lastName);
        this.lastName = lastName;
    }
    //getter methods
    //hashCode() and equals() methods
    public void setPrinter(SetterInfoPrinter printer) {
        this.printer= printer;
    }
}
```

The recording requirement is a cross-cutting concern because it is scattered over many setter methods. To change the recording strategy, or to remove the recording functionality from the code, we need to change all of the classes to update or remove the recording functionality. The following figure depicts the OOP approach for implementing the recording concern in the `Student` class:



The OOP approach of implementing a cross-cutting concern has the following shortcomings:

- **Code duplication**: We need to add duplicate code to every method. In our example, the recording code is put in all of the setter methods.
- **Testing problems**: Testing the final code would be very difficult because the cross-cutting concern is not located in one place, where it could be easily tested. In our example, we need to test all setter methods one-by-one.
- **Maintenance difficulty**: If the requirements change, we must change the code in many places. For example, if a business requirement indicates that no exception handling is needed, or that exceptions should be logged instead of notifying the administrator, we must change many areas of the code to satisfy the new requirement.

Now that you understand what OOP lacks when implementing cross-cutting concerns, we are in a position to dig into the details of AOP and see how it complements OOP. Before we do that, though, let's go over some AOP terminology.

AOP terminology

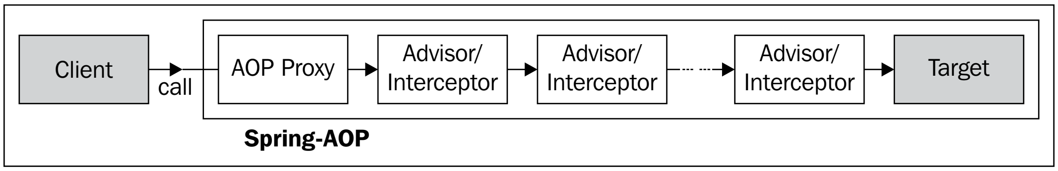
As with other methodologies, AOP has a terminology to describe different participants in an AOP operation. The following list briefly defines AOP terms:

- **Advice**: Advices implement concerns. An advice is a behavior or reaction that is performed at particular points of the application (for example, the recording functionality mentioned in the previous section). Spring supports five different types of advice:
    - Advice executed around method execution, both before and after method invocation.
    - Advice performed before method invocation.

    ◦ Advice performed after method invocation when the method normally returns.
    ◦ Advice executed after method invocation when the method throws an exception and returns abnormally.
    ◦ Advice executed after method invocation. Either the method returns normally or it throws an exception and returns abnormally. (This is only supported by Spring 2.x.)

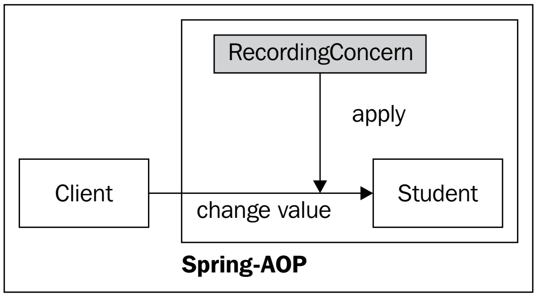
- **Joinpoints**: These are the points in the application where the advice is invoked. Joinpoints are where we insert the cross-cutting logic provided by an advice (setter methods in the previous example). Typical joinpoints are a call to a method, the method invocation itself, class initialization, and object instantiation.
- **Pointcuts**: Any pointcut can be considered a collection of joinpoints. Pointcuts allow us to group joinpoints, and then apply an advice to them. For instance, you can define all methods of a particular class as a pointcut.
- **Advisors and interceptors**: These objects are responsible for executing advice in the pointcuts.
- **Target or advised object**: This is the object that includes the main concern, and to which the cross-cutting concern is added (`Student` objects in the example).
- **Proxy**: This is an intermediate object that resides between the calling object and the target object. It's responsible for applying a chain of advisors or interceptors.

The following figure shows AOP participants and how they relate to the AOP framework:



Now that you've seen all of the AOP participants, you may ask what the *aspect* in Aspect-Oriented Programming refers to. As with many other AOP terms, this term originates from the old frameworks, such as AspectJ, which provide AOP functionality. As you will see later in this chapter, this term has come to Spring from release 2.0, when Spring provides AspectJ-style AOP implementation.

Let's apply AOP to our recording example. AOP allows us to remove recording code from the setter methods, and instead define the recording concern as a distinct class to record generally a property modification. We can tell the AOP framework to apply the recording concern before a setter method is called. The following figure shows the AOP approach to implementing the recording concern:



The process of AOP development can be summarized as follows:

1. Implement the advice: Remove the cross-cutting concern from the main concern and implement it as an individual class.
2. Determine pointcuts: Use Spring's built-in pointcuts or implement custom pointcuts to select target methods. Note that we do not involve the joinpoint objects. Each pointcut object notionally refers to a set of joinpoints with common characteristics. For example, a setter pointcut determines all setter methods, each of which is a joinpoint.
3. Create an advisor: Use Spring's built-in advisors to combine the pointcuts and the advice created in the previous steps.
4. Create a proxy: Use Spring's built-in classes to create a proxy object. Each proxy holds the target object with a set of advisors. The proxy applies the advisors, one-by-one, to the target object. The advisor is responsible for selecting the target methods of the target object (based on the pointcut) and applying the advice. However, each advice can be added directly to the proxy instead of the advisor.

Spring lets you use its IoC container and then configure all of the AOP objects declaratively in the Spring context. This means only the advice implementation is mandatory when you use AOP with Spring IoC.

Next, let's see how these concepts work in practice.

Implementing cross-cutting concerns with AOP

To use AOP, we can simply remove the recording concern from the `Student` class, as shown in the code below:

```
package com.packtpub.springhibernate.ch11;

public class Student {

    private int id;
    private String firstName;
    private String lastName;

    //zero-argument
    public Student() {
    }

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

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

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    //getter methods
    //hashCode() and equals() methods
}
```

Then, we must define the recording concern as an advice class, as shown in the following code:.

```
package com.packtpub.springhibernate.ch11;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

public class RecordingConcern implements MethodBeforeAdvice {

    private SetterInfoPrinter printer; //ideally initialized with
Spring IoC
```

```
        public void before(Method method, Object[] args, Object target)
                                                    throws Throwable {
            if((method.getName().startsWith("set"))
                              &&(target.getClass()==Student.class)) {
                String methodName = method.getName();
                Object newValue = args[0];
                Method getter= getterForSetter (Student.class,
                  methodName);
                Object oldValue = getter.invoke(target, args);
                printer.print(methodName, oldValue, newValue);
            }
        }
        public Method getterForSetter(Class clazz, String methodName) {
            try {
            return clazz.getMethod(methodName.replaceFirst("set", "get"));
            } catch (NoSuchMethodException e) {
                ;
            }
            return null;
        }
        public void setPrinter(SetterInfoPrinter printer){
            this.printer = printer;
        }
    }
```

As you can see, the `RecordingConcern` class implements the `org.springframework.aop.MethodBeforeAdvice` interface. This is a Spring interface that implements a *before advice*, called before method invocation. In our example, we need to call the recording concern to record the method name, the old value, and the new value, so our advice class implements `MethodBeforeAdvice`. As you will see, Spring also provides other types of advice, to be called after method invocation, or after an exception is thrown through the method invocation.

You can now apply `RecordingConcern` to an instance of the `Student` class through Spring AOP. The following code shows a simple class with a static `getStudent()` method. This method, a factory for `Student` instances that instantiates the `Student` instance, uses the Spring AOP API to apply the `RecordingConcern` advice to that instance, and returns the prepared `Student` instance:

```
    package com.packtpub.springhibernate.ch11;
    import org.springframework.aop.framework.ProxyFactory;
    public class StudentFactory {
        public static Student getStudent() {
            Student std = new Student();
            //create and configure the advice object
            RecordingConcern advice = new RecordingConcern();
            SetterInfoPrinter printer = new SetterInfoConsolePrinter();
            advice.setPrinter(printer);
            //create proxy
            ProxyFactory pf = new ProxyFactory();
            //introduce target & advice to proxy
            pf.addAdvice(advice);
            pf.setTarget(std);
            //use proxy object instead of actual target object
            Student proxy = (Student) pf.getProxy();
            return proxy;
        }
    }
```

If you use this factory class now to obtain `Student` instances, instead of directly instantiating the `Student` class, the advice is applied to the `Student` instances. Therefore, just replace:

```
    Student student = new Student();
```

with:

```
    Student student = StudentFactory.getStudent();
```

Although this works, this approach to AOP implementation is not effortless. In practice, Spring IoC and Spring AOP work together to reduce effort and create more effective code.

The next section discusses another, more practical example that relates to our sample educational system application. The example shows how Spring AOP and Spring IoC are integrated to simplify application development, and to produce neat, clean, effective code.

Using Spring AOP with Spring IoC: An example

In Chapter 10, we wrote a notification service that notifies the system administrator whenever a fatal error occurs in the system. Obviously, the notification service is a cross-cutting concern because the notification logic is scattered throughout many methods in many classes. Implementing the notification logic with OOP involves changing the other concerns.

Assume that we have a `StudentService` class in our application, which performs all student-relevant operations, including student registration, profile updating, and so on. A requirement of the application indicates that `StudentService` must notify the system administrator after an exception is thrown in the `StudentService` object.

Let's see how the code looks with OOP, and then explore how this concern can be implemented with AOP to overcome OOP's deficiencies.

Implementing the notification concern with OOP

Following Spring's best-practice implementations, we need an extra interface, `StudentServiceInf`, which exposes all of the `StudentService` methods, lets us configure `StudentService` in the IoC container, and implement IoC-style code. This `StudentService` class uses instances of `StudentDao` and `NotificationServiceInf`, which are provided by the IoC container through setter injection, to perform student-related operations and to notify the administrator when an exception is thrown, respectively.

The following code shows `StudentServiceInf`:

```
package com.packtpub.springhibernate.ch11;
import java.util.List;
public interface StudentServiceInf {
    public List getAllStudents() throws ServiceException;
    public Student getStudent(Long stdId) throws ServiceException;
    public Student saveStudent(Student std) throws ServiceException;
    public Student removeStudent(Student std) throws ServiceException;
    public Student updateStudent(Student std) throws ServiceException;
}
```

And here is the `StudentService` method as the implementation of `StudentServiceInf`. Note that I have intentionally omitted other business methods of the `StudentService` class to keep the example simple:

```
package com.packtpub.springhibernate.ch11;
import java.util.List;
import org.springframework.dao.DataAccessException;
public class StudentService implements StudentServiceInf {
    StudentDao studentDao;
    NotificationServiceInf notificationService;
    public List getAllStudents() throws ServiceException {
        try {
            return studentDao.getAllStudents();
        } catch (HibernateException e) {
            notificationService.notify(e.getMessage());
            throw new ServiceException(e);
        }
    }
    public Student getStudent(Long stdId) throws ServiceException {
        try {
            return studentDao.getStudent(stdId);
        } catch (HibernateException e) {
            notificationService.notify(e.getMessage());
            throw new ServiceException(e);
        }
    }
    public Student saveStudent(Student std) throws ServiceException {
        try {
            return studentDao.saveStudent(std);
        } catch (HibernateException e) {
            notificationService.notify(e.getMessage());
            throw new ServiceException(e);
        }
    }
    public Student removeStudent(Student std) throws ServiceException
{
        try {
            return studentDao.removeStudent(std);
        } catch (HibernateException e) {
           notificationService.notify(e.getMessage());
            throw new ServiceException(e);
        }
    }
    public Student updateStudent(Student std) throws ServiceException
{
        try {
            return studentDao.updateStudent(std);
        } catch (HibernateException e) {
            notificationService.notify(e.getMessage());
            throw new ServiceException(e);
        }
    }
    //other student-related business methods
    //setter methods for dependency injection
    public void setStudentDao(StudentDao studentDao) {
        this.studentDao = studentDao;
    }
    public void setNotificationService(NotificationServiceInf
notificationService){
        this.notificationService = notificationService;
    }
}
```

As you can see in the listings above, interacting with any method of `StudentDao` may throw a `HibernateException`, which is caught in the `StudentService` class, and then notifies the administrator. The notification process happens by invoking the `notify()` method of the `NotificationServiceInf` instance with a message as an argument. After that, a new `ServiceException` is thrown, to be propagated to the layer above, which uses the service layer. The following code shows the `ServiceException` class:

```
package com.packtpub.springhibernate.ch11;

public class ServiceException extends RuntimeException {
    public ServiceException() {
    }

    public ServiceException(String message) {
        super(message);
    }

    public ServiceException(Throwable cause) {
        super(cause);
    }

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

`ServiceException` is a runtime exception, so it may be handled by the calling layer or ignored.

We can simply throw a `RuntimeException` so there will no need to implement the `ServiceException` class. However, it would be a good idea to differentiate typical `RuntimeExceptions` that may occur everywhere, and the exceptions which are originated with `HibernateExceptions`.

The code we have written so far mixes two distinct concerns: student-relevant operations and administrator notification. All methods have code to handle `HibernateException` in the same way, by notifying the administrator and throwing a new `ServiceException`.

Administrator notification is a cross-cutting concern because this is a distinct logic outside of the main responsibility of `StudentService` and scattered over all methods.

Let's refactor our implementation with Spring AOP.

Implementing notification concern with AOP

With AOP, `StudentService` is modularized into its main responsibility (student-related operations) and the notification concern. Two modules then can be mixed together by the Spring AOP framework when a `StudentService` method is called.

The following code shows the `StudentService` class, with the notification concern code omitted:

```
package com.packtpub.springhibernate.ch11;

import java.util.List;

import org.hibernate.HibernateException;

public class StudentService implements StudentServiceInf {
    StudentDao studentDao;

    public List getAllStudents() throws HibernateException {
        return studentDao.getAllStudents();
    }

    public Student getStudent(Long stdId) throws HibernateException{
        return studentDao.getStudent(stdId);
    }

    public Student saveStudent(Student std) throws HibernateException{
        return studentDao.saveStudent(std);
    }

    public Student removeStudent(Student std) throws
HibernateException{
        return studentDao.removeStudent(std);
    }

    public Student updateStudent(Student std) throws
HibernateException {
        return studentDao.updateStudent(std);
    }

    //other student-related business methods

    public void setStudentDao(StudentDao studentDao) {
        this.studentDao = studentDao;
    }

    public void setNotificationService(NotificationServiceInf
notificationService){
        this.notificationService = notificationService;
    }
}
```

In the following code, the notification concern is implemented as another class, `NotificationThrowsAdvice`, to be called by the Spring AOP framework after a `HibernateException` is thrown:

```
package com.packtpub.springhibernate.ch11;
import org.hibernate.HibernateException;
import org.springframework.aop.ThrowsAdvice;
public class NotificationThrowsAdvice implements ThrowsAdvice {
    NotificationServiceInf notificationService ;//initialized via IoC
    public void afterThrowing(HibernateException ex) throws Throwable
{
        notificationService.notify(ex.getMessage());
        throw new ServiceException(ex);
    }
    public void setNotificationService(NotificationServiceInf
notificationService){
        this.notificationService = notificationService;
    }
}
```

As you can see, this class implements the `org.springframework.aop.ThrowsAdvice` interface. This interface is just a marker interface, not enforcing to implement any method in the implemented class. This interface is a Spring interface to implement advice called after a particular exception is thrown in the target objects. The `NotificationThrowsAdvice` class has an `afterThrowing()` method, which takes a `HibernateException` object as an argument. Spring calls this method after a `HibernateException` is thrown in the target methods. As we will discuss, this class can have other `afterThrowing()` methods to handle exceptions of different types. For example, the `NotificationThrowsAdvice` class may have an additional `afterThrowing()` method with the following signature to handle exceptions of type `java.io.IOException` when they are thrown in the target object:

```
public void afterThrowing(java.io.IOException ex) throws Throwable {
    notificationService.notify(ex.getMessage());
    throw new ServiceException(ex);
}
```

Now that the advice has been implemented, it can be applied either programmatically or declaratively to the target objects. In the previous code, you saw how an advice can be applied programmatically. However, it would be more flexible to use the Spring IoC container with Spring AOP to configure and apply the advice to the target object. This mechanism lets you apply the advice with minimal Java code, as well you can modify or remove the advice or insert another advice without having to change the Java code.

Here are the bean definitions for the `NotificationService` and `NotificationThrowsAdvice` objects in the Spring context:

```
<bean id="notificationThrowsAdvice"
        class="com.packtpub.springhibernate.ch11.
NotificationThrowsAdvice">
    <property name="notificationService">
        <ref local="notificationService"/>
    </property>
</bean>
<bean id="notificationService"
        class="com.packtpub.springhibernate.ch11.NotificationService">
    <property name="notifier">
        <ref local="notifier"/>
    </property>
    <property name="retryOnFail">
        <value>true</value>
    </property>
</bean>
<bean id="notifier" class="com.packtpub.springhibernate.ch11.
EmailNotifier">
</bean>
```

The configured advice can now be applied to any number of objects, including the `StudentService` instance, managed by the Spring container. Here's how you would apply the implemented advice to the configured `StudentService` object:

```
<bean id="studentServiceTarget"
      class="com.packtpub.springhibernate.ch11.StudentService">
  ...
</bean>
<bean id="studentService"
      class="org.springframework.aop.framework.autoproxy.
BeanNameAutoProxyCreator">
    <property name="beanNames">
        <value>studentServiceTarget</value>
    </property>
    <property name="interceptorNames">
        <list>
            <value>notificationThrowsAdvice</value>
        </list>
    </property>
</bean>
```

You can look up the IoC container to use `StudentService` object as follows:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
                      "com/packtpub/springhibernate/ch11/beans.xml");
StudentServiceInf studentService =
                    (StudentServiceInf)ctx.getBean("studentService");
```

Note that the bean with the name `studentService` refers to an object of `org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator`, instead of `StudentService`, which works as a proxy for the `StudentService` instance configured by the IoC container. This proxy is responsible for applying all advice specified through the `interceptorNames` property. The `BeanNameAutoProxyCreator` instance also uses another property with the name `beanNames`, referring to the target objects to which the advice should be applied (`studentServiceTarget` in our example, which refers to the configured `StudentService` object).

You have many options when applying a Spring advice to target objects. However, the result will be the same. Don't worry about the details. We will discuss those later in this chapter.

> In this example, we have not specified the target methods to which the advice should be applied. In such cases, the advice is applied to all methods of the target object. However, as you will see, Spring lets you select the target methods through pointcut objects.

Now that you have sufficient background in AOP implementation, let's explore the Spring AOP framework in more detail.

Spring's AOP framework

Spring provides a rich and powerful AOP framework. In this section, we will discuss the different kinds of advice Spring provides, how advice can join with a target object through joinpoints, and how the joinpoints can be accumulated as pointcuts. Finally, we will look at the different proxy generator classes Spring provides.

Advice

The most important part of AOP is advice. An advice defines the custom behavior or reaction that is accomplished at joinpoints. Spring wraps a method invocation in a chain of interceptors behind the proxy object. Each interceptor is responsible for applying an advice.

Spring allows us to use different kinds of advice, and each is useful for a particular case. The following table lists these advice types:

Advice Type	Description
Around advice	This type of advice provides an opportunity to implement custom code to be executed before or after method invocation. This advice is an implementation of the `org.aopalliance.intercept.MethodInterceptor` interface. This interface does not belong to Spring. Instead, it is part of the AOP Alliance API (discussed in this chapter).
Before advice	This type of advice is exposed through the `org.springframework.aop.BeforeAdvice` interface. It lets us implement an advice to be executed before method invocation.
After returning advice	This type of advice is exposed through the `org.springframework.aop.AfterReturningAdvice` interface. It lets us implement an advice to be executed after normal execution of the method when no exception is thrown.
Throws advice	This type of advice is exposed through the `org.springframework.aop.ThrowsAdvice` interface. It lets us implement an advice to be executed after abnormal execution of the method when an exception is thrown.

Additionally, you can define a custom advice in situations when none of the types listed in the table above suits your needs. Let's discuss how each of these advice types is implemented and used.

The AOP alliance

The AOP Alliance is an API that includes a small set of interfaces and provides a common foundation for different AOP implementations. This means different AOP implementations use this API to provide reusability for advice or other AOP components. For the most part, Spring defines its own kinds of advice based on this API. It allows us to reuse the implemented Spring advice with other AOP frameworks that support the AOP Alliance API. The most significant interface in this API is `org.aopalliance.intercept.MethodInterceptor`, which is used to define an advice.

The AOP Alliance libraries are shipped with the Spring distribution. You can check out this API's source files at `http://aopalliance.sourceforge.net`.

Around advice

The **around advice** is a basic advice that lets us implement code to be executed before, after, or before and after the method invocation. This advice is provided by the AOP Alliance API through the `MethodInterceptor` interface, shown in the following code:

```
package org.aopalliance.intercept;

import org.aopalliance.intercept.Interceptor;
import org.aopalliance.intercept.MethodInvocation;

public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

This interface includes only one `invoke()` method, which takes an object of `org.aopalliance.intercept.MethodInvocation` as an argument, and uses `Object` as the return type. The `MethodInvocation` argument exposes the target method being invoked with its arguments and the target joinpoint. The return value of the `invoke()` method represents the result of the target method invocation. To execute code before or after method execution, you need to implement custom code before or after the `MethodInvoation.proceed()` call, respectively. The following code shows an example of `MethodInterceptor` implementation, which measures and logs the time taken by the target method invocation:

```
package com.packtpub.springhibernate.ch11;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class PerformanceInterceptor implements MethodInterceptor {
    private static Logger logger =
             Logger.getLogger(PerformanceInterceptor.class.getName());

    public Object invoke(MethodInvocation invocation) throws Throwable
{
        long start = System.currentTimeMillis();
        try {
            Object result = invocation.proceed();
            return result;
        }
        finally {
            long end = System.currentTimeMillis();
            long timeMs = end - start;
```

```
                logger.log(Level.INFO, "Method: " +
                           invocation.getMethod().getName() +
                           " took: " + timeMs + "ms.");
            }
        }
    }
```

The call of the `proceed()` method on the `MethodInvocation` instance calls the next associated interceptor in the chain. If no other interceptor is associated with the joinpoint, the joinpoint itself is called. You can also throw an exception if you do not want the target method to be called. This mechanism is particularly useful when you want to control the client's access to the target method.

Before advice

Spring provides the `org.springframework.aop.MethodBeforeAdvice` interface, which can be used to implement a method interceptor that is always executed before the joinpoint. The following code shows the `MethodBeforeAdvice` interface, extending the generic `org.springframework.aop.BeforeAdvice` interface used for any kind of joinpoint:

```
package org.springframework.aop;

import java.lang.reflect.Method;

public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method method, Object[] args, Object target) throws
Throwable;
}
```

The only `before()` method in this interface takes three arguments of type `java.lang.reflect.Method`, `Object[]`, and `Object`, representing the target method being called, the target method arguments, and the target object, respectively. The return type of this method is `void`, which means this advice does not allow changing of the method's return value. However, the `before()` method throws `Throwable` exception, which lets you throw an exception to break the interceptor chain.

The following code shows a simple before advice, which logs the method invocation's starting time:

```
package com.packtpub.springhibernate.ch11;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
```

```
public class LogBeforeAdvice implements MethodBeforeAdvice {
    private static Logger logger =
                  Logger.getLogger(LogBeforeAdvice.class.getName());

    public void before(Method method, Object[] args, Object target)
                                              throws Throwable  {
        long start = System.currentTimeMillis();
        logger.log(Level.INFO, method.getName()+" starts at " +
start);
    }
}
```

After returning advice

The `org.springframework.aop.AfterReturningAdvice` interface, shown in the following code, lets you implement an advice to be executed after the method has finished successfully, without throwing an exception:

```
package org.springframework.aop;

import java.lang.reflect.Method;

public interface AfterReturningAdvice extends AfterAdvice {
    void afterReturning(Object returnValue,
                        Method method,
                        Object[] args,
                        Object target) throws Throwable;
}
```

The `afterReturning()` method of this interface takes instances of `Object`, `java.lang.reflect.Method`, `Object[]`, and `Object` as arguments, which represent, the return value of the target method, the target method, the target method's arguments, and the target object, respectively. As with the `before()` method in the `BeforeMethodAdvice`, the `afterReturning()` method returns `void`; and it does not allow changing of the return value.

The following code shows an implementation of `AfterReturningAdvice`, which logs the finishing time of the method invocation:

```
package com.packtpub.springhibernate.ch11;

import org.springframework.aop.AfterReturningAdvice;
import java.lang.reflect.Method;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class LogAfterAdvice implements AfterReturningAdvice {
    private static Logger logger = .
                Logger.getLogger(LogAfterAdvice.class.getName());
    public void afterReturning(Object object, Method method,
                               Object[] args, Object target)
                                                throws Throwable
{
        long end = System.currentTimeMillis();
        logger.log(Level.INFO, method.getName()+ " ends at " + end);
    }
}
```

This `AfterReturningAdvice` is only applied to target methods that have finished successfully without throwing an exception. The implemented advice in the code above only logs the finishing time of successful method execution.

