SAMPLE CHAPTER

SPRING IN ACTION

Craig Walls
Ryan Breidenbach

MANNING

Spring in Action

by Craig Walls

and

Ryan Breidenbach

Sample Chapter 1

brief contents

1 A Spring jump start

This chapter covers

- Creating simpler J2EE applications using Spring
- Decoupling components with inversion of control
- Managing cross-cutting concerns with aspect-oriented programming
- Comparing the features of Spring and EJB

It all started with a bean.

In 1996 the Java programming language was still a young, exciting, up-and-coming platform. Many developers flocked to the language because they had seen how to create rich and dynamic web applications using applets. But they soon learned that there was more to this strange new language than animated juggling cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but they stayed for the components.

It was in December of that year that Sun Microsystems published the JavaBeans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components, they have been primarily used as a model for building user interface widgets. They seemed too simple to be capable of any "real" work. Enterprise developers wanted more.

Sophisticated applications often require services such as transaction support, security, and distributed computing—services not directly provided by the JavaBeans specification. Therefore, in March 1998, Sun published the 1.0 version of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing the much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. In fact, except in name, EJB bears very little resemblance to the original JavaBeans specification.

Despite the fact that many successful applications have been built based on EJB, EJB never really achieved its intended purpose: to simplify enterprise application development. Every version of the EJB specification contains the following statement: "Enterprise JavaBeans will make it easy to write applications." It is true that EJB's declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. But EJBs are complicated in a different way by mandating deployment descriptors and plumbing code (home and remote/local interfaces). Over time many developers became disenchanted with EJB. As a result, its popularity has started to wane in recent years, leaving many developers looking for an easier way.

Now Java component development is coming full circle. New programming techniques, including aspect-oriented programming (AOP) and inversion of control (IoC), are giving JavaBeans much of the power of EJB. These techniques furnish JavaBeans with a declarative programming model reminiscent of EJB, but without all of EJB's complexity. No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.

And that's where Spring steps into the picture.

1.1 Why Spring?

If you are reading this book, you probably want to know why Spring would be good for you. After all, the Java landscape is full of frameworks. What makes Spring any different? To put it simply, Spring makes developing enterprise applications easier. We don't expect that to convince you at face value, so first let's take a look at life without Spring.

1.1.1 A day in the life of a J2EE developer

Alex is a Java developer who has just started on his first enterprise application. Like many Java 2 Enterprise Edition (J2EE) applications, it is a web application that serves many users and accesses an enterprise database. In this case, it is a customer management application that will be used by other employees at his company.

Eager to get to work, Alex fires up his favorite integrated development environment (IDE) and starts to crank out his first component, the `CustomerManager` component. In the EJB world, to develop this component Alex actually has to write several classes—the home interface, the local interface, and the bean itself. In addition, he has to create a deployment descriptor for this bean.

Seeing that creating each of these files for *every* bean seems like a lot of effort, Alex incorporates XDoclet into his project. XDoclet is a code generation tool that can generate all of the necessary EJB files from a single source file. Although this adds another step to Alex's development cycle, his coding life is now much simpler.

With XDoclet now handling a lot of the grunt work for him, Alex turns his attention to his real problem—what exactly should the `CustomerManager` component do? He jumps in with its first method, `getPreferredCustomer()`. There are several business rules that define exactly what a preferred customer is, and Alex dutifully codes them into his `CustomerManager` bean.

Wanting to confirm that his logic is correct, Alex now wants to write some tests to validate his code. But then it occurs to him: the code he is testing will be running within the EJB container. Therefore, his tests need to execute within the container as well. To easily accomplish this, he concocts a servlet that will be responsible for executing these tests. Since all J2EE containers support servlets, this will allow him to execute his tests in the same container as his EJB. Problem solved!

So Alex fires up his J2EE container and runs his tests. His tests fail. Alex sees his coding error, fixes it, and runs the tests again. His tests fail again. He sees another error and fixes it. He fires up the container and runs the tests again. As Alex is going through this cycle, he notices something. The fact that he has to start the J2EE container for each batch of testing really slows down his development cycle. The development cycle should go code, test, code, test. This pattern has now been replaced with code, wait, test, code, wait, test, code, wait, get increasingly frustrated…

While waiting for the container to start during another test run, Alex thinks, "Why am I using EJB in the first place?" The answer, of course, is because of the services it provides. But Alex isn't using entity beans, so he is not using persistence services. Alex is also not using the remoting or security services. In fact, the only EJB service Alex is going to use is transaction management. This leads Alex to another question: "Is there an easier way?"

1.1.2 Spring's pledge

The above story was a dramatization based on the current state of J2EE—specifically EJB. In its current state, EJB is complicated. It isn't complicated just to be complicated. It is complicated because EJBs were created to solve complicated things, such as distributed objects and remote transactions.

Unfortunately, a good number of enterprise projects do not have this level of complexity but still take on EJB's burden of multiple Java files and deployment descriptors and heavyweight containers. With EJB, application complexity is high, regardless of the complexity of the problem being solved—even simple applications are unduly complex. With Spring, the complexity of your application is proportional to the complexity of the problem being solved.

However, Spring recognizes that EJB does offer developers valuable services. So Spring strives to deliver these same services while simplifying the programming model. In doing so, it adopts a simple philosophy: J2EE *should* be easy to use. In keeping with this philosophy, Spring was designed with the following beliefs:

- Good design is more important than the underlying technology.
- JavaBeans loosely coupled through interfaces is a good model.
- Code should be easy to test.

Okay. So how does Spring help you apply this philosophy to your applications?

Good design is more important than the underlying technology

As a developer, you should always be seeking the best design for your application, regardless of the implementation you choose. Sometimes the complexity of EJB is warranted because of the requirements of the application. Often, this is not the case. Many applications require few, if any, of the services provided by EJB yet are still implemented using this technology for technology's sake. If an application does not require distribution or declarative transaction support, it is unlikely that EJB is the best technology candidate. Yet many Java developers feel compelled to use EJB for every Java enterprise application.

The idea behind Spring is that you can keep your code as simple as it needs to be. If what you want are some plain-vanilla Java objects to perform some services supported by transparent transactions, you've got it. And you don't need an EJB container, and you don't have to implement special interfaces. You just have to write your code.

JavaBeans loosely coupled through interfaces is a good model

If you are relying on EJBs to provide your application services, your components do not just depend on the EJB business interface. They are also responsible for retrieving these EJB objects from a directory, which entails a Java Naming and Directory Interface (JNDI) lookup and communicating with the bean's `EJBHome` interface. This is not creating a decoupled application. This is tightly coupling your application to a specific implementation, namely EJB.

With Spring, your beans depend on collaborators through interfaces. Since there are no implementation-specific dependencies, Spring applications are very decoupled, testable, and easier to maintain. And because the Spring container is responsible for resolving the dependencies, the active service lookup that is involved in EJB is now out of the picture and the cost of programming to interfaces is minimized. All you need to do is create classes that communicate with each other through interfaces, and Spring takes care of the rest.

Code should be easy to test

Testing J2EE applications can be difficult. If you are testing EJBs within a container, you have to start up a container to execute even the most trivial of test cases. Since starting and stopping a container is expensive, developers may be tempted to skip testing all of their components. Avoiding tests because of the rigidness of a framework is not a good excuse.

Because you develop Spring applications with JavaBeans, testing is cheap. There is no J2EE container to be started since you will be testing a POJO. And since Spring makes coding to interfaces easy, your objects will be loosely coupled, making testing even easier. A thorough battery of tests should be present in all of your applications; Spring will help you accomplish this.

1.2 What is Spring?

Spring is an open-source framework, created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development*.[1] It was created to address the complexity of enterprise application development. Spring makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

NOTE To avoid ambiguity, we'll use the term "EJB" when referring to Enterprise JavaBeans. When referring to the original JavaBean, we'll call it "JavaBean," or "bean" for short. Some other terms we may throw around are "POJO" (which stands for "plain old Java object") or "POJI" (which means "plain old Java interface").

Put simply, Spring is a lightweight inversion of control and aspect-oriented container framework. Okay, that's not so simple a description. But it does summarize what Spring does. To make more sense of Spring, let's break this description down:

- *Lightweight*—Spring is lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a single JAR file that weighs in at just over 1 MB. And the processing overhead required by Spring is negligible. What's more, Spring is nonintrusive: objects in a Spring-enabled application typically have no dependencies on Spring-specific classes.
- *Inversion of control*—Spring promotes loose coupling through a technique known as inversion of control (IoC). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. You can think of IoC as JNDI in reverse—instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

[1] In this book, Spring was originally called "interface21."

- *Aspect-oriented*—Spring comes with rich support for aspect-oriented programming that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.
- *Container*—Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. You can configure how your each of your beans should be created—either create one single instance of your bean or produce a new instance every time one is needed based on a configurable prototype—and how they should be associated with each other. Spring should not, however, be confused with traditionally heavyweight EJB containers, which are often large and cumbersome to work with.
- *Framework*—Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration, etc.), leaving the development of application logic to you.

All of these attributes of Spring enable you to write code that is cleaner, more manageable, and easier to test. They also set the stage for a variety of subframeworks within the greater Spring framework.

1.2.1 Spring modules

The Spring framework is made up of seven well-defined modules (figure 1.1). When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But you do not have to base your application fully on the Spring framework. You are free to pick and choose the modules that suit your application and ignore the rest.

As you can see, all of Spring's modules are built on top of the core container. The container defines how beans are created, configured, and managed—more of the nuts-and-bolts of Spring. You will implicitly use these classes when you configure your application. But as a developer, you will most likely be interested in the other modules that leverage the services provided by the container. These modules will provide the frameworks with which you will build your application's services, such as AOP and persistence.

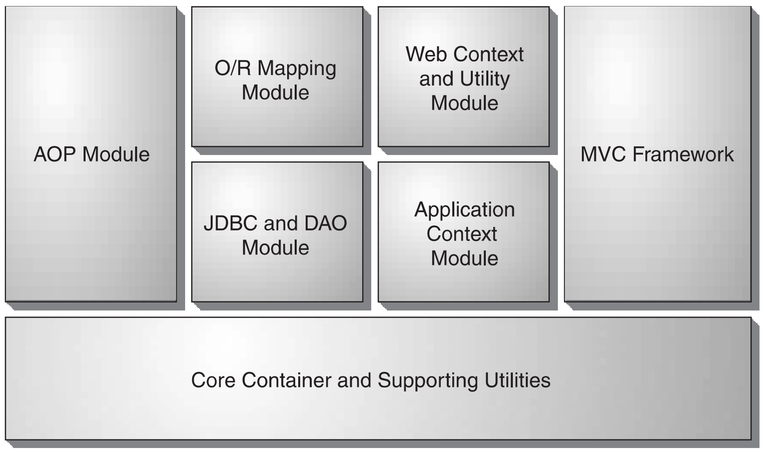
AOP Module
O/R Mapping Module
JDBC and DAO Module
Web Context and Utility Module
Application Context Module
MVC Framework
Core Container and Supporting Utilities

Figure 1.1 The Spring framework is composed of several well-defined modules.

The core container

Spring's core container provides the fundamental functionality of the Spring framework. In this module you'll find Spring's `BeanFactory`, the heart of any Spring-based application. A `BeanFactory` is an implementation of the factory pattern that applies IoC to separate your application's configuration and dependency specifications from the actual application code.

We'll be discussing the core module (the center of any Spring application) throughout this book, starting in chapter 2, when we cover bean wiring using IoC.

Application context module

The core module's `BeanFactory` makes Spring a container, but the context module is what makes it a framework. This module extends the concept of `BeanFactory`, adding support for internationalization (I18N) messages, application life cycle events, and validation.

In addition, this module supplies many enterprise services such as e-mail, JNDI access, EJB integration, remoting, and scheduling. Also included is support for integration with templating frameworks such as Velocity and FreeMarker.

Spring's AOP module

Spring provides rich support for aspect-oriented programming in its AOP module. This module serves as the basis for developing your own aspects for your Spring-enabled application.

To ensure interoperability between Spring and other AOP frameworks, much of Spring's AOP support is based on the API defined by the AOP Alliance. The

AOP Alliance is an open-source project whose goal is to promote adoption of AOP and interoperability among different AOP implementations by defining a common set of interfaces and components. You can find out more about the AOP Alliance by visiting their website at http://aopalliance.sourceforge.net.

The Spring AOP module also introduces metadata programming to Spring. Using Spring's metadata support, you are able to add annotations to your source code that instruct Spring on where and how to apply aspects.

JDBC abstraction and the DAO module

Working with JDBC often results in a lot of boilerplate code that gets a connection, creates a statement, processes a result set, and then closes the connection. Spring's JDBC and Data Access Objects (DAO) module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources. This module also builds a layer of meaningful exceptions on top of the error messages given by several database servers. No more trying to decipher cryptic and proprietary SQL error messages!

In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application.

Object/relational mapping integration module

For those who prefer using an object/relational mapping (ORM) tool over straight JDBC, Spring provides the ORM module. Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, JDO, and iBATIS SQL Maps. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

Spring's web module

The web context module builds on the application context module, providing a context that is appropriate for web-based applications. In addition, this module contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. It also cotains integration support with Jakarta Struts.

The Spring MVC framework

Spring comes with a full-featured Model/View/Controller (MVC) framework for building web applications. Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring's MVC framework uses IoC to provide for a clean separation of controller logic from business objects. It also allows you to declaratively bind request parameters to your business objects, What's more, Spring's MVC framework can take advantage of any of Spring's other services, such as I18N messaging and validation.

