In this chapter, you will learn how to integrate *object/relational mapping (ORM)* frameworks into your Spring applications. Spring supports most of the popular ORM frameworks, including Hibernate, JDO, TopLink, iBATIS, and JPA. The focus of this chapter will be on Hibernate and the Java Persistence API (JPA). However, Spring’s support for these ORM frameworks is consistent, so you can easily apply the techniques in this chapter to other ORM frameworks as well.

ORM is a modern technology for persisting objects into a relational database. An ORM framework persists your objects according to the mapping metadata you provide, such as the mappings between classes and tables, properties and columns, and so on. It generates SQL statements for object persistence at runtime, so you needn’t write database-specific SQL statements unless you want to take advantage of database-specific features or provide optimized SQL statements of your own. As a result, your application will be database independent, and it can be easily migrated to another database in the future. Compared to the direct use of JDBC, an ORM framework can significantly reduce the data access effort of your applications.

*Hibernate* is a popular open source and high-performance ORM framework in the Java community. Hibernate supports most JDBC-compliant databases and can use specific dialects to access particular databases. Beyond the basic ORM features, Hibernate supports more advanced features like caching, cascading, and lazy loading. It also defines a querying language called *HQL (Hibernate Query Language)* for you to write simple but powerful object queries.

*JPA* defines a set of standard annotations and APIs for object persistence in both the Java SE and Java EE platforms. JPA is defined as part of the EJB 3.0 specification in JSR-220. JPA is just a set of standard APIs that require a JPA-compliant engine to provide persistence services. You can compare JPA to the JDBC API and a JPA engine to a JDBC driver. Hibernate can be configured as a JPA-compliant engine through an extension module called *Hibernate EntityManager*. This chapter will mainly demonstrate JPA with Hibernate as the underlying engine.

At the time of writing, the latest version of Hibernate is 3.2. Spring 2.0 supports both Hibernate 2.x and 3.x. The support for Hibernate 2.x is provided by the classes and interfaces in the *org.springframework.orm.hibernate* package, while the support for 3.x is in the *org.springframework.orm.hibernate3* package. You must be careful when importing the classes and interfaces to your application. Spring 2.5 supports only Hibernate 3.1 or higher. That means Hibernate 2.1 and Hibernate 3.0 won’t be supported any more.

Upon finishing this chapter, you will be able to take advantage of Hibernate and JPA for data access in your Spring applications. You will also have a thorough understanding of Spring’s data access module.
9-1. Problems with Using ORM Frameworks Directly

Suppose you are going to develop a course management system for a training center. The first class you create for this system is Course. This class is called an entity class or a persistent class, as it represents a real-world entity and its instances will be persisted to a database. Remember that for each entity class to be persisted by an ORM framework, a default constructor with no argument is required.

```java
package com.apress.springrecipes.course;
...
public class Course {
    private Long id;
    private String title;
    private Date beginDate;
    private Date endDate;
    private int fee;

    // Constructors, Getters and Setters
    ...
}
```

For each entity class, you must define an identifier property to uniquely identify an entity. It’s a best practice to define an auto-generated identifier, as this has no business meaning and thus won’t be changed under any circumstances. Moreover, this identifier will be used by the ORM framework to determine an entity’s state. If the identifier value is null, this entity will be treated as a new and unsaved entity. When this entity is persisted, an insert SQL statement will be issued; otherwise an update statement will. To allow the identifier to be null, you should choose a primitive wrapper type like java.lang.Integer and java.lang.Long for the identifier.

In your course management system, you need a DAO interface to encapsulate the data access logic. Let’s define the following operations in the CourseDao interface:

```java
package com.apress.springrecipes.course;
...
public interface CourseDao {
    public void store(Course course);
    public void delete(Long courseId);
    public Course findById(Long courseId);
    public List<Course> findAll();
}
```

Usually, when using ORM for persisting objects, the insert and update operations are combined into a single operation (e.g., store). This is to let the ORM framework (not you) decide whether an object should be inserted or updated.

In order for an ORM framework to persist your objects to a database, it must know the mapping metadata for the entity classes. You have to provide mapping metadata to it in its supported format. The native format for Hibernate is XML. However, as each ORM framework may have its own format for defining mapping metadata, JPA defines a set of persistent