Throws advice

The final type of advice provided by Spring is exposed through the `org.springframework.aop.ThrowsAdvice` interface, shown in the following code:

```
package org.springframework.aop;

public interface ThrowsAdvice extends AfterAdvice {
}
```

This allows you to implement an advice that is applied when the target method throws an exception. Note that the `ThrowsAdvice` interface does not expose any method to be implemented. You have an option to implement any number of methods with the following signature:

```
afterThrowing([Method, args, target,] Throwable)
```

`Method`, `args`, and `target` represent, the target method being called, the target method's arguments, and the target object, respectively. `Throwable` exception represents a subclass of the `java.lang.Throwable` class that is thrown in the target method and causes this advice to be called. In all of the methods that implement this signature, only `Throwable` is mandatory. The other arguments are optional, meaning you can implement all three of the other arguments, or omit them all.

The following code shows a `ThrowsAdvice` that logs the type of the exception thrown by the target method:

```
package com.packtpub.springhibernate.ch11;
import org.springframework.aop.ThrowsAdvice;
import java.lang.reflect.Method;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
public class LogsThrowsAdvice implements ThrowsAdvice {
    private static Logger logger =
                Logger.getLogger(LogsThrowsAdvice.class.getName());
    public void afterThrowing(Method method,
                              Object[] args,
                              Object target,
                              Exception ex) throws Throwable {
        logger.log(Level.INFO, ex.getClass()+" thrown in " + method);
    }
}
```

The `afterThrowing()` method can be overloaded with different types of exception as arguments to handle different exceptions differently. The following code shows a throws advice that logs `java.io.IOException`, but notifies the system administrator when an `org.hibernate.HibernateException` is thrown:

```
package com.packtpub.springhibernate.ch11;
import org.springframework.aop.ThrowsAdvice;
import java.io.IOException;
import org.hibernate.HibernateException;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
public class CustomExceptionsThrowsAdvice implements ThrowsAdvice {
    private static Logger logger =
       Logger.getLogger(CustomExceptionsThrowsAdvice.class.getName());
    NotificationServiceInf notificationService;
    public void afterThrowing(HibernateException ex) throws Throwable
{
        notificationService.notify(ex.getMessage());
    }
    public void afterThrowing(IOException ex) throws Throwable {
        logger.log(Level.INFO, "An IOException occured");
    }
    public void setNotificationService(NotificationServiceInf
notificationService){
        this.notificationService = notificationService;
    }
}
```

Pointcuts

As mentioned earlier, a **pointcut** is an object comprising a set of joinpoints, which identifies where advice applies. Here are some examples of pointcuts:

- All methods of a class
- All methods of a class with names that start with `do`
- All getter or setter methods of a class
- All methods returning `void`
- All methods that do not have any arguments

All of these pointcuts are called **static** because they are identified based on static information before runtime. However, it is possible to determine a pointcut based on runtime information. For instance, as you will see with an upcoming example, you can define a pointcut that identifies all methods that are passed `null` as their first argument at runtime. Such pointcuts are called **dynamic**.

Pointcuts only specify the target methods, regardless of the target object to which the methods may belong.

The following table briefly describes all pointcuts shipped with Spring:

Spring Pointcut	**Description**
Setter and getter pointcut	The `org.springframework.aop.support.Pointcuts` class has two `final static` members, which determine pointcuts for the getter and setter methods.
Name matched pointcut	This type is presented through the `org.springframework.aop.support.NameMatchMethodPointcut` class, allowing us to select the target methods based on their names. With this pointcut, regular expressions cannot be used for method names.
Regular expression pointcut	This type is provided through `org.springframework.aop.support.JdkRegexpMethodPointcut` class, allowing us to select the target methods with the Java 1.4 regular expression API. The target method name(s) are specified with regular expressions through the `pattern` or `patterns` properties, respectively.

Spring Pointcut	Description
Static matcher pointcut	This type is provided through the `org.springframework.aop.support.StaticMethodMatcherPointcut` class to select target methods based on static information, such as the method names, the number of arguments, the arguments' types, and so on.
Dynamic matcher pointcut	This type, provided through the `org.springframework.aop.support.DynamicMethodMatcherPointcut` class, allows us to select target methods based on the argument values at runtime, in addition to static information provided by the static matcher pointcut.

As you will see, the pointcuts listed in the table above may be used for programmatic proxy creation, as well as configuration in a Spring factory through setter injection.

The sections that follow explain each pointcut and how it is used, both programmatically (through the Java code) and declaratively (through the Spring context).

Setter and getter pointcut

The `org.springframework.aop.support.Pointcuts` class defines two static members for declaring a setter or getter method as a pointcut. These members are: `GETTERS` and `SETTERS`. Each specifies JavaBean-style getter methods and setter methods, respectively, of any class. For instance, you may create a pointcut object programmatically as follows:

```
Pointcut setterPointcut = org.springframework.aop.support.Pointcuts.
SETTERS;
Pointcut getterPointcut = org.springframework.aop.support.Pointcuts.
GETTERS;
```

Alternatively, you may define a pointcut object declaratively in the Spring context as follows:

```
<bean id="getterPointcut"
      class="org.springframework.aop.support.Pointcuts.GETTERS"/>
```

This pointcut specifies all getter methods of the target object.

Name matched pointcut

The `org.springframework.aop.support.NameMatchMethodPointcut` class allows us to create a pointcut that selects target methods with a specific name. This class provides three methods for specifying the target methods' names:

```
public void setMappedName(String methodName)
public void setMappedNames(String[] methodNames)
public NameMatchMethodPointcut addMethodName(String s)
```

`setMappedName()` and `setMappedNames()` take a `String` or an array of `String`s that represents the name or the names of target method or methods, respectively. Additionally, the `addMethodName()` method allows us to specify the target method names multiple times. Here is an example of programmatic creation of `NameMatchMethodPointcut`, which specifies all of the methods with the name `saveStudent` or `updateStudent` in the target object:

```
NameMatchMethodPointcut nmpc = new NameMatchMethodPointcut();
nmpc.setMappedNames(new String[]{"saveStudent", "updateStudent"});
```

The same object may be created through the `addMethodName()` as follows:

```
NameMatchMethodPointcut nmpc =
        new NameMatchMethodPointcut().addMethodName("saveStudent").
                                         addMethodName("updateStudent");
```

Alternatively, you may configure an instance declaratively in the Spring context, as follows:

```
<bean id="studentDaoPointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedNames">
        <list>
            <value>saveStudent</value>
            <value>updateStudent</value>
        </list>
    </property>
</bean>
```

Regular expression pointcuts

This type of pointcut lets you specify the target method names with Java regular expressions. A regular expression allows you to describe the names of the target methods with strings. These strings may use wildcards to match or exclude groups of characters and markers that are required for matching in particular places. This type of pointcut is provided through the `org.springframework.aop.support.JdkRegexpMethodPointcut` class. The following are the most useful methods of this class:

```
public void setPattern(String pattern)
public void setPatterns(String[] patterns)
public void setExcludedPattern(String pattern)
public void setExcludedPatterns(String[] patterns)
```

The `setPattern()`and `setPatterns()` methods allow us to specify, a regular expression or an array of regular expressions that the target method names should match. Two other methods, `setExcludedPattern()` and `setExcludedPatterns()`, let us determine a regular expression and an array of regular expressions that the target method names should *not* match.

[Refer to `http://java.sun.com/docs/books/tutorial/essential/regex` for more information about the regular expression syntax and wildcards.]

Here is an example of using `JdkRegexpMethodPointcut` to select all methods whose names end with either `Student` or `Course`:

```
JdkRegexpMethodPointcut pc = new JdkRegexpMethodPointcut();
pc.setPatterns(new String[]{".*Student", ".*Course"});
```

This is the declarative approach for selecting the same methods:

```
<bean  class="org.springframework.aop.support.
JdkRegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*Student</value>
            <value>.*Course</value>
        </list>
    </property>
</bean>
```

Static matcher pointcut

Using this pointcut, you can select target methods based on static information of target methods and target classes, the information that can be estimated at compile time. This information includes the package name, class name, method name, the type of method's arguments, and so on. This type of pointcut is provided through `org.springframework.aop.support.StaticMethodMatcherPointcut`. To use it, create a subclass of `StaticMethodMatcherPointcut`, either as an individual class or on the fly (as you will see in the example), and implement its abstract `matches()` method. The `matches()` method has this signature:

```
public boolean matches(Method method, Class targetClass) {
    //select the target method
}
```

The `matches()` method takes two arguments: an instance of `java.lang.reflect.Method` and a `java.lang.Class` instance. The first argument specifies the selected target method and the second argument determines the class to which the target method belongs. Here's an example:

```
public static Pointcut myStaticPointcut = new
StaticMethodMatcherPointcut() {
    public boolean matches(Method method, Class targetClass) {
        String className = targetClass.getName();
        String methName = method.getName();
        if((className.indexOf("Test")==-1)&&(methName.
indexOf("Student")>-1)){
            return true;
        }
        return false;
    }
};
```

This pointcut selects the methods whose name includes the word `Student`, and whose classname does not contain the word `Test`.

Dynamic Matcher Pointcut

A dynamic matcher pointcut lets you create pointcuts that select target methods based on -information at runtime, which includes the actual values passed to the methods at runtime. This kind of pointcut is a superset of static matcher pointcut that was already discussed. This means you can use dynamic matcher pointcut to select targets based on the static information, estimated at compile time, and dynamic information at runtime. To implement this type of pointcut, create a subclass of `org.springframework.aop.support.DynamicMethodMatcherPointcut` and override its abstract `matches()` methods. The `matches()` methods in the class have these signatures:

```
public boolean matches(Method method, Class targetClass) {
    // select the target method
}
public boolean matches(Method method, Class targetClass, Object[]
args) {
    // select the target method
}
```

The two-argument `matches()` method is the same as you already saw for the static matcher pointcut, with the same functionality. The other variant of this method takes an array of `Object` as an extra argument, representing the arguments of the target method. You may evaluate these arguments to check whether the current method satisfies your criteria. Note that the three-argument `matches()` method is only called for target methods that satisfy the two-argument method.

As with the static matcher pointcut, you may create an individual subclass, or implement a subclass on the fly. Here is an example:

```
public static Pointcut myDynamicPointcut = new
DynamicMethodMatcherPointcut() {
    public boolean matches(Method method, Class targetClass) {
        String className = targetClass.getName();
        String methName = method.getName();
        if((className.indexOf("Test")==-1) && (methName.
          indexOf("Student")>-1)) {
            return true;
        }
        return false;
    }
    public boolean matches(Method method, Class targetClass,
     Object[] args) {
        if ((args.length > 0) && (args[0] != null)) {
            return true;
        }
        return false;
    }
};
```

This pointcut selects methods whose name includes the word `Student`, that have at least one argument (the first of which is not `null`), and whose classname does not contain the word `Test`.

Pointcut composition

It is possible to combine pointcuts to create a complex pointcut. The resulting pointcut may use a combination of logic to select the target methods. The `org.springframework.aop.support.Pointcuts` class provides the `union()` and `intersection()` methods, which can be used for this purpose as follows:

```
public static Pointcut union(Pointcut p1, Pointcut p2)
public static Pointcut intersection(Pointcut p1, Pointcut p2)
```

The `union()` result of two pointcuts is a pointcut that selects any method which satisfies at least one of the input pointcuts. The `intersection()` result is a pointcut that selects only methods matched by both pointcuts.

Unfortunately, there is no way to compose pointcuts declaratively.

Advisor

Spring uses an extra object, called an **advisor**, to apply advice to the target methods. This object wraps an advice, and a pointcut and is responsible for applying the advice to the pointcut.

To create an advisor, you need to implement the `org.springframework.aop.Advisor` interface. Spring provides a set of convenient implementations, which relieve you from having to implement a new advisor. A common implementation that can be used with any Spring advice type is `org.springframework.aop.support.DefaultPointcutAdvisor`. A `DefaultPointcutAdvisor` instance may be created programmatically with an already created pointcut and advice as follows:

```
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut,
advice);
```

Alternatively, you can create the advisor declaratively in the Spring context as follows:

```
<bean name ="advisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
        <ref local="pointcut" />
    </property>
    <property name="advice">
        <ref local="advice" />
    </property>
</bean>
```

You may use an advice without specifying any pointcut. When you do, the advice is applied to all methods of the target object. Additionally, Spring lets you use an advice without an advisor. The advice is automatically wrapped in an advisor object and applied to all methods of the target object.

Proxy configuration and creation

The proxy object starts the invocation chain of advisors, which ends with the target object. Like many other AOP components, Spring allows us to create proxy objects both programmatically and declaratively. To create a proxy object, you need to configure it with any already defined Spring proxy classes. There are proxy factories to create a proxy object whenever it is needed. The following table lists the essential proxy factory types:

Proxy Factory Type	**Description**
`ProxyFactory`	This is used to create proxy objects programmatically, without an IoC container.
`ProxyFactoryBean`	This is used to configure and create proxy objects in the IoC manner with the Spring application context.
`AbstractAutoProxyCreator`	This abstract class defines a proxy factory used to implement automatic proxying.
`TransactionProxyFactoryBean`	This is a specialization of the `ProxyFactoryBean` used to create a transactional proxy, discussed in Chapters 12 and 13.

Let's see how each factory can be used to create proxy objects.

Using ProxyFactory

Spring provides the `org.springframework.aop.framework.ProxyFactory` class to create proxy objects programmatically. At the start of this chapter, you saw a simple example. The following table describes some of its useful methods:

Method	Description
`void addAdvice(Advice advice)`	This method takes an `org.aopalliance.aop.Advice` object as its argument, and adds it to the tail of the advice chain.
`void addAdvisor(Advisor advisor)`	This method adds an advisor of type `org.springframework.aop.Advisor`.
`void addInterface(Class aClass)`	This method adds a new proxied interface. If it is not used, the AOP framework automatically proxies all interfaces implemented by the target.
`void setTarget(Object o)`	This method sets the given object as the target to which all advice is applied.
`java.lang.Object getProxy()`	This method creates and returns a new proxy according to the settings used for this factory.

Using ProxyFactoryBean

The most commonly used proxy factory in Spring's AOP framework is `ProxyFactoryBean`. It allows us to configure the proxy factory in the Spring context as a bean of type `org.springframework.aop.framework.ProxyFactoryBean`.

To use this proxy factory, you need to configure the required objects, such as advisors, advice, and pointcuts, as individual beans in the Spring context. Then, configure the `ProxyFactoryBean` instance with its required properties, which are explained in the following table:

Property Name	Meaning
`proxyInterfaces`	This property specifies the interfaces implemented by the target object, which should be proxied.
`interceptorNames`	This property indicates a list of names of advice, advisors, or interceptors linking the advice/advisor/interceptor chain together.
`target`	This property determines the target object to which the advisor/interceptor chain is applied.

Here is an example of `ProxyFactoryBean` configuration:

```
<bean id="studentService"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>com.packtpub.springhibernate.ch11.StudentServiceInf</
value>
    </property>
    <property name="target">
        <ref local="studentService"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>advisor</value>
            <value>advice</value>
        </list>
    </property>
</bean>
```

The advice, advisors, or interceptors specified through the `interceptorNames` property are applied based on the order in which they have been specified.

Assembling the AOP components

We can now arrange all of the AOP components and see what they look like. The following code shows the Spring context that includes all of the AOP component definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd">
    <!--advice declaration -->
    <bean id="advice" class="com.packtpub.springhibernate.ch11.
LogAfterAdvice">
    </bean>
    <!-- pointcut declaration -->
    <bean  id="pointcut"
           class="org.springframework.aop.support.
JdkRegexpMethodPointcut">
        <property name="pattern">
            <value>.get*</value>
        </property>
    </bean>
    <!--advisor declaration, combined of the advice and pointcut-->
    <bean name ="advisor"
            class="org.springframework.aop.support.
DefaultPointcutAdvisor">
        <property name="pointcut">
            <ref local="pointcut" />
        </property>
        <property name="advice">
            <ref local="advice" />
        </property>
    </bean>
    <bean id="notifier"
            class="com.packtpub.springhibernate.ch11.EmailNotifier">
    </bean>
    <bean id="notificationService"
            class="com.packtpub.springhibernate.ch11.
NotificationService">
        <property name="notifier">
            <ref local="notifier"/>
        </property>
        <property name="retryOnFail">
            <value>true</value>
        </property>
    </bean>
    <!-- target object declaration -->
    <bean id="studentServiceTarget"
            class="com.packtpub.springhibernate.ch11.StudentService">
        <property name="studentDao">
            <bean class="com.packtpub.springhibernate.ch11.
StudentDaoImpl"/>
        </property>
        <property name="notificationService">
            <ref local="notificationService" />
        </property>
    </bean>
    <!-- proxy object configuration -->
    <bean id="studentService"
            class="org.springframework.aop.framework.
ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.packtpub.springhibernate.ch11.
StudentServiceInf</value>
        </property>
        <property name="target">
            <ref local="studentServiceTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>advisor</value>
            </list>
        </property>
    </bean>
</beans>
```

As you can see, the configuration file declares all of the AOP components, such as advice, pointcut, advisor, target, and the proxy object. In the code above, all of these components are highlighted.

Moving to Spring 2.x's AOP

So far in this chapter, we have discussed only AOP implementations that were introduced in Spring 1.2.x. Spring 2.0 comes with a different AOP framework. The new AOP framework is fully integrated with AspectJ. AspectJ (`http://www.eclipse.org/aspectj`), an extension to the Java language, provides aspect-oriented programming features. The most valuable feature of AspectJ, in addition to its simplicity and maturity, is its pointcut expression language, which lets us express pointcuts with simple string literals.

With its AspectJ support, Spring 2.0 adds a new term to its AOP terminology: **aspect**. As you will see, an aspect is basically an advisor which encapsulates a pointcut and an advice.

[With Spring 2.0, you can still use the classic AOP features provided by Spring 1.2.x.]

The most significant improvements of the Spring 2.0 AOP can be summarized as follows:

- **Easier AOP configuration**: Spring 2.*x* introduces new schema support for defining aspects backed by aspect definition with the general `<bean>` element.
- **Support for AspectJ integration**: Spring 2.*x* takes advantage of AspectJ, a powerful, full-blown AOP framework for Java. Additionally, Spring 2.*x* supports aspects defined using the `@AspectJ` annotations. These aspects, which can be shared between AspectJ and Spring AOP, require only simple configuration.
- **Support for the bean name pointcut element**: Spring 2.5 introduces support for the `bean(...)` pointcut element, matching specific named beans according to Spring-defined bean names.

Let's look at AOP configuration with the AOP schema, and explore how to use new features of the Spring AOP framework.

Classic AOP development with Spring allows us to implement and use only four types of advice: `MethodInterceptor`, `MethodBeforeAdvice`, `AfterReturningAdvice`, and `ThrowsAdvice`. Spring 2.0 has added another kind of advice, called **after advice**, which is called when the joinpoint method returns successfully, or breaks with an exception.

Moreover, with Spring 2.x you must implement a subclass of a Spring or an AspectJ-specific class to define an advice. However, you can turn any method of any class into an advice method. This means that any ordinary class can be an advice. To keep things simple, let's assume that we are going to apply a simple advice to pointcuts. The advice prints out a message in the application console when the joinpoint method executes. The following code shows the `MessageWriter` class that does this with its `writeMessage()` method:

```
package com.packtpub.springhibernate.ch11;

public class MessageWriter {
    public void writeMessage() {
        System.out.print("Simple Message");
    }
}
```

Let's see how this advice can be configured in the Spring context and applied to the joinpoint method.

To use the new features of Spring 2.0 AOP, you need the AspectJ library, distributed with Spring, in your application classpath.

AOP configuration with the AOP schema

To use Spring 2.0 AOP's new features, you must declare the AOP namespace (`aop`) with the actual schema location (`http://www.springframework.org/schema/aop/spring-aop-2.5.xsd`) in the Spring context, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
                 http://www.springframework.org/schema/beans
                 http://www.springframework.org/schema/beans/
                      spring-beans-2.5.xsd
```

```
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/
                             spring-aop-2.5.xsd">
</beans>
```

This configuration lets you use proprietary AOP XML elements in the Spring configuration file. However, you still have an option to configure AOP components classically, as discussed with Spring 1.2.x.

To configure AOP components, add the `<aop:config>` element to the Spring configuration file, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
                    http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
                    http://www.springframework.org/schema/aop
                    http://www.springframework.org/schema/aop/spring-
aop-2.5.xsd">
    <aop:config>
    </aop:config>
</beans>
```

All of the AOP components are configured through nested elements within the `<aop:config>`. These elements are as follows:

- `<aop:aspect>`
- `<aop:pointcut>`
- `<aop:before>`
- `<aop:after-returning>`
- `<aop:after-throwing>`
- `<aop:after>`
- `<aop:around>`

The subsequent sections discuss how each of these elements is used to declare an AOP component.

Defining aspects

The `<aop:aspect>` element is used within the `<aop:config>` element to define an aspect. Each `<aop:aspect>` element comes with two attributes, `id` and `ref`. The `id` attribute specifies an identifier for the aspect, and `ref` refers to another bean defined in the configuration file. The bean referred by the `ref` attribute is a simple Java object defined as an ordinary bean inside the configuration file. Actually, this bean specifies the advice object. In our example, this bean represents the `MessageWriter` object, and can be configured as follows:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>
<aop:config>
    <aop:aspect id="aspect1" ref="messageWriter">
      ...
    </aop:aspect>
</aop:config>
```

This code simply defines an aspect with the `aspect1` identifier. This aspect refers to the ordinary `messageWriter` bean as the advice. This aspect configuration won't do anything. We must complete the `<aop:aspect>` configuration with pointcuts and advice. Let's see how the other elements come into play.

Defining pointcuts

The `<aop:pointcut>` element defines a pointcut. A pointcut can be defined either inside an `<aop:config>` element, to be shared across several aspects and advisors, or directly inside an `<aop:aspect>` element, to be used proprietarily for the aspect specified through the `<aop:aspect>`.

The `<aop:pointcut>` element takes `id` and `expression` as attributes. `id` assigns an identifier to the pointcut, which lets you refer to the pointcut when the aspect is defined. The `expression` attribute determines the joinpoints with which the pointcut is associated through the AspectJ pointcut language. The following code shows an example of a pointcut definition directly within the `<aop:config>` element:

```
<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
ch11.*Service.*(..))"/>
</aop:config>
```

This pointcut selects all of the methods defined in the service classes, the classes located in the `com.packtpub.springhibernate.ch11` package whose names end with `Service`.

The AspectJ language's rules are easy and simple. They can be summarized as follows:

- **Pointcut designators**: The most significant designators are `execution()` and `within()`. The `execution()` method allows matching execution joinpoints. The `within()` method limits matching to joinpoints with certain types. For example, `execution(public * *(..))` selects all public methods, and `within(com.service.*)` selects all methods located in classes in the `com.service` package.
- **Pointcut compositors**: These are the AND (`&&`), OR(||), and NOT (`!`) operators that narrow down the selection. For example, `execution(* get(..))&&execution(* set(..))` selects all setter and getter methods.
- **Visibility selectors**: These are `public` and `protected` operators that narrow down the selection to only public or protected methods. For example, `execution(public * *(..))` selects all public methods.
- **Package, class, and method selectors**: These selectors are the full or partial name of a package, a class, and a method that narrow down the selection to specific packages, classes, and methods. For example, `execution (* set*(..))` selects all methods starting with the word `"set"` in any class and in any package, and `* com.packtpub.springhibernate.ch11.*Service.*(..)` selects all methods of the service classes.
- **Annotations**: These are annotations in Java code to select methods. For example, `@target(org.springframework.transaction.annotation.Transactional)` selects all methods in the target object annotated with `@Transactional`.

For a full discussion of AspectJ's pointcut language, see the AspectJ Programming Guide at `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

Defining an advice

As mentioned earlier, with Spring 2.0 you can define five kinds of advice. Each advice is defined as an individual element within `<aop:config>`. The following subsections discuss how to implement and define these kinds of advice.