Now that you know what Spring is all about, let's jump right into writing Spring applications, starting with the simplest possible example that we could come up with.

1.3 Spring jump start

In the grand tradition of programming books, we'll start by showing you how Spring works with the proverbial "Hello World" example. Unlike the original Hello World program, however, our example will be modified a bit to demonstrate the basics of Spring.

NOTE To find out how to download Spring and plug it into your project's build routine, refer to appendix A.

Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application. What makes Spring-enabled applications different, however, is how these classes are configured and introduced to each other. Typically, a Spring application has an XML file that describes how to configure the classes, known as the Spring configuration file.

The first class that our Springified Hello World example needs is a service class whose purpose is to print the infamous greeting. Listing 1.1 shows GreetingService.java, an interface that defines the contract for our service class.

Listing 1.1 The `GreetingService` interface separates the service's implementation from its interface.

```
package com.springinaction.chapter01.hello;

public interface GreetingService {
  public void sayGreeting();
}
```

GreetingServiceImpl.java (listing 1.2) implements the `GreetingService` interface. Although it's not necessary to hide the implementation behind an interface, it's highly recommended as a way to separate the implementation from its contract.

Listing 1.2 GreetingServiceImpl.java: Responsible for printing the greeting

```
package com.springinaction.chapter01.hello;

public class GreetingServiceImpl implements GreetingService {
  private String greeting;

  public GreetingServiceImpl() {}

  public GreetingServiceImpl(String greeting) {
    this.greeting = greeting;
  }

  public void sayGreeting() {
    System.out.println(greeting);
  }

  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
}
```

The `GreetingServiceImpl` class has a single property: the `greeting` property. This property is simply a `String` that holds the text that is the message that will be printed when the `sayGreeting()` method is called. You may have noticed that the `greeting` can be set in two different ways: by the constructor or by the property's setter method.

What's not apparent just yet is who will make the call to either the constructor or the `setGreeting()` method to set the property. As it turns out, we're going to let the Spring container set the `greeting` property. The Spring configuration file (hello.xml) in listing 1.3 tells the container how to configure the greeting service.

Listing 1.3 Configuring Hello World in Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="greetingService"
      class="com.springinaction.chapter01.hello.GreetingServiceImpl">
    <property name="greeting">
      <value>Buenos Dias!</value>
    </property>
  </bean>
</beans>
```

The XML file in listing 1.3 declares an instance of a `GreetingServiceImpl` in the Spring container and configures its `greeting` property with a value of "Buenos Dias!" Let's dig into the details of this XML file a bit to understand how it works.

At the root of this simple XML file is the `<beans>` element, which is the root element of any Spring configuration file. The `<bean>` element is used to tell the Spring container about a class and how it should be configured. Here, the `id` attribute is used to name the bean `greetingService` and the `class` attribute specifies the bean's fully qualified class name.

Within the `<bean>` element, the `<property>` element is used to set a property, in this case the `greeting` property. By using `<property>`, we're telling the Spring container to call `setGreeting()` when setting the property.

The value of the greeting is defined within the `<value>` element. Here we've given the example a Spanish flair by choosing "Buenos Dias" instead of the traditional "Hello World."

The following snippet of code illustrates roughly what the container does when instantiating the greeting service based on the XML definition in listing 1.3:[2]

```
GreetingServiceImpl greetingService = new GreetingServiceImpl();
greetingService.setGreeting("Buenos Dias!");
```

Similarly, we may choose to have Spring set the `greeting` property through `GreetingServiceImpl`'s single argument constructor. For example:

```
<bean id="greetingService"
    class="com.springinaction.chapter01.hello.GreetingServiceImpl">
  <constructor-arg>
    <value>Buenos Dias!</value>
  </constructor-arg>
</bean>
```

The following code illustrates how the container will instantiate the greeting service when using the `<constructor-arg>` element:

```
GreetingServiceImpl greetingService =
    new GreetingServiceImpl("Buenos Dias");
```

The last piece of the puzzle is the class that loads the Spring container and uses it to retrieve the greeting service. Listing 1.4 shows this class.

[2] The container actually performs other activities involving the life cycle of the bean. But for illustrative purposes, these two lines are sufficient.

Listing 1.4 The Hello World main class

```
package com.springinaction.chapter01.hello;

import java.io.FileInputStream;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class HelloApp {
  public static void main(String[] args) throws Exception {
    BeanFactory factory =
        new XmlBeanFactory(new FileInputStream("hello.xml"));

    GreetingService greetingService =
        (GreetingService) factory.getBean("greetingService");

    greetingService.sayGreeting();
  }
}
```

The `BeanFactory` class used here is the Spring container. After loading the hello.xml file into the container, the `main()` method calls the `getBean()` method on the `BeanFactory` to retrieve a reference to the greeting service. With this reference in hand, it finally calls the `sayGreeting()` method. When we run the Hello application, it prints (not surprisingly)

```
Buenos Dias!
```

This is about as simple a Spring-enabled application as we can come up with. But it does illustrate the basics of configuring and using a class in Spring. Unfortunately, it is perhaps too simple because it only illustrates how to configure a bean by injecting a `String` value into a property. The real power of Spring lies in how beans can be injected into other beans using IoC.

1.4 Understanding inversion of control

Inversion of control is at the heart of the Spring framework. It may sound a bit intimidating, conjuring up notions of a complex programming technique or design pattern. But as it turns out, IoC is not nearly as complex as it sounds. In fact, by applying IoC in your projects, you'll find that your code will become significantly simpler, easier to understand, and easier to test.

But what does "inversion of control" mean?

1.4.1 Injecting dependencies

In an article written in early 2004, Martin Fowler asked what aspect of control is being inverted. He concluded that it is the acquisition of dependent objects that is being inverted. Based on that revelation, he coined a better name for inversion of control: dependency injection.[3]

Any nontrivial application (pretty much anything more complex than HelloWorld.java) is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). As you'll see, this can lead to highly coupled and hard-to-test code.

Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are *injected* into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

1.4.2 IoC in action

If you're like us, then you're probably anxious to see how this works in code. We aim to please, so without further delay…

Suppose that your company's crack marketing team culled together the results of their expert market analysis and research and determined that what your customers need is a knight. That is, they need a Java class that represents a knight. After probing them for requirements, you learn that what they specifically want is for you to implement a class that represents an Arthurian knight of the Round Table that embarks on brave and noble quests to find the Holy Grail.

This is an odd request, but you've become accustomed to the strange notions and whims of the marketing team. So, without hesitation, you fire up your favorite IDE and bang out the class in listing 1.5.

Listing 1.5 KnightOfTheRoundTable.java

```
package com.springinaction.chapter01.knight;

public class KnightOfTheRoundTable {
  private String name;
  private HolyGrailQuest quest;
```

[3] Although we agree that "dependency injection" is a more accurate name than "inversion of control," we're likely to use both terms interchangeably in this book.

```
  public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();   <-- A knight gets its own quest
  }

  public HolyGrail embarkOnQuest()
      throws GrailNotFoundException {
    return quest.embark();
  }
}
```

In listing 1.5 the knight is given a name as a parameter of its constructor. Its constructor sets the knight's quest by instantiating a `HolyGrailQuest`. The implementation of `HolyGrailQuest` is fairly trivial, as shown in listing 1.6.

Listing 1.6 HolyGrailQuest.java

```
package com.springinaction.chapter01.knight;
public class HolyGrailQuest {
  public HolyGrailQuest() {}

  public HolyGrail embark() throws GrailNotFoundException {
    HolyGrail grail = null;
    // Look for grail
    …
    return grail;
  }
}
```

Satisfied with your work, you proudly check the code into version control. You want to show it to the marketing team, but deep down something doesn't feel right. You almost dismiss it as the burrito you had for lunch when you realize the problem: you haven't written any unit tests.

Knightly testing

Unit testing is an important part of development. It not only ensures that each individual unit functions as expected, but it also serves to document each unit in the most accurate way possible. Seeking to rectify your failure to write unit tests, you put together the test case (listing 1.7) for your knight class.

Listing 1.7 Testing the KnightOfTheRoundTable

```
package com.springinaction.chapter01.knight;

import junit.framework.TestCase;

public class KnightOfTheRoundTableTest extends TestCase {

  public void testEmbarkOnQuest() {
    KnightOfTheRoundTable knight =
        new KnightOfTheRoundTable("Bedivere");

    try {
      HolyGrail grail = knight.embarkOnQuest();

      assertNotNull(grail);

      assertTrue(grail.isHoly());
    } catch (GrailNotFoundException e) {
      fail();
    }
  }
}
```

After writing this test case, you set out to write a test case for `HolyGrailQuest`. But before you even get started, you realize that the `KnightOfTheRoundTableTest` test case indirectly tests `HolyGrailQuest`. You also wonder if you are testing all contingencies. What would happen if `HolyGrailQuest`'s `embark()` method returned `null`? Or what if it were to throw a `GrailNotFoundException`?

Who's calling who?

The main problem so far with `KnightOfTheRoundTable` is with how it obtains a `HolyGrailQuest`. Whether it is instantiating a new `HolyGrail` instance or obtaining one via JNDI, each knight is responsible for getting its own quest (as shown in figure 1.2). Therefore, there is no way to test the knight class in isolation. As it

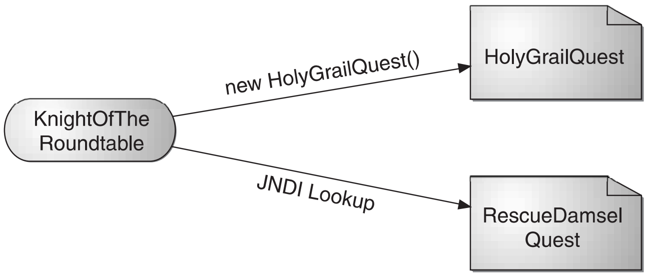

Figure 1.2 A knight is responsible for getting its own quest, through instantiation or some other means.

stands, every time you test KnightOfTheRoundTable, you will also indirectly test HolyGrailQuest.

What's more, you have no way of telling HolyGrailQuest to behave differently (e.g., return null or throw a GrailNotFoundException) for different tests. What would help is if you could create a mock implementation of HolyGrailQuest that lets you decide how it behaves. But even if you were to create a mock implementation, KnightOfTheRoundTable still retrieves its own HolyGrailQuest, meaning you would have to make a change to KnightOfTheRoundTable to retrieve the mock quest for testing purposes (and then change it back for production).

Decoupling with interfaces

The problem, in a word, is *coupling*. At this point, KnightOfTheRoundTable is statically coupled to HolyGrailQuest. They're handcuffed together in such a way that you can't have a KnightOfTheRoundTable without also having a HolyGrailQuest.

Coupling is a two-headed beast. On one hand, tightly coupled code is difficult to test, difficult to reuse, difficult to understand, and typically exhibits "whack-a-mole" bugs (i.e., fixing one bug results in the creation of one or more new bugs). On the other hand, completely uncoupled code doesn't do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary, but it should be managed very carefully.

A common technique used to reduce coupling is to hide implementation details behind interfaces so that the actual implementation class can be swapped out without impacting the client class. For example, suppose you were to create a Quest interface:

```
package com.springinaction.chapter01.knight;

public interface Quest {
  public abstract Object embark() throws QuestException;
}
```

Then, you change HolyGrailQuest to implement this interface. Also, notice that embark now returns an Object and throws a QuestException.

```
package com.springinaction.chapter01.knight;

public class HolyGrailQuest implements Quest {
  public HolyGrailQuest() {}

  public Object embark() throws QuestException {
    // Do whatever it means to embark on a quest
    return new HolyGrail();
  }
}
```

Also, the following method must also change in KnightOfTheRoundTable to be compatible with these Quest types:

```
private Quest quest;
…
public Object embarkOnQuest() throws QuestException {
  return quest.embark();
}
```

Likewise, you could also have `KnightOfTheRoundTable` implement the following `Knight` interface:

```
public interface Knight {
  public Object embarkOnQuest() throws QuestException;
}
```

Hiding your class's implementation behind interfaces is certainly a step in the right direction. But where many developers fall short is in how they retrieve a `Quest` instance. For example, consider this possible change to `KnightOfTheRoundTable`:

```
public class KnightOfTheRoundTable implements Knight {

  private Quest quest;
  …

  public KnightOfTheRoundTable(String name) {
    quest = new HolyGrailQuest();
  …
  }

  public Object embarkOnQuest() throws QuestException {
    return quest.embark();
  }
}
```

Here the `KnightOfTheRoundTable` class embarks on a quest through the `Quest` interface. But, the knight still retrieves a specific type of `Quest` (here a `HolyGrailQuest`). This isn't much better than before. A `KnightOfTheRoundTable` is stuck going only on quests for the Holy Grail and no other types of quest.

Giving and taking

The question you should be asking at this point is whether or not a knight should be responsible for obtaining a quest. Or, should a knight be given a quest to embark upon?

Consider the following change to `KnightOfTheRoundTable`:

```
public class KnightOfTheRoundTable implements Knight {
  private Quest quest;
  …

  public KnightOfTheRoundTable(String name) {
    …
  }

  public HolyGrail embarkOnQuest() throws QuestException {
    …
    return quest.embark();
  }

  public void setQuest(Quest quest) {
    this.quest = quest;
  }
}
```

Notice the difference? Compare figure 1.3 with figure 1.2 to see the difference in how a knight obtains its quest. Now the knight is *given* a quest instead of retrieving one itself. `KnightOfTheRoundTable` is no longer responsible for retrieving its own quests. And because it only knows about a quest through the `Quest` interface, you could give a knight any implementation of `Quest` you want. In a production system, maybe you would give it a `HolyGrailQuest`, but in a test case you would give it a mock implementation of `Quest`.

In a nutshell, that is what inversion of control is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves. And that's where lightweight container frameworks, such as Spring, come into play.

Assigning a quest to a knight

Now that you've written your `KnightOfTheRoundTable` class to be given any arbitrary `Quest` object, how can you specify which `Quest` it should be given?

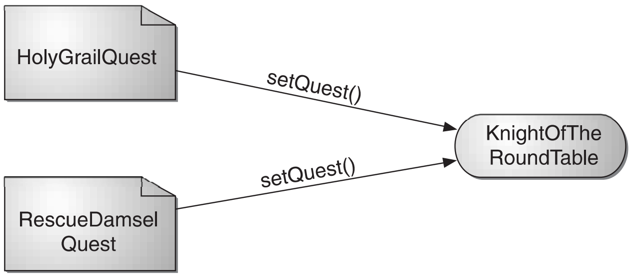

Figure 1.3
A knight is given a quest through its `setQuest()` method.

The act of creating associations between application components is referred to as *wiring*. In Spring, there are many ways to wire components together, but the most common approach is via XML. Listing 1.8 shows a simple Spring configuration file, knight.xml, that gives a quest (specifically, a `HolyGrailQuest`) to a `KnightOfTheRoundTable`.

Listing 1.8 Wiring a quest to a knight in knight.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>                                                    Define a quest
  <bean id="quest"
      class="com.springinaction.chapter01.knight.HolyGrailQuest"/>

  <bean id="knight"
      class="com.springinaction.chapter01.knight.KnightOfTheRoundTable">
                                                         Define a knight
    <constructor-arg>
      <value>Bedivere</value>    <-- Set the knight's name
    </constructor-arg>
    <property name="quest">
      <ref bean="quest"/>    <-- Give the knight a quest
    </property>

  </bean>
</beans>
```

This is just a simple approach to wiring beans. Don't worry too much about the details of it right now. In chapter 2 we'll explain more about what is going on here, as well as show you even more ways you can wire your beans in Spring.

Now that we've declared the relationship between a knight and a quest, we need to load up the XML file and kick off the application.

Seeing it work

In a Spring application, a `BeanFactory` loads the bean definitions and wires the beans together. Because the beans in the knight example are declared in an XML file, an `XmlBeanFactory` is the appropriate factory for this example. The `main()` method in listing 1.9 uses an `XmlBeanFactory` to load `knight.xml` and to get a reference to the "knight" object.

Listing 1.9 Running the knight example

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class KnightApp {
  public static void main(String[] args) throws Exception {
    BeanFactory factory =
        new XmlBeanFactory(new FileInputStream("knight.xml"));     Load the XML beans file

    KnightOfTheRoundTable knight =
        (KnightOfTheRoundTable) factory.getBean("knight");         Retrieve a knight from the factory

    knight.embarkOnQuest();    <— Send knight on its quest
  }
}
```

Once the application has a reference to the `KnightOfTheRoundTable` object, it simply calls the `embarkOnQuest()` method to kick off the knight's adventure. Notice that this class knows nothing about the quest the knight will take. Again, the only thing that knows which type of quest will be given to the knight is the `knight.xml` file.

It's been a lot of fun sending knights on quests using inversion of control, but now let's see how you can use IoC in your real-world enterprise applications.[4]

1.4.3 IoC in enterprise applications

Suppose that you've been tasked with writing an online shopping application. Included in the application is an `OrderServiceBean`, implemented as a stateless session bean. Now you want to have a class that creates an `Order` object from user input (likely an HTML form) and call the `createOrder()` method on your `OrderServiceBean`, as shown in listing 1.10.

Listing 1.10 Creating an order using EJB

```
...
private OrderService orderService;

public void doRequest(HttpServletRequest request) {
  Order order = createOrder(request);
  OrderService orderService = getOrderService();
  orderService.createOrder(order);
}
```

[4] This assumes that your real-world applications do not involve knights and quests. In the event that your current project does involve knights and quests, you may disregard the next section.

```
private OrderService getOrderService() throws CreateException {
  if (orderService == null) {
    Context initial = new InitialContext();
    Context myEnv = (Context) initial.lookup("java:comp/env");
    Object ref = myEnv.lookup("ejb/OrderServiceHome");
    OrderServiceHome home = (OrderServiceHome)
          PortableRemoteObject.narrow(ref, OrderService.class);
    orderService = home.create();
  }
  return orderService;
}
...
```

Get the JNDI Context

Retrieve an EJB Home from JNDI

Get the Remote object from the Home object

Notice that it took five lines of code *just* to get your `OrderService` object. Now imagine having to do this everywhere you need an `OrderService` object. Now imagine you have ten other EJBs in your application. That is a lot of code! But duplicating this code everywhere would be ridiculous, so a `ServiceLocator` is typically used instead. A `ServiceLocator` acts as a central point for obtaining and caching EJBHome references:

```
private OrderService getOrderService() {
  OrderServiceHome home =
    ServiceLocator.locate(OrderServiceHome);
  OrderService orderService = home.create();
}
```

While this removes the need to duplicate the lookup code everywhere in the application, one problem still remains: we always have to explicitly look up our services in our code.

Now let's see how this would be implemented in Spring:

```
private OrderService orderService;

public void doRequest(HttpServletRequest request) {
  Order order = createOrder(request);
  orderService.createOrder(order);
}

public void setOrderService(OrderService orderService) {
  this.orderService = orderService;
}
```

No lookup code! The reference to `OrderService` is given to our class by the Spring container through the `setOrderService()` method. With Spring, we never have to trouble ourselves with fetching our dependencies. Instead, our code can focus on the task at hand.

But inversion of control is only one of the techniques that Spring offers to JavaBeans. There's another side to Spring that makes it a viable framework for enterprise development. Let's take a quick look at Spring's support for aspect-oriented programming.

1.5 Applying aspect-oriented programming

While inversion of control makes it possible to tie software components together loosely, aspect-oriented programming enables you to capture functionality that is used throughout your application in reusable components.

1.5.1 Introducing AOP

Aspect-oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often, however, these components also carry additional responsibility beyond their core functionality. System services such as logging, transaction management, and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as *cross-cutting concerns* because they tend to cut across multiple components in a system.

By spreading these concerns across multiple components, you introduce two levels of complexity to your code:

- The code that implements the systemwide concerns is duplicated across multiple components. This means that if you need to change how those concerns work, you'll need to visit multiple components. Even if you've abstracted the concern to a separate module so that the impact to your components is a single method call, that single method call is duplicated in multiple places.
- Your components are littered with code that isn't aligned with their core functionality. A method to add an entry to an address book should only be concerned with how to add the address and not with whether it is secure or transactional.

Figure 1.4 illustrates this complexity. The business objects on the left are too intimately involved with the system services. Not only does each object know that it is being logged, secured, and involved in a transactional context, but also each object is responsible for performing those services for itself.

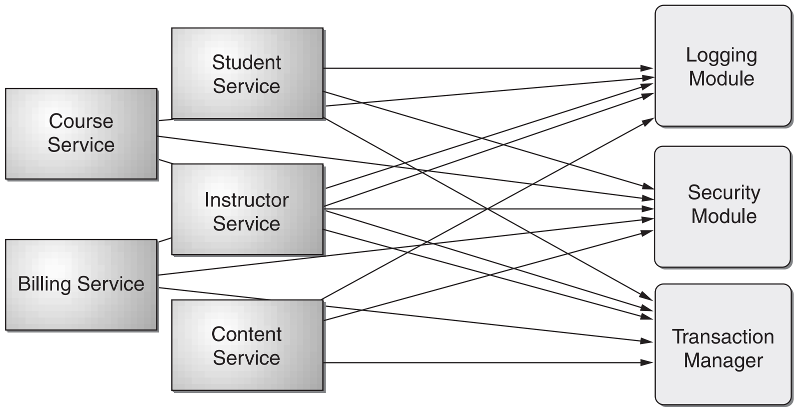

Figure 1.4 Calls to system-wide concerns such as logging and security are often scattered about in modules where those concerns are not their primary concern.

AOP makes it possible to modularize these services and then apply them declaratively to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved.

As shown in figure 1.5, it may help to think of aspects as blankets that cover many components of an application. At its core, an application is comprised of modules that implement the business functionality. With AOP, you can then cover

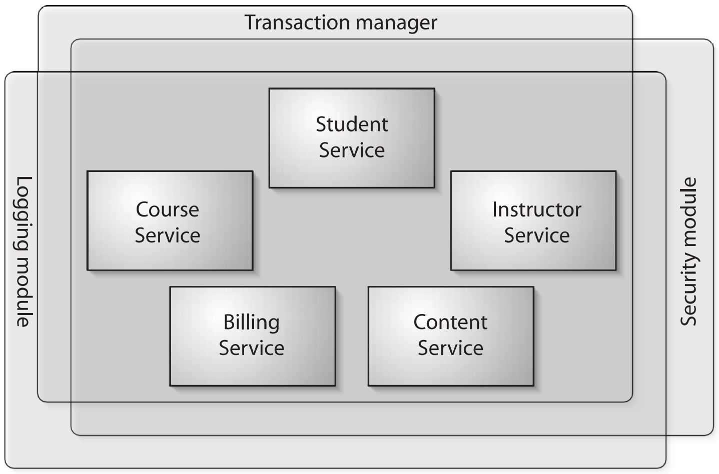

Figure 1.5 Using AOP, systemwide concerns blanket the components that they impact.

your core application with layers of functionality. These layers can declaratively be applied throughout your application in a flexible manner without your core application even knowing they exist. This is a very powerful concept.

1.5.2 AOP in action

Let's revisit our knight example to see how AOP works with Spring. Suppose that after showing your progress to marketing, they came back with an additional requirement. In this new requirement, a minstrel must accompany each knight, chronicling the actions and deeds of the knight in song.[5]

To start, you create a `Minstrel` class:

```
package com.springinaction.chapter01.knight;

import org.apache.log4j.Logger;

public class Minstrel {
  Logger song = Logger.getLogger(KnightOfTheRoundTable.class);
  public Minstrel() {}

  public void compose(String name, String message) {
    song.debug("Fa la la! Brave " + name + " did " + message + "!");
  }
}
```

In keeping with the IoC way of doing things, you alter `KnightOfTheRoundTable` to be given an instance of `Minstrel`:

```
public class KnightOfTheRoundTable {
…
  private Minstrel minstrel;
  public void setMinstrel(Minstrel minstrel) {
    this.minstrel = minstrel;
  }

…

  public HolyGrail embarkOnQuest() throws QuestException {
    minstrel.compose(name, "embark on a quest");
    return quest.embark();
  }
}
```

[5] Think of minstrels as musically inclined logging systems of medieval times.

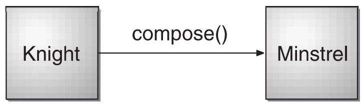

Figure 1.6 Without AOP, a knight must tell his minstrel to compose songs.

There's only one problem. As it is, each knight must stop and tell the minstrel to compose a song before the knight can continue with his quest (as in figure 1.6). Ideally a minstrel would automatically compose songs without being explicitly told to do so. A knight shouldn't know (or really even care) that their deeds are being written into song. After all, you can't have your knight being late for quests because of a lazy minstrel.

In short, the services of a minstrel transcend the duties of a knight. Another way of stating this is to say that a minstrel's services (song writing) are orthogonal to a knight's duties (embarking on quests). Therefore, it makes sense to implement a minstrel as an aspect that adds its song-writing services to a knight. Probably the simplest way to create an aspect-oriented minstrel is to change the minstrel class to be an implementation of `MethodBeforeAdvice`, as shown in listing 1.11.

Listing 1.11 An aspect-oriented minstrel

```
package com.springinaction.chapter01.knight;

import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class MinstrelAdvice
    implements MethodBeforeAdvice {
  public MinstrelAdvice() {}

  public void before(Method method, Object[] args,      <-- Advise method
      Object target) throws Throwable {                     before call

    Knight knight = (Knight) target;

    Logger song =
        Logger.getLogger(target.getClass());       <-- Get the advised
                                                       class's logger
    song.debug("Brave " + knight.getName() +
        " did " + method.getName());
  }
}
```

As a subclass of `MethodBefore-Advice`, the `MinstrelAdvice` class will intercept calls to the target object's methods, giving the `before()` method an opportunity to do something before the target method gets called. In this case, `MinstrelAdvice` naively assumes that the target object is a `KnightOfTheRoundTable` and uses log4j as its mechanism for chronicling the knight's actions. As illustrated in figure 1.7, the knight needn't worry about how he is being sung about or even that the minstrel is writing the song.

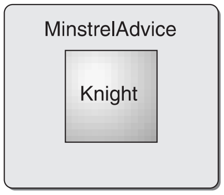

Figure 1.7
An aspect-oriented minstrel covers a knight, chronicling the knight's activities without the knight's knowledge of the minstrel.

The knight no longer needs to tell this new aspect-oriented minstrel to sing about the knight's activities. In fact, the knight doesn't even need to know that the minstrel exists. But how does `MinstrelAdvice` know that it is supposed to intercept calls to a `Knight`?

Weaving the aspect

Notice that there's nothing about `MinstrelAdvice` that tells the `Minstrel` what object it should sing about. Instead, a `Minstrel`'s services are applied to a `Knight` declaratively. Applying advice to an object is known as *weaving*. In Spring, aspects are woven into objects in the Spring XML file, much in the same way that beans are wired together. Listing 1.12 shows the new knight.xml, modified to weave `MinstrelAdvice` into a `KnightOfTheRoundTable`.

Listing 1.12 Weaving MinstrelAdvice into a knight

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="quest"
      class="com.springinaction.chapter01.knight.HolyGrailQuest"/>

  <bean id="knightTarget"
      class="com.springinaction.chapter01.knight.KnightOfTheRoundTable">
    <constructor-arg><value>Bedivere</value></constructor-arg>

    <property name="quest"><ref bean="quest"/></property>
  </bean>

  <bean id="minstrel"
      class="com.springinaction.chapter01.knight.MinstrelAdvice"/>
```

Create a minstrel instance

```
  <bean id="knight"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <list>
        <value>com.springinaction.chapter01.knight.Knight</value>   <-- Intercept calls to the knight
      </list>
    </property>
    <property name="interceptorNames">
      <list>
        <value>minstrel</value>   <-- Let minstrel handle call first
      </list>
    </property>
    <property name="target"><ref bean="knightTarget"/></property>   <-- Then let the knight handle the call
  </bean>
</beans>
```

Notice that the id of KnightOfTheRoundTable has changed from knight to knightTarget and now knight points to a Spring class called ProxyFactoryBean. What this means is that when the container is asked for a knight object, it will return an object that intercepts calls to the target KnightOfTheRoundTable object, giving MinstrelAdvice a shot at handling method calls first. Once MinstrelAdvice is finished, control is returned to KnightOfTheRoundTable to perform the knightly task.