Before advice

A before advice executes before a joinpoint method's execution. This advice is defined through the `<aop:before>` element within `<aop:aspect>`. Here is an example:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>
<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
ch11.*Service.*(..))"/>
    <aop:aspect id="aspect1" ref="messageWriter">
         <aop:before pointcut-ref="servicesPointcut"
method="writeMessage" />
    </aop:aspect>
</aop:config>
```

The `pointcut-ref` attribute refers to a configured pointcut. The `method` attribute specifies the method of the `messageWriter` bean to be used as the advice method. The `<aop:before>` element turns the `writerMessage()` method on the `messageWriter` bean into an advice method, so the `MessageWriter.writeMessage()` method is invoked before any method matching the joinpoints is executed.

After returning advice

An after returning advice executes after the method successfully returns without throwing an exception. The `<aop:after-returning>` element is used within `<aop:aspect>` to define an after returning advice. Here's an example:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>

<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
ch11.*Service.*(..))"/>

    <aop:aspect id="aspect1" ref="messageWriter">
         <aop:after-returning pointcut-ref="servicesPointcut"
method="writeMessage"/>

    </aop:aspect>
</aop:config>
```

The after returning advice can be used with an additional attribute called `returning`, as shown here:

```
<aop:after-returning pointcut-ref="servicesPointcut"
method="writeMessage"
 returning="retValue"/>
```

This attribute lets us get hold of the return value within the advice body. The returning attribute specifies the name of the parameter of the advice to which the return value of the method matched by the pointcut is bound. With the preceding configuration, the `writeMessage()` method would have this signature:

```
public void writeMessage(Object retValue)
```

After throwing advice

This advice executes after the joinpoint throws an exception instead of returning normally. To define this kind of advice, use the `<aop:after-throwing>` element within the `<aop:aspect>` element. The `<aop:throwing>` element uses an additional `throwing` attribute, which specifies the name of the method parameter to which the thrown exception should be bound. Here is an example:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>

<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
        ch11.*Service.*(..))"/>

    <aop:aspect id="aspect1" ref="messageWriter">
        <aop:after-throwing pointcut-ref="servicesPointcut"
          method="writeMessage" throwing="thrownException"/>
    </aop:aspect>
</aop:config>
```

With the preceding configuration, the `writeMessage()` method would have this signature:

```
public void writeMessage(Exception thrownException)
```

Here, the exception thrown by the method matched by the pointcut is passed to the `writeMessage()` method through the `thrownException` argument.

After advice

An after advice executes after method execution, whether the method returns successfully or throws an exception. This kind of advice is analogous to the `finally` block in Java code, which is always executed when the `try` or the `catch` block executes. This advice is defined through the `<aop:after>` element within `<aop:aspect>`, as in this example:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>

<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
ch11.*Service.*(..))"/>

    <aop:aspect id="aspect1" ref="messageWriter">
        <aop:after pointcut-ref="servicesPointcut"
method="writeMessage"/>
    </aop:aspect>
</aop:config>
```

The `writeMessage()` method on the `messageWriter` instance is called whenever the method matched by the pointcut finishes.

Around advice

This kind of advice lets you execute code before or after joinpoints. Similar to what you have already seen in Spring 1.2.x, the around advice can prevent the execution of the joinpoint method. The `<aop:around>` element is used within `<aop:aspect>` to define the around advice. Here's an example:

```
<bean id="messageWriter"
    class="com.packtpub.springhibernate.ch11.MessageWriter">
</bean>

<aop:config>
    <aop:pointcut id="servicesPointcut"
       expression="execution(* com.packtpub.springhibernate.
ch11.*Service.*(..))"/>

    <aop:aspect id="aspect1" ref="messageWriter">
        <aop:around pointcut-ref="servicesPointcut"
method="writeMessage"/>
    </aop:aspect>
</aop:config>
```

The around advice is the only advice that needs to be AspectJ-aware. With this kind of advice, the method of the advice class is implemented with an extra argument of type `org.aspectj.lang.ProceedingJoinPoint`. The `ProceedingJoinPoint` method is analogous to the AOP Alliance's `MethodInterceptor`, which lets you control the joinpoint method's execution, whether the joinpoint method executes or is ignored. The following code shows the `MessageWriter` class, which now is an AspectJ around advice:

```
package com.packtpub.springhibernate.ch11;

import org.aspectj.lang.ProceedingJoinPoint;

public class MessageWriter {

    public void writeMessage(ProceedingJoinPoint call) throws
Throwable {
        System.out.println("Before Target");
        call.proceed();
        System.out.println("After Target");
    }
}
```

Note that the custom code executes before and after invocation of the `call.proceed()` method.

Advice parameters

You may wonder how an advice can access joinpoint methods. The answer is through an additional attribute named `arg-names`. This attribute lets you specify the joinpoint method's arguments which are provided for the advice method. For instance, suppose the joinpoint method uses two arguments: `beforeMessage` and `afterMessage`. The `<aop:advice>` element can be configured as follows, so that both arguments are provided for the advice method:

```
<aop:aspect id="aspect1" ref="messageWriter">
    <aop:around pointcut-ref="servicesPointcut" method="writeMessage"
       arg-names="beforeMessage, afterMessage"/>
</aop:aspect>
```

As you can see, the arguments are specified as a comma-separated list of argument names. By this, the `writeMessage()` method of the advice would have the following signature:

```
public void writeMessage(String beforeMessage, String afterMessage)
```

Note that the `beforeMessage` and `afterMessage` arguments, which are the joinpoint method's arguments, are provided for manipulation in the advice method.

If you've tried to execute the examples in this chapter, you've seen that they work without a proxy object defined. Spring 2.0 uses automatic proxy-type detection for all advised objects.

Summary

In this chapter, we discussed Aspect-Oriented Programming (AOP) technology as a complementary approach to Object-Oriented Programming (OOP). AOP technology defines the aspect, a new concept in system development.

AOP development involves five components: advice, pointcut, advisor, target object, and proxy. Advice specifies the logic that is performed. A pointcut determines the target methods that are advised. An advisor, or aspect, pulls an advice and a pointcut together, and is responsible for applying the advice to the pointcut. The target object is the object that includes the advice methods. A proxy, the object delegated by the method invocation, calls advisors behind the scenes.

Spring provides three types of advice in addition to the `MethodInterceptor` inherited from the AOP Alliance: `ThrowsAdvice`, `BeforeAdvice`, and `AfterReturningAdvice`. `ThrowsAdvice` allows us to implement an advice that is called when the target method throws an exception. `BeforeAdvice` is used to implement advice called before method execution. `AfterReturningAdvice` is used to implement advice called after normal execution of the target method.

To create a pointcut, you must create instances of the `Pointcut` implementations. These are `NameMatchMethodPointcut`, `JdkRegexpMethodPointcut`, `StaticMethodMatcherPointcut`, `DynamicMethodMatcherPointcut`, which allow you to select the target methods based on their names, Java regular expressions, static information, or dynamic information at runtime, respectively. Additionally, you can use `Pointcuts.SETTERS` and `Pointcuts.GETTERS` to select target methods, which are setter and getters, respectively.

Spring's advisors are implementations of the `Advisor` interface. The most common implementation is `DefaultPointcutAdvisor`, which can be used with any Spring advice type and the AOP Alliance's `MethodInterceptor`.

You must create a proxy object to delegate method invocation and call any advice, advisor, or interceptor behind the scenes. Proxy objects are not created on their own, but are created by proxy factory objects. Spring provides a range of different proxy factory classes: `ProxyFactory`, `ProxyFactoryBean`, `AbstractAutoProxyCreator`, and `TransactionProxyFactoryBean`, which are used, respectively, to create a proxy object programmatically, to create one declaratively, to automate proxy creation, and to create a transactional proxy.

Spring 2.0 comes with full support for AspectJ integration. Spring 2.0 provides a proprietary XML schema to configure and wire AOP components together. The main features of Spring 2.0 for AOP implementation are easier XML configuration and support for comprehensive advice. To use Spring 2.0 AOP, you must declare the Spring AOP schema with its location. You can then use `<aop:config>` as the root element for AOP configuration, `<aop:aspect>` to define an aspect, `<aop:pointcut>` to declare a pointcut, and `<aop:before>`, `<aop:after>`, `<aop:after-returning>`, `<aop:after-throwing>`, and `<aop:around>` to define types of advice.

12
Transaction Management

Enterprise applications let users insert new data or manipulate existing data. In most situations, this data is crucial, so the application must perform data manipulation reliably. This reliability is guaranteed through *transactions*, the subject of this chapter.

A transaction is defined as a unit of work, a sequence of operations that must be completed all together, that must be performed reliably. Although this is not a comprehensive definition of a transaction, it's sufficient to give us an idea of what transactions aim to provide.

Databases are the main context in which transactions are discussed. In a database, a transaction implies the reliable performance of a unit of persistence operations. As applications interact with databases, these interactions involve transactions, as well. Applications do not need to implement transactions themselves. Instead, they rely on transaction services exposed by the underlying database or by the application server. This means the application's job is managing transactions, not implementing them. Transaction management includes two activities:

- **Configuring transactions**: How are persistence operations grouped as transactions? How are they isolated from each other?
- **Determining application reactions**: What is the application's response when a transaction's execution succeeds or fails?

Java, and other enterprise languages, provide APIs to configure transactions and determine how the application reacts to them. This is the role of a language in managing transactions.

This chapter explains concepts related to transactions and how transactions are managed in native and Spring-based Hibernate applications. It also discusses transactional and non-transactional caching as two level caching supported by Hibernate. In particular, the chapter covers these topics:

- Transaction essentials
- The Hibernate Transaction API
- The Spring Transaction Abstraction API
- Caching

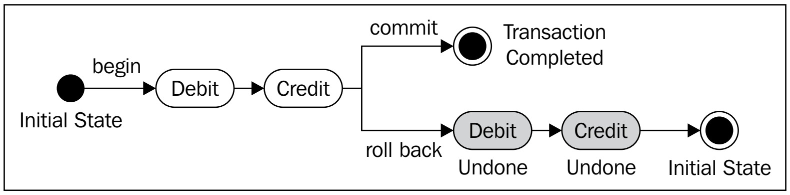
Transaction essentials

Put simply, a transaction is a set of operations that must be considered as a single unit of work. All of the operations must be completed successfully, or none must be carried out. If, for any reason, one or more operations fail, any successful operations in the transaction (or their effects) must be undone.

If the unit of work includes one operation, a transaction for that unit means either that the operation must be completed successfully, or that no persistent state must ever be affected. Additionally, that operation must be performed in isolation from other operations associated with other transactions and executed concurrently. The same statements are true for a unit of work with multiple operations. All of the transaction's operations must be performed successfully, or all must be failed together as if nothing ever happened. The operations associated with one transaction must be isolated from concurrent operations associated with another transaction.

A common example of a transaction is transferring money from one account to another. As debiting the first account and crediting the second must be considered a single job, we expect these actions to be done as parts of a single transaction. In other words, if one operation fails, we expect that the other operation will also fail or will be undone. Otherwise, the transfer is not balanced.

Each transaction *starts* from an initial state. The initial state is the data's state before any changes are applied during the transaction. All other transactions that work with this data see the data as it is in this state. The transaction is executed, and the data may change. When all changes are successfully completed, the transaction is *committed*, meaning all changes are applied and the transaction moves to a stable state. If a change is not successfully completed, the transaction is *rolled back*, meaning that all changes are undone and the transaction moves to the initial state from which it started. The following figure shows this process for our example transaction:



Transactions are often described using a set of properties called ACID, which stands for Atomicity, Consistency, Isolation, and Durability. The ACID properties, described in the following list, are the rules that transactions must obey:

- **Atomicity**: Each transaction is done as a unit of work. If one step of the unit fails, other steps must not be carried out, or must be undone.
- **Consistency**: Each transaction must obey the rules of the affected resource and move it to a consistent state.
- **Isolation**: Each transaction must operate in isolation from other, concurrent transactions. If two transactions are executed concurrently, one cannot affect the other's results before each transaction is committed. For instance, in the educational system application, each course is associated with only a single teacher. When a course is assigned to a teacher in a transaction, another transaction cannot assign the same course to another teacher while the first transaction is not committed. The isolation aspect of transactions guarantees that any transaction data cannot be changed by another transaction while it is being processed.
- **Durability**: Each transaction result must be maintained until another transaction is performed against the result.

Of these properties, only atomicity and isolation are configurable properties that affect application code. Consistency and durability (permanent, fixed properties that are not configurable) are provided and managed by databases.

Managing transactions in Java

Java applications offer a range of different APIs for managing transactions. Based on your application type or your application requirements, you may prefer (or be forced) to use a particular API. Transaction APIs can be summarized as follows:

- **JDBC Transaction API**: This is the core Java API for managing transactions that are associated with a single database. This API is exposed through the `java.sql.Connection` interface, which provides `setAutoCommit()`, `commit()`, and `rollback()` methods to manage transactions.

- **Java Transaction API (JTA)**: This is the Java EE standard API for managing transactions, particularly in distributed environments such as application servers. This API, which is exposed through the `javax.transaction.UserTransaction` interface, is useful in situations where application involves multiple transactional resources.
- **Hibernate Transaction API**: This is a Hibernate-specific API and JPA Transaction API for transaction management in Hibernate applications. These APIs, which are respectively exposed through the `org.hibernate.Transaction` and `javax.persistence.EntityTransaction` interfaces, transmit transaction calls to associated `java.sql.Connection` instances.
- **Spring Transaction Abstraction API**: Part of the Spring API, this is for abstracting the application from the underlying transaction API.

Local versus global transactions

Transactions come in two flavors: local and global. **Local transactions** involve only a single transactional resource (for example, a single database). There may be many transactional resources, but if each resource has its own transactions isolated from other resources' transactions, the transactions are local ones. The code to manage this type of transactions is simple because all transaction implementation is located in a single resource. In the case of a database, the application code simply uses the JDBC API with `commit()` and `rollback()` methods to manage transactions.

In contrast, **global transactions** (also called distributed transactions) involve multiple transactional resources (for example, multiple databases). To manage distributed transactions, the application uses JTA to interact with a third entity, which is called the **transaction manager**. A transaction manager is typically an application server, such as IBM WebSphere or BEA WebLogic, and is responsible for coordinating transactions among resources and providing a two-phase commit process. This two-phase commit ensures that all involved resources commit their associated transactions successfully or, if one transaction fails, all other transactions are rolled back. JTA defines a standard API to interact with different transaction managers. Distributed transactions are more difficult to manage, and, of course, they are executed with overhead that decreases performance.

JTA can be used to manage local transactions, as well. However, JTA needs a JTA provider (for example, Tomcat and any standard application server) and always has overhead.

Transaction demarcation

When managing transactions, you need to mark where each transaction starts and ends. This is called **transaction demarcation**, and there are two different approaches to it: programmatic and declarative. Next, we'll discuss how these approaches are implemented in Hibernate applications.

Programmatic transaction demarcation

Transactions can be demarcated programmatically. JTA or JDBC APIs can be used directly in the application code to begin and commit or roll back transactions. With JDBC, the `setAutoCommit(false)` method is called on the `Connection` object to begin a transaction. The started transaction can be committed or rolled back via `commit()` or `rollback()` on the `Connection` instance, respectively. In the case of a global transaction, a `UserTransaction` object is obtained from the transaction manager through looking up the manager. The `UserTransaction` instance is used as a handler to signal the manager where the transaction starts, `UserTransaction.begin()`, and where the transaction is committed, `UserTransaction.commit()`, or rolled back, `UserTransaction.rollback()`.

When the application uses Hibernate to interact with a single database, you obtain an `org.hibernate.Transaction` object from the `Session` instance as a handler to work with the underlying JDBC transaction API. Although it is still possible to use the JDBC API directly in Hibernate applications, the applications should always rely on the Hibernate API instead of binding to the JDBC API. Global transactions are always managed in the same way, whether the application uses Hibernate or another O/R framework.

Declarative transaction demarcation

The declarative approach makes it possible to mark transaction boundaries declaratively through XML files or Java annotations instead of calling a particular API, such as JDBC or JTA. There are actually two approaches to declarative transaction demarcation: using EJB as a part of the Java EE specification, and using Spring. Using EJB is outside of this book's scope. However, at the end of this chapter, we'll take a look at Spring and how it manages transactions.

Transaction attributes

Transactions can be configured with various attributes to provide more control of transaction execution. These attributes include the following:

- **Isolation level**: This is the first and most significant transaction attribute. Although the underlying database uses a particular isolation level by default, you may decide which isolation level suits your application's needs. The following table shows the isolation levels that are supported by JDBC and most databases. All isolation levels are defined as static members of the `java.sql.Connection` interface.
- **Propagation behavior**: This describes how one or more transactions participate in a bigger transaction and affect its result. The table, shown after isolation level table, lists the different propagation behaviors.
- **Timeout**: This specifies how long a transaction is permitted to run. If the execution time for that transaction exceeds the specified timeout, the transaction must be canceled and rolled back.
- **Read-only**: This specifies whether the transaction is allowed to modify the resource data. Setting this attribute to true allows the transaction provider (for example, Hibernate) to provide some optimizations.

Here are the isolation levels and their equivalent values:

Isolation Level	Equivalent Value
`TRANSACTION_NONE`	0
`TRANSACTION_READ_UNCOMMITTED`	1
`TRANSACTION_READ_COMMITTED`	2
`TRANSACTION_REPEATABLE_READ`	4
`TRANSACTION_SERIALIZABLE`	8

And the following table shows different propagation behaviors:

Propagation Value	Description
`PROPAGATION_REQUIRED`	The method participates in the current transaction. If no transaction exists, a new one is created.
`PROPAGATION_SUPPORTS`	The method participates in the current transaction. If no transaction exists, the method executes without a transaction.
`PROPAGATION_MANDATORY`	The method participates in the current transaction. If no transaction exists, an exception is thrown.
`PROPAGATION_REQUIRES_NEW`	The method executes in a new transaction. If a transaction exists, the transaction will be suspended.

Propagation Value	Description
`PROPAGATION_NOT_SUPPORTED`	The method executes without a transaction. If a transaction exists, the transaction will be suspended.
`PROPAGATION_NEVER`	The method executes without a transaction. If a transaction exists, an exception is thrown.
`PROPAGATION_NESTED`	The method executes within a nested transaction if a current transaction exists. Otherwise, it behaves like `PROPAGATION_REQUIRED`.

Now that you've seen some background material for transactions, it's time to look at transaction implementation details, both in native and in Spring-based Hibernate applications.

Transactions in Hibernate applications

With JDBC, `java.sql.Connection` objects are always used to interact with the database, perform persistent operations, and manage transactions. With Hibernate, `org.hibernate.Session` objects are used to interact with the database. In turn, the `Session` objects use the `Connection` objects.

Each `Session` object uses an individual `Connection` object behind the scenes to perform persistent operations and manage transactions. The following snippet shows a typical procedure for transaction management with Hibernate:

```
Session session = HibernateHelper.getSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  // Unit of work inside the transaction
  tx.commit();
  tx = null;
} catch (Exception e) {
  if (tx != null)
    try {
      tx.rollback();
    } catch (HibernateException innerEx) {
      logger.log("Couldn't roll back transaction", innerEx);
    }
} finally {
  session.close();
}
```

Let's go through the code and see what each line does:

- `Session session = HibernateHelper.getSession()`: A `Session` instance is obtained from the `SessionFactory`. Obtaining a `Session` object notionally means fetching a `Connection` object from the connection pool. However, the `Session` object does not fetch a `Connection` from the pool until the `Session` transaction starts by invoking `Session.beginTransaction()`.
- `Transaction tx = session.beginTransaction()`: The `beginTransaction()` method of `Session` marks the starting point of the transaction. This method returns an `org.hibernate.Transaction` instance as a handler to commit or roll back the transaction at the end. By calling the `beginTransaction()` method, the `Session` is bound to a fresh `Connection` object from the connection pool, the `setAutoCommit(false)` method on the associated `Connection` object is called, and the transaction starts.
- **Performing persistent operations**: The `Session` object is used to perform persistent operations that must be done as a unit of work inside a transaction.
- `tx.commit()`: This is the transaction's end point, where you decide whether to commit all changes to the database. If everything goes well, the transaction is committed. This flushes the `Session` instance, synchronizes it with the database, and releases the `Connection` object to which the `Session` is bound.
- `tx=null`: The `Transaction` instance is thrown away. Assigning `null` to the `Transaction` instance allows us to check in the `catch` block whether the transaction successfully committed. (`tx=null` is invoked when the `tx.commit()`executes without an exception.)
- `catch` **block**: The only operation in the `catch` block is rolling back the transaction if it has not committed successfully.
- `session.close()`: You need to clean up the `Session` instance and release its resources.

Many `Session` and `Transaction` methods throw a `HibernateException`. However, because `HibernateException` is an unchecked exception, you don't need to catch it if you don't want to. Often, we should worry about commit fulfillment, as we have done here, so we catch exceptions only for rolling back the transaction.

Using JTA in Hibernate applications

In rare situations, an application uses more than one database. One valid use case for this situation is when your data is not in one database, and is scattered among many databases. In such situations, the JDBC API, which Hibernate uses internally to manage transactions, is not sufficient. In this case, you must use JTA to manage transactions. However, JTA is available only in managed environments (for example, a Java EE application server, such as IBM WebSphere (`http://www.ibm.com/websphere`), or a stand-alone JTA provider, such as ObjectWeb JOTM (`http://jotm.objectweb.org`)).

To use JTA, Hibernate should be configured as follows:

- In the Hibernate configuration file, set the `hibernate.transaction.factory_class` property to `org.hibernate.transaction.JTATransactionFactory`. (The default value for the `hibernate.transaction.factory_class` property is `org.hibernate.transaction.JDBCTransactionFactory`, meaning that Hibernate uses the JDBC transaction API by default.)
- Set `hibernate.transaction.manager_lookup_class` to an appropriate `TransactionManagerLookup` implementation corresponding to the managed environment that provides transactions. (For instance, use `org.hibernate.transaction.WebSphereTransactionManagerLookup` when you are using IBM WebSphere as the managed environment.)
- Configure Hibernate with the right values for `hibernate.jndi.url` and `hibernate.jndi.<JNDIpropertyname>` to use managed connections from the container. (Refer to Chapter 4 for more information.)
- Obtain a `javax.transaction.UserTransaction` object by looking up an environment JNDI node, and use the obtained object to commit or roll back the transaction.

The following code shows how to use JTA to perform persistent operations with two databases:

```
UserTransaction utx =
  (UserTransaction) new InitialContext().lookup("java:comp/env/tx");
Session session1 = null;
Session session2 = null;
try {
  utx.begin();
  session1 = HibernateHelper.getSession1();
  session2 = HibernateHelper.getSession2();
  // interacting with server
  // Unit of work inside the transaction
```

```
      utx.commit();
    } catch (RuntimeException ex) {
      try {
        utx.rollback();
        throw ex;
      } catch (SystemException sysEx) {
        logger.log("Couldn't roll back transaction", sysEx);
      }
    }
```

As you can see, a `UserTransaction` object is obtained through looking up the container. The `begin()` method is called on the obtained `UserTransaction` to start the transaction. After all of the work has been done, the `commit()` method is invoked on the `UserTransaction` object to commit the transaction. Any exception during the databases' interaction caught in the `catch` block rolls back the transaction.

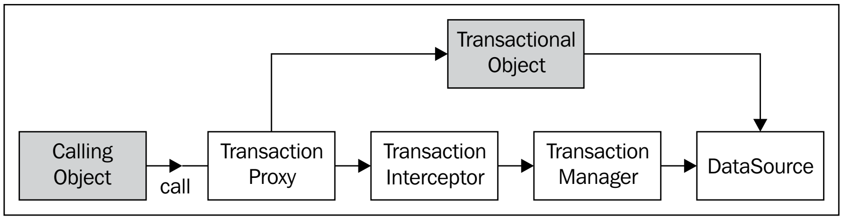
Spring transaction abstraction

So far, you have learned how to manage Hibernate transactions through JDBC (when Hibernate uses a single database) and through JTA (when Hibernate interacts with multiple databases). Spring can simplify transaction management in Hibernate applications by providing a transaction infrastructure to abstract the application code from the underlying transaction strategy. Using Spring, transaction management code that's bound to a specific API, such as JDBC or JTA, is omitted from the application. With Spring, you can manage transactions programmatically or declaratively. However, the declarative approach has certain advantages and is always preferred.

Transaction management with Spring is powerful and flexible because Spring provides a rich set of options for a range of different environments. With Spring, you can easily switch to a different transaction strategy without making significant changes to the application code.

How Spring manages transactions

In the previous chapter, we established that transactions are cross-cutting concerns. Transaction management is logic that is scattered over the entire application. With Spring, we can implement transaction management code as advice applied to particular classes of the application. The following figure shows the individual participants in Spring's transaction management:



[Only the gray objects in the figure above are used in the application code. The white objects are wholly configured, instantiated, and managed by the Spring container.]

The following table describes the role of each object in the above figure:

Object	**Role**
Data source	This object represents the `java.sql.Connection` factory. It is configured as a bean of type `javax.sql.DataSource`.
Transaction manager	This object abstracts the application code from the underlying transaction API. There is an individual implementation of `org.springframework.transaction.PlatformTransactionManager` for each persistence technology supported by Spring.
Transaction interceptor	This is an around advice that binds methods to transactions.
Transactional object	This is the object that is bound to the transaction interceptor.
Transaction proxy	This is an intermediate object, residing between the calling object and the transactional object. It is responsible for applying the transaction interceptor.
Calling object	This is the object that calls the transactional object. Actually, the calling object interacts with a transaction proxy to call the transactional object.

Let's go through this list and see how each object is configured.