Don't worry if this doesn't make sense yet. We'll explain Spring's AOP support in more detail in chapter 3. For now, suffice it to say that even though a knight's every move is being observed by a minstrel, the knight's activities are in no way hampered because of the minstrel's presence.

But Spring's AOP can be used for even more practical things than composing ageless sonnets about knights. As you'll see, AOP can be used to provide enterprise services such as declarative transactions and security.

1.5.3 AOP in the enterprise

Enterprise applications often require certain services such as security and transactional support. One way of applying these services is to code support for them directly into the classes that use them. For example, to handle transactions, you may place the following snippet throughout your code:

```
UserTransaction transaction = null;
try {
  transaction = ... {retrieve transaction}

  transaction.begin();

  ... do stuff...
```

```
    transaction.commit();
  } catch (Exception e) {
    if (transaction != null) transaction.rollback();
  }
```

The problem with handling transactions this way is that you may repeat the same transaction handling code several times—once for each time you need a transactional context. What's more, your application code is responsible for more than its core functionality.

EJB simplifies things by making it possible to declare these services and their policies in the EJB deployment descriptor. With EJB it is possible to write components that are ignorant of the fact that they are in a transactional context or being secured and then declare the transactional and security policies for those components in the EJB deployment descriptor. For example, to ensure that a method is transactional in EJB, you simply place the following in the deployment descriptor:

```
<container-transaction>
  <method>
    <ejb-name>Foo</ejb-name>
    <method-intf>Remote</method-inf>
    <method-name>doSomething</method-name>
  </method>
  <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
```

EJB has hung its hat on how it simplifies infrastructure logic such as transactions and security. But as we discussed in the introduction to this chapter, EJB has complicated matters in other ways.

Although Spring's AOP support can be used to separate cross-cutting concerns from your application's core logic, its primary job is as the basis for Spring's support for declarative transactions. Spring comes with several aspects that make it possible to declare transaction policies for JavaBeans. And the Acegi Security System (another open-source project associated with Spring) provides declarative security to JavaBeans. As with all Spring configuration, the transactional and security policies are prescribed in a Spring configuration file.

NOTE Although the Spring framework comes packed with several frameworks and support for several enterprise-level services, it does not come with much to assist you with security. The Acegi security system uses Spring's AOP support as the foundation of a framework that adds declarative security to Spring-enabled applications. You will learn more about Acegi in chapter 11.

For example, suppose that instead of a knight your application handles student registration for training courses. Perhaps you have a bean called `StudentServiceImpl` that implements the following interface:

```
public StudentService {
  public void registerForCourse(Student student, Course course);
}
```

This bean may be registered in the Spring bean XML file as follows:

```
<bean id="studentServiceTarget"
    class="com.springinaction.training.StudentServiceImpl"/>
```

`StudentService`'s `registerForCourse()` method should perform the following actions:

1. Verify that there is an available seat in the course.
2. Add the student to the course's roster.
3. Decrement the course's available seat count by 1.
4. Notify the student by e-mail of a successful registration.

All of these actions should happen atomically. If anything goes bad, then all should be rolled back as if nothing happened. Now imagine if instead of a minstrel providing musical logging to this class, you were to apply one of Spring's transaction manager aspects. Applying transactional support to `StudentServiceImpl` might be as simple as adding the lines shown in listing 1.13 to the bean XML file.

Listing 1.13 Declaring StudentService to be transactional

```
<bean id="transactionManager" class=
    "org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory">            Declare transaction manager
    <ref bean="sessionFactory"/>
  </property>
</bean>

<bean id="studentService" class=
"org.springframework.transaction.interceptor.
  ➥ TransactionProxyFactoryBean">

  <property name="target">
    <ref bean="studentServiceTarget"/>   <— Apply transactions
  </property>

  <property name="transactionAttributes">
    <props>
```

```
        <prop key="registerForCourse">                          Declare
            PROPAGATION_REQUIRES_NEW,ISOLATION_DEFAULT           transaction
        </prop>
      </props>
    </property>

    <property name="transactionManager">
        <ref bean="transactionManager"/>   <— Inject transaction
    </property>
  </bean>
```

Here we make use of Spring's `TransactionProxyFactoryBean`. This is a convenience proxy class that allows us to intercept method calls to an existing class and apply a transaction context. In this case we are creating a proxy to our `StudentServiceImpl` class and applying a transaction to the `registerForCourse()` method. We are also using `HibernateTransactionManager`, the implementation of a transaction manager you would most likely use if your application's persistence layer is based on Hibernate.

Although this example leaves a lot to be explained, it should give you a glimpse of how Spring's AOP support can provide plain-vanilla JavaBeans with declarative services such as transactions and security. We'll dive into more details of Spring's declarative transaction support in chapter 5.

1.6 Spring alternatives

Whew! After that whirlwind introduction of Spring, you have a pretty good idea of what it can do. Now you are probably chomping at the bit to get down into the details so you can see how you can use Spring for your projects. But before we do that, we need to cover what else is out there in the world of J2EE frameworks.

1.6.1 Comparing Spring to EJB

Because Spring comes with rich support for enterprise-level services, it is positioned as a viable alternative to EJB. But EJB, as opposed to Spring, is a well-established platform. Therefore, the decision to choose one over the other is not one to be taken lightly. Also, you do not necessarily have to choose only Spring *or* EJB. Spring can be used to support existing EJBs as well, a topic that will be discussed in detail in chapter 7. With that in mind, it is important to know what these two have in common, what sets them apart, and the implications of choosing either.

EJB is a standard

Before we delve into the technical comparisons between Spring and EJB, there is an important distinction that we need to make. EJB is a *specification* defined by the JCP. Being a standard has some significant implications:

- *Wide industry support*—There is a whole host of vendors that are supporting this technology, including industry heavyweights Sun, IBM, Oracle, and BEA. This means that EJB will be supported and actively developed for many years to come. This is comforting to many companies because they feel that by selecting EJB as their J2EE framework, they are going with a safe choice.
- *Wide adoption*—EJB as a technology is deployed in thousands of companies around the world. As a result, EJB is in the tool bag of most J2EE developers. This means that if a developer knows EJB, they are more likely to find a job. At the same time, companies know that if they adopt EJB, there is an abundance of developers who are capable of developing their applications.
- *Toolability*—The EJB specification is a fixed target, making it easy for vendors to produce tools to help developers create EJB applications more quickly and easily. Dozens of applications are out there that do just that, giving developers a wide range of EJB tool options.

Spring and EJB common ground

As J2EE containers, both Spring and EJB offer the developer powerful features for developing applications. Table 1.1 lists the major features of both frameworks and how the implementations compare.

Table 1.1 Spring and EJB feature comparison

Feature	EJB	Spring
Transaction management	■ Must use a JTA transaction manager. ■ Supports transactions that span remote method calls.	■ Supports multiple transaction environments through its `PlatformTransactionManager` interface, including JTA, Hibernate, JDO, and JDBC. ■ Does not natively support distributed transactions—it must be used with a JTA transaction manager.

continued on next page

Table 1.1 Spring and EJB feature comparison *(continued)*

Feature	EJB	Spring
Declarative transaction support	▪ Can define transactions declaratively through the deployment descriptor. ▪ Can define transaction behavior per method or per class by using the wildcard character `*`. ▪ Cannot declaratively define rollback behavior—this must be done programmatically.	▪ Can define transactions declaratively through the Spring configuration file or through class metadata. ▪ Can define which methods to apply transaction behavior explicitly or by using regular expressions. ▪ Can declaratively define rollback behavior per method and per exception type.
Persistence	▪ Supports programmatic bean-managed persistence and declarative container managed persistence.	▪ Provides a framework for integrating with several persistence technologies, including JDBC, Hibernate, JDO, and iBATIS.
Declarative security	▪ Supports declarative security through users and roles. The management and implementation of users and roles is container specific. ▪ Declarative security is configured in the deployment descriptor.	▪ No security implementation out-of-the box. ▪ Acegi, an open source security framework built on top of Spring, provides declarative security through the Spring configuration file or class metadata.
Distributed computing	▪ Provides container-managed remote method calls.	▪ Provides proxying for remote calls via RMI, JAX-RPC, and web services.

For *most* J2EE projects, the technology requirements will be met by either Spring or EJB. There are exceptions—your application may need to be able to support remote transaction calls. If that is the case, EJB may seem like the the way to go. Even then, Spring integrates with a Java Transaction API (JTA) transaction providers, so even this scenario is cut-and-dried. But if you are looking for a J2EE framework that provides declarative transaction management and a flexible persistence engine, Spring is a great choice. It lets you choose the features you want without the added complexities of EJB.

The complexities of EJB

So what are the complexities of EJB? Why is there such a shift toward lightweight containers? Here are a few of the complexities of EJB that turn off many developers:

- *Writing an EJB is overly complicated*—To write an EJB, you have to touch *at least* four files: the business interface, the home interface, the bean implementation, and the deployment descriptor. Other classes are likely to be involved as well, such as utility classes and value objects. That's quite a proliferation of files when all you are looking for is to add some container services to your implementation class. Conversely, Spring lets you define your implementation as a POJO and wire in any additional services needs through injection or AOP.

- *EJB is invasive*—This goes hand in hand with the previous point. In order to use the services provided by the EJB container, you *must* use the `javax.ejb` interfaces. This binds your component code to the EJB technology, making it difficult (if not possible) to use the component outside of an EJB container. With Spring, components are typically not required to implement, extend, or use any Spring-specific classes or interfaces, making it possible to reuse the components anywhere, even in the absence of Spring.
- *Entity EJBs fall short*—Entity EJBs are not as flexible or feature-rich as other ORM tools. Spring recognizes there are some great ORM tools out there, such as Hibernate and JDO, and provides a rich framework for integrating them into your application. And since an entity bean could represent a remote object, the Value Object pattern was introduced to pass data to and from the EJB tier in a course-grained object. But value objects lead to code duplication—you write each persistent property twice: once in the entity bean and once in your value object. Using Spring together with Hibernate or another ORM framework, your application's entity objects are not directly coupled with their persistence mechanism. This makes them light enough to be passed across application tiers.

Again, in most J2EE applications, the features provided by EJB may not be worth the compromises you will have to make. Spring provides nearly all of the services provided by an EJB container while allowing you to develop much simpler code. In other words, for a great number of J2EE applications, Spring makes sense. And now that you know the differences between Spring and EJB, you should have a good idea which framework fits your needs best.

1.6.2 *Considering other lightweight containers*

Spring is not the only lightweight container available. In the last few years, more and more Java developers have been seeking an alternative to EJB. As a result, several lightweight containers have been developed with different methods for achieving inversion of control.

Table 1.2 lists the types of IoC. These were first described with the nondescript "Type X" convention, but have since shifted to more meaningful names. We will always refer to them by the name.

Table 1.2 Inversion of Control types

Type	Name	Description
Type 1	Interface Dependent	Beans must implement specific interfaces to have their dependencies managed by the container.
Type 2	Setter Injection	Dependencies and properties are configured through a bean's setter methods.
Type 3	Constructor Injection	Dependencies and properties are configured through the bean's constructor.

Although the focus of this book is on Spring, it may be interesting to see how these other containers stack up to Spring. Let's take a quick look at some of the other lightweight containers, starting with PicoContainer.

PicoContainer

PicoContainer is a minimal lightweight container that provides IoC in the form of constructor and setter injection (although it favors constructor injection). We use the word *minimal* to describe PicoContainer because, with it small size (~50k), it has a sparse API. PicoContainer provides the bare essentials to create an IoC container and expects to be extended by other subprojects and applications. By itself, you can only assemble components programmatically through PicoContainer's API. Since this would be a cumbersome approach for anything but the most trivial applications, there is a subproject named NanoContainer that provides support for configuring PicoContainer through XML and various scripting languages. However, at the time of this writing, NanoContainer does not appear to be production-ready.

One of the limitations of PicoContainer is that it allows only one instance of any particular type to be present in its registry. This is could lead to problems if you need more than one instance of the same class, just configured differently. For example, you may want to have two instances of a `javax.sql.DataSource` in your application, each configured for a different database. This would not be possible in PicoContainer.