The choice of transaction manager

Spring provides a distinct transaction manager for each persistence technology that it supports. The transaction manager abstracts the details of the underlying transaction management API. The following table shows the different transaction managers that Spring provides. Each transaction manager is an implementation of `org.springframework.transaction.PlatformTransactionManager`:

Transaction Manager Class	**Persistence Technology**
`org.springframework.jdbc.datasource.DataSourceTransactionManager`	This manages pure JDBC transactions associated directly with `java.sql.Connection`.
`org.springframework.orm.hibernate.HibernateTransactionManager`	This manages transactions associated with Hibernate 2's `Sessions`.
`org.springframework.orm.hibernate3.HibernateTransactionManager`	This manages transactions associated with Hibernate 3's `Sessions`.
`org.springframework.orm.jpa.JpaTransactionManager`	This manages transactions associated with JPA's `javax.persistence.EntityManager` objects.
`org.springframework.orm.jdo.JdoTransactionManager`	This manages transactions associated with JDO's `javax.jdo.PersistenceManager` objects.
`org.springframework.orm.toplink.TopLinkTransactionManager`	This manages transactions associated with TopLink's `Sessions`.
`org.springframework.jta.JtaTransactionManager`	This manages transactions associated with JTA's `javax.transaction.UserTransaction` objects.

To use the Spring transaction API, you must use a transaction manager that's appropriate to the persistence technology your application uses. Additionally, you must use an appropriate connection factory, creating connection objects of the target persistence technology. The following table shows the factory classes provided by Spring:

Factory Class	**Persistence Technology**
`org.springframework.orm.hibernate.LocalSessionFactoryBean`	This is a factory for Hibernate 2's `Sessions`.
`org.springframework.orm.hibernate3.LocalSessionFactoryBean`	This is a factory for Hibernate 3's `Sessions`.
`org.springframework.orm.toplink.LocalSessionFactoryBean`	This is a factory for TopLink's `Sessions`.
`org.springframework.orm.jpa.LocalEntityManagerFactoryBean`	This is a factory for JPA's `EntityManager`.
`org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean`	This is a factory for JDO's `PersistenceManager`.

Using the Hibernate transaction manager

Setting up the transaction manger for a Hibernate application includes declaring factory and transaction manager objects in the Spring context, and then wiring them together, as shown here:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
    <value>org.hsqldb.jdbcDriver</value>
  </property>
  <property name="url">
    <value>jdbc:hsqldb:hsql://localhost/hiberdb</value>
  </property>
  <property name="username">
    <value>sa</value>
  </property>
  <property name="password">
    <value></value>
  </property>
</bean>
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="mappingResources">
    <list>
      <value>com/packtpub/springhibernate/ch12/Student.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch12/Teacher.hbm.xml</value>
      <!-- Other hbm files -->
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.transaction.factory_class">
        org.hibernate.transaction.JDBCTransactionFactory
      </prop>
      <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
      </prop>
    </props>
  </property>
  <property name="dataSource">
    <ref local="dataSource"/>
  </property>
</bean>
<bean id="transactionManager"
```

```
      class="org.springframework.orm.hibernate3.
HibernateTransactionManager">
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>
```

As you can see, the configuration file includes three bean definitions: one for the data source, one for the factory object, and one for the transaction manager object. The `SessionFactory` object is configured with three properties: `mappingResources`, `hibernateProperties`, and `dataSource`, which specify the Hibernate mapping paths, the Hibernate configurable properties, and the data source that creates JDBC `Connection`s, respectively. The `TransactionManager` is configured with only one property: `sessionFactory`.

Using the JTA transaction manager

You can use the JTA transaction manager in a Hibernate application when the application works with multiple databases. The configuration for using the JTA transaction manager is very similar to using a normal Hibernate transaction manager, as recently discussed. The difference is using `JtaTransactionManager`, instead of `HibernateTransactionManager`, and configuring it without a `sessionFactory` property. The following code shows how to use the JTA transaction manager:

```
<bean id="dataSource" class="org.springframework.jndi.
JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/MyDB</value>
  </property>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="mappingResources">
    <list>
      <value>com/packtpub/springhibernate/ch12/Student.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch12/Teacher.hbm.xml</value>
      <!-- Other hbm files -->
      </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
```

```
          org.hibernate.dialect.HSQLDialect
        </prop>
      </props>
    </property>
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
  </bean>
```

```
  <bean id="transactionManager"
        class="org.springframework.transaction.jta.
  JtaTransactionManager">
  </bean>
```

Spring transaction configuration

When you've configured the `SessionFactory` and the transaction manager, you need to configure which methods to execute inside transactions. The configuration is similar to what you have already learned in Chapter 11 about AOP configuration. Similarly, you may use the classic bean definitions introduced by Spring 1.x, or the schema-based transaction configuration introduced by Spring 2.x. Additionally, Spring 2.x introduced an annotation-based transaction configuration.

[Although using Spring 2.*x*'s new features is always recommended, you still have the option to use a classic bean configuration provided by Spring 1.*x* when you are using Spring 2.*x*.]

The only difference between Spring 1.*x* and Spring 2.*x* transaction configuration has to do with interceptor and proxy configuration. In the following subsections, we'll look at both classic and schema-based interceptor and proxy configurations.

Transaction configuration in Spring 1.x

The transaction interceptor is configured as an instance of `org.springframework.transaction.interceptor.TransactionInterceptor`. The `TransactionInterceptor` is an around advice, implementing the `MethodInterceptor` interface. `TransactionInterceptor` begins a transaction before target method execution, and commits the transaction after the method executes normally, without throwing an unchecked exception. `TransactionInterceptor` rolls back the transaction if an unchecked exception is thrown. Here's an example of interceptor configuration:

```
  <bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.
  TransactionInterceptor">
```

```
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
      <props>
        <prop key="*">PROPAGATION_REQUIRED</prop>
      </props>
    </property>
  </bean>
```

`TransactionInterceptor` is configured with two properties: `transactionManager` and `transactionAttributes`. The `transactionManager` refers to the configured transaction manager inside the Spring context. `transactionAttributes` lets you select target methods based on their names, and define transaction attributes, such as propagation behavior. In this example, this attribute uses `*` to indicate that all of the target methods must be executed inside transactions. We could specify another pattern, such as `save*`, which means only methods whose names start with the word `save` are executed inside a transaction. `PROPAGATION_REQUIRED`, as you have already seen, specifies how multiple transactions interact with each other.

The `TransactionInterceptor` instance starts a transaction before the target method is invoked, and commits the transaction after the target method returns normally. If the target method throws an unchecked exception (thereby returning abnormally), `TransactionInterceptor` rolls back the transaction. However, if `TransactionInterceptor` encounters a checked exception, it ignores the exception and commits the transaction. You can also configure rollback conditions for the interceptor. For this purpose, append a comma-separated list of exception types, which must make transactions roll back to the transaction's propagation behavior. For example, `transactionInterceptor` can be configured as follows to indicate that throwing any type of exception rolls back a transaction:

```
  <bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.
  TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
      <props>
        <prop key="*">PROPAGATION_REQUIRED, -Throwable</prop>
      </props>
    </property>
  </bean>
```

Each exception type must have a minus (-) symbol as a prefix.

Proxy configuration

The proxy is another object that should be configured. This object intercepts calls to the target methods, and applies the configured transaction interceptor to the target methods. As you have already seen, Spring provides the general `org.springframework.aop.framework.ProxyFactoryBean` class as a proxy object to be configured inside the Spring context. The following code shows how this class can be used:

```
<bean id="transactionalStudentDao"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <ref local="studentDao"/>
  </property>
  <property name="interceptorNames">
    <list>
      <idref bean="transactionInterceptor"/>
    </list>
  </property>
</bean>
```

As you can see, the transaction proxy is configured with two properties: `target` and `interceptorNames`. The `target` property refers to the target object, the method that must be executed inside a transaction. The `interceptorNames` property specifies a list of interceptors, including the `transactionInterceptor`, which must be applied to target method invocations.

Remember to use the proxy, rather than the actual target object, in your other bean definitions that use the DAO bean.

Autoproxy creation

Spring provides an autoproxy approach to proxy advice and interceptors automatically, making XML configuration less verbose. Using this approach, you can declare an `org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator` bean, which wraps all of the target objects and interceptors. When the target objects are invoked, it applies all of the interceptors to the target objects. The following snippet shows how this approach can be used for our example:

```
<bean class="org.springframework.aop.framework.autoproxy.
BeanNameAutoProxyCreator">
  <property name="beanNames">
    <list>
      <idref bean="studentDao"/>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <idref bean="transactionInterceptor"/>
    </list>
  </property>
</bean>
```

As you can see, the `BeanNameAutoProxyCreator` bean is configured with two properties: `beanNames` and `interceptorNames`. The `beanNames` specifies a list of transactional target objects, the objects that must be executed inside transactions. `interceptorNames` specifies the advice, including the transaction advice, applied to the target objects. This time, you can use the original `studentDao` bean in your other beans.

TransactionProxyFactoryBean

Using `TransactionProxyFactoryBean` is the easiest way to wrap a target object into a transaction proxy. It allows us to omit the `TransactionInterceptor` definition from the XML configuration and define a new bean, which wraps the `target`, `interceptors`, and `transactionAttributes`. The following code shows our example, which now is configured with `TransactionProxyFactoryBean`:

```
  <bean id="transactionalStudentDao"
      class="org.springframework.transaction.interceptor.
  TransactionProxyFactoryBean">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    <property name="target">
      <ref local="studentDao"/>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="save*">PROPAGATION_REQUIRED</prop>
      </props>
    </property>
  </bean>
```

To use `TransactionProxyFactoryBean`, you should configure each transactional object as a separate bean. In this case, you must use this in place of the original `studentDao` bean.

It's also possible to define the target object in the `TransactionProxyFactoryBean` definition, so there's no need for another bean:

```
<bean id="transactionalStudentDao"
    class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="target">
    <bean class="com.packtpub.springhibernate.ch12.StudentDaoImpl">
      <property name="sessionFactory">
        <ref local="sessionFactory"/>
      </property>
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Transaction configuration in Spring 2.x

To use the convenient configuration features provided in Spring 2.*x*, you need to import the transaction schema to the Spring context as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/tx
          http://www.springframework.org/schema/tx/spring-tx.xsd">
    <!--transaction configuration here-->
</beans>
```

The specific elements which can come for transaction configuration in the spring context are as follows:

- `spring-tx.xsd`: This defines custom elements to make transaction configuration more convenient. The custom elements used for transaction configuration are as follows:
    - `<tx:annotation-driven>`: This configures the transaction manager with target classes, which are annotated with the Spring `@Transactional` annotation. (This relies on the Java annotation API, so it's not discussed in this book.)
    - `<tx:advice>`: This element declares a transaction as an advice to be applied to the target objects.
    - `<tx:attributes>`: This element configures a transaction advice. The configuration attributes include the propagation behavior, read-only flag, roll back for, and so on.
    - `<tx:method>`: This element specifies methods as target transactional methods.

The following code shows our example, which now has been refactored to use Spring 2.x's custom schema elements:

```
<bean id="dataSource" class="org.springframework.jndi.
JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/MyDB</value>
  </property>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="mappingResources">
    <list>
      <value>com/packtpub/springhibernate/ch12/Student.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch12/Teacher.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch12/Course.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
      </prop>
    </props>
  </property>
  <property name="dataSource">
    <ref local="dataSource"/>
  </property>
</bean>

<bean id="transactionManager"
      class="org.springframework.transaction.jta.
JtaTransactionManager">
</bean>

<!-- Transactional proxy -->
<tx:advice id="transactionInterceptor" transaction-manager=
"transactionManager">
  <tx:attributes>
    <tx:method name="*" rollback-for="Throwable"/>
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:advisor
        pointcut=" execution(* com.packtpub.springhibernate.
ch12.*Service.*(..))"
        advice-ref="transactionInterceptor"/>
</aop:config>
```

As you can see, only the `transactionInterceptor` and `target` declarations have changed. `transactionInterceptor` is defined as an advice. This advice is applied to all methods (`*`) of the target object. The `<aop:advisor>` element defines the target methods to which the declared advice is applied. This element uses a `pointcut` attribute to select target methods.

Another topic that may involve reliable data manipulation is caching. Caching, of course, is managed by Hibernate, so it always provides reliable data. However, this is a good place to look at Hibernate's caching mechanism, and see how it improves performance and provides reliable data.

Caching

Interacting with a database is an expensive operation. This is due to the fact that any database interaction consists of sending a query request over the network, compiling the query on the database server, executing the query, producing the result, and finally sending the result back to the client.

Databases use different strategies to optimize performance. One of these strategies is caching the data for a period of time. Caching prevents unnecessary query compiling and disk I/O. Although the database-side caching strategy can be very effective (because it does not consider network traffic), this strategy has limited value when there are a large number of requests.

A typical strategy in this situation is an application-side cache mechanism. Instead of using the database to cache data, the application caches the data. When direct JDBC is used, the application is responsible for providing and managing the caching. However, Hibernate provides a rich, easy-to-use cache mechanism on two levels. Although the first-level cache is mandatory for all interactions (meaning that all database requests must pass through this level), the second-level cache is optional and configurable.

The first-level cache, provided by the `Session` object, holds the repetitive requested objects in memory and prevents Hibernate from loading an object multiple times. This cache can be discarded for an individual object by invoking the `evict()` method of the session object:

```
public void evict(Object o) throws HibernateException
```

`o` is an object that you want to remove from the cache.

You can also use the `clear()` method to disable this cache for all objects:

```
public void clear()
```

The second-level cache is an external cache provided as a third-party project and plugged into Hibernate. The `Session` object accesses this cache transparently. Last figure in Chapter 3 shows the relationship of two caches with the session object inside Hibernate.

Hibernate lets us use any third-party cache implementation by providing the `org.hibernate.cache.CacheProvider` interface as the handler for interacting with that cache implementation. The implemented class should be specified through the `hibernate.cache.provider_class` property in the configuration file. The following table shows the cache providers which Hibernate currently supports. By default, Hibernate uses Ehcache for JVM-level caching:

Cache	Provider Class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	`HashtableCache Provider`	memory	no	yes
Ehcache	`EhCacheProvider`	memory, disk	yes	yes
OSCache	`OSCacheProvider`	memory, disk	yes	yes
SwarmCache	`SwarmCacheProvider`	clustered (IP multicast)	yes (clustered invalidation)	no
JBoss TreeCache	`TreeCacheProvider`	clustered (IP multicast), transactional	yes (replication)	yes (clock sync req.)

All of the cache provider classes are in the `org.hibernate.cache` package.

Hibernate lets you configure the type of access to the second-level cache. You do this via an object of the `org.hibernate.CacheMode` class that is applied to a `Session` object by invoking its `setCacheMode()` method:

```
public void setCacheMode(CacheMode cacheMode)
```

This tells Hibernate how to interact with the second-level cache. For example, you can use `CacheMode.IGNORE` to tell Hibernate not to use the second-level cache in this particular `Session`. Suppose you are storing a huge number of students in the database. To reduce memory usage, you can disable the second-level cache by setting the cache mode of `Session` to `org.hibernate.CacheMode.IGNORE` as follows:

```
List students = ...//prepared through the business layer
Session session = HibernateHelper.getSession();
Transaction tx = session.beginTransaction();
session.setCacheMode(CacheMode.IGNORE);
for (int i = 0; i < students.size(); i++) {
  session.save(students.get(i));
  if (i % 20 == 0) {
    session.flush();
    session.clear();
  }
}
tx.commit();
session.close();
```

In this example, the first-level cache is flushed for every 20 objects stored.

The following table shows the cache modes that can be applied to `Session` objects. Notice that all of these modes have been defined as static members in the `CacheMode` class:

Cache Mode	Description
`IGNORE`	This mode specifies that items are never read from, or written to the second-level cache.
`REFRESH`	This mode indicates that items are written to the second-level cache, but are not read from it. This mode bypasses the effect of `hibernate.cache.use_minimal_puts`, forcing a refresh of the second-level cache for all items read from the database.
`PUT`	The second-level cache is only applied to put (write) requests, but Hibernate won't read from the second-level cache.
`GET`	The second-level cache is only applied to get (read) requests, but Hibernate won't write to the second-level cache, except when updating data.
`NORMAL`	This is the normal behavior, reading items from and writing items to the second-level cache.

Summary

In this chapter, we discussed the fundamental concepts behind transactions and caching, and how they are implemented in Hibernate.

Conceptually, a transaction is a group of operations that must be done reliably against a database or any other resource. Every transaction is specified by its so-called ACID properties: Atomicity, Consistency, Isolation, and Durability. Of these properties, only atomicity and isolation are of concern to the application developer. The other two are implemented internally by the transactional resource.

Hibernate allows you to configure transaction isolation through the configuration file. This is done through the `hibernate.connection.isolation` entry in the configuration file with a value defined as a static member of the `java.sql.Connection` interface.

Regarding atomicity, you need to obtain a `Transaction` object before any operation by invoking the `beginTransaction()` method of `Session`. You can decide to commit or roll back the transaction by calling either the `commit()` or the `rollback()` method.

Transactions also have other configurable characteristics, called transaction attributes. These include transaction timeout, transaction propagation, and read-only flag.

Spring provides a transaction API to abstract the application code from the specific transaction API exposed by different persistence technologies. Moreover, Spring lets you apply transactions as advice to the target transactional objects. To use the Spring transaction API, you must configure a transaction manager, a transaction interceptor, and target transactional objects. The transaction manager is the object that delegates transaction invocation to the underlying transaction API. The transaction interceptor is an around advice, applying transactions as advice to the target transactional objects. To configure these objects, you may use classic bean configuration exposed through Spring 1.*x*, or schema-based bean configuration provided by Spring 2.*x*.

Hibernate also provides caching in two levels to enhance persistence performance. Although the first-level cache is provided by the session objects and cannot be disabled, the second-level cache is served by third-party cache projects plugged into Hibernate. The second-level cache is configurable and can be disabled. You can also configure Hibernate to indicate how it should interact with the second-level cache. This is done by the `setCacheMode()` method of `Session`, with a value defined as a static member of the `CacheMode` class.

13 Integrating Hibernate with Spring

As you've seen, Spring is a general-purpose framework that plays different roles in many areas of application architecture. One of these areas is persistence. Spring does not provide its own persistence framework. Instead, it provides an abstraction layer over JDBC, and a variety of O/R mapping frameworks, such as iBATIS SQL Maps, Hibernate, JDO, Apache OJB, and Oracle TopLink. This abstraction allows consistent, manageable data-access implementation.

Spring's abstraction layer abstracts the application from the connection factory, the transaction API, and the exception hierarchies used by the underlying persistence technology. Application code always uses the Spring API to work with connection factories, utilizes Spring strategies for transaction management, and involves Spring's generic exception hierarchy to handle underlying exceptions. Spring sits between the application classes and the O/R mapping tool, undertakes transactions, and manages connection objects. It translates the underlying persistence exceptions thrown by Hibernate to meaningful, unchecked exceptions of type `DataAccessException`. Moreover, Spring provides IoC and AOP, which can be used in the persistence layer.

In the previous chapter, you learned about transaction management strategies for a Hibernate application. You also learned how Spring undertakes Hibernate's transactions and provides a more powerful, comprehensive approach to transaction management.

This chapter discusses how Spring affects the application's data-access layer. Our discussion starts with the **Data Access Object (DAO)** pattern. This pattern, which is popular in the Java world, allows for a more manageable, more maintainable data-access tier. Then, we'll discuss how Spring affects application DAO classes when integrated with Hibernate.

The Data Access Object pattern

Although you can obtain a `Session` object and connect to Hibernate anywhere in the application, it's recommended that all interactions with Hibernate be done only through distinct classes. Regarding this, there is a JEE design pattern, called the DAO pattern. According to the DAO pattern, all persistent operations should be performed via specific classes, technically called **DAO classes**. These classes are used exclusively for communicating with the data tier. The purpose of this pattern is to separate persistence-related code from the application's business logic, which makes for more manageable and maintainable code, letting you change the persistence strategy flexibly, without changing the business rules or workflow logic.

The DAO pattern states that we should define a DAO interface corresponding to each DAO class. This DAO interface outlines the structure of a DAO class, defines all of the persistence operations that the business layer needs, and (in Spring-based applications) allows us to apply IoC to decouple the business layer from the DAO class.

Service Facade Pattern

In implementation of data access tier, the Service Facade Pattern is always used in addition to the DAO pattern. This pattern indicates using an intermediate object, called service object, between all business tier objects and DAO objects. The service object assembles the DAO methods to be managed as a unit of work. Note that only one service class is created for all DAOs that are implemented in each use case.

The service class uses instances of DAO interfaces to interact with them. These instances are instantiated from the concrete DAO classes by the IoC container at runtime. Therefore, the service object is unaware of the actual DAO implementation details.

Regardless of the persistence strategy your application uses (even if it uses direct JDBC), applying the DAO and Service Facade patterns to decouple application tiers is highly recommended.

Data tier implementation with Hibernate

Let's now see how the discussed patterns are applied to the application that directly uses Hibernate. The following code shows a sample DAO interface:

```
package com.packtpub.springhibernate.ch13;

import java.util.Collection;

public interface StudentDao {
  public Student getStudent(long id);

  public Collection getAllStudents();

  public Collection getGraduatedStudents();

  public Collection findStudents(String lastName);

  public void saveStudent(Student std);

  public void removeStudent(Student std);
}
```

The following code shows a DAO class that implements this DAO interface:

```
package com.packtpub.springhibernate.ch13;

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

import java.util.Collection;

public class HibernateStudentDao implements StudentDao {

  SessionFactory sessionFactory;

  public Student getStudent(long id) {
    Student student = null;
    Session session = HibernateHelper.getSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      student = (Student) session.get(Student.class, new Long(id));
      tx.commit();
      tx = null;
    } catch (HibernateException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    } finally {
      session.close();
    }
    return student;
  }
  public Collection getAllStudents(){
    Collection allStudents = null;
    Session session = HibernateHelper.getSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Query query = session.createQuery(
             "from Student std order by std.lastName, std.firstName");
      allStudents = query.list();
      tx.commit();
      tx = null;
    } catch (HibernateException e) {
      if (tx != null)
        tx.rollback();
      throw e;    } finally {
      session.close();
    }
    return allStudents;
  }
  public Collection getGraduatedStudents(){
    Collection graduatedStudents = null;
    Session session = HibernateHelper.getSession();
    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Query query = session.createQuery(
                        "from Student std where std.status=1");
      graduatedStudents = query.list();
      tx.commit();
      tx = null;
    } catch (HibernateException e) {
      if (tx != null)
        tx.rollback();
      throw e;
    } finally {
      session.close();
    }
    return graduatedStudents;
  }
```

```
public Collection findStudents(String lastName) {
  Collection students = null;
  Session session = HibernateHelper.getSession();
  Transaction tx = null;
  try {
    tx = session.beginTransaction();
    Query query = session.createQuery(
                "from Student std where std.lastName like ?");
    query.setString(1, lastName + "%");
    students = query.list();
    tx.commit();
    tx = null;
  } catch (HibernateException e) {
    if (tx != null)
      tx.rollback();
    throw e;
  } finally {
    session.close();
  }
  return students;
}
public void saveStudent(Student std) {
  Session session = HibernateHelper.getSession();
  Transaction tx = null;
  try {
    tx = session.beginTransaction();
    session.saveOrUpdate(std);
    tx.commit();
    tx = null;
  } catch (HibernateException e) {
    if (tx != null)
      tx.rollback();
    throw e;
  } finally {
    session.close();
  }
}
public void removeStudent(Student std) {
  Session session = HibernateHelper.getSession();
  Transaction tx = null;
  try {
    tx = session.beginTransaction();
    session.delete(std);
```

```
        tx.commit();
        tx = null;
      } catch (HibernateException e) {
        if (tx != null)
          tx.rollback();
        throw e;
      } finally {
        session.close();
      }
    }

    public void setSessionFactory(SessionFactory sessionFactory) {
      this.sessionFactory = sessionFactory;
    }
  }
```

As you can see, all implemented methods do routines. All obtain a `Session` object at first, get a `Transaction` object, perform a persistence operation, commit the transaction, rollback the transaction if exception occurs, and finally close the `Session` object. Each method contains much boilerplate code that is very similar to the other methods.

Although applying the DAO pattern to the persistence code leads to more manageable and maintainable code, the DAO classes still include much boilerplate code. Each DAO method must obtain a `Session` instance, start a transaction, perform the persistence operation, and commit the transaction. Additionally, each DAO method should include its own duplicated exception-handling implementation. These are exactly the problems that motivate us to use Spring with Hibernate.

Template Pattern

To clean the code and provide more manageable code, Spring utilizes a pattern called **Template Pattern**. By this pattern, a template object wraps all of the boilerplate repetitive code. Then, this object delegates the persistence calls as a part of functionality in the template. In the Hibernate case, `HibernateTemplate` extracts all of the boilerplate code, such as obtaining a Session, performing transaction, and handing exceptions.

Data tier implementation with Spring

With Spring, you do not need to implement code for obtaining `Session` objects, starting and committing transactions, and handling Hibernate exceptions. Instead, you use a `HibernateTemplate` instance to delegate persistence calls to Hibernate, without direct interaction with Hibernate.

Here are some benefits of using Spring in the persistence layer, instead of using direct interaction with Hibernate:

- With Spring, the `HibernateTemplate` object interacts with Hibernate. This object removes the boilerplate code from DAO implementations.
- Any invocation of one of `HibernateTemplate's` methods throws the generic `DataAccessException` exception instead of `HibernateException` (a Hibernate-specific exception).
- Spring lets us demarcate transactions declaratively, instead of implementing duplicated transaction-management code.

The `HibernateTemplate` class uses a `SessionFactory` instance internally to obtain `Session` objects for Hibernate interaction. Interestingly, you can configure the `SessionFactory` object via the Spring IoC container to be instantiated and injected into DAO objects.

The next sections discuss how `HibernateTemplate` is used in DAO classes, and how it is configured with `SessionFactory` in the Spring IoC container. First, let's look at the Spring exception hierarchy.

Spring exception translation

Spring provides its own exception hierarchy, which sits on the exception hierarchies of the O/R mapping tools it supports. It catches any database exception or error that might be thrown through JDBC, or the underlying O/R mapping tool, and translates the caught exception to a corresponding exception in its own hierarchy. The Spring exception hierarchy is defined as a subclass of `org.springframework.dao.DataAccessException`. Spring catches any exception thrown in the underlying persistence technology and wraps it in a `DataAccessException` instance. The `DataAccessException` object is an unchecked exception, because it extends `RuntimeException` and you do not need to catch it if you do not want to.

Refactoring DAO classes to use Spring

Spring provides distinct DAO base classes for the different data-access technologies it supports. For instance, Spring provides `HibernateDaoSupport` for Hibernate, `SqlMapClientDaoSupport` for iBATIS SQL Maps, and `JdoDaoSupport` for JDO. These classes wrap the common properties and methods that are required in all DAO implementation subclasses.

When you use Hibernate with Spring, the DAO classes extend the Spring `org.springframework.orm.hibernate3.support.HibernateDaoSupport` class. This class wraps an instance of `org.springframework.orm.hibernate3.HibernateTemplate`, which in turn wraps an `org.hibernate.SessionFactory` instance. As you will soon see in this chapter, extending the `HibernateDaoSupport` class lets you configure all DAO implementations consistently as beans inside the Spring IoC container. Their `SessionFactory` property is configured and set up via the Spring context.

The following code shows a simple DAO interface for a Spring-based DAO implementation:

```
package com.packtpub.springhibernate.ch13;

import java.util.Collection;

public interface StudentDao {
  public Student getStudent(long id);

  public Collection getAllStudents();

  public Collection getGraduatedStudents();

  public Collection findStudents(String lastName);

  public void saveStudent(Student std);

  public void removeStudent(Student std);
}
```

Here, `StudentDao` is the DAO interface, with the same structure as the interface shown in the following code.

`HibernateException` is thrown for any failure when directly interacting with Hibernate. When Spring is used, `HibernateException` is caught by Spring and translated to `DataAccessException` for any persistence failure. Both exceptions are unchecked, so you do not need to catch them if you don't want to do.

The following code shows the DAO implementation for this DAO interface, which now uses Spring to interact with Hibernate:

```
package com.packtpub.springhibernate.ch13;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import java.util.Collection;
public class HibernateStudentDao extends HibernateDaoSupport
implementsStudentDao {
  public Student getStudent(long id) {
    return (Student) getHibernateTemplate().get(Student.class, new
     Long(id));
  }
  public Collection getAllStudents(){
    return getHibernateTemplate().
      find("from Student std order by std.lastName, std.firstName");
  }
  public Collection getGraduatedStudents(){
    return getHibernateTemplate().find("from Student std where
     std.status=1");
  }
  public Collection findStudents(String lastName) {
    return getHibernateTemplate().
      find("from Student std where std.lastName like ?", lastName +
"%");
  }
  public void saveStudent(Student std) {
    getHibernateTemplate().saveOrUpdate(std);
  }
  public void removeStudent(Student std) {
    getHibernateTemplate().delete(std);
  }
}
```

Notice that the DAO class still implements the DAO interface, which outlines all of the persistent operations implemented by the DAO class. As you can see, all of the persistent methods in the DAO class use the `getHibernateTemplate()` method to access the `HibernateTemplate` object. As you saw in the previous section, `HibernateTemplate` is a Spring convenience class that delegates DAO calls to the Hibernate Session API. This class exposes all of Hibernate's `Session` methods, as well as a variety of other convenient methods that DAO classes may need. Because `HibernateTemplate` convenient methods are not exposed by the `Session` interface, you can use `find()` and `findByCriteria()` when you want to execute HQL or create a `Criteria` object. Additionally, it wraps all of the underlying exceptions thrown by the `Session` method with instances of the unchecked `org.springframework.dao.DataAccessException`.

For convenience, I recommend using the `HibernateDaoSupport` class as the base class for all Hibernate DAO implementations, but you can ignore this class and work directly with a `HibernateTemplate` instance in DAO classes. To do so, define a property of `HibernateTemplate` in the DAO class, which is initialized and set up via the Spring IoC container.

The following code shows the DAO class, which now uses `HibernateTemplate` directly. Note that this approach is not recommended because it gets you involved with the `HibernateTemplate` object in both the DAO class and the DAO configuration in the Spring context:

```
package com.packtpub.springhibernate.ch13;
import org.springframework.orm.hibernate3.HibernateTemplate;
import java.util.Collection;
public class HibernateStudentDao implements StudentDao {
  HibernateTemplate hibernateTemplate;
  public Student getStudent(long id) {
    return (Student) getHibernateTemplate().get(Student.class, new 
Long(id));
  }
  public Collection getAllStudents(){
    return getHibernateTemplate().
      find("from Student std order by std.lastName, std.firstName");
  }
  public Collection getGraduatedStudents(){
    return getHibernateTemplate().find("from Student std where 
std.status=1");
  }
  public Collection findStudents(String lastName) {
    return getHibernateTemplate().
      find("from Student std where std.lastName like "+ lastName + 
"%");
  }
  public void saveStudent(Student std) {
    getHibernateTemplate().saveOrUpdate(std);
  }
  public void removeStudent(Student std) {
    getHibernateTemplate().delete(std);
  }
  public HibernateTemplate getHibernateTemplate() {
    return hibernateTemplate;
  }
  public void setHibernateTemplate(HibernateTemplate 
hibernateTemplate) {
    this.hibernateTemplate = hibernateTemplate;
  }
}
```

The DAO class now has the `setHibernateTemplate()` method to allow Spring to inject the configured `HibernateTemplate` instance into the DAO object.

Moreover, the DAO class can abandon the `HibernateTemplate` class and use the `SessionFactory` instance directly to interact with Hibernate. The following code shows `HibernateStudentDao`, which now works directly with the `SessionFactory` object:

```
package com.packtpub.springhibernate.ch13;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate3.SessionFactoryUtils;

import java.util.Collection;

public class HibernateStudentDao implements StudentDao {

  SessionFactory sessionFactory;

  public Student getStudent(long id) {
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {
      return (Student) session.get(Student.class, new Long(id));
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }

  public Collection getAllStudents(){
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {

      Query query = session.createQuery(
                  "from Student std order by std.lastName, std.
firstName");
      Collection allStudents = query.list();
      return allStudents;
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }
  public Collection getGraduatedStudents(){
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {
      Query query = session.createQuery("from Student std where std.
status=1");
      Collection graduatedStudents = query.list();
      return graduatedStudents;
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }
  public Collection findStudents(String lastName) {
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {
      Query query = session.createQuery(
                  "from Student std where std.lastName like ?");
      query.setString(1, lastName + "%");
      Collection students = query.list();
      return students;
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }
  public void saveStudent(Student std) {
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {
      session.save(std);
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }

  public void removeStudent(Student std) {
    Session session = SessionFactoryUtils.getSession(this.
sessionFactory,
true);
    try {
      session.delete(std);
    } catch (HibernateException ex) {
      throw SessionFactoryUtils.convertHibernateAccessException(ex);
    } finally {
      SessionFactoryUtils.closeSession(session);
    }
  }

  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
}
```

In all of the methods above, the `SessionFactoryUtils` class is used to obtain a `Session` object. The provided `Session` object is then used to perform the persistence operation. `SessionFactoryUtils` is also used to translate `HibernateException` to `DataAccessException` in the catch blocks and close the `Session` objects in the final blocks. Note that this DAO implementation bypasses the advantages of `HibernateDaoSupport` and `HibernateTemplate`. You must manage Hibernate's `Session` manually (as well as exception translation and transaction management) and implement much boilerplate code.

> `org.springframework.orm.hibernate3.SessionFactoryUtils` is a Spring helper class for obtaining `Session`, reusing `Session` within transactions, and translating `HibernateException` to the generic `DataAccessException`.

This way is absolutely not the way to work with `Session` in Spring. Always use the `HibernateTemplate` class to work with `Session` objects behind the scenes. However, in cases where you need to work directly with `Session` objects, you can use an implementation of the `org.springframework.orm.hibernate3.HibernateCallback` interface as the handler to work with `Sessions`.
The following code snippet shows how this approach is:

```
  public void saveStudent(Student std) {
    HibernateCallback callback = new HibernateCallback() {
     public Object doInHibernate(Session session) throws
  HibernateException,
  SQLException {
      return session.saveOrUpdate(std);
     }
    };
    getHibernateTemplate().execute(callback);

   }
```

In this code, an implicit implementation of `HibernateCallback` is created and its only `doInHibernate()` method is implemented. The `doInHibernate()` method takes an object of `Session` and returns the result of persistence operation, `null` if none. The `HibernateCallback` object is then passed to the `execute()` method of `HibernateTemplate` to be executed. The `doInHibernate()` method just provides a handler to work directly with `Session` objects that are obtained and used behind the scenes.

Configuring Hibernate in a Spring context

As discussed in Chapter 10, Spring provides the `LocalSessionFactoryBean` class as a factory for a `SessionFactory` object. The `LocalSessionFactoryBean` object is configured as a bean inside the IoC container, with either a local JDBC `DataSource` or a shared `DataSource` from JNDI.

The local JDBC `DataSource` can be configured in turn as an object of `org.apache.commons.dbcp.BasicDataSource` in the Spring context:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
    <value>org.hsqldb.jdbcDriver</value>
  </property>
  <property name="url">
    <value>jdbc:hsqldb:hsql://localhost/hiberdb</value>
  </property>
  <property name="username">
    <value>sa</value>
  </property>
  <property name="password">
    <value></value>
  </property>
</bean>
```

In this case, the `org.apache.commons.dbcp.BasicDataSource` (the Jakarta Commons Database Connection Pool) must be in the application classpath.

Similarly, a shared `DataSource` can be configured as an object of `org.springframework.jndi.JndiObjectFactoryBean`. This is the recommended way, which is used when the connection pool is managed by the application server. Here is the way to configure it:

```
<bean id="dataSource" class="org.springframework.jndi.
JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/HiberDB</value>
  </property>
</bean>
```

When the `DataSource` is configured, you can configure the `LocalSessionFactoryBean` instance upon the configured `DataSource` as follows:

```
<bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
    …
</bean>
```

Alternatively, you may set up the `SessionFactory` object as a server-side resource object in the Spring context. This object is linked in as a JNDI resource in the JEE environment to be shared with multiple applications. In this case, you need to use `JndiObjectFactoryBean` instead of `LocalSessionFactoryBean`:

```
<bean id="sessionFactory" class="org.springframework.jndi.
JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/hiberDBSessionFactory</value>
  </property>
</bean>
```

`JndiObjectFactoryBean` is another factory bean for looking up any JNDI resource.

When you use `JndiObjectFactoryBean` to obtain a preconfigured `SessionFactory` object, the `SessionFactory` object should already be registered as a JNDI resource. For this purpose, you may run a server-specific class which creates a `SessionFactory` object and registers it as a JNDI resource.

`LocalSessionFactoryBean` uses three properties: `datasource`, `mappingResources`, and `hibernateProperties`. These properties are as follows:

- `datasource` refers to a JDBC `DataSource` object that is already defined as another bean inside the container.
- `mappingResources` specifies the Hibernate mapping files located in the application classpath.
- `hibernateProperties` determines the Hibernate configuration settings.

We have the `sessionFactory` object configured as follows:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="mappingResources">
    <list>
      <value>com/packtpub/springhibernate/ch13/Student.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch13/Teacher.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch13/Course.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.
HSQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.max_fetch_depth">2</prop>
    </props>
  </property>
</bean>
```

The `mappingResources` property loads mapping definitions in the classpath. You may use `mappingJarLocations`, or `mappingDirectoryLocations` to load them from a JAR file, or from any directory of the file system, respectively.

It is still possible to configure Hibernate with `hibernate.cfg.xml`, instead of configuring Hibernate as just shown. To do so, configure `sessionFactory` with the `configLocation` property, as follows:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="configLocation">
    <value>/conf/hibernate.cfg.xml</value>
  </property>
</bean>
```

Note that `hibernate.cfg.xml` specifies the Hibernate mapping definitions in addition to the other Hibernate properties.

When the `SessionFactory` object is configured, you can configure DAO implementations as beans in the Spring context. These DAO beans are the objects which are looked up from the Spring IoC container and consumed by the business layer. Here is an example of DAO configuration:

```
<bean id="studentDao"
class="com.packtpub.springhibernate.ch13.HibernateStudentDao">
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>
```

This is the DAO configuration for a DAO class that extends `HibernateDaoSupport`, or directly uses a `SessionFactory` property. When the DAO class has a `HibernateTemplate` property, configure the DAO instance as follows:

```
<bean id="studentDao" class="com.packtpub.springhibernate.ch13.
HibernateStudentDao">
  <property name="hibernateTemplate">
    <bean class="org.springframework.orm.hibernate3.
HibernateTemplate">
      <constructor-arg>
        <ref local="sessionFactory"/>
      </constructor-arg>
    </bean>
  </property>
</bean>
```

According to the preceding declaration, the `HibernateStudentDao` class has a `hibernateTemplate` property that is configured via the IoC container, to be initialized through constructor injection and a `SessionFactory` instance as a constructor argument.

Now, any client of the DAO implementation can look up the Spring context to obtain the DAO instance. The following code shows a simple class that creates a Spring application context, and then looks up the DAO object from the Spring IoC container:

```
package com.packtpub.springhibernate.ch13;

public class DaoClient {
  public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
                        "com/packtpub/springhibernate/ch13/
applicationContext.xml");
    StudentDao stdDao = (StudentDao)ctx.getBean("studentDao");
    Student std = new Student();

    //set std properties

    //save std
    stdDao.saveStudent(std);
  }
}
```

Spring transaction management

One reason to integrate Hibernate with Spring is transaction management. Spring provides a transaction abstraction layer over the Hibernate transaction API, and enables persistent operations to participate in global transactions. Moreover, Spring provides declarative transaction demarcation, which produces more readable and maintainable Java code. The declarative approach lets us change the transaction strategy easily, without changing the code.

The Spring transaction demarcation API has two classes for working with Hibernate applications:

- `org.springframework.transaction.support.TransactionTemplate` for a programmatic approach.
- `org.springframework.transaction.interceptor.TransactionProxyFactoryBean` for a declarative approach.

Behind this API, you may use one of Spring's two transaction managers:

- `org.springframework.orm.hibernate3.HibernateTransactionManager`: Use this when the application involves a single data source and Hibernate alone is needed. This covers local transactions executed on a single `SessionFactory`. This manager is commonly used, since most Hibernate applications work with a single database.
- `org.springframework.transaction.jta.JtaTransactionManager`: This is Spring's global JTA transaction manager. Use it when the application participates in global transactions in a Java EE environment, in which multiple `SessionFactory` methods are involved, and transactions are scattered over them.

Note that we can develop DAO classes that are not involved in transaction management. This is an advantage Spring provides for the application, so that all DAO classes work with transactional `Session` objects provided behind the scenes by the Spring IoC container.

You configure Spring's transactions by setting up the transaction-manager instance as a bean inside the IoC container. The bean configuration depends on which transaction strategy you are using: local or global.

Lets look at the transaction configuration in detail.

Local transactions

When the application uses only a single data source (a single `SessionFactory` in Hibernate), you can define the transaction manager as an instance of `HibernateTransactionManager` as follows:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.
HibernateTransactionManager">
  <property name="sessionFactory">
    <ref bean="sessionFactory"/>
  </property>
</bean>
```

Then, define DAO and Service beans, respectively, as instances of the DAO and service classes. Here is an example:

```
<bean id="persistenceService"
        class="com.packtpub.springhibernate.ch13.PersistenceService">
    <property name="studentDao">
      <ref bean="studentDao"/>
    </property>
  </bean>
<bean id="studentDao"
      class="com.packtpub.springhibernate.ch13.HibernateStudentDao">
  <property name="sessionFactory">
    <ref bean="sessionFactory"/>
  </property>
</bean>
```

Finally, the DAO instances are wrapped in a proxy transaction. As you saw in Chapter 11, transactions are cross-cutting concerns that are not dedicated to the function of a particular method. Instead, they are scattered over many persistence methods. With Spring, DAO functionality can be split into two modules:

- **DAO implementations**, which perform persistence operations.
- **Transaction advice**, which defines how persistence operations are performed in transactions.

Modularizing DAO implementations to carry out persistence operations, and not transaction management, avoids boilerplate transaction-management code in every persistence method. To apply transaction advice to the target methods in the DAO implementations, we need proxy objects.

Each *proxy* is an object that intermediates between two other objects (a calling object and an invoked object), and applies a concern (a transaction in our case) to the object invocation. Actually, the business layer always works with instances of proxies instead of service objects. Here is an example of the transaction proxy definition:

```
<bean id="studentDao"
class="org.springframework.transaction.interceptor.
       TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="target">
    <ref bean="persistenceService"/>
  </property>
```

```
    <property name="transactionAttributes">
      <props>
        <prop key="save*">PROPAGATION_REQUIRED</prop>
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>
```

As you saw in Chapter 12, the `TransactionProxyFactoryBean` object is configured with three properties:

- `transactionManager`, declared as an individual bean, is an object which provides an abstraction API between the application code and Hibernate transaction API.
- `target` represents the object to which the transaction concern must be applied.
- `transactionAttributes` defines how to select the transactional methods. For instance, `save*` selects all methods of the target object that start with the `save` word.

In our example, `PROPAGATION_REQUIRED` and `readOnly` specify, respectively, how multiple transactions participate, and whether the transaction is allowed to only read the data.

Global transactions

You can configure Spring to use global transactions for synchronizing persistent operations that are performed in a Java EE environment across multiple data sources. The configuration process is similar to configuring a local transaction, but you need to define multiple data sources, as well as multiple `SessionFactory` and DAOs.

The transaction-manager object is now defined as an instance of `JtaTransactionManager`, instead of `HibernateTransactionManager`.
Here is an example:

```
<beans>
  <!-- the datasource1 declartion, registered to node ds1 on the JNDI 
tree-->
<bean id="datasource1" class="org.springframework.jndi.
JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/ds1</value>
    </property>
  </bean>
```

```
    <bean id="datasource2" class="org.springframework.jndi.
  JndiObjectFactoryBean">
      <property name="jndiName">
        <value>java:comp/env/jdbc/ds2</value>
      </property>
    </bean>
    <!-- the sessionFactory1 declaration, which uses datasource1-->
    <bean id="sessionFactory1"
          class="org.springframework.orm.hibernate3.
  LocalSessionFactoryBean">
      <property name="mappingResources">
        <list>
          <value>Student.hbm.xml</value>
        </list>
      </property>
      <property name="hibernateProperties">
        <props>
          <prop key="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
          </prop>
        </props>
      </property>
      <property name="dataSource">
        <ref bean="dataSource1"/>
      </property>
    </bean>
    <!-- the sessionFactory2 declartion, which uses datasource2-->
    <bean id="sessionFactory2"
          class="org.springframework.orm.hibernate3.
  LocalSessionFactoryBean">
      <property name="mappingResources">
        <list>
          <value>Teacher.hbm.xml</value>
          <value>Course.hbm.xml</value>
        </list>
      </property>
      <property name="hibernateProperties">
        <props>
          <prop key="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
          </prop>
        </props>
      </property>
      <property name="dataSource">
        <ref bean="dataSource2"/>
      </property>
    </bean>
    <!-- TransactionManager declaration -->
    <bean id="transactionManager"
```

```
        class="org.springframework.transaction.jta.
JtaTransactionManager"/>
  <!-- A DAO configuration with the sessionFactory1-->
  <bean id="studentDao"
class="com.packtpub.springhibernate.ch13.HibernateStudentDao">
    <property name="sessionFactory">
      <ref bean="sessionFactory1"/>
    </property>
  </bean>
  <!-- A DAO configuration with the sessionFactory2-->
  <bean id="teacherDao"
class="com.packtpub.springhibernate.ch13.HibernateTeacherDao">
    <property name="sessionFactory">
      <ref bean="sessionFactory2"/>
    </property>
  </bean>
  <!-- A service instance configuration which reside in the business
layer -->
  <!-- and works with both StudentDao and TeacherDao instances-->
  <bean id="persistenceServiceTarget"
        class="com.packtpub.springhibernate.ch13.PersistenceService">
    <property name="studentDao">
      <ref bean="studentDao"/>
    </property>
    <property name="teacherDao">
      <ref bean="teacherDao"/>
    </property>
  </bean>
  <!-- A proxy configuration, which applies the transactions on the
DAO methods-->
  <bean
   id="persistenceService"
   class="org.springframework.transaction.interceptor.
          TransactionProxyFactoryBean">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    <property name="target">
      <ref bean="persistenceServiceTarget"/>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="save*">PROPAGATION_REQUIRED</prop>
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>
</beans>
```

Alternatively, you can configure the transaction manager as a bean of type `JndiObjectFactoryBean` in the IoC container. `JndiObjectFactoryBean` is a factory bean which obtains the transaction-manager object by looking up a JNDI resource. If you choose this strategy, the `SessionFactory` configuration changes as follows:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
  <property name="jtaTransactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="mappingResources">
    <list>
      <value>com/packtpub/springhibernate/ch13/Student.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch13/Teacher.hbm.xml</value>
      <value>com/packtpub/springhibernate/ch13/Course.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
      </prop>
    </props>
  </property>
</bean>
```

You must also configure a transaction-manager object:

```
<bean id="transactionManager"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/jdbc/mytm</value>
  </property>
</bean>
```

In this way, we configured `transactionManager` as a bean produced by `JndiObjectFactoryBean`. This factory bean returns transaction-manager objects by looking up a server-specific JNDI location.

Please refer to Chapter 12 for more details about transaction management.

Summary

In this chapter, you learned about the Data Access Object (DAO) pattern. This pattern allows all interaction with the data tier to be done through specific classes, called DAO classes. According to this pattern, a DAO interface is defined to correspond to each DAO class, defining all persistent operations that the business layer needs. The main goal is to decouple the business layer from the data layer. DAO interfaces are the only references that the business layer uses to perform persistent operations. At this point, Spring is configured to instantiate these references with the actual implementations at runtime.

You can integrate Hibernate with Spring to simplify DAO implementation. Spring provides a transaction abstraction layer on the local transaction API provided by Hibernate, and allows declarative transaction demarcation. Moreover, it converts `HibernateException` into Spring's generic `DataAccessException` hierarchy, and allows the use of IoC in the data tier.

Spring supports DAO implementation with two helper classes: `org.springframework.orm.hibernate3.HibernateTemplate` and `org.springframework.orm.hibernate3.support.HibernateDaoSupport`. All DAO classes extend `HibernateDaoSupport` and use `HibernateTemplate` to perform persistent operations. Spring provides `org.springframework.orm.hibernate3.LocalSessionFactoryBean` as a factory bean to configure and set up a `SessionFactory` object in an IoC-style manner. For transaction management, Spring provides `org.springframework.orm.hibernate3.HibernateTransactionManager` and `org.springframework.transaction.jta.JtaTransactionManager` for managing local and global transactions, respectively. Moreover, Spring provides `org.springframework.transaction.interceptor.TransactionProxyFactoryBean`, which acts as a proxy to apply transaction concerns to persistent operations.

14 Web Development with Hibernate and Spring

Most enterprises applications today provide a web-based user interface to interact with end users. This means Hibernate and Spring developers often encounter situations involving web development. The problem is that the JSP and servlet technologies typically used to produce dynamic web content are not simple Java classes to be configured in the Spring context.

This chapter begins by exploring how Model-View-Controller (MVC) web frameworks are integrated with Spring, and then briefly discusses the Spring MVC components and features.

[

When a web application uses Hibernate without Spring, every JSP page or servlet class can simply obtain a service object to interact with the business layer, and persist the entity objects via Hibernate behind the scenes. The problem is when Hibernate and Spring are used together, we expect Spring to inject objects via IoC in JSP or Servlet pages. As soon as you obtain Spring-managed objects in JSP pages or servlet classes, you can interact with Hibernate via the Spring-managed service objects.