Also, you should know that PicoContainer is only a container. It does not offer any of the other powerful features that Spring has, such as AOP and third-party framework integration.

HiveMind

HiveMind is a relatively new IoC container. Like PicoContainer, it focuses on wiring and configuring services with support for both constructor and setter injection. HiveMind allows you to define your configuration in an XML file or in HiveMind's Simple Data Language.

HiveMind also provides an AOP-like feature with its *Interceptors*. This allows you to wrap a service with Interceptors to provide additional functionality. However, this is not nearly as powerful as Spring's AOP framework.

Finally, like PicoContainer, HiveMind is *only* a container. It provides a framework for managing components but offers no integration with other technologies.

Avalon

Avalon was one of the first IoC containers developed. As with many early entrants into a market, some mistakes were made in its design. Mainly, Avalon provides interface-dependent IoC. In other words, in order for your objects to be managed by the Avalon container, they must implement Avalon-specific interfaces. This makes Avalon an invasive framework; you must change *your* code in order for it to be usable by the container. This is not desirable because it couples your code to a particular framework for even the simplest of cases.

We believe that if Avalon does not adopt a more flexible means of managing components, it will eventually fade out of the lightweight container market; there are other ways of achieving the same results with much less rigidity.

1.6.3 Web frameworks

Spring comes with its own very capable web framework. It provides features found in most other web frameworks, such as automatic form data binding and validation, multipart request handling, and support for multiple view technologies. We'll talk more about Spring's web framework in chapter 8. But for now, let's take a look at how Spring measures up to some popular web frameworks

Struts

Struts can probably be considered the de facto standard for web MVC frameworks. In has been around for several years, was the first "Model 2" framework to gain wide adoption and has been used in thousands of Java projects. As a result, there is an abundance of resources available on Struts.

The Struts class you will use the most is the `Action` class. It is important to note that this is a class and not an interface. This means all your classes that handle input will need to subclass `Action`. This in contrast to Spring, which provides a `Controller` interface that you can implement.

Another important difference is how each handles form input. Typically, when a user is submitting a web form, the incoming data maps to an object in your application. In order to handle form submissions, Struts requires you have `ActionForm` classes to handle the incoming parameters. This means you need to create a class solely for mapping form submissions to your domain objects. Spring allows you to map form submissions directly to an object without the need for an intermediary, leading to eaiser maintenance.

Also, Struts comes with built-in support for declarative form validation. This means you can define rules for validating incoming form data in XML. This keeps validation logic out of your code, where it can be cumbersome and messy. Spring does not come with declarative validation. This does not mean you cannot use this within Spring; you will just have to integrate this functionality yourself using a validation framework, such as the Jakarta Commons Validator.

If you already have an investment in Struts or you just prefer it as your web framework, Spring has a package devoted to integrating Struts with Spring.

Furthermore, Struts is a mature framework with a significant following in the Java development community. Much has been written about Struts, including Ted Husted's *Struts in Action* (Manning, 2002).

WebWork

WebWork is another MVC framework. Like Struts and Spring, it supports multiple view technologies. One of the biggest differentiators for WebWork is that it adds another layer of abstraction for handling web requests. The core interface for handling requests is the `Action` interface, which has one method: `execute()`. Notice that this interface is not tied to the web layer in any way. The WebWork designers went out of their way to make the `Action` interface unaware that it could be used in a web context. This is good or bad, depending on your perspective. Most of the time it *will* be used in a web application, so hiding this fact through abstraction does not buy you much.

A feature that WebWork provides that Spring does not (at least, not explicitly) is *action chaining*. This allows you to map a logical request to a series of `Actions`. This means you can create several `Action` objects that all perform discrete tasks and chain them together to execute a single web request.

Tapestry

Tapestry is another open source web framework that is quite different than ones mentioned previously. Tapestry does not provide a framework around the request-response servlet mechanism, like Struts or WebWork. Instead, it is a framework for creating web applications from reusable components (if you are familiar with Apple's WebObjects, Tapestry was inspired by its design).

The idea behind Tapestry is to relieve the developer from thinking about Session attributes and URLs, and instead think of web applications in terms of components and methods. Tapestry takes on the other responsibilities, such as managing user state and mapping URLs to methods and objects.

Tapestry provides a view mechanism as well. That is, Tapestry is not a framework for using JSPs—it is an alternative to JSPs. Much of Tapestry's power lies in its custom tags that are embedded with HTML documents and used by the Tapestry framework. Needless to say, Tapestry provides a unique web application framework. To learn more about Tapestry, take a look at *Tapestry in Action* (Manning, 2004).

1.6.4 Persistence frameworks

There really isn't a direct comparison between Spring and any persistence framework. As mentioned earlier, Spring does not contain any built-in persistence framework. Instead, Spring's developers recognized there were already several good frameworks for this and felt no need to reinvent the wheel. They created an ORM module that integrates these frameworks with rest of Spring. Spring provides integration points for Hibernate, JDO, OJB, and iBATIS.

Spring also provides a very rich framework for writing JDBC. JDBC requires a lot of boilerplate code (getting resources, executing statements, iterating though query results, exception handling, cleaning up resources). Spring's JDBC module handles this boilerplate, allowing you to focus on writing queries and handling the results.

Spring's JDBC and ORM frameworks work within Spring's transaction management framework. This means you can use declarative transactions with just about any persistence framework you choose.

1.7 Summary

You should now have a pretty good idea of what Spring brings to the table. Spring aims to make J2EE development easier, and central to this is its inversion of control. This enables you to develop enterprise applications using simple Java objects that collaborate with each other through interfaces. These beans will be wired together at runtime by the Spring container. It lets you maintain loosely coupled code with minimal cost.

On top of Spring's inversion control, Spring's container also offers AOP. This allows you place code that would otherwise be scattered throughout you application in one place—an aspect. When your beans are wired together, these aspects can be woven in at runtime, giving these beans new behavior.

Staying true to aiding enterprise development, Spring offers integration to several persistence technologies. Whether you persist data using JDBC, Hibernate, or JDO, Spring's DAO frameworks ease your development by providing a consistent model for error handling and resource management for each of these persistence frameworks.

Complementing the persistence integration is Spring's transaction support. Through AOP, you can add declarative transaction support to your application without EJB. Spring also supports a variety of transaction scenarios, including integration with JTA transactions for distributed transactions.

Filling out its support for the middle tier, Spring offers integration with other various J2EE services, such as mail, EJBs, web services, and JNDI. With its inversion of control, Spring can easily configure these services and provide your application objects with simpler interfaces.

To help with the presentation tier, Spring supports multiple view technologies. This includes web presentation technologies like Velocity and JSP as well as support for creating Microsoft Excel spreadsheets and Adobe Acrobat Portable Document Format (PDF) files. And on top of the presentation, Spring comes with a built-in MVC framework. This offers an alternative to other web frameworks like Struts and WebWork and more easily integrates with all of the Spring services.

So without further ado, let's move on to chapter 2 to learn more about exactly how Spring's core container works.

JAVA

SPRING IN ACTION

Craig Walls • Ryan Breidenbach

Spring is a fresh breeze blowing over the Java landscape. Based on a design principle called Inversion of Control, Spring is a powerful but lightweight J2EE framework that does not require the use of EJBs. Spring greatly reduces the complexity of using interfaces, and speeds and simplifies your application development. You get the power and robust features of EJB *and* get to keep the simplicity of the non-enterprise JavaBean.

Spring in Action introduces you to the ideas behind Spring and then quickly launches into a hands-on exploration of the framework. Combining short code snippets and an ongoing example developed throughout the book, it shows you how to build simple and efficient J2EE applications. You will see how to solve persistence problems using the leading open-source tools, and also how to integrate your application with the most popular web frameworks. You will learn how to use Spring to manage the bulk of your infrastructure code so you can focus on what really matters—your critical business needs.

What's Inside

- Persistence using Hibernate, JDO, iBatis, OJB, and JDBC
- Declarative transactions and transaction management
- Integration with web frameworks:
  Struts, WebWork, Tapestry, Velocity
- Accessing J2EE services such as JMS and EJB
- Addressing cross-cutting concerns with AOP
- Enterprise applications best practices

Craig Walls is a software developer with over 10 years' experience and co-author of *XDoclet in Action*. He has sucessfully implemented a number of Spring applications. Craig lives in Denton, Texas. An avid supporter of open source Java technologies, **Ryan Breidenbach** has developed Java web applications for the past five years. He lives in Coppell, Texas.

"... a great way of explaining Spring topics... I enjoyed the entire book."

—Christian Parker
President Adigio Inc.

"... no other book can compare with the practical approach of this one."

—Olivier Jolly
J2EE Architect, Interface SI

"I thoroughly enjoyed the way Spring is presented."

—Norman Richards
co-author of *XDoclet in Action*

"I highly recommend it!"

—Jack Herrington, author of
Code Generation in Action

AUTHOR ONLINE
Ask the Authors

Ebook edition

www.manning.com/walls2

54495
9 781932 394351
ISBN 1-932394-35-4

MANNING $44.95 US/$60.95 Canada

Sample Chapter

Spring in Action

Craig Walls
Ryan Breidenbach

MANNING

Spring in Action

by Craig Walls

and

Ryan Breidenbach

Sample Chapter 7

brief contents

7 Accessing enterprise services

This chapter covers

- Accessing JNDI resources
- Sending and formatting email
- Scheduling tasks
- Integrating with EJBs

There are several enterprise services that Spring doesn't support directly. Instead Spring relies on other APIs to provide the services, but then places them under an abstraction layer so that they're easier to use.

You've already seen a few of Spring's abstraction layers. In chapter 4, you saw how Spring abstracts JDBC and Hibernate. In addition to eliminating the need to write certain boilerplate code, these abstractions eliminated the need for you to catch checked exceptions.

In this chapter, we're going to take a whirlwind tour of the abstraction layers that Spring provides for several enterprise services, including Spring's support for

- Java Naming and Directory Interface (JNDI)
- E-mail
- Scheduling
- Java Message Service (JMS)

We'll begin by looking at Spring's support for JNDI, since this provides the basis for several of the other abstraction layers.

7.1 Retrieving objects from JNDI

JNDI affords Java applications a central repository to store application objects. For example, a typical J2EE application uses JNDI to store and retrieve such things as JDBC data sources and JTA transaction managers.

But why would you want to configure these objects in JNDI instead of in Spring? Certainly, you could configure a `DataSource` object in Spring's configuration file, but you may prefer to configure it in an application server to take advantage of the server's connection pooling. Likewise, if your transactional requirements demand JTA transaction support, you'll need to retrieve a JTA transaction manager from the application server's JNDI repository.

Spring's JNDI abstraction makes it possible to declare JNDI lookups in your application's configuration file. Then you can wire those objects into the properties of other beans as though the JNDI object were just another POJO. Let's take a look at how to use Spring's JNDI abstraction to simplify lookup of objects in JNDI.

7.1.1 Working with conventional JNDI

Looking up objects in JNDI can be a tedious chore. For example, suppose you need to retrieve a `javax.sql.DataSource` from JNDI. Using the conventional JNDI APIs, your might write some code that looks like this:

```
InitialContext ctx = null;
try {
  ctx = new InitialContext();
  DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/myDatasource");
} catch (NamingException ne) {
  // handle naming exception
  …
} finally {
  if(ctx != null) {
    try {
      ctx.close();
    } catch (NamingException ne) {}
  }
}
```

At first glance, this may not look like a big deal. But take a closer look. There are a few things about this code that make it a bit clumsy:

- You must create and close an initial context for no other reason than to look up a `DataSource`. This may not seem like a lot of extra code, but it is extra plumbing code that is not directly in line with the goals of your application code.
- You must catch or, at very least, rethrow a `javax.naming.NamingException`. If you choose to catch it, you must deal with it appropriately. If you choose to rethrow it, then the calling code will be forced to deal with it. Ultimately, someone somewhere will have to deal with the exception.
- You code is tightly coupled with a JNDI lookup. All your code needs is a `DataSource`. It doesn't matter whether or not it comes from JNDI. But if your code contains code like that shown earlier, you're stuck retrieving the `DataSource` from JNDI.
- Your code is tightly coupled with a specific JNDI name—in this case `java:comp/env/jdbc/myDatasource`. Sure, you could extract that name into a properties file, but then you'll have to add even more plumbing code to look up the JNDI name from the properties file.

The overall problem with the conventional approach to looking up objects in JNDI is that it is the antithesis of dependency injection. Instead of your code being given an object, your code must go get the object itself. This means that your code is doing stuff that isn't really its job. It also means that your code is unnecessarily coupled to JNDI.

Regardless, this doesn't change the fact that sometimes you need to be able to look up objects in JNDI. `DataSource`s are often configured in an application server, to take advantage of the application server's connection pooling, and then retrieved by the application code to access the database. How can you get all the benefits of JNDI along with all of the benefits of dependency injection?

7.1.2 Proxying JNDI objects

Spring's `JndiObjectFactoryBean` gives you the best of both worlds. It is a factory bean, which means that when it is wired into a property, it will actually create some other type of object that will wire into that property. In the case of `JndiObjectFactoryBean`, it will wire an object retrieved from JNDI.