]

Problem definition

Let's take a simple portion of our sample application to explain how to use Spring in web applications. Consider this use case: the user submits a web request with a student identifier to see a student's details. The web layer of the application obtains a `StudentService` instance by looking up the Spring context. The `StudentService` is configured as a bean in `applicationContext.xml`, as follows:

```
<bean id="studentService"
      class="com.packtpub.springhibernate.ch14.StudentService">
  <property name="studentDao">
    <ref local="studentDao"/>
  </property>
</bean>
```

The web layer then uses the `getStudent()` method on the `StudentService` instance to get the requested student. The `getStudent()` method signature provided by `StudentService` is as follows:

```
public Student getStudent(String studentId)
```

In upcoming sections, we will discuss how each web technology deals with this problem. Before we do that though, let's look at the basic configuration that is always applied to each framework.

Common configuration for Spring integration

Web applications include JSP pages and servlets, which provide dynamic content for end users. To integrate web applications with Spring, we must access Spring services in these JSP pages and servlets. Fortunately, we can do this using `ServletContext`, a long-lived object initialized at application startup that's accessible in every JSP page and servlet. The servlet specification also defines that we can implement a context listener to execute customized behavior when the `ServletContext` is initialized or destroyed. Using the `ServletContext` and a context listener, we can load the Spring context at application startup, and store the loaded context in `ServletContext`.

Spring provides `org.springframework.web.context.ContextLoaderListener` as the context listener, which loads the Spring context, and stores it in the `ServletContext`. To use this listener, you need to configure it in `web.xml`, as follows:

```
<web-app>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...
</web-app>
```

By default, `ContextLoaderListener` loads the Spring context file named `applicationContext.xml` located in the application's `/WEB-INF` directory. However, you can load a Spring context with another name or location. For this purpose, you can use the `<context-param>` element in `web.xml` to specify the name and location of the Spring context, as follows:

```
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/springContext.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...
</web-app>
```

This example specifies `springContext.xml` as the name of the Spring context file, which is located in the web application's root directory.

Spring provides `org.springframework.web.context.support.WebApplicationContextUtils` as a utility class to access the Spring context in a JSP page. This class provides the following two static methods to access the application context stored in the `ServletContext`:

```
public static WebApplicationContext
  getRequiredWebApplicationContext(ServletContext sc)
```

and

```
public static WebApplicationContext getWebApplicationContext
(ServletContext sc)
```

These methods do the same job with the only difference that if no Spring context is found, the `getWebApplicationContext()` method returns `null`, but `getRequiredWebApplicationContext()` throws an `IllegalStateException`. The following code shows how you can call `getRequiredWebApplicationContext()` in a JSP page to obtain the `ApplicationContext`:

```
ApplicationContext context =
    WebApplicationContextUtils.getRequiredWebApplicationContext(
     application);
```

`application` represents the `ServletContext` object that is provided by the web container in all JSP pages.

[The `getRequiredWebApplicationContext()` method is useful when the context is absolutely required, and you need the application to treat the absence of the Spring context file as an error.]

You may use the same code to obtain the Spring context in a servlet class, but you need to obtain the `ServletContext` via the `getServletContext()` method, instead of using the `application` variable, as follows:

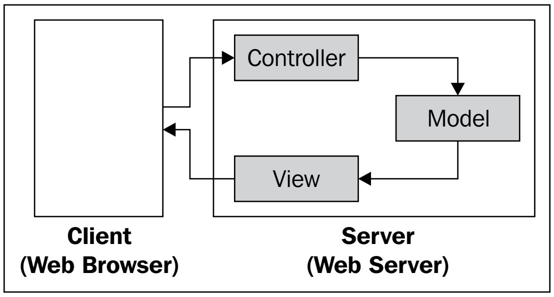
```
ServletContext sContext = getServletContext();
ApplicationContext appContext =
WebApplicationContextUtils.getRequiredWebApplicationContext(sContext);
```

Before discussing how to integrate practical web frameworks with Spring, let's look at the common MVC pattern implemented by web frameworks.

The MVC architectural pattern

The Model-View-Controller (MVC) pattern's purpose is to decouple user interfaces, data, and business logic, which are called views, models, and controllers, respectively. This separation simplifies application architecture and makes development, testing, and maintenance easier. This architecture avoids code duplication in many situations, and provides reusability in views, models, and business logic.

All web frameworks implement the MVC pattern. The following figure depicts the basic scenario that is accomplished in web frameworks for a typical user request:

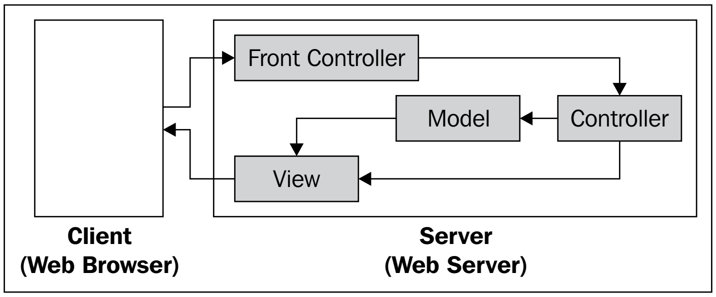


The **controller** is an object that processes the request, updates the model object, and forwards the request to the view. The JSP page uses the updated model to generate output for the user.

The following list provides a summary of the MVC components in web frameworks:

- **Model components**: These components provide an interface to the data and services used by the application. In some cases, model components are simple value objects, represented as instances of simple JavaBeans, transmitted between view and controller components. However, model components can be more complex and include business logic to access and manipulate data, such as Enterprise JavaBeans (EJBs). Model components allow controller components to work transparently with view components.
- **View components**: These components are responsible for generating the application view, which is what the user sees, based on the model components. Most of the time, view components are simple JSP or HTML pages. However, they can be of any other type, such as PDFs.
- **Controller components**: These components have a crucial role in the MVC model. Sometimes, it is possible to omit a model component, or even a view component, but controller components are essential. The controller receives requests, processes them, creates the model, and forwards the prepared model to the view component.

Web frameworks use an additional object, called a **front controller servlet**. This object, which is responsible for receiving requests and dispatching them to the appropriate controllers, allows us to create cleaner code that eliminates unnecessary request processing in controllers. The following figure shows the role of the front controller servlet in a web framework's architecture:



Now that I've given some background material regarding web frameworks, let's look at some actual web frameworks and see how we can integrate them with Spring in order to provide Spring services in a web implementation.

This chapter looks at two primary web frameworks that implement the MVC pattern. First, I'll discuss Spring, with which you can easily use other Spring services in web implementations. Then, I'll explain Struts, an older, very popular web framework. You can find more information about integrating other frameworks in the Spring documentation.

Spring MVC web framework

Spring is a popular MVC-based web framework. Besides many services, such as IoC, AOP, and transaction management provided by Spring and discussed so far, Spring provides full functionality for building robust web applications. This functionality is supported in two ways. First, Spring can be integrated with other web frameworks, such as Struts, WebWork, Java Server Faces, and Tapestry in order to use Spring's services such as IoC and AOP, and second, Spring provides its own MVC web framework.

In this section, we will look at Spring MVC and its various components, including Dispatcher Servlet, Handler Mappings, Controllers, Model and Views, and View Resolvers. As advantages of Spring, Spring MVC is not tightly coupled with Servlet and JSP to render the View to the clients, and can be integrated with other view technologies like Velocity, Freemarker, Excel, and PDF.

Spring MVC workflow

Before digging into Spring MVC details, let's explore Spring MVC architecture and various components involved. The main components are front controller, handler mappings, controllers, view resolvers, and views. The basic workflow to handle a request are as follows:

1. A web request is submitted to the application.
2. The **Front Controller** receives the request and interprets it to find the appropriate handle mapping. The handle mapping maps the request to an appropriate **Controller** object.
3. The front controller forwards the request to the controller object.
4. The controller receives the request, processes it, and returns a **Model and View** object to the front controller.
5. The front controller uses **View Resolver**s to find the appropriate **View** object.
6. The view object is used to render the result which then sends it back to the client.

Now let's continue our discussion with a bit of a detailed explanation on each component.

Front controller

The front controller in Spring MVC is an object of `org.springframework.web.servlet.DispatcherServlet`. As the first step to use Spring MVC, you need to configure front controller in `web.xml` to intercept all of the client requests. This is done as follows:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

This configuration tells the container all requested URLs with `.html` extension must be processed by the `DispatcherServlet` object. In addition to being a front controller object, the `DispatcherServlet` object finds the Spring application context file named as **<servlet-name>**—`servlet.xml` and located in `/WEB-INF` directory. In our case, this means `DispatcherServlet` tries to find `/WEB-INF/action-servlet.xml`, and then loads and binds it to the `ServletContext` object. Alternatively, it is also possible to use `ContextLoaderListener` discussed to load the Spring context files. You need to use this approach if you have different file names, or the context files are located in a different directory than `WEB-INF`.

Handler mappings

As mentioned, the `DispatcherServlet` object finds an appropriate handler mapping object, and then uses it to map the request to a controller. Actually, the handler mapping object tells Spring how the requested URL must be mapped to controllers. The most common types of handler mappings are `BeanNameUrlHandlerMapping`, `ControllerClassNameHandlerMapping`, `SimpleUrlHandlerMapping`. These handler mappings are explained as follows:

BeanNameUrlHandlerMapping

This handler mapping simply maps the requests to controller based on the name of the requested URLs. For instance, for the URL `http://hostname:port/webAppName/viewStudent.html`, the request is passed to a controller configured as:

```
<bean name="/viewStudent.html"
      class="com.packtpub.springhibernate.ch14.ViewStudentController">
</bean>
```

To use this handler mapping, `BeanNameUrlHandlerMapping` must be configured as a bean in the Spring context as:

```
<bean id="beanNameUrl"    class="org.springframework.web.servlet.
handler.BeanNameUrlHandlerMapping"/>
```

ControllerClassNameHandlerMapping

This handler mapping takes the name of the controller from the requested URL with some modifications. For instance, requests for any URLs that follow the pattern `/viewStudent*` will be processed by the controller `ViewStudentController`. This handler mapping simply strips `Controller` off the controller's name to determine the URLs for that controller. To use this kind of mapping, you need to have a bean definition as follows:

```
<bean id="controllerClassName" class=" org.springframework.web.
servlet.mvc.support.ControllerClassNameHandlerMapping "/>
```

SimpleUrlHandlerMapping

Directly maps the URLs to controllers. The following code shows how this mapping is configured and used:

```
<bean id="simpleUrlMapping"
class="org.springframework.web.servlet.handler.
SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/viewStudent.html">
                viewStudentController
            </prop>
            <prop key="/listStudents.html">
               listStudentController
            </prop>
            <prop key="/deleteCourse.html">
               deleteCourseController
            </prop>
        </props>
    </property>
</bean>
```

The URL `/viewStudent.html` will be processed by a controller named `viewStudentController` in the Spring context.

Controllers

Controllers have the responsibility to process the requests, do the business logic, and produce the viewable result. Spring provides the `Controller` interface as the super-interface for all controller implementations. This interface is as follows:

```
public interface Controller  {
ModelAndView handleRequest(HttpServletRequest request,
                      HttpServletResponse response) throws Exception;
}
```

To make it easy, Spring provides a variety of built-in controller implementations, which is useful for a specific purpose. The most common controllers are as follows:

- `AbstractController`
- `AbstractCommandController`
- `SimpleFormController`
- `CancellableFormController`
- `MultiActionController`
- `ParameterizableViewController`
- `ServletForwardingController`
- `ServletWrappingController`
- `UrlFilenameViewController`

Let's have a quick look at some of these controllers and the way they are used.

AbstractController

This class provides the basic implementation for all other Spring built-in controllers. Therefore, you shouldn't implement a custom implementation of the `Controller` interface if you want to implement a custom controller from scratch. The `AbstractController` class provides the `protected handleRequestInternal()` method to be implemented by subclasses. This method receives two parameters as object of `HttpServletRequest` and `HttpServletResponse`, just like servlet methods and returns an object of `ModelAndView`. The `ModelAndView` object is a Spring specific class representing the model, the result of request processing, and information about the target page that will render the model. The `ModelAndView` object doesn't directly refer to a specific page. Instead, it just provides a logical view name to be mapped to the physical path of the target page. The `ModelAndView` class will be discussed later in this chapter.

The following code shows how our `ModelAndView` can be implemented with `AbstractController`:

```
package com.packtpub.springhibernate.ch14;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ViewStudentController extends AbstractController {
  StudentService studentService;

  public ModelAndView handleRequestInternal(HttpServletRequest
request, HttpServletResponse response)
          throws Exception {
    // create a model-and-view object
    ModelAndView modelAndView = new ModelAndView("viewStudent");
    String studentId = request.getParameter("studentId");
    Student student = studentService.getStudent(studentId);
    // add an object to render in the view
    modelAndView.addObject("student", student);
    return modelAndView;
  }

  public void setStudentService(StudentService studentService) {
    this.studentService = studentService;
  }
}
```

Note that the dispatcher servlet always calls the `handleRequest()` method to forward the request to a `AbstractController` object. The `AbstractController` object does some control checking. Namely, checking if the HTTP request method is supported by this controller, checking whether session exists if it needs the session, setting cache control headers, and so on, and then calls the `handleRequestInternal()` method internally to process the request.

AbstractCommandController

This controller is essentially used to process a user request, which always build with parameters. This class provides a `handle()` method which takes an object parameter in addition to `HttpServletRequest` and `HttpServletResponse`. With this kind of controller, we can deal with a command object, including all of the submitted values, instead of working with `HttpServletRequest` to obtain the request parameters. The command object can be a simple POJO object populated with request parameters. The following code shows our simple example, which now is implemented with `AbstractCommandController`:

```
public class ViewStudentController extends AbstractCommandController {

    StudentSeverice service;

    public ViewStudentController() {
        setCommandClass(StudentInfo.class);
    }

    protected ModelAndView handle(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException errors) throws Exception {
        StudentInfo info = (StudentInfo) command;
        Student std = //service object to obtain the student
        if(std == null){
            return new ModelAndView("failure");
        }else{
           return new ModelAndView("success", "student", std);
        }
    }

    public void setService(StudentSeverice service) {
        this.service = service;
    }
}
```

Note how in the constructor we introduced the command class to the controller.

[The `handle()` method takes a `BindException` object, which represents the validation result of command object, if validation has been set for that command.]

The following code shows the command class:

```
public class StudentInfo {
    String id;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
}
```

SimpleFormController

`SimpleFormController` is used to handle request submission. This controller provides a variety of methods to process the user submission.

The following code uses the `onSubmit()` method in the controller to process the request:

```
public ModelAndView onSubmit(Object command) throws Exception{
        StudentInfo info = (StudentInfo) command;
        Student std = //service object to obtain the student
        if(std == null){
            return new ModelAndView("failure");
        }else{
           return new ModelAndView("success", "student", std);
        }
    }
```

The input page, which provides the submission information to this controller, and the success page, which renders the result, are respectively specified via the `formView` and `successView` properties. The following bean configuration shows how `SimpleFormController` is configured:

```
<bean id = "studentView" class="StudentViewController">

    <property name="formView">
        <value>allStudents</value>
    </property>
```

```
        <property name="successView">
            <value>studentDetails</value>
        </property>
    </bean>
```

Note that the values are just logical view names, which are then mapped to the physical page locations.

CancellableFormController

This controller provides more functionality over `SimpleFormController`. By this kind of controller, it is possible to handle the user cancelation for the form submission by its `onCancel()` method. The following code shows how `onCancel()` is implemented:

```
public ModelAndView onCancel(Object command) throws Exception{
        return new ModelAndView("viewAll");
}
```

The `onCancel()` method is called to process the request when the framework finds a request parameter named _`cancel`. You can change this parameter name by the `cancelParamKey` property for the `CancellableFormController` object. The following code shows how `CancellableFormController` is configured:

```
<bean id="viewStudentController" class="com.packtput.springhibernate.
ch14.ViewStudentController">
        <property name="useCacheControlHeader" value="false"/>
        ...
        <!-- Required for CancellableFormController -->
        <property name="cancelParamKey" value="cancel"/>
        <property name="cancelView" value="redirect:myCancel.action"/>
        <!-- Standard FormController properties -->
        <property name="formView" value="studentList.jsp"/>
        <property name="successView" value="success"/>
</bean>
```

Model and View

In the Spring MVC, each controller returns an object of `org.springframework.web.servlet.ModelAndView`, representing the model and the logical view name to render the model. The `ModelAndView` object includes three parts, the model, a map object, and the view, an implementation of the `org.springframework.web.servlet.View` interface. The view can be a simple string literal that is mapped by the view resolver to an actual JSP page.

View resolvers

Finally, when the data is ready to render, the view resolver is invoked to find the actual JSP page responsible to render the result. Similar to controllers, Spring provides a variety of view resolvers, all as implementation of the `org.springframework.web.servlet.ViewResolver` interface. The view resolvers Spring provides are `BeanNameViewResolver`, `FreeMarkerViewResolver`, `InternalResourceViewResolver`, `JasperReportsViewResolver`, `ResourceBundleViewResolver`, `UrlBasedViewResolver`, `VelocityLayoutViewResolver`, `VelocityViewResolver`, `XmlViewResolver`, `XsltViewResolver`. Here, we will just look at two most common view resolvers, `InternalResourceViewResolver` and `BeanNameViewResolver`.

InternalResourceViewResolver

This view resolver simply appends prefix and suffix to the logical view name to find the physical page locations. For instance, the following configured view resolver object maps any logical view name to a JSP page with the same name located in the `WEB-INF` directory:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
    <property name="prefix"><value>/WEB-INF/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

By this code, if the logical view name has been set to `viewStudent`, the `/WEB-INF/viewStudent.jsp` page is chosen to render the result.

BeanNameViewResolver

`BeanNameViewResolver` allows us to dynamically generate in any format, which is something that is not possible with `InternalResourceViewResolver`. For instance, suppose we want to generate a PDF page from the model on the fly. It is possible to use `BeanNameViewResolver` with an implemented `AbstractPdfView` object to generate the PDF. The following shows the code to do so:

```
<bean id="beanNameResolver" class="org.springframework.web.servlet.
view.BeanNameViewResolver"/>
<bean id = "pdf" class = "com.packtpub.springhibernate.ch14.
CustomPdfGenerator"/>
```

Any `ModelAndView` object with the logical `pdf` view name will then be processed by `CustomPdfGenerator`.

Render the result through JSP

Finally, a JSP page can render the result. The following code shows a partial JSP page to render the student instance:

```
<table width="637" border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td>Student Number</td>
        <td><c:out value="${student.stdNo}"/></td>
      </tr>
      <tr>
        <td>First Name</td>
        <td><c:out value="${student.firstName}"/></td>
      </tr>
      <tr>
        <td>Last Name</td>
        <td><c:out value="${student.lastName}"/></td>
      </tr>
      <tr>
        <td>SSN</td>
        <td><c:out value="${student.ssn}"/></td>
      </tr>
      <tr>
        <td>Birthday</td>
        <td><fmt:formatDate value="${student.birthday}"
          type="both"/></td>
      </tr>
      <tr>
        <td>Entrance Date</td>
        <td><fmt:formatDate value="${student.entranceDate}"
             type="both"/></td>
      </tr>
    </table>
```

There is nothing special about this page, only that is uses JSTL to render the student instance.

Summary

This chapter discussed how to use Spring in web applications. We started with a quick overview of the MVC pattern. Our discussion continued with the common configuration to integrate web frameworks with Spring. We used the `org.springframework.web.context.ContextLoaderListener` class to load the Spring application context at application startup, and then store it in the `ServletContext`. In any JSP page or servlet class, the stored `ApplicationContext` can be accessed through Spring utility class: `org.springframework.web.util.WebApplicationContextUtils`.

To develop a web application with the Spring MVC framework, you must configure the `DispatcherServlet` as the front controller. `DispatcherServlet` accepts all web requests, and forwards each to the appropriate controller. You must also configure the web application to use `ContextLoaderListener` to load the Spring context files at application startup. You will be able to implement controller classes as an implementation of `Controller`, and configure them in the Spring context as beans. Additionally, you should specify a resolver, which resolves names to paths, and use handler mapping to map URL patterns to bean names. You can then render the result in JSP pages.

15
Testing

When developing an application, we need to apply a testing mechanism to ensure that the application works as we intend. In practice, different kinds of testing exist, each with different goals, including quality assurance testing, acceptance testing, performance testing, integration testing, and unit testing. Customer satisfaction tests, performed by the quality assurance team, verify that the application meets the customer's needs and expectations. Acceptance tests exercise all operating conditions of the user's environment or features of the system. Performance tests evaluate how the application responds to a large number of requests, specifically testing the application's robustness, memory usage, and response time. Integration tests ensure that all of the application's parts work correctly when they are assembled in a simulated production environment.

However, not all of these tests are the subject of this chapter. Instead, this chapter discusses two types of testing: unit testing, to test all smallest parts of the application work properly, and integration testing, to make sure these smallest parts work correctly when they merge. Unit testing is the first step in the application testing process and uncovers defects and program errors. The developers write and execute simple blocks of code as tests, verifying the functions, behavior, and performance of an object to make sure that the object meets its functional requirements. Unit testing allows developers to produce more productive and effective code with fewer bugs. Integration testing means testing the combination of units.

This chapter discusses the aspects of Hibernate and Spring applications that require testing and how to implement test cases for those aspects. Our discussion begins with a quick overview of unit testing and the popular unit test framework JUnit. We'll then discuss how JUnit lets us create effective test code and run it automatically.

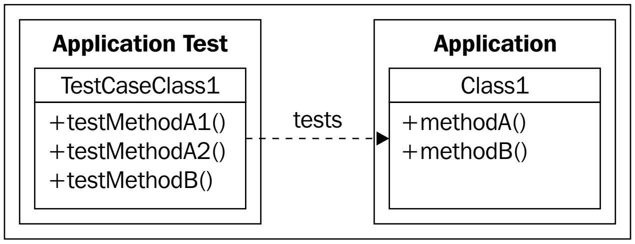
An introduction to unit testing

A **unit test** is a chunk of code that examines the functionality of a very small part of the application code: in practice, a method. Developers write unit tests to find defects and verify that their code meets its requirements in isolation from other code in the application. The test should provide all of the possible circumstances, situations, and conditions in which the code may be executed and that may potentially break the code. Unit testing is part of the philosophy that, when we're developing code, we should find any defects as soon as possible. The longer a bug exists, the more dramatically the cost of finding and fixing a bug increases. Using unit testing, developers can learn, as soon as possible, where and how their code breaks, fix the problem, and test again and again until no problem exists.

During application development, when you add a component to the application, you must implement and add a unit test, as well. Each unit test guarantees that a component works as expected. After the application has been fully developed, you will have a set of unit tests for all of the application's components. If all components pass the tests without any failures, you can conclude that the application's components work properly in isolation from each other.

In practice, we implement a test class for each class of the application. Each test class implemented with JUnit is called a **test case**. For each method of the class being tested, we implement one or more test methods in the test class. To examine whether a method works as it should, we must call the method with its required parameters and conditions in a test method, and then verify that the method's result is what we expect.

The following figure shows the relationship between application classes and unit tests:



Test classes include test methods. Each test method provides a particular condition that may break the method under test, invokes the method under test, and verifies the result.

Unit testing with JUnit

JUnit is a Java framework for unit testing. The benefits of using JUnit, rather than implementing unit tests on your own code, can be summarized as follows:

- Create simple and effective test programs.
- Create test programs in a public and standard structure.
- Receive a report of the test's results.
- Execute the tests automatically with Ant.

JUnit is the foundation for more specialized unit-testing tools, such as Cactus, mocks, JMeter, Jester, JUnitPerf, DbUnit, and HttpUnit.

JUnit can be found at `http://www.junit.org`, along with sample usages and documentation.

The structure of test classes with JUnit

New releases of JUnit support annotation-based unit tests. Therefore, we can develop unit test in a different style of old releases of JUnit. Here we will discuss the old and new approaches and for that we will use JUnit 3.8 as the old version and JUnit 4.0 as the new version.

JUnit provides a basic test class, called `junit.framework.TestCase`. To develop with JUnit 3.8, we must always create a subclass of `TestCase` as a test class for each class being tested with JUnit. `TestCase` is a JUnit utility class which provides basic assertion methods (discussed soon) to verify test results. Moreover, JUnit's test runners identify test classes that extend the `TestCase` class in order to execute them.

As a convention, name test classes after the class being tested with a common prefix or suffix, such as `TestCase`. This naming convention allows you to identify the test classes throughout the application and give Ant a pattern for automatic test execution. As a mandatory naming convention with JUnit 3.8 and before, the name of the test method is the name of the method being tested with the `test` prefix. This convention allows JUnit's test runners to distinguish the test methods from other helper methods inside test classes. For instance, a method that tests the `saveStudent()` method would be named `testSaveStudent()`. If more than one test method exists for a method, you can append a number: `testSaveStudent1()`, `testSaveStudent2()`, and so on.