To illustrate how this works, let's revisit an example from chapter 4 (section 4.1.2). There you used `JndiObjectFactoryBean` to retrieve a `DataSource` from JNDI:

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean"
    singleton="true">
  <property name="jndiName">
    <value>java:comp/env/jdbc/myDatasource</value>
  </property>
</bean>
```

The `jndiName` property specifies the name of the object in JNDI. Here the full JNDI name of `java:comp/env/jdbc/myDatasource` is specified. However, if the object is a Java resource, you may choose to leave off `java:comp/env/` to specify the name more concisely. For example, the following declaration of the `jndiName` property is equivalent to the previous declaration:

```
<property name="jndiName">
  <value>jdbc/myDatasource</value>
</property>
```

With the `dataSource` bean declared, you may now inject it into a `DataSource` property. For instance, you may use it to configure a Hibernate session factory as follows:

```
<bean id="sessionFactory" class="org.springframework.orm.
      ➥ hibernate.LocalSessionFactoryBean">
  <property name="dataSource">
```

```
      <ref bean="dataSource"/>
    </property>
  …
  </bean>
```

When Spring wires the `sessionFactory` bean, it will inject the `DataSource` object retrieved from JNDI into the session factory's `dataSource` property.

The great thing about using `JndiObjectFactoryBean` to look up an object in JNDI is that the only part of the code that knows that the `DataSource` is retrieved from JNDI is the XML declaration of the `dataSource` bean. The `sessionFactory` bean doesn't know (or care) where the `DataSource` came from. This means that if you decide that you would rather get your `DataSource` from a JDBC driver manager, all you need to do is redefine the `dataSource` bean to be a `DriverManagerDataSource`.

We'll see even more uses of JNDI later in this chapter. But first, let's switch gears a bit and look at another abstraction provided by the Spring framework—Spring's e-mail abstraction layer.

7.2 Sending e-mail

Suppose that the course director of Spring Training has asked you to send her a daily e-mail outlining all of the upcoming courses, including a seat count and how many students have enrolled in the course. She'd like this report to be e-mailed at 6:00 a.m. every day so that she can see it when she first gets to work. Using this report, she'll schedule additional offerings of popular courses and cancel courses that aren't filling up very quickly.

As laziness is a great attribute of any programmer,[1] you decide to automate the e-mail so that you don't have to pull together the report every day yourself.

The first thing to do is to write the code that sends the e-mail (you'll schedule it for daily delivery in section 7.3).

To get started, you'll need a mail sender, defined by Spring's `MailSender` interface. A mail sender is an abstraction around a specific mail implementation. This decouples the application code from the actual mail implementation being used. Spring comes with two implementations of this interface:

[1] The other two attributes of a programmer are impatience and hubris. See *Programming Perl, 3rd Edition*, by Larry Wall et al. (O'Reilly & Associates, 2000).

- `CosMailSenderImpl`—Simple implementation of an SMTP mail sender based on Jason Hunter's COS (`com.oreilly.servlet)` implementation from his *Java Servlet Programming b*ook (O'Rielly, 1998).
- `JavaMailSenderImpl`—A JavaMail API-based implementation of a mail sender. Allows for sending of MIME messages as well as non-SMTP mail (such as Lotus Notes).

Either `MailSender` implementation is sufficient for the purposes of sending the report to the course director. But we'll choose `JavaMailSenderImpl` since it is the more versatile of the two. You'll declare it in your Spring configuration file as follows:

```
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host">
    <value>mail.springtraining.com</value>
  </property>
</bean>
```

The `host` property specifies the host name of the mail server, in this case Spring Training's SMTP server. By default, the mail sender assumes that the port is listening on port 25 (the standard SMTP port), but if your SMTP server is listening on a different port, you can set it using the `port` property of `JavaMailSenderImpl`.

The `mailSender` declaration above explicitly names the mail server that will send the e-mails. However, if you have a `javax.mail.MailSession` in JNDI (perhaps placed there by your application server) you have the option to retrieve it from JNDI instead. Simply use `JndiObjectFactoryBean` (as described in section 7.1) to retrieve the mail session and then wire it into the `mailSession` property as follows:

```
<bean id="mailSession"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/mail/Session</value>
  </property>
</bean>

<bean id="mailSender"
    class="org.springrframework.mail.javamail.JavaMailSenderImpl">
  <property name="session"><ref bean="mailSession"/></property>
</bean>
```

Now that the mail sender is set up, it's ready to send e-mails. But you might want to declare a template e-mail message:

```
<bean id="enrollmentMailMessage"
    class="org.springframework.mail.SimpleMailMessage">
  <property name="to">
    <value>coursedirector@springtraining.com</value>
  </property>
  <property name="from">
    <value>system@springtraining.com</value>
  </property>
  <property name="subject">
    <value>Course enrollment report</value>
  </property>
</bean>
```

Declaring a template e-mail message is optional. You could also create a new instance of `SimpleMailMessage` each time you send the e-mail. But by declaring a template in the Spring configuration file, you won't hard-code the e-mail addresses or subject in Java code.

The next step is to add a `mailSender` property to `CourseServiceImpl` so that `CourseServiceImpl` can use it to send the e-mail. Likewise, if you declared an e-mail template you should add a `message` property that will hold the message template bean:

```
public class CourseServiceImpl implements CourseService {
…
  private MailSender mailSender;
  public void setMailSender(MailSender mailSender) {
    this.mailSender = mailSender;
  }

  private SimpleMailMessage mailMessage;
  public void setMailMessage(SimpleMailMessage mailMessage) {
    this.mailMessage = mailMessage;
  }
…
}
```

Now that `CourseServiceImpl` has a `MailSender` and a copy of the e-mail template, you can write the `sendCourseEnrollementReport()` method (listing 7.1) that sends the e-mail to the course director. (Don't forget to add a declaration of `sendCourse-EnrollmentReport()` to the `CourseService` interface.)

Listing 7.1 Sending the enrollment report e-mail

```
public void sendCourseEnrollmentReport() {
  Set courseList = courseDao.findAll();

  SimpleMailMessage message =                        | Copy mail
      new SimpleMailMessage(this.mailMessage);       | template
```

```
    StringBuffer messageText = new StringBuffer();
    messageText.append(
        "Current enrollment data is as follows:\n\n");

    for(Iterator iter = courseList.iterator(); iter.hasNext(); ) {
      Course course = (Course) iter.next();
      messageText.append(course.getId() + "    ");
      messageText.append(course.getName() + "    ");
      int enrollment = courseDao.getEnrollment(course);
      messageText.append(enrollment);
    }

    message.setText(messageText.toString());   <— Set mail text

    try {
      mailSender.send(message);   <— Send e-mail
    } catch (MailException e) {
      LOGGER.error(e.getMessage());
    }
  }
```

The sendCourseEnrollmentReport() starts by retrieving all courses using the CourseDao. Then, it creates a working copy of the e-mail template so that the original will remain untouched. It then constructs the message body and sets the message text. Finally, the e-mail is sent using the mailSender property.

The final step is to wire the mailSender and enrollmentMailMessage beans into the courseService bean:

```
<bean id="courseService"
    class="com.springinaction.training.service.CourseServiceImpl">
…
  <property name="mailMessage">
    <ref bean="enrollmentMailMessage"/>
  </property>

  <property name="mailSender">
    <ref bean="mailSender"/>
  </property>
</bean>
```

Now that the courseService bean has everything it needs to send the enrollment report, the job is half done. Now the only thing left is to set it up on a schedule to send to the course director on a daily basis. Gee, it would be great if Spring had a way to help us schedule tasks…

7.3 Scheduling tasks

Not everything that happens in an application is the result of a user action. Sometimes the software itself initiates an action.

The enrollment report e-mail, for example, should be sent to the course director every day. To make this happen, you have two choices: You can either come in early every morning to e-mail the report manually or you can have the application perform the e-mail on a predefined schedule. (We think we know which one you would choose.)

Two popular scheduling APIs are Java's `Timer` class and OpenSymphony's Quartz scheduler.[2] Spring provides an abstraction layer for both of these schedulers to make working with them much easier. Let's look at both abstractions, starting with the simpler one, Java's `Timer`.

7.3.1 Scheduling with Java's Timer

Starting with Java 1.3, the Java SDK has included rudimentary scheduling functionality through its `java.util.Timer` class. This class lets you schedule a task (defined by a subclass `java.util.TimerTask`) to occur every so often.

Creating a timer task

The first step in scheduling the enrollment report e-mail using Java's `Timer` is to create the e-mail task by subclassing `java.util.TimerTask`, as shown in listing 7.2.

Listing 7.2 A timer task to e-mail the enrollment report

```
public class EmailReportTask extends TimerTask {
  public EmailReportTask() {}

  public void run() {
    courseService.sendCourseEnrollmentReport();    <— Send the report
  }

  private CourseService courseService;
  public void setCourseService(CourseService courseService) {
    this.courseService = courseService;                Inject the
  }                                                      CourseService
}
```

[2] Quartz is an open source job scheduling system from the OpenSymphony project. You can learn more about Quartz at http://www.opensymphony.com/quartz/.

The `run()` method defines what to do when the task is run. In this case, it calls the `sendCourseEnrollmentReport()` of the `CourseService` (see listing 7.1) to send the enrollment e-mail. As for the `CourseService`, it will be supplied to `EmailReport-Task` via dependency injection.

Declare the `EmailReportTask` in the Spring configuration file like this:

```
<bean id="reportTimerTask"
    class="com.springinaction.training.schedule.EmailReportTask">
  <property name="courseService">
    <ref bean="courseService"/>
  </property>
</bean>
```

By itself, this declaration simply places the `EmailReportTask` into the application context and wires the `courseService` bean into the `courseService` property. It won't do anything useful until you schedule it.

Scheduling the timer task

Spring's `ScheduledTimerTask` defines how often a timer task is to be run. Since the course director wants the enrollment report e-mailed to her every day, a `ScheduledTimerTask` should be wired as follows:

```
<bean id="scheduledReportTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">
  <property name="timerTask">
    <ref bean="reportTimerTask"/>
  </property>
  <property name="period">
    <value>86400000</value>
  </property>
</bean>
```

The `timerTask` property tells the `ScheduledTimerTask` which `TimerTask` to run. Here it is wired with a reference to the `reportTimerTask` bean, which is the `Email-ReportTask`. The `period` property is what tells the `ScheduledTimerTask` how often the `TimerTask`'s `run()` method should be called. This property, specified in milliseconds, is set to `86400000` to indicate that the task should be kicked off every 24 hours.

Starting the timer

The final step is to start the timer. Spring's `TimerFactoryBean` is responsible for starting timer tasks. Declare it in the Spring configuration file like this:

```
<bean class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
```

```
      <list>
        <ref bean="scheduledReportTask"/>
      </list>
    </property>
  </bean>
```

The `scheduledTimerTasks` property takes an array of timer tasks that it should start. Since you only have one timer task right now, the list contains a single reference to the `scheduledReportTask` bean.

Unfortunately, even though the task will be run every 24 hours, there is no way to specify what time of the day it should be run. `ScheduledTimerTask` does have a `delay` property that lets you specify how long to wait before the task is first run. For example, to delay the first run of `EmailReportTask` by an hour:

```
<bean id="scheduledReportTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">
  <property name="timerTask">
    <ref bean="reportTimerTask"/>
  </property>
  <property name="period">
    <value>86400000</value>
  </property>
  <property name="delay">
    <value>3600000</value>
  </property>
</bean>
```

Even with the delay, however, the time that the `EmailReportTask` will run will be relative to when the application starts. How can you have it sent at 6:00 a.m. every morning as requested by the course director (aside from starting the application at 5:00 a.m.)?

Unfortunately, that's a limitation of using Java's `Timer`. You can specify how often a task runs, but you can't specify exactly when it will be run. In order to specify precisely when the e-mail is sent, you'll need to use the Quartz scheduler instead.

7.3.2 Using the Quartz scheduler

The Quartz scheduler provides richer support for scheduling jobs. Just as with Java's `Timer`, you can use Quartz to run a job every so many milliseconds. But Quartz goes beyond Java's `Timer` by enabling you to schedule a job to run at a particular time and/or day.

For more information about Quartz, visit the Quartz home page at http://www.opensymphony.com/quartz.

Let's start working with Quartz by defining a job that sends the report e-mail.

Creating a job

The first step in defining a Quartz job is to create the class that defines the job. For that, you'll subclass Spring's `QuartzJobBean`, as shown in listing 7.3.

Listing 7.3 Defining a Quartz job

```
public class EmailReportJob extends QuartzJobBean {

  public EmailReportJob() {}

  protected void executeInternal(JobExecutionContext context)
      throws JobExecutionException {

    courseService.sendCourseEnrollmentReport();    <-- Send enrollment report
  }

  private CourseService courseService;
  public void setCourseService(CourseService courseService) {
    this.courseService = courseService;                    Inject
  }                                                 CourseService
}
```

A `QuartzJobBean` is the Quartz equivalent of a Java `TimerTask`. It is an implementation of the `org.quartz.Job` interface. The `executeInternal()` method defines the actions that the job does when its time comes. Here, just as with `EmailReportTask`, you simply call the `sendCourseEnrollmentReport()` method on the `courseService` property.

Declare the job in the Spring configuration file as follows:

```
<bean id="reportJob"
    class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass">
    <value>com.springinaction.training.
        ➥ schedule.EmailReportJob</value>
  </property>
  <property name="jobDataAsMap">
    <map>
      <entry key="courseService">
        <ref bean="courseService"/>
      </entry>
    </map>
  </property>
</bean>
```

Notice that you don't declare an `EmailReportJob` bean directly. Instead you declare a `JobDetailBean`. This is an idiosyncrasy of working with Quartz.

JobDetailBean is a subclass of Quartz's org.quartz.JobDetail, which requires that the Job object be set through the jobClass property.

Another quirk of working with Quartz's JobDetail is that the courseService property of EmailReportJob is set indirectly. JobDetail's jobDataAsMap takes a java.util.Map that contains properties that are to be set on the jobClass. Here, the map contains a reference to the courseService bean with a key of course-Service. When the JobDetailBean is instantiated, it will inject the courseService bean into the courseService property of EmailReportJob.

Scheduling the job

Now that the job is defined, you'll need to schedule the job. Quartz's org.quartz.Trigger class decides when and how often a Quartz job should run. Spring comes with two triggers, SimpleTriggerBean and CronTriggerBean. Which trigger should you use? Let's take a look at both of them, starting with SimpleTriggerBean.

SimpleTriggerBean is similar to ScheduledTimerTask. Using it, you can specify how often a job should run and (optionally) how long to wait before running the job for the first time. For example, to schedule the report job to run every 24 hours, with the first run starting after one hour, declare it as follows:

```
<bean id="simpleReportTrigger"
    class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail">
    <ref bean="reportJob"/>
  </property>
  <property name="startDelay">
    <value>3600000</value>
  </property>
  <property name="repeatInterval">
    <value>86400000</value>
  </property>
</bean>
```

The jobDetail property is wired to the job that is to be scheduled, here the reportJob bean. The repeatInterval property tells the trigger how often to run the job (in milliseconds). Here, we've set it to 86400000 so that it gets triggered every 24 hours. And the startDelay property can be used (optionally) to delay the first run of the job. We've set it to 3600000 so that it waits an hour before firing off for the first time.

Scheduling a `cron` job

Although you can probably think of many applications for which SimpleTrigger-Bean is perfectly suitable, it isn't sufficient for e-mailing the enrollment report. Just as with ScheduledTimerTask, you can only specify how often the job is run—not exactly when it is run. Therefore, you can't use SimpleTriggerBean to send the enrollment report to the course directory at 6:00 a.m. every day.

CronTriggerBean, however, gives you more precise control over when your job is run. If you're familiar with the Unix cron tool, then you'll feel right at home with CronTriggerBean. Instead of declaring how often a job is run you get to specify exact times (and days) for the job to run. For example, to run the report job every day at 6:00 a.m., declare a CronTriggerBean as follows:

```
<bean id="cronReportTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail">
    <ref bean="reportJob"/>
  </property>
  <property name="cronExpression">
    <value>0 0 6 * * ?</value>
  </property>
</bean>
```

As with SimpleTriggerBean, the jobDetail property tells the trigger which job to schedule. Again, we've wired it with a reference to the reportJob bean. The cron-Expression property tells the trigger when to fire. If you're not familiar with cron, this property may seem a bit cryptic, so let's examine this property a bit closer.

A cron expression has at least 6 (and optionally 7) time elements, separated by spaces. In order from left to right, the elements are defined as follows:

1. Seconds (0–59)
2. Minutes (0–59)
3. Hours (0–23)
4. Day of month (1–31)
5. Month (1–12 or JAN–DEC)
6. Day of week (1–7 or SUN–SAT)
7. Year (1970–2099)

Each of these elements can be specified with an explicit value (e.g., 6), a range (e.g., 9–12), a list (e.g., 9,11,13), or a wildcard (e.g., *). The day of the month and day of the week elements are mutually exclusive, so you should also indicate which one of these fields you don't want to set by specifying it with a question mark (?). Table 7.1 shows some example cron expressions and what they mean.

Table 7.1 Some sample `cron` expressions

Expression	What it means
0 0 10,14,16 * * ?	Every day at 10 a.m., 2 p.m., and 4 p.m.
0 0,15,30,45 * 1–10 * ?	Every 15 minutes on the first 10 days of every month
30 0 0 1 1 ? 2012	30 seconds after midnight on January 1, 2012
0 0 8-5 ? * MON–FRI	Every working hour of every business day

In the case of `cronReportTrigger`, we've set `cronExpression` to `0 0 6 * * ?` You can read this as "at the zero second of the zero minute of the sixth hour on any day of the month of any month (regardless of the day of the week), fire the trigger." In other words, the trigger is fired at 6:00 a.m. every day.

Using `CronTriggerBean`, you are able to adequately meet the course director's expectations. Now all that's left is to start the job.

Starting the job

Spring's `SchedulerFactoryBean` is the Quartz equivalent to `TimerFactoryBean`. Declare it in the Spring configuration file as follows:

```
<bean class="org.springframework.scheduling.
      ➥ quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref bean="cronReportTrigger"/>
    </list>
  </property>
</bean>
```

The `triggers` property takes an array of triggers. Since you only have a single trigger at this time, you simply need to wire it with a list containing a single reference to the `cronReportTrigger` bean.

At this point, you've satisfied the requirements for scheduling the enrollment report e-mail. But in doing so, you've done a bit of extra work. Before we move on, let's take a look at a slightly easier way to schedule the report e-mail.

7.3.3 Invoking methods on a schedule

In order to schedule the report e-mail you had to write the `EmailReportJob` bean (or the `EmailReportTask` bean in the case of timer tasks). But this bean does little more than make a simple call to the `sendCourseEnrollmentReport()` method of `CourseService`. In this light, `EmailReportTask` and `EmailReportJob` both seem a bit superfluous. Wouldn't it be great if you could specify that the `sendCourse-EnrollmentReport()` method be called without writing the extra class?

Good news! You can schedule single method calls without writing a separate `TimerTask` or `QuartzJobBean` class. To accomplish this, Spring has provided `MethodInvokingTimerTaskFactoryBean` and `MethodInvokingJobDetailFactoryBean` to schedule method calls with Java's timer support and the Quartz scheduler, respectively.

For example, to schedule a call to `sendCourseEnrollmentReport()` using Java's timer service, re-declare the `scheduledReportTask` bean as follows:

```
<bean id="scheduledReportTask">
    class="org.springframework.scheduling.timer.
          ➥ MethodInvokingTimerTaskFactoryBean">
  <property name="targetObject">
    <ref bean="courseService"/>
  </property>
  <property name="targetMethod">
    <value>sendCourseEnrollmentReport</value>
  </property>
</bean>
```

Behind the scenes, `MethodInvokingTimerTaskFactoryBean` creates a `TimerTask` that calls the method specified by the `targetMethod` property on the object specified by the `targetObject` property. This is effectively the same as the `EmailReportTask`.

With `scheduledReportTask` declared this way, you can now eliminate the `EmailReportTask` class and its declaration in the `reportTimerTask` bean.

`MethodInvokingTimerTaskFactoryBean` is good for making simple one-method calls when you are using a `ScheduledTimerTask`. But you're using Quartz's `CronTriggerBean` so that the report will be sent every morning at 6:00 a.m. So instead of using `MethodInvokingTimerTaskFactoryBean`, you'll want to re-declare the `reportJob` bean as follows:

```
<bean id="courseServiceInvokingJobDetail">
    class="org.springframework.scheduling.quartz.
          MethodInvokingJobDetailFactoryBean">
  <property name="targetObject">
    <ref bean="courseService"/>
  </property>
  <property name="targetMethod">
    <value>sendCourseEnrollmentReport</value>
  </property>
</bean>
```

`MethodInvokingJobDetailFactoryBean` is the Quartz equivalent of `MethodInvokingTimerTaskFactoryBean`. Under the covers, it creates a Quartz `JobDetail` object that makes a single method call to the object and method specified in the `targetObject` and `targetMethod` properties. Using `MethodInvokingJobDetail-FactoryBean` this way, you can eliminate the superfluous `EmailReportJob` class.

7.4 Sending messages with JMS

Most operations that take place in software are performed synchronously. In other words, when a routine is called the program flow is handed off to that routine to perform its functionality. Upon completion, control is returned to the calling routine and the program proceeds. Figure 7.1 illustrates this.

But sometimes, it's not necessary (or even desirable) to wait for the called routine to complete. For example, if the routine is slow, it may be preferable to send a message to a routine and then just assume that the routine will process the message or to check on its progress sometime later.

When you send a message to a routine and do not wait for a result, it is said to be *asynchronous*. Asynchronous program flow is illustrated in figure 7.2.

The Java Messaging Service (JMS) is a Java API for asynchronous processing. JMS supports two types of messaging: point-to-point and publish-subscribe.

A point-to-point message is placed into a message *queue* by the message producer and later pulled off the queue by the message consumer. Once the message is pulled from the queue, it is no longer available to any other message consumer that is watching the queue. This means that even though several consumers may observe a queue, a single consumer will consume each point-to-point message.

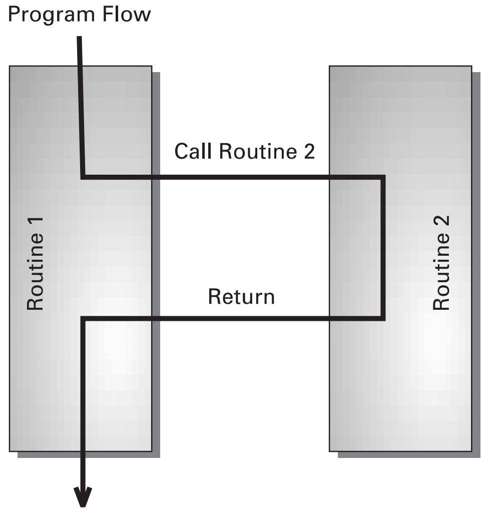

Figure 7.1 Synchronous program flow

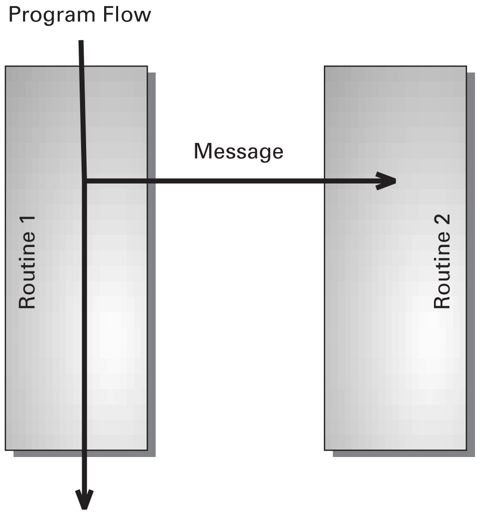

Figure 7.2 Asynchronous program flow

The publish-subscribe model invokes images of a magazine publisher who sends out copies of its publication to multiple subscribers. This is, in fact, a good analogy of how publish-subscribe messaging works. Multiple consumers subscribe to a message *topic*. When a message producer publishes a message to the topic, all subscribers will receive the message and have an opportunity to process it.

Spring provides an abstraction for JMS that makes it simple to access a message queue or topic (abstractly referred to as a *destination*) and publish messages to the destination. Moreover, Spring frees your application from dealing with `javax.jms.JMSException` by rethrowing any JMS exceptions as unchecked `org.springframework.jms.JmsExceptions`.

Let's see how to apply Spring's JMS abstraction.

7.4.1 Sending messages with JMS templates

In chapter 6 you learned to use Spring's remoting support to perform credit card authorization against the Spring Training payment service. Now you're ready to settle the account and receive payment.

When authorizing payment, it was necessary to wait for a response from the credit card processor, because you needed to know whether or not the credit card's issuing bank would authorize payment. But now that authorization has been granted, payment settlement can be performed asynchronously. There's no need to wait for a response—you can safely assume that the payment will be settled.

The credit card processing system accepts an asynchronous message, sent via JMS, for the purposes of payment settlement. The message it accepts is a `javax.jms.MapMessage` containing the following fields:

- `authCode`—The authorization code received from the credit card processor
- `creditCardNumber`—The credit card number
- `customerName`—The card holder's name
- `expirationMonth`—The month that the credit card expires
- `expirationYear`—The year that the credit card expires

Spring employs a callback mechanism to coordinate JMS messaging. This callback is reminiscent of the JDBC callback described in chapter 4. The callback is made up of two parts: a message creator that constructs a JMS message (`javax.jms.Message`) and a JMS template that actually sends the message.

Using the template

The first thing to do is to equip the `PaymentServiceImpl` class with a `JmsTemplate` property:

```
private JmsTemplate jmsTemplate;
public void setJmsTemplate(JmsTemplate jmsTemplate) {
  this.jmsTemplate = jmsTemplate;
}
```

The `jmsTemplate` property will be wired with an instance of `org.springframework.jms.core.JmsTemplate` using setter injection. We'll show you how to wire this a little bit later. First, however, let's implement the service-level method that sends the settlement message.

`PaymentServiceImpl` will need a `sendSettlementMessage()` method to send the settlement message to the credit card processor. Listing 7.4 shows how `sendSettlementMessage()` uses the `JmsTemplate` to send the message. (The `PaySettlement` argument is a simple JavaBean containing the fields needed for the message.)

Listing 7.4 Sending a payment settlement via the JMS callback

```
public void sendSettlementMessage(final PaySettlement settlement) {
  jmsTemplate.send(    <- Send message

      new MessageCreator() {    <- Define message creator
        public Message createMessage(Session session)
            throws JMSException {
```

```
            MapMessage message = session.createMapMessage();
            message.setString("authCode",
                settlement.getAuthCode());
            message.setString("customerName",
                settlement.getCustomerName());
            message.setString("creditCardNumber",
                settlement.getCreditCardNumber());
            message.setInt("expirationMonth",
                settlement.getExpirationMonth());
            message.setInt("expirationYear",
                settlement.getExpirationYear());

            return message;
          }
        }
    );
  }
```

Construct message

The sendSettlementMessage() method uses the JmsTemplate's send() method to send the message. This method takes an instance of org.springframework.jms.core.MessageCreator, here defined as an anonymous inner class, which constructs the Message to be sent. In this case, the message is a javax.jms.MapMessage. To construct the message, the MessageCreator retrieves values from the PaySettlement bean's properties and uses them to set fields on the MapMessage.

Wiring the template

Now you must wire a JmsTemplate into the PaymentServiceImpl. The following XML from the Spring configuration file will do just that:

```
<bean id="paymentService"
    class="com.springinaction.training.service.PaymentServiceImpl">
…
  <property name="jmsTemplate">
    <ref bean="jmsTemplate"/>
  </property>
<bean>
```

The declaration of the jmsTemplate bean is as follows:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory">
    <ref bean="jmsConnectionFactory"/>
  </property>
  <property name="defaultDestination">
    <ref bean="destination"/>
    </property>
  </bean>
```

Notice that the jmsTemplate bean is wired with a JMS connection factory and a default destination. The connectionFactory property is mandatory because it is how JmsTemplate gets a connection to a JMS provider. In the case of the Spring Training application, the connection factory is retrieved from JNDI, as shown in the following declaration of the connectionFactory bean:

```
<bean id="jmsConnectionFactory"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>connectionFactory</value>
  </property>
</bean>
```

Wired this way, Spring will use JndiObjectFactoryBean (see section 7.1) to look up the connection factory in JNDI using the name java:comp/env/connection-Factory. (Of course, this assumes that you have a JMS implementation with an instance of JMSConnectionFactory registered in JNDI.)