With JUnit 4.0, the test case class doesn't need to extend the `TestClass`, and the test method's name no longer needs to start with test. Any class can be used as a test case, in which test methods are marked with the `@Test` annotation.

Whether you are using JUnit 3.8 or JUnit 4, test classes must follow these steps, in order:

1. Provide the required conditions and circumstances that are needed to test the method.
2. Call the method being tested in the test method.
3. Verify the method's result to check whether it operated as expected.
4. Perform finalization operations.

Let's go through this process.

Setting up the preconditions

Each test case has its own preconditions, which must be provided before execution of any test. With JUnit 3.8, you need to implement the `TestCase`'s `setUp()` method as follows:

```
protected void setUp();
```

This method is automatically called by JUnit before each test method is executed. You can override it to provide the preconditions for executing the test methods (all methods in the test class named with the `test` prefix). Typically, you may use this method to create the required objects for invoking the method being tested, look up a resource, and so on.

With JUnit 4, you can mark any method with the `@org.junit.Before` annotation to act as the `setUp()` method.

Call the method being tested and verify the result

With JUnit 3.8, test methods with the `test` prefix call the methods being tested with the required parameters and check for the expected result. Here is an example of a test method:

```
public void testSaveStudent() {
    //call the method being tested
    //verify the result
}
```

Test methods have `void` as the return type and take no argument. These methods call the methods of the object(s) prepared by the `setUp()` method. To verify the result, the `TestCase` class provides some helper methods, called **assertion methods**. Assertion methods help you compare the result with the value you expected.
For instance, if you expect the return value of a method to be 1, you may use the following code to verify it:

```
assertEquals("Should be 1", 1, result);
```

The `result` is the actual returned value of the method invocation. JUnit checks whether the result equals 1. If it does, the test succeeds, and if not, the test fails and the message `"Should be 1"` is reported via the application console, the log files, or another user interface (based on how the test is executed). The message argument for assertion methods, `"Should be 1"` in this case, is optional.

JUnit provides many other useful assertion methods. The following table shows some significant `TestCase` assertion methods:

Method	**Description**
`assertEquals([String message],expected, actual)`	Verifies that the `actual` value is the expected one. For primitive types, such as `boolean`, `int`, `short`, and so on, this method performs a simple value comparison. For objects, this method uses the `equals()` method to perform comparison.

`assertNull([String message], Object o)`
`assertNotNull([String message], Object o)`
	Checks whether the object `o` is `null` or is not `null`.

`assertSame([String message], expected, actual)`
`assertNotSame([String message], expected, actual)`
	Checks whether the `expected` and `actual` objects have the same reference, or do not have the same reference.

`assertTrue([String message], boolean condition)`
`assertFalse([String message], boolean condition)`
	Checks whether the `condition` variable is `true` or `false`.

`fail([String message])`	Fails the test intentionally when an undesired condition happens. For instance, this method can be used in a section of code that must never be reached (for example, in a `catch` block if you do not expect an exception be thrown, or immediately before a `catch` block if you do expect an exception to be thrown).

The message argument shown as [String message] in all of the methods listed in the table above is optional. This argument represents the message that is reported if the test fails.

With JUnit 4, test methods are simple public methods marked with the @Test annotation. Here is an example:

```
@Test
public void saveStudent() {
    //call the method being tested
    //verify the result
}
```

To make assertion, JUnit 4 provides Assert class with public static assert methods to evaluate the test results. The assert methods provided by the Assert class are similar to the TestCase's assert methods with the same meaning and syntax.

Perform finalization operations

The TestCase class provides the tearDown() method, similar to the setUp() method, and is called by JUnit after each test method is executed. The tearDown() method's signature is as follows:

```
protected void tearDown();
```

With JUnit 3.8, you can override this method in test classes to perform finalization operations, such as object clean up or releasing a resource such as a file. With JUnit 4 and later, you can simply mark a method with the @org.junit.After annotation to act as tearDown() method.

Running JUnit tests

Almost all **Integrated Development Environments (IDEs)** are integrated with JUnit, and you may easily use the IDE to run test cases. However, we never want to make our development dependent on a particular product. Instead of using a particular IDE, you can rely on JUnit itself for running the tests. JUnit provides the org.junit.runner.JUnitCore class for running test cases. You can execute test cases through the java command with the JUnitCore class as follows:

```
java org.junit.runner.JUnitCore TestClass1.class [TestClass2.class ...]
```

You can also use the `JUnitCore` class in application code, as follows:

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class [,TestClass2.
class, ...]);
```

In practice, Ant and Maven are used to automatically run unit tests as a part of the build process.

Integration testing data-access layer

When developing with Hibernate, we need to test whether the application's data-access layer works properly. This involves testing the following:

- Whether classes are persistent
- Whether all classes' fields are persistent
- Whether HQL expressions return valid results
- Whether persistent operations are cascaded as they are configured in the mapping definitions

Let's take the `HibernateStudentDao` class as a representation of the data-access layer and see how test cases are implemented for this class. The following code shows the `StudentDao` interface implemented by `HibernateStudentDao`:

```
package com.packtpub.springhibernate.ch15;

import java.util.Collection;

public interface StudentDao {
    public Collection getAllStudents();
    public Student getStudent(Long studentId);
    public Student saveStudent(Student student);
    public Student removeStudent(Student student);
    public Student updateStudent(Student student);
}
```

We create the `TestCaseHibernateStudentDao` class as a test class for `HibernateStudentDao`, and implement appropriate test methods. All we need to do in the test methods is to verify that the DAO methods work properly when they are invoked. For example, the `saveStudent()` method stores a newly instantiated `Student` instance with all of its persistent properties. To do so, we can use a `HibernateStudentDao` instance and a Hibernate `Session` object concurrently to verify, for example, `getAllStudents()` returns all persistent students, and `getStudent()` returns the persisted `Student` instance correctly with all of its persistent fields. If we do this, we must provide a `Session` instance in the `setUp()` method, and close the `Session` instance in the `tearDown()` method, as the start and end of each test method's process. The following code shows the basic structure of the test class:

```
package com.packtpub.springhibernate.ch15;

import junit.framework.TestCase;
import org.hibernate.Session;

public class TestCaseHibernateStudentDao extends TestCase {

    Session session;
    StudentDao dao;

    public void setUp() throws Exception {
       //a Hibernate Session and a HibernateStudentDao object are used
       //by all the test methods, so those object creation are
         performed here
        session = HibernateHelper.getSession();
        session.connection().setAutoCommit(true);
        dao = new HibernateStudentDao();
    }

    public void tearDown() throws Exception {
        //as the finalized step we need to close the Hibernate Session
        // instance to release the Session's resources
        if(session.isOpen())
            session.close();
    }
}
```

The `tearDown()` method checks whether the session is open before closing it. This is reasonable. If we do not use the session in the test method (for example, to start any transaction with the session), the session remains closed.

Next, I'll explain how a test method is implemented for each method.

Verify that the entity class is persistent

To test the `saveStudent()` method, we create a `testSaveStudent()` method in the test class as follows:

```
public void testSaveStudent() {
    Student student = createADummyStudent();
    Student retStudent = dao.saveStudent(student);

    //the value returned by the DAO must be equal to
    //the original value
    assertEquals("The DAO's returned value of saving"+
                 "must be equal to original",
                                   student, retStudent);

    //look inside the database through direct Hibernate
    //the session is auto-commit, so we are sure that the
    //transaction is committed, the session is flushed
    //after each operation
    Student actualStudent = (Student)session.get(Student.class,
                                         retStudent.getId());

    //the actual Student object in the database must be
    //equal to the original value
    assertEquals("Actual Student and DAO's returned "+
                 "Student must be equal",
                  student, actualStudent);
}

private Student createADummyStudent() {
    Student student = new Student();
    Calendar birthday = Calendar.getInstance();
    birthday.set(Calendar.YEAR, 1982);
    birthday.set(Calendar.MONTH, 3);
    birthday.set(Calendar.DAY_OF_MONTH, 21);
    student.setBirthdate(birthday.getTime());
    Calendar entranceTime = Calendar.getInstance();
    birthday.set(Calendar.YEAR, 2005);
    birthday.set(Calendar.MONTH, 7);
    birthday.set(Calendar.DAY_OF_MONTH, 1);
    student.setEntranceDate(entranceTime.getTime());
    student.setFirstName("John");
    student.setLastName("White");
    student.setSsn("121-21212-211");
    student.setStdNo("832423472");
    return student;
}
```

As you can see, a `Student` instance is created and is stored via the DAO object. The object returned by the DAO must be equal to the original object. The `Session` instance uses the returned value to load the `Student` object from the database. Loaded instances can be used to verify that the actual data in the database holds the correct values for the `Student` instance.

> In all test methods, we look at DAO classes as black boxes. It does not matter whether a DAO class uses Hibernate or any other O/R mapping technology. We test the DAO's functionality regardless of the technology it uses.

Verify that all entity fields are persistent

To verify all properties are persistent, we can simply compare the original object with the persistent object with the `assertEquals()` method. The `testSaveStudent1()` shows how it can be done:

```
public void testSaveStudent() {
    Student student = createADummyStudent();
    Student retStudent = dao.saveStudent(student);

    assertNotNull(retStudent);

    //set the identifier to let the equals() method proceed.
    student.setId(retStudent.getId());

    //the returned value by the DAO must be equal to
    // the original value
    assertEquals("The DAO's returned value of saving" +
                 " must be equal to original",
                                     student, retStudent);

    //look inside the database using Hibernate directly
    Student actualStudent =
            (Student)session.get(Student.class, retStudent.getId());

    //the actual Student object in the database must
    // be equal to the original value
    assertEquals("Actual Student and DAO's returned" +
                 " Student must be equal",
                               student, actualStudent);

}
```

The `assertEquals()` method uses the `Student`'s `equals()` method to compare two objects.

Although you should test every behavior of the class, the setter and getter methods commonly just assign and return the field values, so these are easily tested. However, if these methods have side effects, or nontrivial functionality, they should be tested.

If you are using a Hibernate `Session` in each test method to perform a distinct persistent operation, it is a good idea to put the `Session` obtaining code in the `setUp()` and close the `Session` in the `tearDown()` method. However, if the test methods take the `Session` obtaining or the `Session` closing as a part of the test logic, you cannot put obtaining and closing the `Session` instance generally in the `setUp()` and `tearDown()` methods.

Verify that HQL works properly

While testing the data-access layer, you need to verify that HQL expressions load persistent objects correctly. For instance, in our example you must verify that the `searchStudentsByFirstName()` method takes a `String` parameter as the student's first name, and retrieves all `Student` instances with the specified first name. The `testSearchStudentsByFirstName()` method verifies this:

```
public void testSearchStudentsByFirstName(){
    Student student = createADummyStudent();
    dao.saveStudent(student);

    Collection students =
           dao.searchStudentsByFirstName(student.getFirstName());
    assertNotNull(allStudents);
    String hql = "from Student s where s.firstName like :firstName";
    Query query = session.createQuery(hql);
    query.setString("firstName", student.getFirstName());
    Collection actualStudents =
                    query.list();
    assertEquals("All students should be the same",
                                  allStudents, actualStudents);

}
```

Verify cascading operation

Verifying the cascading operation is one of the important aspects. We want to make sure when an object is stored if its association are persistent properly. With what we have done so far, doing it is very easy and straightforward. We are just going to test the `save-update` cascading operation. You will then be able to test all other cascading operations.

Suppose the `Student` object is associated with an `Address` object, and their relation is mapped with the `save-update` cascading operation. When a Student object with its Address associated is stored, we expect both objects persisted properly:

```
public void testSaveStudent() {
    Student student = createADummyStudent();
    Address address = createADummyAddress();
    student.setAddress(address);

    Student retStudent = dao.saveStudent(student);

    //look inside the database through direct Hibernate
    //the session is auto-commit, so we are sure that the
    //transaction is committed, the session is flushed
    //after each operation
    Student actualStudent = (Student)session.get(Student.class,
                                       retStudent.getId());
    Address actualAddress = actualStudent.getAddress();
    assertNotNull(actualAddress);

    //set the identifier, to let the equals() method proceed.
    address.setId(actualAddress.getId());
    student.setId(actualStudent.getId());

    //the actual Student object in the database must be
    //equal to the original value
    assertEquals("Actual Student and DAO's returned "+
                 "Student must be equal",
                  student, actualStudent);
    assertEquals("Actual Address and DAO's returned "+
                 "Address must be equal",
                  address, actualAddress);

}
```

It is assumed that the `Student`'s `equals()` method does not care about the equality of the associated `Address` object. We have used an extra `assertEquals()` to verify, if the persistent associated `Address` object is equal to the original `Address` object.

Testing Inversion of Control

One of Spring's benefits is that most of the objects managed by Spring are unaware of the Spring framework API. This makes it easy to use Java code to create and wire together an instance of the class being tested with its required dependencies.

Look at the Spring context with two configured objects, as shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-
             2.5.xsd">
    <bean id="studentService"
        class="com.packtpub.springhibernate.ch15.StudentService">
        <property name="studentDao">
            <ref local="studentDao"/>
        </property>
    </bean>

    <bean id="studentDao"
        class="com.packtpub.springhibernate.ch15.HibernateStudentDao">
    </bean>
</beans>
```

The test class to initialize the Spring context and obtain a `StudentService` instance is shown in the following code:

```
package com.packtpub.springhibernate.ch15;

import org.springframework.context.support.
ClassPathXmlApplicationContext;
import org.springframework.context.ApplicationContext;

import junit.framework.TestCase;

public class TestCaseStudentService extends TestCase {

    ApplicationContext ctx;

    public void setUp() throws Exception {
        ctx = new ClassPathXmlApplicationContext(
"com/packtpub/springhibernate/ch15/applicationContext.xml");
    }

    public void testStudentDaoInjection() throws Exception {
        StudentService ss =
                (StudentService) ctx.getBean("studentService");
        assertNotNull("StudentService should not be null", ss);
        assertNotNull("StudentDao should not be null",
                                   ss.getStudentDao());
        assertEquals(ss.getStudentDao().getClass(),
                                 HibernateStudentDao.class);
    }
}
```

Note that, we still need to implement typical test methods to verify the functionality of `StudentService` methods.

Unit testing using mocks

With unit testing, as a rule, it is always necessary to test each object without relying on other application's objects which need testing as well. However, in many situations, the object being tested is tightly coupled with other objects. For example, the object being tested uses other objects to do its job. In such cases, *mock objects* help us to test the object conveniently without relying on other application's objects. Mock objects are also used in situations where testing an object needs to create other objects, such as connecting to a database, starting an application server, providing a socket object, and so on, that are costly. Mock objects work similarly to real objects, but we are sure about their operations and do not need to test them. This means that we can be confident in using them, instead of real objects, to test a dependent object.

For instance, how can we ensure that `StudentService` always uses the `StudentDao` instance, when there is no approach to look inside the `StudentService` class and verify that the `StudentDao` instance is always used? In this case, we can create a mock implementation of the `StudentDao` interface, and manually inject it into the `StudentService` instance. Based on the objects returned by the mock `StudentDao` implementation, we can determine whether `StudentDao` is used. The following code shows this approach:

```
package com.packtpub.springhibernate.ch15;

import java.util.Calendar;
import java.util.Collection;
import  java.util.ArrayList;

import org.hibernate.HibernateException;

import org.springframework.context.support.
ClassPathXmlApplicationContext;
import org.springframework.context.ApplicationContext;

import junit.framework.TestCase;

public class TestCaseStudentService1 extends TestCase {

    ApplicationContext ctx;
    StudentService ss;

    public void setUp() throws Exception {
        ctx = new ClassPathXmlApplicationContext(
       "com/packtpub/springhibernate/ch15/applicationContext.xml");

        ss = (StudentService) ctx.getBean("studentService");
    }
```

```
public void testStudentDaoUsageWithMock() throws Exception {
    ss.setStudentDao(new mockHibernateStudentDao());
    //verify if StudentDao instance is called in
    //the getAllStudents() method
    Collection allStudents = ss.getAllStudents();
    assertNotNull(allStudents);
    assertEquals(allStudents.size(), 1);
    assertEquals(allStudents.toArray()[0],
                         createADummyStudent());
    //verify if StudentDao instance is called in
    // the saveStudent() method
    Student student = createADummyStudent();
    Student retStudent = ss.saveStudent(student);
    assertSame(student, retStudent);
    //verify if StudentDao instance is called in
    // the removeStudent() method
    retStudent = ss.removeStudent(student);
    assertSame(student, retStudent);
    //verify if StudentDao instance is called in
    // the updateStudent() method
    retStudent = ss.updateStudent(student);
    assertSame(student, retStudent);
}
private Student createADummyStudent() {
    Student student = new Student();
    Calendar birthday = Calendar.getInstance();
    birthday.set(Calendar.YEAR, 1982);
    birthday.set(Calendar.MONTH, 3);
    birthday.set(Calendar.DAY_OF_MONTH, 21);
    student.setBirthdate(birthday.getTime());
    Calendar entranceTime = Calendar.getInstance();
    birthday.set(Calendar.YEAR, 2005);
    birthday.set(Calendar.MONTH, 7);
    birthday.set(Calendar.DAY_OF_MONTH, 1);
    student.setEntranceDate(entranceTime.getTime());
    student.setFirstName("John");
    student.setLastName("White");
    student.setSsn("121-21212-211");
    student.setStdNo("832423472");
    return student;
}
class mockHibernateStudentDao implements StudentDao {
```

```
            public Collection getAllStudents() throws
                                              HibernateException {
                Student std = createADummyStudent();
                ArrayList list = new ArrayList();
                list.add(std);
                return list;
            }
            public Student getStudent(Long stdId) throws
                                              HibernateException {
                Student std = createADummyStudent();
                std.setId(stdId);
                return std;
            }
            public Student saveStudent(Student std) throws
                                              HibernateException {
                return std;
            }
            public Student removeStudent(Student std) throws
                                              HibernateException {
                return std;
            }
            public Student updateStudent(Student std) throws
                                              HibernateException {
                return std;
            }
        }
    }
```

In this class, we simply created a mock implementation of the `StudentDao` interface. Although, you can always use this approach to create mock objects, it is not always as simple and straightforward as in the example. For instance, to create a mock implementation of interfaces with many methods, you need to implement all of the methods, even if you do not need them all. For example, when you are developing a web application and dealing with the `HttpServletRequest` and `HttpServletResponse` interfaces, it is not simple to create a concrete implementation of these interfaces as mock classes. Fortunately, Spring includes some mock classes for common interfaces, such as `MockHttpSession`, `MockHttpServletRequest`, and `MockHttpServletResponse` as mock implementations of the `HttpSession`, `HttpServletRequest`, `HttpServletResponse` interfaces, respectively.

Additionally, you can use easy mock objects. EasyMock (`http://www.easymock.org`) is a library for creating mock objects dynamically without implementing mock classes. You specify the method, which the mock object should provide, and the value that method should return when called. Then, you can use the object, which acts similarly to the real object. These objects work in two modes: record mode and replay mode. Record mode is the state in which the object records a method and the return value it should provide. Record mode can switch to replay mode, in which the object is ready to be used.

To use a mock object, follow these steps:

1. Create a mock object for the interface you want to simulate.
2. Record the expected behavior.
3. Switch the mock object to the replay state.

The following code shows the `StudentServiceTest2` class, which uses mock objects to test the `StudentService` class:

```
package com.packtpub.springhibernate.ch15;

import junit.framework.TestCase;
import org.easymock.EasyMock;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;

public class TestCaseStudentService2 extends TestCase {
    StudentService ss;
    StudentDao dao;

    public void setUp() throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
       "com/packtpub/springhibernate/ch15/applicationContext.xml");
        ss = (StudentService) ctx.getBean("studentService");

        //create a mock object which represents
        // the StudentDao instance
        dao = EasyMock.createMock(StudentDao.class);
    }

    public void testStudentDaoUsageEasyMock() throws Exception {
        //create a dummy student
        Student student = createADummyStudent();
        Collection allStudents = new ArrayList();
        allStudents.add(student);

        //the DAO instance is expected to return allStudents
        //when its getAllStudents() method is called

       EasyMock.expect(dao.getAllStudents()).andReturn(allStudents);

        //switch the mock object to the replay state
        EasyMock.replay(dao);
        ss.setStudentDao(dao);

        //verify our expectations when the StudentService is called
        Collection retStudents = ss.getAllStudents();

        assertNotNull("The returned collection of Students" +
                      " are not null", retStudents);

        assertEquals("The returned collection has the " +
                       "expected size", retStudents.size(), 1);

        assertSame("The returned collection has the expected value",
                            retStudents.toArray()[0], student);

        //verify if the dao instance is called
        EasyMock.verify(dao);
    }

    private Student createADummyStudent() {
        Student student = new Student();
        Calendar birthday = Calendar.getInstance();
        birthday.set(Calendar.YEAR, 1982);
        birthday.set(Calendar.MONTH, 3);
        birthday.set(Calendar.DAY_OF_MONTH, 21);
        student.setBirthdate(birthday.getTime());
        Calendar entranceTime = Calendar.getInstance();
        birthday.set(Calendar.YEAR, 2005);
        birthday.set(Calendar.MONTH, 7);
        birthday.set(Calendar.DAY_OF_MONTH, 1);
        student.setEntranceDate(entranceTime.getTime());
        student.setFirstName("John");
        student.setLastName("White");
        student.setSsn("121-21212-211");
        student.setStdNo("832423472");
        return student;
    }
}
```

In this example, we call the `createMock()` static method of the `org.easymock.EasyMock` class to create a `StudentDao` mock object. We then register a method call on the mock object, specify that the mock should return the test data, and feed the mock object to the `StudentService` instance provided by the Spring IoC container. Later, we can call `EasyMock.verify()` to ask the mock object to verify that the method we registered was actually called.

Automating tests with Ant

Ant provides several tasks to run test classes from the build file and generate test reports. To use these tasks, the `junit.jar` file must be in the Ant classpath (in the `ANT_HOME/lib` directory) or in the application classpath (in the application's `lib` directory).

The `junit` task is an Ant task that allows us to execute the test classes automatically with the application's build process. Here is a typical usage of the `junit` task inside the `build.xml` file, introduced in Chapter 2:

```
<target name="all-tests" depends="compile"
                         description="Run All Test Cases">
    <javac destdir="${target.test-classes}" debug="true"
                                 classpathref="class.path">
        <src path="${test.java}"/>
        <classpath>
            <pathelement location="${target.classes}"/>
        </classpath>
    </javac>

    <copy todir="${target.test-classes}">
        <fileset dir="${test.etc}"/>
    </copy>

    <copy todir="${target.test-classes}">
        <fileset dir="${test.webapp}/WEB-INF"/>
    </copy>

    <junit printsummary="on">
        <classpath>
            <fileset dir="lib" includes="${lib}">
                <include name="**/*.jar"/>
            </fileset>
            <pathelement location="${target.classes}"/>
            <pathelement location="${target.test-classes}"/>
        </classpath>
        <formatter type="xml"/>
        <formatter type="plain"/>
```

```
            <batchtest todir="${target.test-reports}">
                <fileset dir="${test.java}">
                    <include name="**/TestCase*.java"/>
                </fileset>
            </batchtest>
        </junit>
        <junitreport todir="${target.test-reports}">
            <fileset dir="${target.test-reports}">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="frames" todir="${target.test-reports}"/>
        </junitreport>
    </target>
```

The `junit` and `junitreport` tasks run test cases and create a document to show the test case's result, respectively. Here are short explanations for these tasks:

- The `<junit>` element can be used with the optional `printsummary` attribute, which determines whether to print one line statistics for each test case. The nested `<formatter>` elements within the `<junit>` element lets you choose different formats for the test result. The valid values, which are specified through the `type` attribute of the `<formatter>` element, are `plain`, `xml`, and `brief`. The `xml` prints the test results in XML format. `plain` and `brief` emit plain text, but `brief` provides a shorter description for tests than `plain` does. The nested `<batchtest>` element lets you select test cases based on pattern matching.
- `junitreport` lets you create a document to show test results. It merges the individual XML files generated by the `junit` task, and applies a style sheet to the result to produce browsable test-case results. The `todir` attribute determines the location of the XML file aggregated from the results of all tests. The nested `<fileset>` element lets you select individual XML files generated by the `junit` task. The nested `<report>` element generates a browsable report based on the document created by the merge.

For a more in-depth discussion about the `junit` and `junitreport` tasks, see the JUnit documentation, or *Pro Apache Ant by Matthew Moodie* (Apress, 2005).

Summary

Unit testing, a critical part of application development, ensures that the application works as expected. A good practice for unit testing is to execute test classes automatically.

A test class is implemented for each Java class. Each test class includes the required methods for testing an application class in isolation from other application and test classes.

JUnit is a Java framework that simplifies test development. Using JUnit, you can easily verify that the method invocation results match the expected values. Extensions exist for the JUnit framework, such as Cactus, JMeter, DbUnit, HttpUnit, and so on.