The defaultDestination property defines the default JMS destination (an instance of javax.jms.Destination) that the message will be published to. Here it is wired with a reference to the destination bean. Just as with the connection-Factory bean, the destination bean will be retrieved from JNDI using a Jndi-ObjectFactoryBean:

```
<bean id="destination"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>creditCardQueue</value>
  </property>
</bean>
```

The defaultDestination property is optional. But because there's only one JMS destination for credit card messages, it is set here for convenience. If you do not set a default destination, then you must pass a Destination instance or the JNDI name of a Destination when you call JmsTemplate's send() method. For example, you'd use this to specify the JNDI name of the JMS destination in the call to send():

```
jmsTemplate.send(
    "creditCardQueue", new MessageCreator() { … });
```

Working with JMS 1.0.2

Until now, the jmsTemplate bean has been declared to be an instance of JmsTemplate. Although it isn't very apparent, this implies that the JMS provider implementation adheres to version 1.1 of the JMS specification. If your JMS provider is 1.0.2-compliant and not 1.1-compliant, then you'll want to use JmsTemplate102 instead of JmsTemplate.

The big difference between JmsTemplate and JmsTemplate102 is that JmsTemplate102 needs to know whether you're using point-to-point or publish-subscribe messaging. By default, JmsTemplate102 assumes that you'll be using point-to-point messaging, but you can specify publish-subscribe by setting the pubSubDomain property to true:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
…
  <property name="pubSubDomain">
    <value>true</value>
  </property>
</bean>
```

Other than that, you use JmsTemplate102 the same as you would JmsTemplate.

Handling JMS exceptions

An important thing to notice about using JmsTemplate is that you weren't forced to catch a javax.jms.JMSException. Many of JmsTemplate's methods (including send()) catch any JMSException that is thrown and converts it to an unchecked runtime org.springframework.jms.JmsException.

7.4.2 Consuming messages

Now suppose that you are writing the code for the receiving end of the settlement process. You're going to need to receive the message, convert it to a PaySettlement object, and then pass it on to be processed. Fortunately, JmsTemplate can be used for receiving messages as well as sending messages.

Listing 7.5 demonstrates how you might use JmsTemplate to receive a settlement message.

Listing 7.5 Receiving a PaySettlement message

```
public PaySettlement processSettlementMessages() {
  Message msg = jmsTemplate.receive("creditCardQueue");     <- Receive message
```

```
    try {
      MapMessage mapMessage = (MapMessage) msg;
      PaySettlement paySettlement = new PaySettlement();

      paySettlement.setAuthCode(mapMessage.getString("authCode"));
      paySettlement.setCreditCardNumber(
          mapMessage.getString("creditCardNumber"));
      paySettlement.setCustomerName(
          mapMessage.getString("customerName"));
      paySettlement.setExpirationMonth(
          mapMessage.getInt("expirationMonth"));
      paySettlement.setExpirationYear(                Map message to
          mapMessage.getInt("expirationYear"));       PaySettlement

      return paySettlement;
    } catch (JMSException e) {
      throw JmsUtils.convertJmsAccessException(e);
    }
  }
```

The `receive()` method of `JmsTemplate` attempts to receive a `Message` from the specified `Destination`. As used earlier, `receive()` will try to receive a message from the `Destination` that has a JNDI name of `creditCardQueue`.

Once the `Message` is received, it is cast to a `MapMessage` and a `PaySettlement` object is initialized with the values from the fields of the `MapMessage`.

By default, `receive()` will wait indefinitely for the message. However, it may not be desirable to have your application block while it waits to receive a message. It'd be nice if you could set a timeout period so that `receive()` will give up after a certain time.

Fortunately, you can specify a timeout by setting the `receiveTimeout` property on the `jmsTemplate` bean. For example:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
  <property name="receiveTimeout">
    <value>10000</value>
  </property>
</bean>
```

The `receiveTimeout` property takes a value that is the number of milliseconds to wait for a message. Setting it to `10000` specifies that the `receive()` method should give up after 10 seconds. If no message is received in 10 seconds, the `JmsTemplate` will throw an unchecked `JmsException` (which you may choose to catch or ignore).

7.4.3 Converting messages

In listing 7.4, the `MessageCreator` instance was responsible for mapping the properties of `PaySettlement` to fields in a `MapMessage`. The `processSettlement()` message in listing 7.5 performs the reverse mapping of a `Message` to a `PaySettlement` object. That'll work fine, but it does result in a lot of mapping code that may end up being repeated every time you need to send or receive a `PaySettlement` message.

To avoid repetition and to keep the send and receive code clean, it may be desirable to extract the mapping code to a separate utility object.

Converting PaySettlement messages

Although you could write your own utility object to handle message conversion, Spring's `org.springframework.jms.support.converter.MessageConverter` interface defines a common mechanism for converting objects to and from JMS `Messages`.

To illustrate, `PaySettlementConverter` (listing 7.6) implements `MessageConverter` to accommodate the conversion of `PaySettlement` objects to and from JMS `Message` objects.

Listing 7.6 Convert a `PaySettlement` to and from a JMS `Message`

```
public class PaySettlementConverter implements MessageConverter {
  public PaySettlementConverter() {}

  public Object fromMessage(Message message)
      throws MessageConversionException {
    MapMessage mapMessage = (MapMessage) message;
    PaySettlement settlement = new PaySettlement();

    try {
      settlement.setAuthCode(mapMessage.getString("authCode"));
      settlement.setCreditCardNumber(
          mapMessage.getString("creditCardNumber"));
      settlement.setCustomerName(
          mapMessage.getString("customerName"));
      settlement.setExpirationMonth(
          mapMessage.getInt("expirationMonth"));
      settlement.setExpirationYear(
          mapMessage.getInt("expirationYear"));
    } catch (JMSException e) {
      throw new MessageConversionException(e.getMessage());
    }

    return settlement;
  }
```

Convert Message to PaySettlement

Rethrow as runtime exception

```
  public Message toMessage(Object object, Session session)
      throws JMSException, MessageConversionException {

    PaySettlement settlement = (PaySettlement) object;
    MapMessage message = session.createMapMessage();
    message.setString("authCode", settlement.getAuthCode());
    message.setString("customerName",
        settlement.getCustomerName());
    message.setString("creditCardNumber",
        settlement.getCreditCardNumber());
    message.setInt("expirationMonth",
        settlement.getExpirationMonth());
    message.setInt("expirationYear",
        settlement.getExpirationYear());

    return message;
  }
}
```

Convert PaySettlement to Message

As its name implies, the `fromMessage()` method is intended to take a `Message` object and convert it to some other object. In this case, the `Message` is converted to a `PaySettlement` object by pulling the fields out of the `MapMessage` and setting properties on the `PaySettlement` object.

The conversion is performed in reverse by the `toMessage()` method. This method takes an `Object` (in this case, assumed to be a `PaySettlement` bean) and sets elements in the `MapMessage` from the properties of the `Object`.

Wiring a message converter

To use the message converter, you first must declare it as a bean in the Spring configuration file:

```
<bean id="settlementConverter" class="com.springinaction.
      ➥ training.service.PaySettlementConverter">
…
</bean>
```

Next, the `JmsTemplate` needs to know about the message converter. You tell it about the `PaySettlementConverter` by wiring it into `JmsTemplate`'s `messageConverter` property:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
…
  <property name="messageConverter">
    <ref bean="settlementConverter"/>
  </property>
</bean>
```

Now that JmsTemplate knows about PaySettlementConverter, you're ready to send messages converted from PaySettlement objects.

Sending and receiving converted messages

With a converted message wired into PayServiceImpl, the implementation of sendSettlementMessage() becomes significantly simpler:

```
public void sendSettlementMessage(PaySettlement settlement) {
  jmsTemplate.convertAndSend(settlement);
}
```

Instead of calling JmsTemplate's send() method and using a MessageCreator to construct the Message object, you simply call JmsTemplate's convertAndSend() method passing in the PaySettlement object. Under the covers, the convertAndSend() method creates its own MessageCreator instance that uses PaySettlementConverter to create a Message object from a PaySettlement object.

Likewise, to receive converted messages, you call the JmsTemplate's receiveAndConvert() method (instead of the receive() method) passing the name of the JMS message queue:

```
PaySettlement settlement = (PaySettlement)
    jmsTemplate.receiveAndConvert("creditCardQueue");
```

Other than automatically converting Message objects to application objects, the semantics of receiveAndConvert() are the same as receive().

Using SimpleMessageConverter

Spring comes with one prepackaged implementation of the MessageConverter interface. SimpleMessageConverter converts MapMessages, TextMessages, and ByteMessages to and from java.util.Map collections, Strings, and byte arrays, respectively.

To use SimpleMessageConverter to convert PaySettlement objects to and from JMS Messages, replace the settlementConverter bean declaration with the following declaration:

```
<bean id="settlementConverter" class="org.springframework.jms.
      ➥ support.converter.SimpleMessageConverter">
…
</bean>
```

Although this converter's function is quite simple, it may prove useful when your messages are simple and do not correspond directly to an object in your application's domain.

7.5 Summary

Even though Spring provides functionality that eliminates much of the need to work with EJBs, there are still many enterprise services that Spring doesn't provide direct replacements for. In those cases, Spring provides abstraction layers that make it easy to wire those services into your Spring-enabled applications.

In this chapter, you've seen how to obtain references to objects that are kept in JNDI. These references could then be wired into bean properties as though they were locally defined beans. This proved to be useful throughout the chapter as you used Spring's JNDI abstraction to look up such things as mail sessions and JMS connection factories.

You've also seen how to send e-mails using Spring's e-mail abstraction and how to schedule tasks using either Java's `Timer` or OpenSymphony's Quartz scheduler.

Finally, you saw how to send and receive asynchronous messages using Spring's JMS abstraction.

In the next chapter, we'll move our focus to the presentation layer of our application, learning how to use Spring's MVC framework to develop web applications.

JAVA

SPRING IN ACTION

Craig Walls • Ryan Breidenbach

Spring is a fresh breeze blowing over the Java landscape. Based on a design principle called Inversion of Control, Spring is a powerful but lightweight J2EE framework that does not require the use of EJBs. Spring greatly reduces the complexity of using interfaces, and speeds and simplifies your application development. You get the power and robust features of EJB *and* get to keep the simplicity of the non-enterprise JavaBean.

Spring in Action introduces you to the ideas behind Spring and then quickly launches into a hands-on exploration of the framework. Combining short code snippets and an ongoing example developed throughout the book, it shows you how to build simple and efficient J2EE applications. You will see how to solve persistence problems using the leading open-source tools, and also how to integrate your application with the most popular web frameworks. You will learn how to use Spring to manage the bulk of your infrastructure code so you can focus on what really matters—your critical business needs.

What's Inside

- Persistence using Hibernate, JDO, iBatis, OJB, and JDBC
- Declarative transactions and transaction management
- Integration with web frameworks:
  Struts, WebWork, Tapestry, Velocity
- Accessing J2EE services such as JMS and EJB
- Addressing cross-cutting concerns with AOP
- Enterprise applications best practices

Craig Walls is a software developer with over 10 years' experience and co-author of *XDoclet in Action*. He has sucessfully implemented a number of Spring applications. Craig lives in Denton, Texas. An avid supporter of open source Java technologies, **Ryan Breidenbach** has developed Java web applications for the past five years. He lives in Coppell, Texas.

"… a great way of explaining Spring topics… I enjoyed the entire book."

—Christian Parker
President Adigio Inc.

"… no other book can compare with the practical approach of this one."

—Olivier Jolly
J2EE Architect, Interface SI

"I thoroughly enjoyed the way Spring is presented."

—Norman Richards
co-author of *XDoclet in Action*

"I highly recommend it!"

—Jack Herrington, author of
Code Generation in Action

AUTHOR ONLINE
Ask the Authors

Ebook edition

www.manning.com/walls2

54495
9 781932 394351
ISBN 1-932394-35-4

MANNING $44.95 US/$60.95 Canada