To implement a test class with JUnit 3.8 and before, you only need to create a subclass of `TestCase`, and then implement the initialization process in the `setUp()` method and the finalization process in the `tearDown()` method. Then, implement the test methods, whose names begin with the `test` prefix. With JUnit 4 or later, you can simply use any class as a test class. In the test class, any public method which doesn't take any parameter and returns `void` can be marked with `@Test` to be a test method. It is also possible to mark methods with the `@Before` and `@After` annotations to act as `setUp()` and `tearDown()` methods.

JUnit automatically explores the test classes to find and run test methods.

Some of Hibernate's Advanced Features

Throughout this book, you have learned essential information about Hibernate. Hibernate is flexible enough to be used in any environment, providing advanced features that are the subject of this appendix.

This appendix discusses the event/listener model that Hibernate implements. Using this model, you can customize normal persistence behavior exposed by Hibernate. We also look at filters, which let you work with a subset of persistent data. Finally, we'll explore some useful Hibernate properties that you may need in advanced cases.

Hibernate's Event/Listener model

At times, you'll need to perform a task when a persistence operation is done. Normally, you may change the persistence operation and insert the task's code before or after the operation, based on your requirements. Although this solves the problem, it may not suit your needs: the task may not be part of a persistence operation, or it may be temporary. You want a method to add and remove the task easily without changing the other code.

In Chapter 11, you learned how to create and apply an advice. Advice is useful when you want to perform a different concern from the main concern. In addition to implementing and using advice, Hibernate 3 provides an event/listener model which is useful in such cases. Using this feature, you can create a listener class that is notified when a particular persistence event occurs. You should then register the listener programmatically with the `Configuration` instance that you use to build `SessionFactory`. It is also possible to configure the listener declaratively in the `hibernate.cfg.xml` file.

Like the event/listener model common in the Java world, any *listener* is an instance of a class that implements a particular interface. An *event* is also an instance of a particular event class. The following table shows the listener interfaces, the events that notify them, and the associated persistence operations that fire the events:

Listener	**Type of Operation**	**Description**
`PreInsertEventListener`	pre-insert	May be called by the `Session.save()` and `Session.saveOrUpdate()` methods *before* an item is inserted into the database
`PostInsertEventListener`	post-insert	May be called by the `Session.save()` and `Session.saveOrUpdate()` methods *after* an item is inserted into the database
`PreUpdateEventListener`	pre-update	May be called by the `Session.update()` and `Session.saveOrUpdate()` methods *before* an item is inserted into the database
`SaveOrUpdateEventListener`	save–update	May be called by `Session.save()`, `Session.update()`, or `Session.saveOrUpdate()` *when* an item is saved or updated
`PostUpdateEventListener`	post-update	May be called by the `Session.update()` and `Session.saveOrUpdate()` methods *after* an item is inserted into the database
`PreDeleteEventListener`	pre-delete	Called by the `Session.delete()` method *before* an item is removed from the database
`DeleteEventListener`	delete	Called by the `Session.delete()` method *when* an item is removed from the database
`PostDeleteEventListener`	post-delete	Called by the `Session.delete()` method *after* an item is removed from the database
`PreLoadEventListener`	pre-delete	May be called by the `Session.load()` or `Session.get()` method *before* property values are injected into a newly loaded entity instance

Listener	Type of Operation	Description
`LoadEventListener`	load	May be called by the `Session.load()` or `Session.get()` method *when* an instance is loaded into the application, either from memory or from the database
`PostLoadEventListener`	post-load	May be called by the `Session.load()` or `Session.get()` method *after* an instance is fully loaded into memory
`MergeEventListener`	merge	Called by the `Session.merge()` method
`LockEventListener`	lock	Called by the `Session.lock()` method
`AutoFlushEventListener`	auto-flush	Called when the session is automatically flushed
`DirtyCheckEventListener`	dirty check	Called when the session is checked for dirty objects, particularly before flushing the session
`EvictEventListener`	evict	Called by the `Session.evict()` method
`FlushEventListener`	flush	Called by the `Session.flush()` method
`InitializeCollectionEventListener`	load collection	Called when a collection is initialized before it is loaded from the database
`RefreshEventListener`	refresh	Called by the `Session.refresh()` method
`ReplicateEventListener`	replicate	Called by the `Session.replicate()` method
`PersistEventListener`	persist	Called by the `Session.persist()` method

The `org.hibernate.event` package defines all listener interfaces and event classes.

To use any of the listeners listed in the table above, you must implement the appropriate listener interface and override the required methods. You should then apply the listener either declaratively, through the Hibernate configuration file, or programmatically, via the `Configuration` object. The following code shows an example:

```
package com.packtpub.springhibernate.appendix;

import org.hibernate.HibernateException;
import org.hibernate.event.SaveOrUpdateEvent;
import org.hibernate.event.SaveOrUpdateEventListener;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class SaveNewCourseListener implements
SaveOrUpdateEventListener {

  static Logger logger = Logger.getLogger(SaveNewCourseListener.
class);

  public void onSaveOrUpdate(SaveOrUpdateEvent event) throws
HibernateException {
    if (event.getObject() instanceof Course) {
      Course c = (Course) event.getObject();
      logSaveOrUpdate(c);
    }
  }
  private void logSaveOrUpdate (Course c) {
    // prepare the message
    String message = c + " is added.";
    logger.log(Level.INFO, message);
  }
}
```

In the previous table, you can see the `SaveOrUpdateEventListener` listener is called when `Session.save()`, `Session.update()`, or `Session.saveOrUpdate()` is called. This means `onSaveOrUpdate()` is called by these `Session` methods.

After all, you may apply the implemented listener declaratively in the Hibernate configuration file as follows:

```
<hibernate-configuration>
    <session-factory>
        <listener type="save-update"              class="com.packtpub.
springhibernate.appendix.SaveNewCourseListener"/>
        …
    </session-factory>
</hibernate-configuration>
```

The `<session-factory>` element can nest with an arbitrary number of `<listener>` elements. Each `<listener>` element identifies a particular listener with the `type` and `class` attributes, which specify, respectively, the type of the listener and the class of the listener.

Alternatively, you can apply the implemented interface programmatically to the `Configuration` object as follows:

```
Configuration config = new Configuration();
config.setListener("save-update", new SaveNewCourseListener());
```

Interceptor

Hibernate provides full control of an object's states in its life cycle, allowing you to define custom behaviors when the object's state changes. Hibernate can do this thanks to interceptors and listeners. An *interceptor* is an old mechanism used by Hibernate 2.x. Hibernate 3 introduced the event/listener model instead, but has not removed support for interceptors. Although any interceptor lets you define custom behaviors for all object states, listeners are used at a particular state of an object. To define an interceptor, first implement the `Interceptor` interface and override the appropriate methods, and then apply it to the `Configuration` or `Session` object. Notice that the implemented interceptor cannot be declared declaratively through either the configuration file or the mapping files. Here are some methods of the `org.hibernate.Interceptor` interface:

- `afterTransactionBegin()`: Is called after the transaction is started via calling of `Session.beginTransaction()` This is called if Spring manages transactions.
- `beforeTransactionCompletion()`: Is called before a transaction is completed, but not before rollback. This method can be used to roll back the transaction.
- `afterTransactionCompletion()`: Is called after the transaction is committed or rolled back.
- `instantiate()`: Is always called when the `Session` needs to create a new instance of the persistent class. This method is useful when the persistent class does not have a default constructor and Hibernate does not know how to instantiate the persistent class. If the persistent class does not have a constructor, create and return an instance of the persistent class, otherwise return null.

- `onSave()`: Is called before a persistent instance is saved. You may use this method to change the persistent instance's state. If you want to change the state of the persistent instance before the instance is saved, return true, otherwise return false.
- `onDelete()`: Is called before a persistent instance is removed.
- `onLoad()`: Is called immediately before a persistent instance is initialized from the database.
- `getEntity()`: Is called by the `Session` object to look up the second-level cache when a persistent instance is not found in the `Session`'s own cache.
- `getEntityName()`: Is called by the `Session` to determine a persistent object's name.
- `isTransient()`: Is called by the `Session` to determine whether the instance being saved is a transient instance.
- `preFlush()`: Is called immediately before `Session.flush()` is called.
- `onFlushDirty()`: Is called when the session is flushed after the persistent instance is identified as dirty.
- `findDirty()`: Is called when `Session.flush()` is called. This method returns an array of `int`, which indicates that the indices of the dirty properties have changed and need to update, or it returns `null` when no property needs to update. This method may be used to change the persistent instance's state. To change the instance's state return true, otherwise return false.
- `postFlush()`: Is called after a flush.

Hibernate provides a default implementation class for the `Interceptor` interface, called `org.hibernate.EmptyInterceptor`. This default avoids implementing unnecessary methods when you do not actually need them.

Let's look at a simple example to see how to implement a Hibernate interceptor.

Suppose we want the application to log every student's result when the system administrator enters exam results (passed, failed, or conditional) into the system. The results are then stored in the database. Because printing the results and storing the objects are clearly distinct concerns, we may prefer to use an interceptor to implement the reporting rule. The following code shows an interceptor class in which the `onSave()` and `afterTransactionCompletion()` methods have been implemented. Notice that the `onSave()` method holds all of the entities saved in a `HashSet` object. After the transaction has been committed, all held objects are retrieved from the `HashSet` and an appropriate form printed for each:

```
package com.packtpub.springhibernate.appendix;
import org.hibernate.CallbackException;
import org.hibernate.Transaction;
import org.hibernate.type.Type;
import org.hibernate.EmptyInterceptor;
import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
public class CourseSavingInterceptor extends EmptyInterceptor {
  static Logger logger = Logger.getLogger(CourseSavingInterceptor.
class);
  public CourseSavingInterceptor() {
  }
  private ThreadLocal<Collection> stored =
          new ThreadLocal<Collection>();
  public void afterTransactionBegin(Transaction tx) {
    stored.set(new HashSet());
  }
  public void afterTransactionCompletion(Transaction tx) {
    if (tx.wasCommitted()) {
      Iterator i =
          ((Collection) stored.get()).iterator();
      while (i.hasNext()) {
        Course c = (Course) i.next();
        logCompletion (c);
      }
    }
    stored.set(null);
  }
  public boolean onSave(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types)
          throws CallbackException {
    stored.get().add(entity);
    return false;
  }
  private void logCompletion (Course c) {
    // prepare the message
    String message = c + " is added.";
    logger.log(Level.INFO, message);  }
}
```

After an interceptor has been implemented, it should either be applied to the `Configuration` or a `Session` object. Notice that when it applies to the `Configuration` object, it means that the interceptor should be applied to the persisting objects of all entity classes. In contrast, when the interceptor applies to a `Session` object, it only applies to the persisting objects that are persisted by that session.

In our example, we are only concerned with `Student` objects, so we apply the implemented interceptor only to the `Session` object that is specifically used to save `Student` objects:

```
Course course = …//new instantiated course
CourseSavingInterceptor interceptor = new CourseSavingInterceptor();
Configuration cfg = new Configuration();
cfg.configure();
SessionFactory sf = cfg.buildSessionFactory();
Session session = sf.openSession(interceptor);
Transaction tx = session.beginTransaction();
session.save(course); // Triggers onSave() of the Interceptor
tx.commit();
session.close();
```

Spring AOP can do the same work. Typically, Spring AOP is preferred because it allows you to do all of your work declaratively.

Alternatively, the interceptor may be applied to the `Configuration` object. This means that the interceptor will be applied to all of the `Session`:

```
new Configuration().setInterceptor(new CourseSavingInterceptor());
```

Filters

There are situations where you may want to work with a subset of objects, instead of the entire set of objects. In such situations, you may implement appropriate HQL expressions to load the desired objects. This is not a good practice, because you always need to call HQL expressions to load the objects. In our example of an education system application, suppose a requirement indicates that the application must be implemented so that we can configure it to view and use the information for only current students, only those students who have graduated, or only those students older than a specified age.

Hibernate allows us to filter the returned objects at the `Session` level. To use filters, follow these three steps:

1. **Define a filter**: This step includes filter definition in the mapping files through the `<filter-def>` elements.
2. **Apply the filter**: This step includes using the `<filter>` element inside of the mapping definitions to refer to the defined filter.
3. **Enable the filter and set up parameters**: You do this by invoking the `enableFilter()` method of `Session`, which returns an object of `org.hibernate.Filter` used for setting up the filter's parameters. You must specify the filter's runtime criteria restrictions.

The sections that follow elaborate on these steps.

Defining a filter

Hibernate filters are defined through `<filter-def>` elements inside the mapping files. This element assigns a name to the filter. After that time, the filter is referred to by that name. The `<filter-def>` element comes with an arbitrary number of `<filter-param>` elements, which specify the filter's parameter names. Each `<filter-param>` element comes with `name` and type `attributes`, which specify the name and the type of each parameter, respectively. The following snippet shows a filter definition:

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

<hibernate-mapping>
  <class ...
  </class>

  <filter-def name="statusFilter">
    <filter-param name="status" type="boolean"/>
  </filter-def>

</hibernate-mapping>
```

Applying the filter

In the second step, apply the defined filter to the target class or classes through the mapping definition. For this purpose, you need to insert the `<filter>` element inside the `<class>` element, and refer back to the defined filter using its name and parameters. The `<filter>` element comes with two attributes, `name` and `condition`, which specify, respectively, the filter being used and the restrictions the filter uses. The following snippet shows how to do this step:

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

<hibernate-mapping>
  <class …
    <filter name="statusFilter" condition=":status=studentStatus"/>
  </class>

  <filter-def name="statusFilter">
    <filter-param name="status" type="boolean"/>
  </filter-def>

</hibernate-mapping>
```

Enabling the filter and setting up parameters

Although you have called a filter in the mapping of a persistent class, the filter will not be applied unless it is enabled programmatically. To use the filter, you must enable it and pass the parameters' values. The `Session` interface provides the `enableFilter()` method, which takes the filter name as a `String` argument, and returns an object of type `org.hibernate.Filter` to be used as a reference to set the parameters:

```
org.hibernate.Filter enableFilter(String filterName)
```

The `org.hibernate.Filter` interface provides the following methods for setting up parameter values:

```
org.hibernate.Filter setParameter(String paramName,
                                  Object paramValue)
org.hibernate.Filter setParameterList(String paramName,
                                  Collection paramValues)
org.hibernate.Filter setParameterList(String paramName,
                                  Object[] paramValues)
```

The first `setParameter()` methods set `paramValue` for a given parameter, with the name `paramName`. Two other methods are used for setting parameter values using an `IN` clause.

The following snippet shows how the filter defined in the previous examples is applied to `Student` objects:

```
Filter f = session.enableFilter("statusFilter");
f.setParameter("status", Boolean.TRUE);
List results = session.createQuery("from Student").list();
```

More on Hibernate configuration

Although database properties, dialect, and mapping files are the most important parts of a Hibernate configuration file, Hibernate's configuration properties include far more than what you have seen so far. This section discusses some of these configuration properties.

JDBC properties

These properties relate directly to the `java.sql.Connection` and `java.sql.Statement` objects, which Hibernate uses behind the scenes to interact with the database:

- `hibernate.jdbc.fetch_size`: This property specifies the number of database rows that are buffered with every fetch. Using this property can improve performance when the application retrieves many rows. The optimized value can be achieved by balancing memory usage and network traffic. To use this property, the database driver must support this functionality. The default value depends on the `java.sql.Statement` implementation used by the database driver. However, you can set the desired value by calling `Statement.setFetchSize()`.
- `hibernate.jdbc.batch_size`: This property determines the maximum batch size for updates. When the session is flushed, modified objects are synchronized with the database through a JDBC batch. Configure this property with the maximum batch updates that may potentially occur: the recommended values are between 5 and 30.
- `hibernate.jdbc.batch_versioned_data`: If the used JDBC driver has been properly implemented, and returns the correct row count for `executeBatch()`, `true` is used for this option. Hibernate uses the return value of this method to check whether a batch update was successful. The default value is `false`.

- `hibernate.jdbc.use_scrollable_resultset`: This property determines whether Hibernate uses the JDBC2 scrollable resultsets for a user-provided JDBC. This property is only necessary when you are using user-supplied JDBC connections. Otherwise, Hibernate uses connection metadata.
- `hibernate.jdbc.use_streams_for_binary`: This property specifies whether the binary data or `Serializable` types are read or written over JDBC as streams.
- `hibernate.jdbc.use_get_generated_keys`: This allows Hibernate to use natively generated keys after SQL `INSERT` statements if the JDBC driver supports JDBC3 autogenerated key feature. Set it to `false` if your driver has problems with the Hibernate identifier generators. If this property is not specified, Hibernate tries to determine the driver capabilities by using connection metadata.
- `hibernate.connection.provider_class`: This indicates a class that implements the `org.hibernate.connection.ConnectionProvider` interface and provides `java.sql.Connection` objects for Hibernate. Use this option if you want to provide `Connection` objects in a specific way.
- `hibernate.connection.isolation`: This determines the transaction isolation level that JDBC connections use in Hibernate. The valid values have been defined as static members in `java.sql.Connection`. The default isolation level of the DBMS is usually read as committed or repeatable read. It can change either on the application side (through this property) or on the database side (through database configuration). Not all databases support all isolation levels. Look at Chapter 12 for more details.
- `hibernate.connection.autocommit`: This property turns autocommit behavior on or off for JDBC connections. Autocommit is disabled by default.
- `hibernate.connection.<propertyName>`: This configures the `propertyName` of the JDBC connection used by Hibernate.
- `hibernate.jndi.<propertyName>`: This passes the property `propertyName` to the JNDI `InitialContextFactory`s. Use this option when Hibernate uses a container-managed data source.

Hibernate properties

This section describes properties directly related to Hibernate. Use these configuration settings to optimize Hibernate's performance:

- `hibernate.show_sql`: This property enables or disables the printing of all SQL executed by Hibernate to the console. This option can be a problem in a high transactional system, so it is useful only in the development time, and not deployment. The right approach to monitor SQL is to enable SQL-logging of Hibernate, as you will see in the coming section.
- `hibernate.default_schema`: This specifies the default database schema for which Hibernate generates unqualified table names in the generated SQL.
- `hibernate.session_factory_name`: This binds the Hibernate `SessionFactory` to a JNDI node when `Configuration.buildSessionFactory()` is called. It then lets you access the `SessionFactory` object by looking up the bound `SessionFactory` object anywhere in the application.
- `hibernate.max_fetch_depth`: This property determines the maximum depth size to which Hibernate will go when fetching the outer-join fetch tree. By default, no limit is set. The recommended values are between 1 and 5. A value of 0 disables join fetching.
- `hibernate.default_batch_fetch_size`: This sets a default size for Hibernate batch fetching of associations. The recommended values are 4, 8, 16.
- `hibernate.order_updates`: This property forces Hibernate to order SQL updates by the primary key value of the items being updated, resulting in fewer transaction deadlocks in highly concurrent systems.

- `hibernate.generate_statistics`: Hibernate can be configured to collect statistics about the persistent operations that are performed. These statistics are useful for performance tuning. Typical information includes flush count, delete count, insert count, query execution count, second-level cache look up counts, and so on. `hibernate.generate_statistics` indicates whether Hibernate collects these statistics. The collected statistics, represented as an object of `org.hibernate.stat.Statistics`, can be obtained programmatically by the `SessionFactory`'s `getStatistics()` method. It is also possible to publish these statistics through a JMX MBean whether the Hibernate application is a standalone application or a JEE application. For information about how to use JMX with Hibernate statistics, look at `https://www.hibernate.org/216.html`. The following code shows an example of how to obtain these statistics:

```
Configuration cfg = new Configuration();
cfg.configure();
SessionFactory sf = cfg.buildSessionFactory();
Statistics stats =sf.getStatistics();
stats.setStatisticsEnabled(true);
stats.getSessionOpenCount();
stats.logSummary();//logs main statistics in the log file
long flushcount = stats.getFlushCount();
long deletecount = stats.getEntityDeleteCount();
long insertcount = stats.getEntityInsertCount();
long queryCount = stats.getQueryExecutionCount();
long lookupCacheCount = stats.getSecondLevelCacheHitCount();
```

- `hibernate.use_sql_comments`: This property causes Hibernate to put comments inside all SQL statements executed by Hibernate, which may help in debugging code. You can also set a custom comment for query expressions through the `setComment()` method for all of the `org.hibernate.Query`, `org.hibernate.SQLQuery`, `org.hibernate.Criteria` instances, as follows:

```
Query query =  session.createQuery("from Student");
query.setComment("HQL-Select All Students: ");

SQLQuery sqlQuery = session.createSQLQuery("SELECT * FROM
STUDENT");
sqlQuery.setComment("SQL-Select All Students: ");

Criteria criteria = session.createCriteria(Student.class);
criteria.setComment("Criteria-Select All Students: ");
```

  This option is useful for tracking Hibernate-generated SQL for optimization.

Cache properties

The properties described in this section relate to second-level cache configuration:

- `hibernate.cache.provider_class`: This specifies the second-level cache provider for Hibernate by determining the name of a class that implements the `org.hibernate.cache.CacheProvider` interface.
- `hibernate.cache.use_minimal_puts`: This property enhances second-level cache performance by forcing Hibernate to add an item to the cache, only after ensuring that the item isn't already cached. This option is helpful when you're using cluster cache providers and writing to the cache is more expensive than reading.
- `hibernate.cache.use_query_cache`: This property triggers Hibernate to cache each query result in queries for which caching is enabled programmatically through the `setCacheable(true)` method, as follows:

  ```
  Query query =  session.createQuery("from Student");
  query.setCacheable(true);

  SQLQuery sqlQuery = session.createSQLQuery("SELECT * FROM
  STUDENT");
  sqlQuery.setCacheable(true);

  Criteria criteria = session.createCriteria(Student.class);
  criteria.setCacheable(true);
  Note that no query result is cached by default.
  ```

- `hibernate.cache.use_second_level_cache`: This enables or disables second-level cache service. Using false for this property disables all other cache-related settings.
- `hibernate.cache.region_prefix`: Use this property when the application works with multiple databases, and therefore with multiple `SessionFactory`s. Because a persistent instance may belong to different databases, you must refer to each instance with a prefix. This property specifies a region-name prefix for a particular `SessionFactory`. For instance, with the prefix `db1`, `Student` objects are cached in an area of the cache named `db1.com.packtpub.springhibernate.appendix.Student`.

Transaction properties

This section describes configuration settings for transaction management:

- `hibernate.transaction.factory_class`: This setting determines if an implementation of `org.hibernate.transaction.TransactionFactory` is used by Hibernate to obtain `Transaction` objects. The default value is `org.hibernate.transaction.JDBCTransactionFactory`, used for the transaction API in Java SE and in direct JDBC.
- `jta.UserTransaction`: This determines a JNDI name that points to a `javax.transaction.UserTransaction` object, held by the application server, and looked up by `JTATransactionFactory`.
- `hibernate.transaction.manager_lookup_class`: This property specifies a class which implements the `org.hibernate.transaction.TransactionManagerLookup` interface. Hibernate uses this class to find the JTA implementation on which the application deploys.
- `hibernate.transaction.flush_before_completion`: This determines whether the session is automatically flushed before the transaction is complete. If set to `true`, the property makes flushing part of the transaction manager's internal synchronization procedure.
- `hibernate.transaction.auto_close_session`: This determines whether the session is automatically closed after the transaction is complete. If set to `true`, the property makes session closing part of the transaction manager's internal synchronization procedure.

Logging configuration in Hibernate

To provide logging, Hibernate uses SLF4J (`http://www.slf4j.org`) as the logging framework. SLF4J does not provide a logging alternative to Log4j, the Java logging API, or to any other. Instead, it is an API that sits on top of these other APIs to provide you with a common logging API to use in the source code. It forwards all logging events to either a logging framework, such as Log4j, the Java logging API, JCL, or so on. For each supported logging framework, SLF4J provides a JAR file, called *SLF4J Binding*, that must be in the application classpath when that framework is used as the underlying logging framework. For instance, SLF4J provides `slf4j-log4j12.jar` for Log4j that must be in the classpath when SLF4J is used with Log4j. Note that to use Log4j, you must put `log4j.properties`, the Log4j configuration file, in the root of the application classpath.

The following table shows Hibernate's categories for log messages:

Category	Function
`org.hibernate.SQL`	Log all SQL DML statements when they are executed
`org.hibernate.type`	Log all JDBC parameters
`org.hibernate.tool.hbm2ddl`	Log all SQL DDL statements as they are executed
`org.hibernate.cache`	Log all second-level cache activity
`org.hibernate.transaction`	Log transaction-related activity
`org.hibernate.jdbc`	Log all JDBC resource acquisitions
`org.hibernate.secure`	Log all JAAS authorization requests
`org.hibernate`	Log everything (a lot of information, but very useful for troubleshooting)

The practical approach to monitor the generated SQL is using the debug level for the category `org.hibernate.SQL`. Do not enable the `hibernate.show_sql` property in the production system.

Index